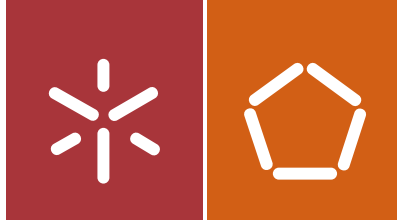




Universidade do Minho
Escola de Engenharia

Gil Fernando Ferreira da Cunha

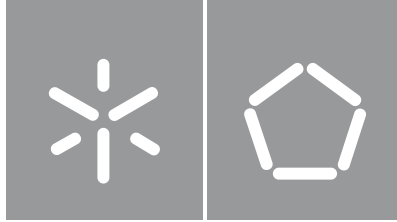
**Data Analysis and Recommender
System Architecture for
E-Commerce platforms**



Universidade do Minho
Escola de Engenharia

Gil Fernando Ferreira da Cunha

**Data Analysis and Recommender
System Architecture for
E-Commerce Platforms**



Universidade do Minho

Escola de Engenharia

Gil Fernando Ferreira da Cunha

**Data Analysis and Recommender
System Architecture for
E-Commerce Platforms**

Master's Dissertation

Integrated Master's in Informatics Engineering

Dissertation supervised by

Professor Dr. Hugo Daniel Abreu Peixoto

Professor Dr. José Manuel Ferreira Machado

COPYRIGHTS AND CONDITIONS OF USE BY THIRD PARTIES

This is an academic work that can be used by third parties as long as the internationally accepted rules and good practices are respected, with regard to copyright and related rights.

Thus, this work can be used under the terms provided for in the license below.

If the user needs permission to be able to use the work under conditions not provided for in the indicated license, he must contact the author, through the *RepositóriUM* of the University of Minho.



Attribution-NonCommercial

CC BY-NC

<https://creativecommons.org/licenses/by-nc/4.0/>

"Knowledge is power."

- Francis Bacon (1597)

"Knowledge is power.

Information is power.

*The secreting or hoarding of knowledge or information
may be an act of tyranny camouflaged as humility."*

- Robin Morgan

"Information is power only if you can take action with it.

Then, and only then, does it represent knowledge

and, consequently, power."

- Daniel Burrus

"Knowledge is power.

Information is liberating.

Education is the premise of progress,

in every society, in every family"

- Kofi Annan

ACKNOWLEDGEMENTS

AGRADECIMENTOS

First of all, I would like to thank my parents, Fernando and Elsa, for the tireless support that they gave me not only on this journey, but also continue to give throughout my life; for believing in me and making an effort to invest in my success, my gratitude. Following, I appreciate my brother Rafael for his friendship and attention whenever I needed it and for his confidence in my progress. I also thank my girlfriend Anabela, for her encouragement, her patience with me and for helping me to overcome the most difficult moments.

I also want to appreciate my family and friends in general for being present when I needed to and keeping my motivation in the project. I also acknowledge my advisors, Hugo and Francisco, for guiding me on this project and to my colleague Vitor for having accompanied me throughout the course.

Finally, I would like to thank Beevo and all the people who work there, my co-workers, for introducing me into the business world, for the opportunity to develop this project and for all the support they gave me on a personal and professional level.

To all these people, a big thank you and let this dissertation represent the outcome of all the support, dedication and affection that you have given me during this journey.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Gil Fernando Ferreira da Cunha

ABSTRACT

Data Analysis and Recommender System Architecture for E-Commerce platforms

E-commerce is constantly expanding, leading to greater market competitiveness. The number of online platforms offering products or services is increasing; so there is a growing need for companies to stand out from the competition, which leads to the application of various marketing strategies. However, not all are adequate and mismanagement, as well as a bad investment of these strategies, may prejudice companies.

Hence the implementation of recommendation systems in e-commerce platforms, as a safe and economical strategy. By investing in a good recommendation mechanism, one can provide better user experience, taking his interests into account. As a result, more traffic on the platforms is ensured, which may result in a higher sales rate and, consequently, a higher number of revenues.

However, to develop a recommendation system, the first step must consist in obtaining information about the sales platform, where data about its users and products/services form the basis of recommendations. But not all information is useful, which can influence the accuracy of the forecasting models used by the system to produce results.

Following this perspective, a data analysis methodology is proposed, as well as an architecture of a recommendation system, which allows to extract and treat relevant data, in order to integrate a recommendation engine for most e-commerce platforms.

Keywords: recommender system, data analysis, software architecture, e-commerce, business intelligence

RESUMO

Análise de Dados e Arquitetura de um Sistema de Recomendação para plataformas de Comércio Eletrônico

O comércio eletrônico (*e-commerce*) está em constante expansão, o que leva a uma maior competitividade no mercado. Existem cada vez mais plataformas de venda online e, conseqüentemente, há uma crescente necessidade das empresas se destacarem da concorrência, o que leva à aplicação das mais variadas estratégias de marketing. Porém, nem todas são adequadas e uma má gestão e investimento destas estratégias pode causar prejuízo às empresas.

Daí surge a implementação de sistemas de recomendação nas plataformas de venda, como uma estratégia segura e econômica. Ao investir num bom mecanismo de recomendação, é possível proporcionar uma melhor experiência para o utilizador, tendo em conta os seus interesses. Desta forma, assegura-se um maior tráfego nas plataformas, o que poderá resultar numa maior taxa de vendas e, conseqüentemente, num maior número de receitas.

No entanto, para desenvolver um sistema de recomendação é necessário, em primeiro lugar, obter informação sobre a plataforma de vendas, onde os dados sobre os seus utilizadores e produtos/serviços constituem a base das recomendações. Mas nem toda a informação é útil, o que pode influenciar a acurácia dos modelos de previsões utilizado pelo sistema.

Seguindo esta perspectiva, propõe-se uma metodologia de análise de dados, assim como uma arquitetura de um sistema de recomendação, que permitam extrair e tratar dados relevantes de modo a integrar um motor de recomendação para a generalidade das plataformas de *e-commerce*.

Palavras-chave: sistema de recomendação, análise de dados, arquitetura de software, comércio eletrônico, *business intelligence*

TABLE OF CONTENTS

Acknowledgements	viii
Abstract	x
Resumo	xi
List of Figures	xiv
List of Tables	xvi
List of Acronyms	xviii
1 Introduction	1
1.1 Context and Problems	1
1.2 Motivation	2
1.3 Objectives	3
1.4 Document Structure	4
2 State of the Art	5
2.1 Background	5
2.1.1 Cloud Computing	6
2.1.1.1 Software as a Service (SaaS)	7
2.1.1.2 Platform as a Service (PaaS)	7
2.1.2 Monolithic vs Microservices Architecture	8
2.1.2.1 Monolithic Application	9
2.1.2.2 Microservices	9
2.1.3 Representational State Transfer (REST)	10
2.1.4 Recommendation methods	13
2.1.5 Exploratory Data Analysis	14
2.1.6 Business Intelligence	15
2.2 Related Work	15
2.2.1 Amazon	16
2.2.1.1 Amazon Web Services (AWS)	16
2.2.2 Netflix	17
2.2.2.1 Architecture Overview	18
2.2.3 eBay	20
2.2.4 SaaS and PaaS Recommender Systems	21
2.2.4.1 Yusp	21
2.2.4.2 Strands Retail	22
2.2.4.3 Commerce Cloud Einstein Product Recommendations	23
2.2.4.4 Amazon Personalize	23

2.3	Summary	25
3	The Proposal	27
3.1	General Overview	27
3.1.1	Challenges	27
3.1.2	Functionalities	29
3.2	System Requirements	30
3.3	Proposed Approach	32
3.3.1	Architecture Description	33
3.3.1.1	Architectural Approaches	33
3.3.1.2	Architectural Solution	34
3.3.1.3	Architecture Diagram	35
3.4	Summary	36
4	Development	37
4.1	Technology Used	37
4.1.1	Infrastructure Requirements	39
4.2	Product Recommendations	41
4.2.1	Recommendation Types and Structure	41
4.2.2	Recommendation Storage	43
4.2.3	Filters	44
4.2.4	Online and Offline Computation	46
4.2.5	Offline recommendations	47
4.2.6	Online recommendations	48
4.2.7	Nearline recommendations	49
4.3	System Communication Process	50
4.4	Beevo's Business Intelligence Application	53
4.4.1	Application configurations	55
4.4.2	Database population process	57
4.4.3	Event triggers	59
4.4.4	Storefront widgets	60
4.5	Security	63
4.6	Summary	66
5	Methods	67
5.1	Data Extraction Strategy	67
5.1.1	Logstash	67
5.1.2	Debezium (Change Data Capture)	68
5.2	Exploratory Data Analysis (EDA)	69
5.2.1	Variables Selection and Filtering	71
5.3	API Documentation	74

5.4	Summary	75
6	Case Studies / Experiments	76
6.1	Experiment setup	76
6.1.1	Data Contract	82
6.1.2	Kibana Data Analysis	82
6.2	Results	85
6.2.1	Performance Tests	86
6.2.2	Business Intelligence Dashboards	93
6.3	Discussion	97
7	Conclusion	100
	References	103
	Appendices	109

LIST OF FIGURES

2.1	Monolithic application architecture template	9
2.2	Microservices application architecture template	10
2.3	REST vs SOAP: Web search interest rate comparison between REST and SOAP, from 2004 until 2019, worldwide	11
2.4	REST API Model example diagram	12
2.5	Collaborative and Content-based Filtering examples	13
2.6	Applications covered by Amazon Web Services (1) products.	17
2.7	Netflix recommender system overview - based on Netflix Tech Blog post (2)	19
2.8	Strands Retail product recommendation system workflow overview	22
2.9	Commerce Cloud Einstein product recommendations process	23
2.10	Amazon Personalize "How it works" diagram	24
2.11	Amazon Personalize high level architecture	25
3.1	Recommender System Architecture diagram	35
4.1	Node.js Application Modular Structure Design, according to Separation of Concerns (3)	38
4.2	Recommender System Entities	41
4.3	MongoDB recommendation documents structure	44
4.4	User Interaction Activity Diagram	47

4.5	Recommender System Activity Diagram	47
4.6	Communication process between Recommender's API an Engine - option 1	51
4.7	Communication process between Recommender's API an Engine - option 2	51
4.8	Communication process between BI App and Recommender System	53
4.9	Beevos' Business Intelligence Application File Structure	54
4.10	Homepage recommendations vitrine generic template. In this example, four recommended products are displayed to the user: two from ' <i>Clothes</i> ' category (shirt and pants) and two from ' <i>Drinks</i> ' category (iced tea and smoothie). Products are ordered by score, with the product on the left having the highest score, i.e., is more likely to be bought by the user.	60
4.11	Product details page recommendations vitrine generic template. In this example, the user selected a shirt, thus be presented products similar to it in the recommendation vitrine, where the product on the left is the most similar to the selected shirt (ordered by score).	61
4.12	Side cart recommendations vitrine generic template. In this example, the user added a shirt and pants to the shopping cart, so the recommendation vitrine displays some products which are commonly bought together with the cart's current content.	61
4.13	Product listing page recommendations vitrine generic template. In this example, the user browses for products within the " <i>Drinks</i> " category and orders them with the " <i>Recommended</i> " option. Ergo, products related to " <i>Drinks</i> " are listed, ordered by recommendation score, i.e., the products that would appeal the most according to the user's profile are shown first.	62
6.1	Deeply Back-Office - RS user association to tenant	77
6.2	Deeply Back-Office - Recommender engine configurations	78
6.3	Deeply Back-Office - Attribute selection and Recommender Population	78
6.4	Deeply MongoDB recommendation documents examples	79
6.5	Deeply Homepage - Popularity recommendations example	80
6.6	Deeply Product Details - Similar-products recommendations example	80
6.7	Deeply Side-Cart - Complementary-products suggestions example	81
6.8	Deeply Product Listing - Hybrid recommendations example	81
6.9	Execution of load tests: Workload, System Under Test (SUT) and Metrics.	86
6.10	<i>Locust</i> load test structure.	88
6.11	Number of Users over time	91
6.12	Rate of total requests per secod over time	92
6.13	Response time value over time	92
6.14	Total Orders by Country - Map	93
6.15	Total Orders by Country - Bars graph	94

6.16	Average spend by Country	94
6.17	Average quantity and spend per order	95
6.18	Client genders	95
6.19	Promotion Tracking	96
A.1	NodeJS Application Structure	110
A.2	API Documentation with Swagger - part 1	111
A.3	API Documentation with Swagger - part 2	112
A.4	User login page of the Recommender System	113
A.5	User registration page of the Recommender System	113
A.6	Beevo Business Intelligence Data Contract - part 1	114
A.7	Beevo Business Intelligence Data Contract - part 2	115
A.8	Kibana's data analysis on Clients data - part 1	116
A.9	Kibana's data analysis on Clients data - part 2	117
A.10	Kibana's data analysis on Products data - part 1	118
A.11	Kibana's data analysis on Products data - part 2	119
A.12	Kibana's data analysis on Order-items data - part 1	120
A.13	Kibana's data analysis on Order-items data - part 2	121
A.14	Kibana's Business Intelligence Dashboards - part 1	122
A.15	Kibana's Business Intelligence Dashboards - part 2	123

LIST OF TABLES

4.1	Version table of docker images used in RS architecture	40
4.2	Server host machine hardware specifications. Note that the server is hosted in a virtual machine, emulated using <i>QEMU</i> (4). <i>QEMU</i> allows to run operating systems for any machine, on any supported architecture, with near native performance.	40
4.3	Beevo's business intelligence application configurations	56
4.4	RS Access Control List: existing roles, resources and permissions. As it can be observed, Tenant Users are not allowed to access Users and ACL resources. On the other hand, Tenant Admins are not allowed to edit system users information nor create or remove ACL elements (roles, resources and permissions). The System Admins are able to see the roles and permissions of all users. They have full access to the ACL, which allows them to manage all system's policies.	64
5.1	Fields selection for Client entity	71

5.2	Fields selection for Product entity	72
5.3	Fields selection for Order-item entity	72
6.1	Deeply clients' gender distribution	83
6.2	Deeply top 5 country values	83
6.3	Top 5 values of colors, sizes and categories of Deeply products	84
6.4	Top 5 values of colors, sizes and categories of ordered products	85
6.5	Simulated users in the load tests	87
6.6	System's server load test performance results	87
6.7	Locust table that translates the values of metrics collected during the load test, discriminated by each of the routes/endpoints provided by the Recommender API . .	90

LIST OF ACRONYMS

A

API Artificial Intelligence
API Application Programming Interface
AWS Amazon Web Services

B

B2B Business to Business
B2C Business to Client
BI Business Intelligence

C

CDC Change Data Capture

D

DB Database

E

EDA Exploratory Data Analysis

I

IaaS Infrastructure as a Service
IT Information Technology

J

JWT JSON Web Tokens

M

ML Machine Learning

P

PaaS Platform as a Service

R

RS Recommender System

S

SaaS Software as a Service
SLA Service Level Agreement
SOAP Simple Object Access Protocol

1. INTRODUCTION

Information is power. Nowadays, information is one of the most valuable assets that companies can have to improve their business and stand out on the market. The power of knowing customers, as well as their needs and behaviors, depending on the context or even their surroundings, and the ability to correlate it with the products or services companies can offer, provides them great means to exploit their full potential on the market.

This concept is most applied in *e-commerce platforms*, which design their websites to draw customers' attention, by suggesting and recommending their products or services, depending on the customers' activity in the online store.

In this chapter, an introduction to the work developed under the context of this Master's dissertation is presented. First, it is introduced the Context and Problems where this project was framed, then the Motivation is exposed, followed by the project's Objectives, listed in a generic and simple way. Lastly, the Document Structure is described.

1.1 Context and Problems

The concept of *electronic commerce* (5), or e-commerce, can be defined as transactions of goods or services via the Internet, by any electronic means, including transfers of money and data implied in these processes. Thus, it is generally described as any kind of commercial transaction executed through the Internet. However, in the context of this thesis, it will be referred to as the sale of physical products by online stores.

There are several types of e-commerce of which we highlight the following two: *Business To Consumer (B2C)* and *Business To Business (B2B)*. These two concepts are the most commonly explored in online stores and other e-commerce areas.

B2C is the process of selling products or services directly to consumers from a platform. In this type of commerce, the consumer browses and buys products for personal use, providing some personal information to checkout and pay, finishing the purchase. For this reason, the system aims to be simple and attractive to the buyer, appealing the emotion to influence their decision on purchasing, focusing on products' characteristics to satisfy their needs.

On the other hand, B2B stands for the process of selling goods or services to other businesses, where buyers purchase products on behalf of their companies. The format of orders and the value of products can vary with the customer, which makes this type of system more complex than the previous one. It is logic-driven, which means that focus on the product details and potential to benefit buyers' businesses. It also prioritizes saving time, money and resources, improving productivity and those are the main features companies are looking for.

E-commerce is constantly expanding, leading to greater market competitiveness. There are more and more online platforms offering products or services; so there is a growing need for companies to stand out from the competition. However, many companies lack the necessary information about their clients and products, and even their competitors, which can make a difference in the current e-commerce environment. This data can help companies to increase clientele and, consequently, profit. This may lead to the implementation of various marketing strategies, but not all are adequate and mismanagement, as well as a bad investment of these strategies, can be harmful to businesses.

The project of this dissertation will be developed under the context proposed by the company Beevo (6). Beevo provides e-commerce B2C, B2B and B2E (Business to Employee) solutions, for mid-market and large companies, offering a digital platform for their *e-business*. The company builds and maintains other companies' e-commerce platforms, including online stores and respective business logic (customers, products and orders management), marketing, support, CRM (Customer Relationship Management) technologies and business analysis.

1.2 Motivation

For companies to have a competitive digital business, Beevo offers more than an online store; it also provides a set of professional apps that allows fast and simple growth.

With the e-commerce competitiveness in mind, Beevo proposed the development of provisional models to grow their arsenal of professional apps and boost their Business Intelligence strand. These provisional models were later translated to what is now the core of this project: a Recommender System (**RS**) for e-commerce platforms of Beevo's domain.

The need to make e-commerce platforms more appealing to its clients makes integrating a recommender system a logical approach. Its function is to provide a pleasant user experience, trying to

create a connection with the buyer, showing that the system understands them and is appreciated as a customer by the store. This *special* connection to users increases their loyalty to the store and keeps their interest in coming back for more. As a result, more traffic on the platforms is ensured, which may increase the sales rate and, therefore, a higher number of revenues is expected.

However, in order to develop a recommender system, it is necessary to define which information meets the requirements for obtaining trusted recommendations, as well as the data flow between the system and platforms.

Several companies defend that most sold products or used services are *recommended* to clients by their platform. As evidence of that fact, we see quite popular platforms such as *Netflix* (2) and *Amazon* (7), having advanced recommendation engines in their fields.

1.3 Objectives

In order to integrate a recommender system into e-commerce platforms, the first step must consist in obtaining information from the selling platform, where data about its users and products/services form the basis of recommendations. But not all information is useful, which may influence the accuracy of the forecasting models used by the system to make recommendations. Furthermore, it is crucial to define a good workflow to collect the data and attend the platforms' needs for recommendations, so we can run a smooth communication between systems.

Following this perspective, the recommender system can be divided into two parts:

- **Architecture:** the recommender system architecture contemplates its infrastructure, responsible to set and handle the communication between the system and the e-commerce platforms, as well as the data management, essential to produce recommendations.

- **Engine:** the recommender system engine is the component responsible for consuming data, preserved in the architectural process, and calculate recommendations based upon users, products and services of the selling platforms.

This pair of *architecture* and *recommendation engine* form the system developed in the context of this dissertation. Although there are several studies and papers exploring various recommendation algorithms and trying different combinations and techniques, to get the best recommendation results, there's a lack of investigation on the integration of recommender systems within online platforms.

Ergo, this Master's dissertation is focused on the recommender's *Architecture* component. With this in mind, the main objectives are listed as followed:

1. Understand the concepts of *Recommender System*, *Business Intelligence*, *Cloud Computing* and other terms related to a system's architecture;
2. Implement an architecture of a recommender system, which allows to extract, preserve and analyze relevant data from Beevo's e-commerce platforms, to integrate a recommendation engine. This architecture will manage the data flow on the communication between platforms and the recommender engine. Summarizing, it will be responsible for delivering data to the engine and return the results to platforms;
3. Explore and define a data analysis methodology.
4. Understand the importance of selecting data and its impact on recommendations performance and accuracy;
5. Explore the potential of the e-commerce data collected on business intelligence applications to better understand and improve businesses.

1.4 Document Structure

This dissertation is structured in seven different chapters:

- **Introduction:** This chapter introduces the context of this dissertation project, its motives and what it aims to achieve;
- **State of the Art:** In this chapter it is explored some terms and definitions related to this dissertation theme, as well as exposed some work related to the context of the project.
- **The Proposal:** This third chapter describes the proposed solution to overcome some challenges in order to successfully achieve the defined objectives.
- **Development:** Here all development stages of the system architecture are described, including the technologies used, the structure of recommendations, communication processes and all the decisions taken along the course.
- **Methods:** This part of the document presents the various strategies for extracting and analyzing data and making results available to e-commerce platforms.
- **Case Studies / Experiments:** In order to demonstrate the different functionalities that the system has to offer, a test case is presented in one of the company's online stores.
- **Conclusion:** In this final chapter, a global review is presented on the project developed, as well as several improvements that can be done in the future.

2. STATE OF THE ART

As mentioned in the previous chapter, this thesis focus mainly on the recommender system infrastructure, this is, its architectural component. It covers not only the establishment of communication between the system and platforms but also data analysis strategies to process data before being transferred on this communication.

The first part of this chapter sets the Background of this work: it defines the key concepts and vocabulary for the rest of the Master thesis, such as Cloud Computing, Monolithic vs Microservices Architecture, Representational State Transfer (REST), Exploratory Data Analysis and Business Intelligence. In the second part, it is exposed some Related Work with the project theme and objectives. Examples of famous companies like Amazon and Netflix are explored, giving some insights on how existing recommender systems work in today's market.

2.1 Background

Recommender systems, or recommendation systems, emerged as an independent field of research in the mid-1990s and derived from different other areas, such as cognitive science, approximation and forecasting theories, information retrieval and also have links to management science (8).

This area is widely explored because it constitutes a problem-rich research field and due to the abundance of practical applications, which can help users to deal with information overload as well as provide personalized recommendations, content, and services to them.

An e-commerce RS is a *machine learning* (**ML**) mechanism that relies on a variety of data, related to users, products or services, processes it and creates personalized suggestions for the intended user. Its purpose is to assist the user in their purchasing decisions, recommending the products or services that best suit their interests, but can be manipulated according to the company's intentions. Therefore, it acts as a forecasting model dedicated to calculate and making recommendations, such

as predicting the likelihood of a product being bought by a certain user, taking into account their preferences and other information.

2.1.1 Cloud Computing

Cloud Computing (9) refers to the distribution of power, database space, applications, and other resources using a service platform, via the Internet. It is regarded as on-demand delivery of IT resources, where end-users subcontract and benefit from hosted services, without worrying about storage space and power consumption.

Before the idea of cloud computing emerge, servers stored all the resources (software applications, data, and services) for client/server computing. It was a centralized storage, hence for users to access data, they needed to gain access to the server. As the concept of distributed computing was introduced, resource sharing was made possible, contributing to the evolution of cloud computing. Cloud Computing appeared in the 1950's, when mainframe computers were accessed via dummy terminals into a central computer, so users could gain access. Yet, mainframes' production and maintenance were too expensive, thus the urgent need of sharing resources to reduce costs.

Cloud services provide *flexibility* for businesses with developing or shifting bandwidth needs. The service storage capacity can be easily modified to meet companies' demands, being advantageous to businesses over contenders. It also guarantees a more secure atmosphere and data centralization.

Some examples of what can be performed via cloud are enlisted below (9):

- Creating new applications and services;
- Hosting website and blogs;
- Streaming live video and audio;
- Storage, back-up, and data recovery;
- Software delivery on demand;
- Data Analysis and predictions;

2.1.1.1 Software as a Service (SaaS)

The term *Software as a Service* (10) refers to a distribution model for deployed software, by third-party providers that host applications and makes them available to users, over the Internet. **SaaS** is one of three main categories of Cloud Computing, alongside *Infrastructure as a Service (IaaS)* and *Platform as a Service (PaaS)* (10).

According to the *Handbook of Industry 4.0 and SMART Systems* (11):

"In the software on-demand SaaS model, the provider gives customers network-based access to a single copy of an application that the provider created specifically for SaaS distribution. The application's source code is the same for all customers and when new features or functionalities are rolled out, they are rolled out to all customers. Depending upon the Service Level Agreement (SLA), the customer's data for each model may be stored locally, in the cloud or both locally and in the cloud. Organizations can integrate SaaS applications with other software using Application Programming Interfaces (APIs)."

This means that the application runs on the SaaS provider's servers, freeing the customers' side from several responsibilities, as they can focus on building their business and not worrying about maintaining the subscribed application. It also means that it's capable of exchanging data with customers, interacting with their web services via **APIs**. An Application Programming Interface can be described as a set of definitions and protocols that provide an interface between two systems. It allows to build and integrate application software in existing systems, granting communication between services, without knowing how they are implemented (12).

Most of SaaS providers, e.g. Salesforce¹, Oracle², Red Hat³, Microsoft⁴ and Amazon⁵ offer applications for fundamental business technologies, such as email, sales management, Customer Relationship Management (CRM), financial management, Human Resource Management (HRM), billing and collaboration (11).

2.1.1.2 Platform as a Service (PaaS)

With *Platform as a Service* (13), cloud service providers offer a platform to clients, enabling them to concentrate on building and deploying their business applications, without the need to create and maintain the infrastructure typically required by such software development processes.

¹<https://www.salesforce.com>

²<https://www.oracle.com>

³<https://www.redhat.com>

⁴<https://www.microsoft.com>

⁵<https://www.amazon.com/>

Clients have control over the software deployment while the cloud provider delivers all components needed to host the applications, including servers, storage systems, networks, operating systems, and databases. This differs from the Software as a Service model, as in SaaS most of the services are managed by service providers and the amount of configuration in the client's end is minimal.

Generally, SaaS target end-users while PaaS are addressed to software developers, providing the tools and capabilities they need to build and deploy an application without having to concern about the underlying infrastructure.

PaaS vendors tend to be the biggest technology companies, who can offer a broad range of capabilities for their clients on a platform (14). Some examples include Google App Engine⁶, Oracle Cloud Platform⁷ and the Salesforce-owned Heroku⁸.

A few common use cases for PaaS can be listed as followed (13):

- **API development and management:** companies can use PaaS to develop, run, manage, and secure APIs and microservices.
- **Business intelligence and analytics:** Tools provided as PaaS let companies analyze their data to explore their business and discover patterns that can help to make better decisions and more accurately anticipate future events such as market demand for products or trends.
- **Data Management:** A PaaS can manage essential business information of a company, providing a single point of reference for data. Such data might be related to customer transactions and analytical data to support decision making.
- **Databases:** A PaaS provider can deliver services such as setting up and maintaining an organization's database.

2.1.2 Monolithic vs Microservices Architecture

Microservices are an important and trending architectural approach used in the Information Technology (**IT**) sector, representing a crucial shift in how IT approaches software development. This architecture has been successfully adopted by organizations like Netflix, Google, Amazon, and others, but what are microservices' advantages over a monolithic architecture? (15)

The right architectural approach depends on the application context and objectives.

⁶<https://cloud.google.com/appengine/docs/>

⁷<https://www.oracle.com/cloud/>

⁸<https://www.heroku.com/>

2.1.2.1 Monolithic Application

Traditionally, applications were built as **monolith** (16), i.e., a single application was packaged and deployed with all different logical components: presentation, business logic, database access and application integration. This method makes it simple to develop, test and deploy, as well as to scale *horizontally* by running multiple instance copies of the application and using a *load balancer*.

However, this approach has a few limitations as to the application size and complexity. If the application is too large, it can have problems with performance and scalability, as different modules have conflicting resource requirements. The entire application must be redeployed on each update, which makes continuous deployment difficult, and a minor bug in any module can impact heavily the rest. So, it's not a very flexible nor reliable architecture, if the application is of great complexity, being hard to adopt new technologies and integrate new frameworks or languages since it affects the entire operation and is expensive in both time and cost (16).

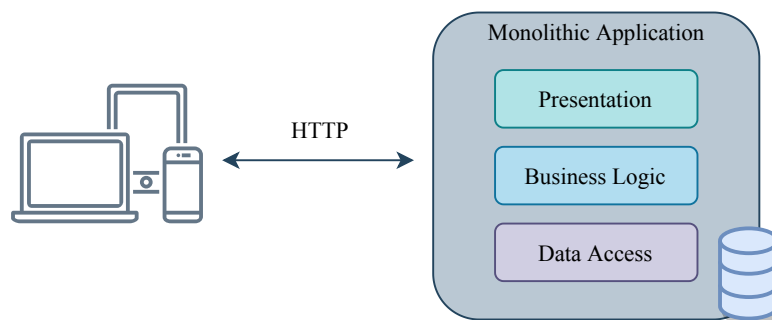


Figure 2.1: Monolithic application architecture template

Nonetheless, it is important to understand monolithic architectures since it is the basis for a microservices architecture, where each service by itself is implemented according to a monolithic architecture.

2.1.2.2 Microservices

With the **microservices** approach, applications consist of a set of smaller, independent and interconnected services, instead of a single monolithic application (16). This way, it's possible to build large applications with low complexity, breaking it into a set of manageable services that are faster to develop and maintain, splitting the effort across different developing teams. Microservices architecture enables each *microservice* to be deployed and scale independently from others, which makes easier to update the application and to integrate new services and functionalities.

But this approach also presents some drawbacks; thus the importance of weighing the benefits of both architectural approaches and choose the one that best matches the context and implications of

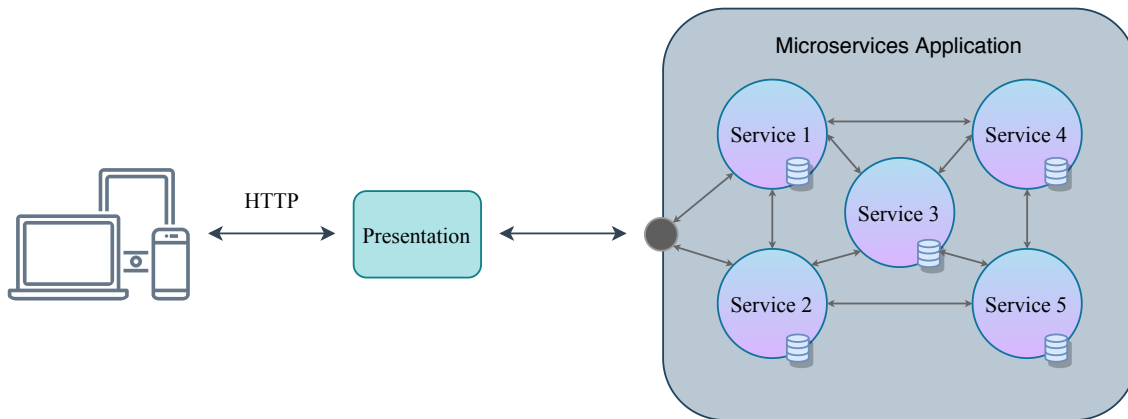


Figure 2.2: Microservices application architecture template

the problem considered. One needs to evaluate the purpose of the application and assess if it's worth adding the complexity that microservices architecture brings as a *distributed system*. Managing all different services and communication between them may not compensate for the effort, time and cost invested in developing a microservice architecture (16).

In a microservice architecture, when some service is changed, it's essential to thoroughly plan and coordinate the outcome of those changes to each of the other services, while in a monolithic application it's possible to simply change the corresponding modules and integrate the changes.

Deploying a microservice-based application is also more complex, in contrast to a monolithic, because each service will have multiple runtime instances. In turn, each instance needs to be configured, deployed, scaled, and monitored. Additionally, it may be necessary to implement a service discovery mechanism, thus requiring a high level of automation (16).

2.1.3 Representational State Transfer (REST)

State Transfer (17), or REST, is essentially a design concept for a web service architecture. It's a very popular architectural style due to its simplicity and the fact that it builds upon existing systems and functionalities from the application layer protocol Hypertext Transfer Protocol (HTTP) in order to achieve its objectives, instead of creating new standards, frameworks and technologies.

Over the years, more and more companies develop their web services based on REST, as opposed to the traditional web services with Simple Object Access Protocol (SOAP). This can be demonstrated with a Google Trend Analysis, given REST and SOAP keywords, which shows an increasing interest on REST compared with SOAP, as followed:



Source: Google Trend Analysis (trends.google.com/trends)

Figure 2.3: REST vs SOAP: Web search interest rate comparison between REST and SOAP, from 2004 until 2019, worldwide

In systems that follow the REST paradigm, both client and server can be implemented independently, keeping them separated and modular. They are also stateless, meaning that the server does not need to acknowledge the client's state and vice versa. Each time a client accesses a resource through an endpoint, the API provides the same response. It does not remember the client's last request neither takes that into account when providing the new response. A client is supposed to enter a REST service without any knowledge of the API, except for the entry point and the media type. In SOAP, applications can be stateless, but usually they are stateful, meaning the client needs previous knowledge on everything it will be using, or it won't even begin the interaction.

Responses can also be cached in REST APIs to increase performance. If the browser's cache already contains the information asked for in the request, the browser can just return the information from the cache instead of getting the resource from the server again (18). This does not happen in SOAP APIs.

Every REST architecture must implement hypermedia and HATEOAS. Hypermedia is a generalization of hypertext for content, like HTML, XML, JSON, etc. Documents containing hypertext are intended to be parsed by an automated client who will also follow links and actions like a human would do with a browser. HATEOAS means the interaction of a client with a REST application must be driven by hypermedia, i.e., the client should obtain all Uniform Resource Identifiers (URIs) for every resource it needs by following links in the representation of resources themselves.

In spite of representing different concepts to approach a system architecture implementation, a resumming comparison between REST and SOAP characteristics is presented below, enlisting the advantages and disadvantages of each (19):

- **Design:** SOAP is a standardized protocol with pre-defined rules to follow, while REST is an architectural style;
- **Approach:** SOAP is function-driven - transfers structured information - and REST is data-driven - accesses resources for data;

- **Caching:** In REST, API calls can be cached, but not in SOAP;
- **Security:** SOAP supports WS-Security with SSL and has built-in ACID compliance. In the other hand, REST lacks ACID compliance and supports HTTPS and SSL;
- **Performance:** SOAP requires more bandwidth and computing power, while REST needs fewer resources, being lightweight;
- **Message format:** REST permits many data formats, including plain text, HTML, XML, JSON and others. SOAP only supports XML;
- **Advantages:** SOAP is standardized and has high security and extensibility, while REST has better performance, scalability and flexibility;
- **Disadvantages:** SOAP is more complex and has poorer performance and flexibility. However, REST may be less suitable and secure for distributed environments.

Overall, REST offers several advantages over SOAP, for building communication channels between systems, by being simple, flexible and scalable, allowing a greater variety of data formats and performance.

When using REST over HTTP, it's possible to resort to standard HTTP security and authentication. By combining it with *JSON Web Tokens (JWT)* for authentication and authorization of user's to validate their requests, an efficient way of secure the communication can be achieved.

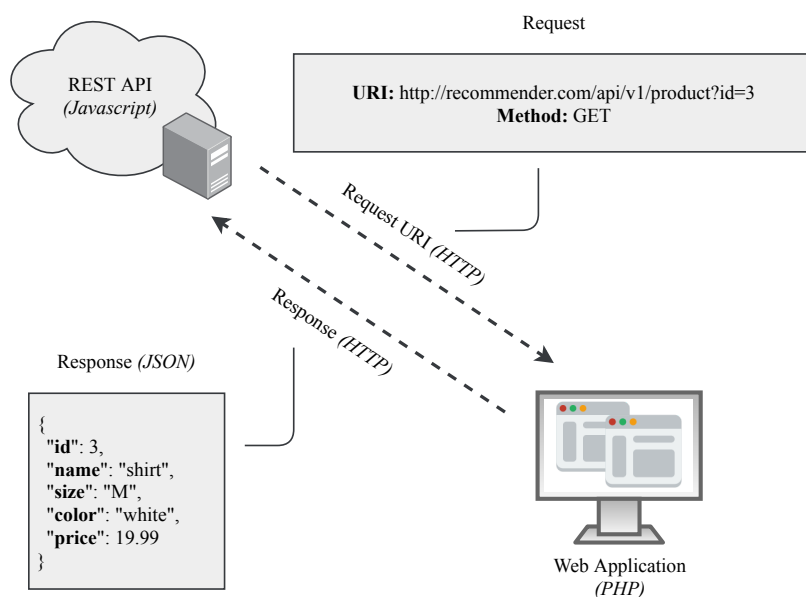


Figure 2.4: REST API Model example diagram

According to the example displayed in the diagram above, there's a request and a response between a client and the server's API. The client and server can be based in any language but it doesn't matter because the message request and response are made through a common HTTP web protocol. This *request-and-response* pattern is fundamentally how REST APIs work.

2.1.4 Recommendation methods

There are several approaches that can be used to create a recommendation model, the most popular being **Content-based Filtering** (20) and **Collaborative Filtering** (21).

The premise of the *Collaborative Filtering* approach is to search for similarities between clients, according to their actions and preferences. *User-based* (22) recommendations take into account the similarity between the clients' profile, i.e., items purchased by a certain client will be recommended to another client who has similar tastes and behaviors. On the other hand, *item-based* (23) recommendations are supported in products' characteristics. For example, user A has a similar buying pattern as the user B. Consequently, items with similar attributes to those that user A has purchased in the past, may be suggested to user B.

In *Content-based Filtering*, the user's shopping history is important. The characteristics of items from previous purchases made by the client are analyzed and compared with the remaining candidate items, available in the store. Items that have more in common with those that the user has purchased are recommended. For example, in a certain online book store, a user bought some books in the Sci-Fi category. According to the user's shopping history, the system may recommend other similar books, i.e. of the same category (Sci-Fi), to that user in the next visit to the online store.

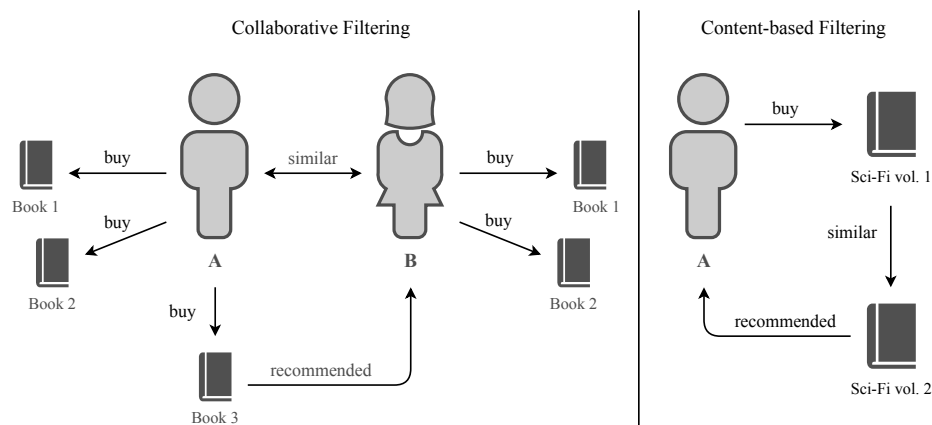


Figure 2.5: Collaborative and Content-based Filtering examples

Each different recommendation technique has its benefits and drawbacks in terms of efficiency and results accuracy, depending on the context in which they are applied. In order to fill the gaps that each approach presents and to make the system more robust, a **Hybrid Approach** is usually adopted by several techniques, in this case, collaborative and content-based filtering. Following the previous example, with a hybrid approach, the system would recommend not only other Sci-Fi books to the client (content-based filtering), but would also take into account the book's rating given by the other users (collaborative filtering).

Another strategy widely used in e-commerce is *Customer Segmentation* (24). By using clustering algorithms, such as *k-means*, it is possible to trace the profiles of several clients and group them according to the similarity between their characteristics (gender, age, etc). Obtaining an ideal number of clusters can be extremely useful since different clients belonging to the same cluster can be treated as a single entity, thus saving a lot of resources in the production of recommendations and targeted marketing.

Since each of these approaches consumes different types of information, it is essential to select the most significant data, corresponding not only to the type of the selling platform, but also to the approach itself. Hence the importance of the quality of the data set used to train the model, as it will influence the accuracy of recommendations.

2.1.5 Exploratory Data Analysis

To select the most relevant data, data analysis techniques are usually applied, according to a certain approach previously defined. *Exploratory Data Analysis (EDA)* (25) refers to the initial investigation of a data set, to understand it, so as to extract patterns, detect anomalies, filter outliers, evaluate hypothesis and to verify assumptions with the support of summary statistics and graphical representations.

This is an important step in the data analysis process, to help data analysts to comprehend the big picture, in this case, the main characteristics of each different e-commerce platform, setting up the context before any machine learning operation. This way, one can create a model that fits the given context and increase the system efficiency.

By examining and treating data, conforming that context, one can develop visual panels as an information management tool to visually track, analyze and display *key performance indicators* (26), metrics and other key data points to monitor the status and performance of a business. These panels are called *dashboards* and prove that the considered data allows going beyond recommendations, and explore the vast area of *Business Intelligence*.

2.1.6 Business Intelligence

In 1865, *Business Intelligence* (**BI**) was used to describe how the banker Sir Henry Furnese took advantage of existing information by collecting and acting on it before his competitors, in the *Cyclopædia of Commercial and Business Anecdotes*, by Richard Millar Devens. Years later, in 1958, IBM computer scientist Hans Peter Luhn wrote an article describing the potential of Business Intelligence through the use of technology and this field has evolved since then. The number of BI vendors grew in the 1980's, as business people discovered the value of Business Intelligence and, consequently, an assortment of tools were developed during this time, with the goal of accessing and organizing data in simpler ways (27).

Nowadays, we can define *Business Intelligence* (28) as a set of concepts, methods, processes and technologies that gather and store *raw* data, and transform it into relevant and useful knowledge for business purposes. BI can handle large amounts of information to help companies identify and develop new opportunities, as well as planning and making decisions. Thus, studying BI solutions can provide a competitive market advantage and long-term stability, helping to make the right decision.

From this, we may conclude that data is a very powerful resource, when well manipulated, giving companies the necessary knowledge to overcome obstacles and competition. However, this brings up some security concerns, where companies can be targeted by rivals, stealing their information. Hence, it is very important to build secure communication channels, where data can flow inside the company's systems, protecting private business information.

2.2 Related Work

As mentioned in the previous chapter, nowadays most popular online platforms have recommender systems, each one adapted to the platform's background. For example, *Youtube* (29) is a video-sharing and streaming platform, featuring video recommendations to its users, depending on what they previously watched. Along this line, *Spotify* (30) is an audio streaming platform and recommends songs and playlists to users, matching genres they use to listen. On the other hand, *Facebook* (31) is an online social network service, which recommends other users' contacts, so one can be "*friends*" based on their current connections.

In this section, we highlight *Amazon* (7), *Netflix* (2) and *eBay* (32) systems, as well as other different architectural approaches, in order to achieve a general perspective of how recommender systems work in today's market.

2.2.1 Amazon

In light of the theme of this dissertation, *Amazon* (7) is the best example to consider. Amazon is the world's largest online retailer, which sells a wide variety of products to customers, like the e-commerce platforms that this project intends to target.

Amazon started as an online book store and one of its main advantages, compared to physical stores, was the *infinite shelf-space* capacity as a platform over the Internet. This allowed it to make a great number of sales from books beyond the inventories of physical stores. But due to the massive quantity of different books stored, customers might have missed some good opportunities to discover new relevant books that they might have bought. Hence, the emergence of the recommender system as a great tool to suggest new books - and other products later on - to customers, from this *infinite book shelf*, and consequently increase sales.

By the year of 2012, JP Mangalindan claims in a Fortune's article (33) that:

“Judging by Amazon's success, the recommendation system works. The company reported a 29% sales increase to \$12.83 billion during its second fiscal quarter, up from \$9.9 billion during the same time last year. A lot of that growth arguably has to do with the way Amazon has integrated recommendations into nearly every part of the purchasing process...” - JP Mangalindan (33)

Currently, Amazon uses a combination of user-based and item-based collaborative filtering in their recommendation algorithms, to suggest products to customers along the purchase process, via e-mails, browse pages, product details pages and even at the end of an order.

Additionally, according to Ian MacKenzie (34), 35% of Amazon.com's revenue is generated by its recommendation engine. That is why companies are increasingly investing in RS to deploy in their online platforms.

2.2.1.1 Amazon Web Services (AWS)

It is also worth to mention that, as one of the biggest cloud computing service providers in today's market, Amazon holds the company *Amazon Web Services (AWS)* which provides on-demand cloud computing APIs and platforms. The clients that may be interested on develop complex and efficient applications, with great flexibility and reliability, can use AWS products for cloud ecosystems - a mix of IaaS, PaaS and SaaS.

Some of the features that AWS presents and appeal to clients the most are the following:

- Security: AWS is one of the safest cloud platforms on the market;

- Experience: Users can get a hands-on experience of AWS free of charge;
- Hosting: AWS can host static websites also for free;
- Scalability: AWS has a great scaling capacity.

Products can be combined to create a scalable cloud application without having to concern about problems related to infrastructure maintenance (compute, storage, and network) and management (10).

Amazon Web Services offers products in the areas illustrated below:

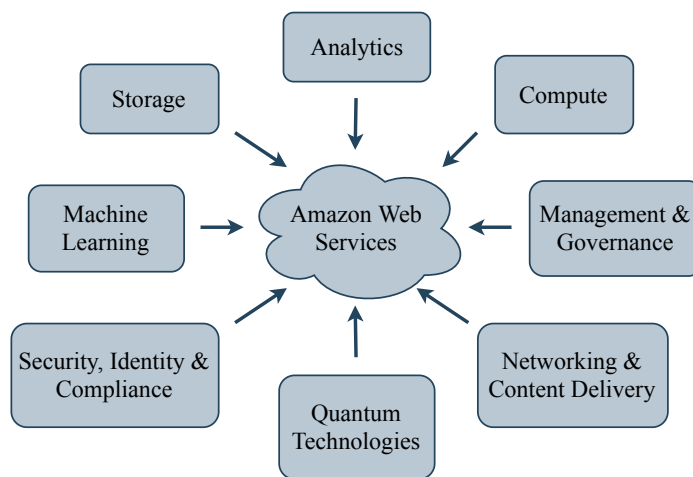


Figure 2.6: Applications covered by Amazon Web Services (1) products.

2.2.2 Netflix

Netflix (2) is a streaming service that allows members to watch a wide variety of TV shows, movies, documentaries, etc. Recommendation algorithms have been the core of the Netflix product from very early on. Because of its importance, the company continually seeks to improve recommendations results by advancing the state-of-the-art in the field.

Netflix's RS combines collaborative and content-based filtering through similar habits of users and higher rates of shared movie characteristics. The company, Netflix Inc., released a contest in 2006 - The Netflix Prize -, offering a reward of one million US dollars to enhance the recommender system. The team who could succeed to decrease the value of *root-mean-square error* (RMSE) for a data set by 10 percent, would win the prize. Bellkor's Pragmatic Chaos team succeed in achieving an RMSE of 0.8554 with a 10.06% improvement over the Netflix system. This challenge grew up the attention on recommender systems beyond Computer Science.

In an interview with *MobileSyrup* (35), Todd Yellin said:

“We found the typical Netflix member on average will only look at 40 or 50 titles before deciding what they want to watch, even though there are thousands of titles available. So it’s important we present the right content to the right member at the right time.” - Todd Yellin, Netflix’s vice-president of product innovation

Netflix uses recommender systems so extensively that, in 2015 Chief Product Officer, Neil Hunt, indicated that more than *80 percent* of movies watched on Netflix came through recommendations (36) and placed the value of Netflix recommendations at more than *US\$1 billion per year*. This proves the power and importance of recommender systems in a e-business.

2.2.2.1 Architecture Overview

In Netflix Tech Blog⁹, an article published by Amatriain and Basilico (37) explains how Netflix tackles some of the challenges of maintaining a software architecture capable of handling large volumes of data, responsive to user interactions and flexible to new recommendation approaches.

An overall RS architecture is described, where the whole infrastructure runs across the public *Amazon Web Services* cloud. The system’s diagram (38) is presented in figure 2.7.

The system’s architecture can be divided into three parts: **Online, Offline and Nearline computation**. These distinguish the types of processes and recommendations that the recommender system computes.

⁹<https://medium.com/netflix-techblog>

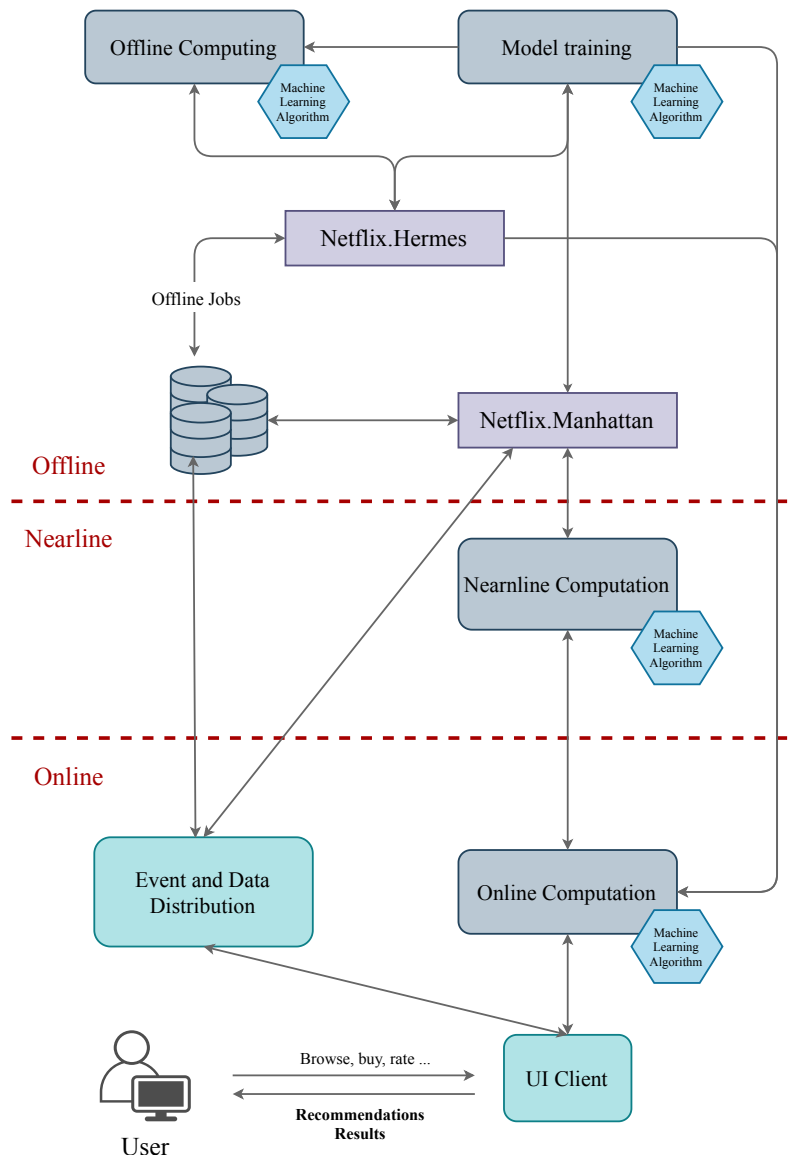


Figure 2.7: Netflix recommender system overview - based on Netflix Tech Blog post (2)

- **Online computing** must have fast responses to events and use the most recent data to fulfill the availability and response time required by the client-side. This constrains the implementation of complex and computationally costly algorithms, limits the amount of data that can be processed and, consequently, recommendations' accuracy. Moreover, using solely online computation may fail to meet some of the requirements, hence the importance to have a *offline computation mechanism* as a fallback solution to fit those requirements.
- **Offline computing** is less limited in terms of data processing, computational costs, and complexity, having more flexibility on the implementation requirements and a wider range of algorithms to choose from. However, because of heavy processing in this approach, offline computing does not have a fast response to changes from new events or data. Eventually,

this can lead to staleness that may degrade the user experience. It also requires having infrastructure for storing, computing, and accessing large sets of pre-computed results.

- **Nearline** computation can be seen as a compromise between the two previous approaches. In this approach, computation is performed exactly like in the online case, yet results are posteriorly stored, allowing it to be asynchronous as in offline mode. Hence, the requirement of short response time is excluded, allowing to explore the potential of more complex processing, while still enabling the system to be responsive to user events. After receiving a request, the system computes the results and may store them in an intermediate caching or storage back-end.

Model training is commonly applied in offline mode, consisting of generating predictive models based on existing data, that will later be used to create suggestions and other personalized results. Another part of the architecture describes how the different elements communicate with each other, handling events (user interactions and activities) - *Event and Data Distribution System*. This near-real-time event flow is managed through an internal framework called *Manhattan*. A related issue is the data flow in the process of obtaining **Recommendation Results**, across the offline, nearline, and online regimes. This is managed by *Hermes*, a publish-subscribe mechanism, which allows data to be delivered to subscribers in near real-time.

Such a complex system shows the importance of planning the software architecture in which the recommender will be deployed. It's a flexible and sophisticated architecture, capable of handle great amounts of data and manage complex machine learning algorithms, while always having recommendations ready for quick responses. Finding the right balance is not trivial: " It requires a thoughtful analysis of requirements, careful selection of technologies, and a strategic decomposition of recommendation algorithms to achieve the best outcomes" (37).

2.2.3 eBay

eBay (32), as an online auction and shopping platform, presents a scalable RS architecture for recommending items with a short life span (e.g.: auctions), controlling the trade-off between relevance and quality (39). The architecture can be divided into two layers: *Offline Model Generation* and *Real-time Performance System*. The offline layer creates models using clustering ML algorithms, while the online layer combines those models with dynamic characteristics obtained from users information and activities in e-commerce. In the paper (39), the authors emphasize two main approach scenarios: *pre-purchase recommendation* and *after-purchase recommendation*. In the *pre-purchase scenario*, the RS recommends alternative products which are similar to the ones recently viewed by the user. In the *post-purchase* scenario, the RS recommends complementary products related to the one that the user has recently purchased.

Both layers use the same data store, providing two versions of similar services. Data stored may be related not only to basic information such as users, items, and user actions (navigation, access to auctions, etc.), but also to clustering results, such as which group a set of similar items belong. The *real-time layer* has two components: *Similar Item Recommender* (SIR) and *Related Item Recommender* (RIR). Both receive an item as input and return a set of similar or related items in return, respectively. As the response must occur in real time, all the computational complexity is in the *offline layer*, consisting of *Apache Hadoop* running *map reduce jobs* (40), queries and K-means algorithm.

2.2.4 SaaS and PaaS Recommender Systems

The tendency of e-commerce stores to search for recommender systems, in order to increase their sales volume and revenue, is growing over the years. However, developing a good RS can be expensive and time-consuming. This leads companies to reach for *SaaS and PaaS Recommender Systems* (41). Instead of having a large upfront investment, companies can pay as they use a SaaS model of a recommender system. The integration is usually straightforward and there are continuous cycles of improvement (42).

In this dissertation, we emphasize *Yusp*, *Strands Retail*, *Commerce Cloud Einstein* and *Amazon Personalize* as examples for using different approaches in their architectures and recommendation techniques.

2.2.4.1 Yusp

Yusp (43) is a personalization engine, developed by Gravity R&D company, the same team that tied for first place at Netflix Prize (44) - improving the Netflix algorithm by more than 10%. This service offers customization features for e-commerce platforms, having several case studies from large companies in which their revenues have increased significantly thanks to these solutions. To produce recommendations, the engine consumes data from online activities and habits of both known and first-time customers, the properties of products - such as name, price, category and other attributes - and contextual information of the customer browsing like the location or the time of the day. Due to the importance of this data, *Yusp* has security measures to protect the privacy of their clients.

It provides control dashboards where the client can customize and adapt the recommendation engine to the needs of their platform and their customers. It is also possible to obtain detailed analysis reports, thus giving several insights about the business to help make better decisions for the future.

2.2.4.2 Strands Retail

Strands Retail (45) is a SaaS that provides a *plug-and-play* recommender system focused in product and e-commerce activities. The system allows real-time recommendations, multiple personalization strategies and other options that enable users to customize recommendation results. In brief, it provides global personalization and recommendation solutions that empower online retailers, in order to achieve superior customer experience within their digital channels.

The system works by including tracking scripts on the e-commerce platform and recommendations widgets, using Strands Recommender javascript library¹⁰. This library is intended to facilitate the integration of the recommender by automatically handling important concepts like the user management and by offering a broad set of functions to interact with the recommendations API. Since the code runs in customers' browser it adds no delay to the normal rendering of the page.

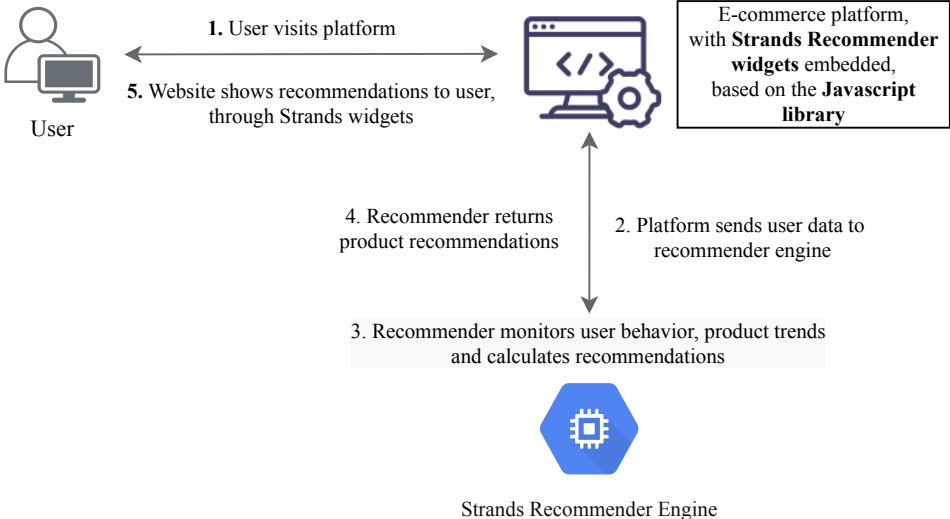


Figure 2.8: Strands Retail product recommendation system workflow overview

The more visible widgets are, the stronger their impact will be in helping customers find what they are looking for and the more likely to reach new potential clients. Most common and effective placements are the platform's home, item, category, shopping cart and order confirmation web pages.

¹⁰<http://retail.strands.com/resources/javascript-library/>

2.2.4.3 Commerce Cloud Einstein Product Recommendations

Commerce Cloud Einstein ¹¹, developed by Salesforce (46), is an artificial intelligence (AI) tool embedded right in e-commerce platforms that run on Salesforce B2C Commerce¹². Salesforce clients can easily have access to predictive intelligence and personalization without having to hire a data scientist or use a costly third-party recommendation provider.

Einstein Product Recommendations (47) provides personalized product recommendations based on a shopper's onsite behaviour and preferences, but also recommends current popular products by tracking general shopping trends. To achieve this, the client must simply create and assign a recommender to his platform; then whenever a customer visits the platform, Commerce Cloud Engine is called. Commerce Cloud Einstein learns about products, attributes, prices and inventory (*Product data*), discovers relationships between products and users (*Order data*) and collects session information (i.e. customers behavior and actions while shopping - *Clickstream data*). After "digesting" this data, it uses machine learning algorithms (e.g. collaborative filtering, unsupervised and supervised learning and deep learning) to process it. When this process is finished, it returns the recommended product IDs to the platform where storefront pages display the received product recommendations.

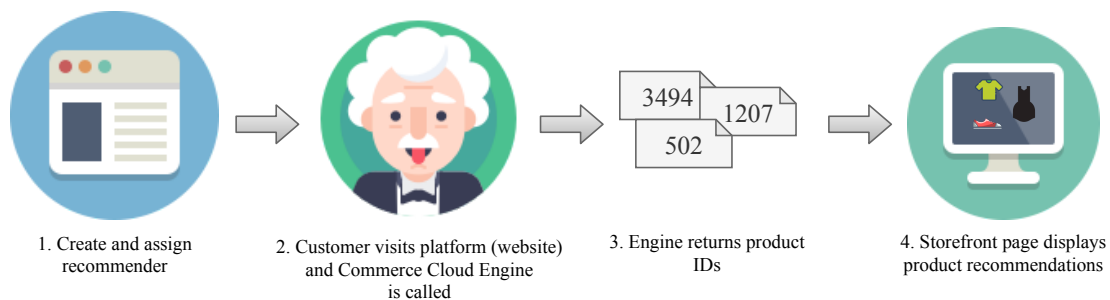


Figure 2.9: Commerce Cloud Einstein product recommendations process

2.2.4.4 Amazon Personalize

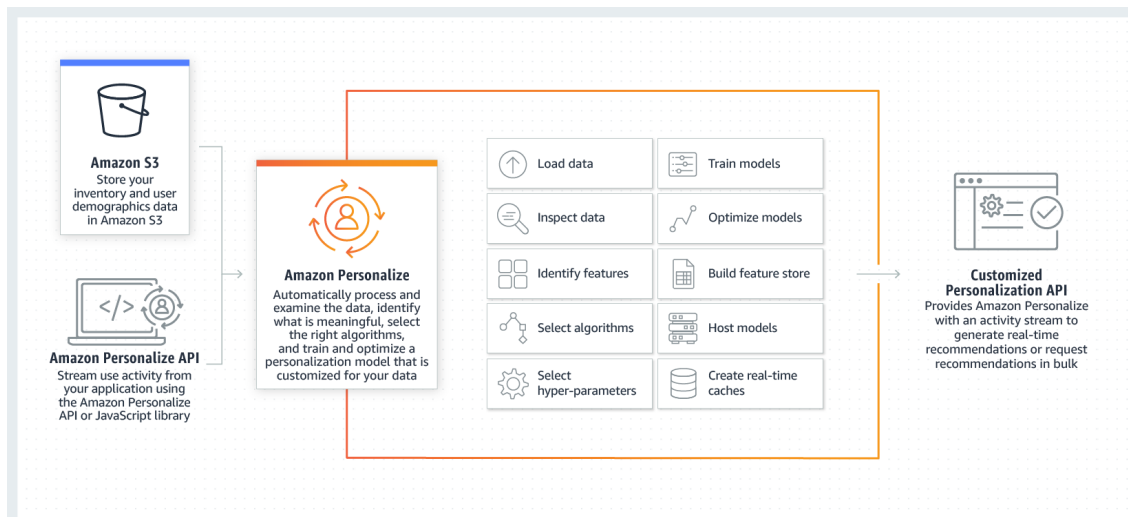
*Amazon Personalize*¹³ is a machine learning service that gives developers the capacity to integrate and personalize their own recommender system in online platforms. It creates real-time individualized recommendations for clients using their applications and allows them to personalize search and notifications for a better marketing communication. This product is available at Amazon Web Services (1) and is based on the same technology used at Amazon.com platform.

The following diagram illustrates how the Amazon Personalize service works:

¹¹<https://www.salesforce.com/products/commerce-cloud/commerce-cloud-einstein/>

¹²<https://www.salesforce.com/products/commerce-cloud/b2b-e-commerce/>

¹³<https://aws.amazon.com/personalize>



Source: Amazon Personalize, AWS (48)

Figure 2.10: Amazon Personalize "How it works" diagram

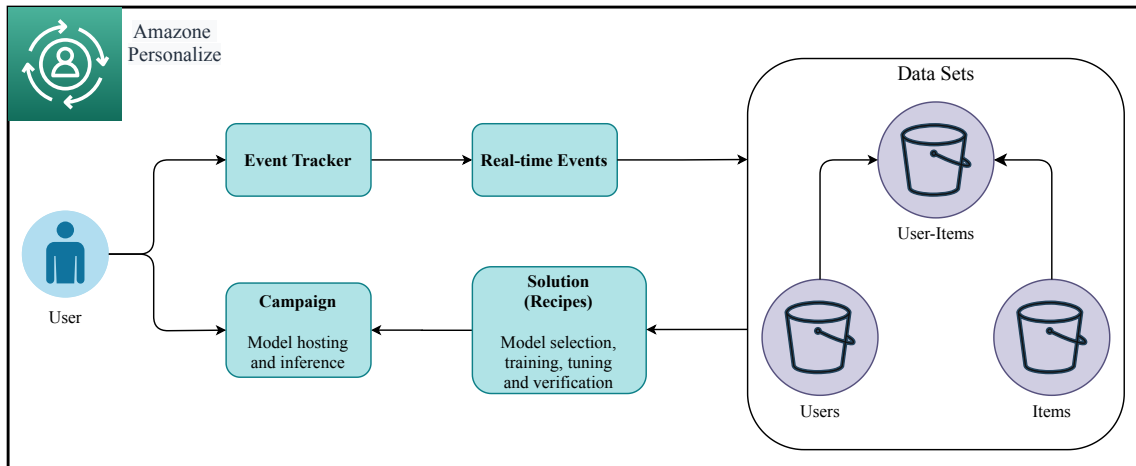
The application streams user activities to the service, through the Amazon Personalize API. It also stores demographic data in Amazon S3. The service then receives and processes this data, creating a model that best fits the context of the application. Once the service is ready, the application may request recommendations through a customized personalization API, available to communicate with the service.

After a model and artifacts are defined and trained, Amazon Personalize allows developers to deploy a *campaign* - a solution version consisting of an engine inference for the model and the trained artifacts - as a *PaaS*, because it's possible to customize its API. The campaign allows Amazon Personalize to make recommendations for users, returning a REST API that developers can use to get recommendations (49). Amazon Personalize also allows to use a JavaScript library.

To sum up, according to Amazon Personalize documentation (*Developer Guide* (50)), the Amazon Personalize workflow for training, deploying, and getting recommendations from a campaign is as followed:

1. Create related data sets and a data set group;
2. Get training data;
3. Import historical data to the data set group;
4. Record user events to the data set group;
5. Create a solution version (trained model) using a recipe;

6. Evaluate the solution version using metrics;
7. Create a campaign (deploy the solution version);
8. Provide recommendations for users.



Source: Based on Basford's article (49), Inawisdom (51)

Figure 2.11: Amazon Personalize high level architecture

2.3 Summary

Summarizing the concepts and definitions covered in this section: an e-commerce recommender system is a ML mechanism that produces recommendations based on data related to clients, products and services of a certain e-commerce store. There are two major approaches to build a RS model: Content-based Filtering and Collaborative Filtering. The model consumes data, processes it trying to match products' attributes to the clients' profile - content-based filtering - or taking into account clients' purchase history and find similar items/users - collaborative-filtering - generating recommendations.

Recommendations' accuracy depends on how suitable the selected data is to the e-commerce platform background. Data can be processed first, in order to find relevant information, by extracting patterns, detecting anomalies and other statistics - Exploratory Data Analysis. The output can be useful not only to be consumed by the RS, but also to be applied using other different methods and techniques of Business Intelligence, that companies might want to explore.

Since this dissertation will focus on the *architectural* aspect of recommender systems, it must be discussed how it will interact with e-commerce platforms. Two widely spoken concepts of communication architectures are REST and SOAP, despite being different paradigms. SOAP is a

communication protocol whereas REST is a design concept for a web service architecture and more popular nowadays, being a good option to implement.

To build a RS that matches the objectives of this project, it is important to understand the definition of Platform as a Service, as well as the difference between monolithic and microservice applications. PaaS are service platforms hosted by cloud service providers and can be centralized (monolith) or distributed (microservices) systems.

Recommender systems have been successfully adopted by many organizations, such as Netflix and Amazon, which became popular and affluent due to the efficiency of their product recommendations. Moreover, many companies invest in developing recommender systems as PaaS or hire as SaaS, which proves the great potential of RS in the e-business area.

3. THE PROPOSAL

This project consists of the design, development and implementation of an architecture for a recommender system, combined with a methodology of data analysis and processing, that will feed a certain recommendation engine and make it available to most e-commerce platforms.

The results produced by the recommendation engine are expected to target some use cases, such as marketing campaigns, product recommendations (e.g. discovering relationships between clients and/or products), search and browsing experience personalization.

In this chapter, a General Overview of the recommender system to consider is presented, followed by the Proposed Approach and the Architecture Description.

3.1 General Overview

The architecture of a recommendation system for an e-commerce platform addresses not only the communication between both entities, but also the communication between the elements of the RS itself. Therefore, it is important to develop an organized and efficient structure to manage and handle the great amount of data that will be generated by this communication, promoting a good data flow on the system.

3.1.1 Challenges

A common practice in big e-commerce companies is the construction of its own RS focused on their own business (39). However, this project presents a generic architecture in order to cover most e-commerce online platforms.

Although each of these companies' RS has its individual architecture with distinct implementations, all architectures share similar issues:

- **Personalized data sets:** Collect and preserve various types of data from different e-commerce entities. This data will be consumed by the recommender engine to produce recommendations;
- **Data Analysis:** The data analysis tools used must be versatile and capable of processing variables of various types, from the referred custom data sets, to create informative and useful dashboards;
- **Cold Start problem:** When entities (users or items) are recently registered in the platform, the recommender system has limited information about them to be able to produce accurate recommendations. Nevertheless, new customers should get relevant recommendations and new products should be included in recommendations;
- **Recommendations availability and up-to-date:** Recommendation results must always be up-to-date, according to the latest platform activities and available at platforms' demand;
- **Scalability:** As mentioned in the previous chapter, RS tends to increase the number of visitors on the platforms. Recommendations should scale across hundreds of clients and products. Thus, the recommender system must handle a great number of items and active users, simultaneously, keeping a short response time and good performance;
- **Multi-tenancy:** The multi-tenancy problem (52) refers to a software architecture in which one application instance is hosted on a server and serves multiple tenants. A tenant can be a set of one or multiple users who share common access to the software instance. With a multi-tenant architecture, an application is designed to provide each tenant with a dedicated share of the instance. The multi-tenancy definition opposes to multi-instance architecture definition, where separate software instances serve different tenants individually.

In this sense, we need to consider the following scenarios:

1. *Deploy an instance (copy) of the RS stack for each e-commerce platform (tenant):*

Pros: Simple to develop and higher performance individually. It allows delivering more personalized service to each platform, being easier to manage business logic inside each platform scope.

Cons: Global system management more complex and it consumes many resources. It is necessary to configure an additional instance each time a new tenant is considered.

2. Create an instance of the RS to handle multiple e-commerce platforms:

Pros: Allows the system to be generic and flexible. Only a type of architecture must be developed and maintained for the whole system to handle multiple tenants. Consumes less resources.

Cons: More complex to develop, but easier to configure to communicate with several platforms, at the final stage. It is expected to be difficult to separate and manage the business logic of the different platforms.

- **Security:** Data used to produce recommendations must be secured, because it is based in sensitive information such as clients' personal information, products details and orders history. Hence, privacy must be protected.

To get the most of the architecture's potential, it must be generic, flexible and capable of being deployed and tailored to most e-commerce platforms dedicated to product or service transactions.

Furthermore, data analysis methods and techniques must be able to handle different types of entities and attributes.

3.1.2 Functionalities

The recommender system architecture comprehends not only the communication between the *recommendation engine* and e-commerce platforms but also between the elements within the system. Thus, it is necessary to organize a structure capable of managing large volumes of data to be transmitted in its communication flux.

This communication process can be divided into the following steps:

1. Extract data from e-commerce platforms, necessary to train recommendation models at the engine. This data is related to clients' profiles (gender, age, addresses, ...), shopping activities (page views, browsing, client's shopping history, orders, ...) and item inventory (product characteristics, hits, ...).
2. Store the collected data in a database capable of adapting to different contexts of distinct e-commerce platforms. The system should present tools capable of processing and analyzing the collected data, in order to provide various Business Intelligence features to the company, such as future perspectives on its business and support in decision making.
3. After recommendations being generated by the engine, the system must store and make them available at platforms' demand, whenever necessary.

Throughout the communication flow, the architecture must take security measures to protect data against potential threats and reduce vulnerabilities in the system with authentication and authorization mechanisms.

3.2 System Requirements

E-commerce platforms have strict requirements that must be met in order to increase and maintain quality and offer a good experience to their clients. An important aspect to take into account is the **response time**, as it will influence the loading time of web pages. Online shopping should deliver a smooth and clear experience to their users, where results are presented as quickly as possible since 40% of shoppers will abandon a website that takes more than three seconds to load, according to a study conducted by *Forrester Consulting on behalf of Akamai* (53). Furthermore, a short page loading time is a key factor in a consumer's loyalty to an e-commerce site. The study reveals that 79% of users who have not had a good experience are less prone to return to that platform while 27% are less likely to buy from the platform's physical store, suggesting that a poor online experience may have a great impact in store sales.

Therefore, the recommender system must fulfill some requirements as well, so as not to impair the normal functioning of the e-commerce platforms and the *Service Level Agreement (SLA)* continues to be complied with. In conclusion, the impact of generating and obtaining recommendation results should be minimal on the loading time of a web page.

Taking into consideration the challenges raised in the previous section (3.1.1), the requirements defined for the system follow as below:

Functional requirements

1. The recommender system must return a list of IDs related to *hot products*. These are the most popular products at the moment and with the highest likelihood of being purchased by clients.
2. Given a client ID, the RS must return the list of product IDs recommended to that specific client, sorted by *purchasing probability*.
3. Given a certain product ID, the RS must return a list of product IDs that are similar to that specific product, sorted by *similarity score*.
4. The RS must be able to do *complementary product* suggestions. It should return a list of product IDs that best suit a purchase, according to the products currently selected by the client.

5. The RS must be flexible and capable of delivering recommendations according to the context that is required by the platform. With this in mind, the system must be capable of filtering recommendations by *category*, returning only products which belong to that filter. For example, when a user searches for women products, only products of that category should be included in the recommendation.
6. The RS must be able to return recommendations up to the maximum limit established by the tenant. If this limit is not specified, the default quantity of recommended products to retrieve in the server response is 100 items.

Non-functional requirements

1. The RS must provide a tool capable of analyzing data stored in the system's database, by allowing developers to create dashboards and calculated statistics, to develop new prospects about the future of their business and market.
2. The RS must always collect recent data from the platforms' database. Whenever a certain event occurs, the RS database must be updated, to maintain its consistency and be in sync with the platform's. Such events can be described as the registration of new clients or products, modifying their information, creation of new shopping orders and changes in order status.
3. The server must respond in JSON format.
4. The system must be functional 99% of the time in a year (361 in 365 days).
5. As for the response time, the RS must return a result in less than 100 milliseconds to 95% of the requests. No answer should take more than 150 milliseconds.
6. The RS must implement data security techniques, mainly authentication and authorization mechanisms, to protect the company's business information and its clients' privacy.
7. The RS must not only handle large amounts of data related to clients, products, orders and other shopping activities, but also be able of serving multiple platforms at the same time, scaling as necessary to keep a good performance and a short response time.
8. The RS must allow its users to configure and manipulate the training of *recommendation engine models*.

3.3 Proposed Approach

The proposed approach to build this architecture was to assemble a set of components, all interconnected by a core *JavaScript API*, supported on the concepts of *PaaS*, *REST* and the State of the Art explored in the previous chapter.

The idea resided on creating a service, having the *recommender system* hosted on a server held by Beevo, where e-commerce platforms of the company's domain can request for recommendations produced by it. This way, the RS works as a *Platform as a Service*, delivering all the infrastructure, components and logic needed to obtain recommendations. It does all the computational effort, while tenants (e-commerce platforms) only need to communicate with the RS and integrate the results in their applications. Another advantage is that the RS is developed and updated independently, not compromising the platforms' functioning.

The interaction between the RS and the platforms is possible through an application programming interface (API), which manages and describes the communication process. This communication dwells on the exchange of data, in JSON format and via HTTP requests, between platforms and the RS endpoints delimited by its API. The e-commerce platform sends business information about its clients, products and orders history to the recommender, which preserves this data for later analysis and processing.

The architecture of the recommender system stores the data, replicating part of the platform's database, forming data sets useful for two main operations: *data analysis* and *recommendations results*.

Exploratory data analysis is performed on data stored by the RS with the objective to discover and extract useful information that may help to understand the market, consumers and adopt better strategies. This data will optimize the company efficiency in decision making and business predictions, opening the door to *Business Intelligence* functionalities.

Then, the recommender engine *ingests* that data, transforming and using it to train predictive models, which create recommendations for the platform's clients. Those recommendation results are stored in a database of the system's architecture component, which later displays them on platform's demand. Although recommendations are saved in different formats, according to their type, they present the same format when they reach the platforms. Whenever a platform makes a request to the RS server, it responds with the according result, but always in the same format, regardless of the type and context of the recommendation. This format, in turn, consists of a *list of product IDs* that platforms receive and use to display the recommended products to users.

This API was developed based on REST principles, due to its advantages and increasing usage, although it's not RESTful. In order for the RS to be deployed on e-commerce platforms, an effort

from both sides is required. Company developers must know how to conceive the communication between platforms and the recommender. To accomplish this, they expect to have knowledge about the API's functionalities, in spite of the API's *documentation* contradicting some of the fundamental rules of REST.

Citing Giessler's "*Best Practices for the Design of RESTful Web Services*" article (17):

"A documentation for Web APIs is a debatable topic in the context of RESTful web services since it represents an out-of-band information, which should be prevented according to Fielding (54): Any effort spent describing what method to use on what URIs of interest should be entirely defined within the scope of the processing rules for a media type"

Usually, as mentioned in section 2.1.3, tenants should not have knowledge about servers' REST API. Describing a server REST API, using a description language to make it machine or human-readable, disregards two of the constraints of REST: self-describing messages and HATEOAS (hypermedia as the engine of application state).

Nevertheless, in this case, tenants are not normal users, but company developers and since they need to integrate the RS withing the platforms, the most common approach is to document all URIs, HTTP methods supported, and structures of representations (e.g. as JSON) so that tenant-application developers can rely on such documentation to program (55).

3.3.1 Architecture Description

Before developing the architecture to execute the functionalities mentioned in section 3.1.2, the first step relied on planning the approach to follow and the implications it brings to the system's performance.

3.3.1.1 Architectural Approaches

In this dissertation, two main architectural approaches we considered - Monolithic vs Microservices Architecture. In light of these definitions, the following options were elaborated:

- **Monolithic architecture:** With this approach, the recommender system would consist of a single application instance, which handled and managed all the functions mentioned previously (communication, data storage and analysis, authentication, etc). This could represent an issue due to the great number of requests to be exchanged between platforms and RS's architecture components. As an example, a scenario to consider could be *Black Friday*,

where thousands of users' activities would generate a huge amount of traffic on the communication network. To fight this problem, application redundancy can be applied, i.e., create multiple copies of the same application instance and then a load balancer would manage the resources, keeping the flow consistent and avoid *bottleneck* problems. This would also depend on the infrastructure that hosts the system software. Any additions or changes to the software can be of great complexity as the various components can depend on each other. Modifying one part of the software could imply changing other parts that interact with the first, as well.

- **Microservices architecture:** This option, although more complex to implement than the first, and perhaps less practical, is more robust in terms of processing and congestion control of data and requests. As initially stated, this approach consists of dividing the architecture component into several small and independent services (microservices), each dedicated to a function: one for communication, another for data storage, other for authentication, and so on. It is expected to be slightly slower than the first option, when the traffic flux does not exceed a regular day, because each request has to be filtered, validated and processed by each microservice. However, as declared before, the monolithic option can be slower on special days, with promotions like Black Friday, where there is more activity on the platforms and the data flux increases considerably.

3.3.1.2 Architectural Solution

By evaluating the requirements of this project, considering the need of response to the challenges in section 3.1.1, and by assessing the several situations and risks, as well as the characteristics of Beevo's e-commerce platforms, it was made the following decision:

A *microservices* architecture was adopted, rather than a monolithic approach, because a modular system enables the RS to be very flexible, easily implemented and extended depending on the e-commerce platforms different needs (56). This modular approach allows the usage of different technologies and methods on each component, thus facilitating the best implementation to solve a problem and fit to the companies necessities.

As stated before, a monolithic architecture may present data congestion problems (*bottleneck*) when faced with a scenario where platforms are visited by numerous users and, consequently, many events, generating a great amount of traffic in the network. With the microservice approach, a single architectural service will serve multiple clients - *multi-tenancy* -, instead of creating a copy of the service for each e-commerce platform considered as each platform is served by one instance - *single-tenancy*.

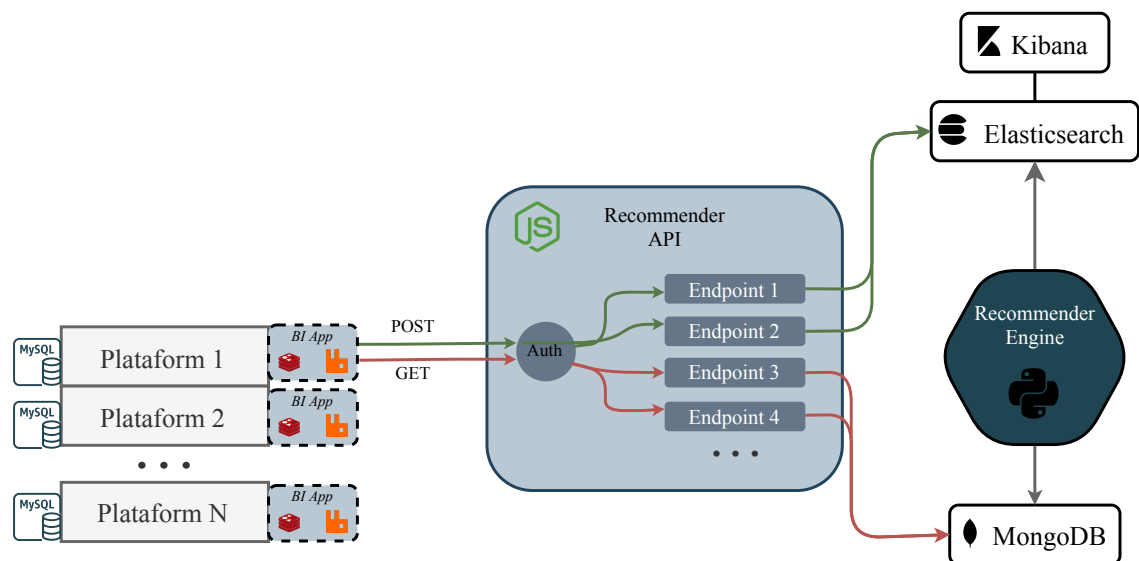
With the scaling and multi-tenancy problems solved, we focused on how to perform data analysis

on custom data sets. To make this possible, it's important to decide which database types should the RS hold. Two types of databases were contemplated: *relational and non-relational DBs*. A non-relational database revealed to be the best choice, since the objective is to build a generic and flexible architecture, capable of adapting to any type of data from different online platforms. Non-relational databases confer flexible storage to the system, accepting various variables with distinct types. On the other hand, relational databases must be planned and structured beforehand, defining what entities and respective fields is the system going to work with.

As for security concerns, it were implemented authentication and authorization mechanisms with JWT (JSON Web Tokens - RFC 7519 standard (57)) in the API.

3.3.1.3 Architecture Diagram

This section presents a detailed explanation of the system's workflow. The diagram in figure 3.1 illustrates the sequence of interactions between online platforms and components of the recommender system, data flow and other mechanisms that occur in the process.



Source: Improving Performance of Recommendation System Architecture (Appendix I - Publications)

Figure 3.1: Recommender System Architecture diagram

As it can be observed, several online e-commerce platforms connect to the recommender system through its API. In turn, this API contains multiple endpoints, each one related to a certain type of entity (client, product or event) corresponding to a given RS action. It was also recognized the existence of a *recommendation engine*, assuming that it was able to connect to the databases used in the architecture - *Elasticsearch* (58) and *MongoDB* (59). The recommender engine is capable

of storing data in these databases in the format that the architecture requires to communicate with platforms, according to a Data Contract previously determined.

For security concerns, each request is filtered by an *authentication* and *authorization* mechanism, before reaching an endpoint. This security measures validate and ensure that the author of the request has the right permissions to make use of the API, protecting it from potential threats. It was implemented authentication, with JWT (*JSON Web Tokens* (57)), and authorization mechanisms with an *Access Control List* (ACL), further explained at section 4.5.

3.4 Summary

Before the conception of the proposed architecture, several potential problems were addressed, that may arise during the development of the system, such as personalized data sets, data analysis, cold start problem, availability, scalability, multi-tenancy and security.

It was also gathered the requirements that the recommendation system must satisfy before it can be employed as a service by e-commerce platforms.

It was assumed, therefore, the development of an architecture based on a microservices approach, supported by non-relational databases, thus giving great flexibility to the system. An API was defined to serve as a bridge between the e-commerce platforms and the recommendation engine, to manage the communication between tenants and the service and ensure the privacy of the data involved.

4. DEVELOPMENT

This chapter reflects the stages of development and implementation of this project. At first, the Technology Used in the architecture is identified, described and complemented with the list of infrastructure requirements for the host machine. Posteriorly, Product Recommendations are explored, more specifically the concept and structure of each type developed for this system. In order to complete this section, *online* and *offline* computing modes are also compared. Afterwards, the System Communication Process is explained, i.e., the interaction between service and tenants, ending up focusing the e-commerce platforms side through the Beevo's Business Intelligence Application. To conclude this chapter, Security mechanisms applied by the recommendation system in the preservation and management of data are specified in detail.

4.1 Technology Used

In this section is presented a description of all components of the recommender system architecture, illustrated in figure 3.1, and their role in the process of obtaining recommendations for an e-commerce platform.

The **recommender API** acts as an intermediary between e-commerce platforms and the recommender engine. It's a communication channel between both elements, handling platforms' requests and delivering recommendation results, produced by the engine. It can be designated as the *RS manager*. To build this component, **Node.js** (60) was used as the server engine with great performance, scalability and lightweight, supporting the API with the **Express** framework (61), which has a great potential to assemble a microservices architecture. The server's application modular structure - appendix A.1 - was developed based on the *Principle of Separation of Concerns* (62), having been created three layers: *Controllers Layer*, *Service Layer* and *Data Access Layer*.

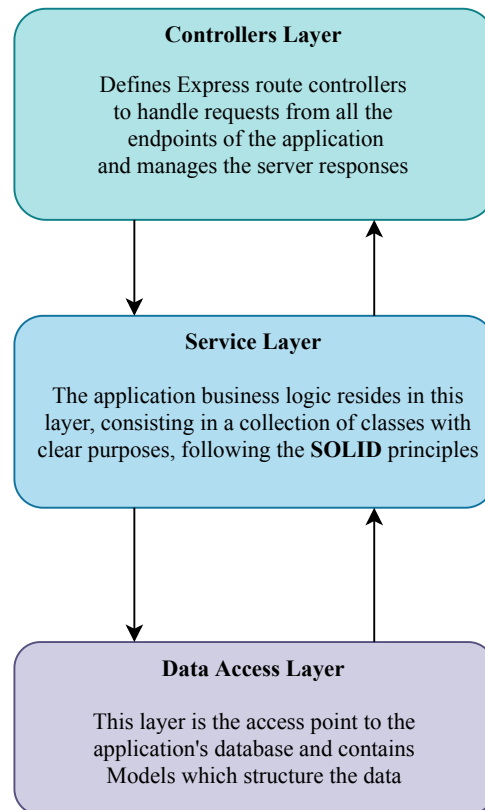


Figure 4.1: Node.js Application Modular Structure Design, according to Separation of Concerns (3)

The API allows any external entity to access the service, as long it is properly authenticated. In this case, as it can be observed on diagram 3.1, each e-commerce platform communicates with the RS through a Beevo's application, designated as **Business Intelligence Application**. The *BI App* was specifically developed to operate as a communication point for the platforms, connecting them to the RS through its *API*. Thus, both these components (BI App and Recommender API) form the communication channel between online stores and the service, handling platforms' requests and delivering recommendation results, produced by the *recommender engine*. Recommended products are then displayed in vitrines placed throughout the stores' web pages. The BI application uses **RabbitMQ** for *asynchronous communication* and *load balancing* purposes, as well as **Redis** for caching results. Another alternative for RabbitMQ would be *Apache Kafka* (63), as both technologies stand out as a distributed messaging system based on the publish-subscribe model capable of playing as an event distributor. However, it was decided to use the RabbitMQ and Redis components that were already developed and implemented by the company. These software functionalities will be explored further in section 4.4.

Elasticsearch (58) is a full-text search engine, based on Lucene¹ library, built to handle large volumes of data. Among the many existing database types, Elasticsearch was chosen because its

¹<https://lucene.apache.org/>

characteristics match with the database defined in the Architectural Solution section, to handle the unstructured data expected in the recommendation process, the need for a generic architecture and a highly scalable and flexible database.

Elasticsearch is a non-relational database that operates with schema-free JSON documents and scales very efficiently. It allows to index documents and search them in near-real-time. Using flexible, JSON-like documents, means that fields can vary from document to document and data structure can be changed over time. Therefore, it has advantages both in storage structure, performance and flexibility, being capable of handling entities with distinct attributes, e.g., different products may have different characteristics. It also provides a REST API, which means it can be deployed on any system regardless of the platform. Besides, this API is useful to access and perform analysis on data stored in Elasticsearch, opening up many opportunities to explore in the BI area.

In order to complement Elasticsearch, a **Kibana** (64) component was considered in the architecture as it is a powerful data analytics and visualization tool, that works along with this database. It enables users to create bar, line and scatter plots, or pie charts and maps, on top of the content stored on Elasticsearch, thus increasing its great potential to explore e-commerce *Business Intelligence* opportunities. This will answer the need for data analysis, addressing the Challenges list in the previous section.

On the other hand, **MongoDB** (59) is a NoSQL and document-oriented database. It stores data in the form of JSON style documents, which keeps the consistency with data format used in HTTP requests and Elasticsearch. MongoDB has highly query performance and is easy to scale, making it a good choice to preserve recommendation results. This data must be always ready on platform demand, being possible with MongoDB's replication, high availability and fast access to data.

Finally, the **recommender Engine** is a set of Python scripts responsible to produce recommendations from various types, based on platform e-commerce data. Although we won't focus on the Machine Learning techniques, algorithms and models used by the engine, it is important to have into consideration the data necessary to "feed" this component. The architecture must be prepared to handle any type of data required by the engine. Moreover, it is known that this engine is capable of interacting with the databases used in the architecture (Elasticsearch and MongoDB) and produces different recommendation models specifically to each platform. A more detailed explanation of the recommender engine workflow can be found in the article *Integrating a Data Mining Engine into Recommender Systems* (65), where this component is exhaustively described.

4.1.1 Infrastructure Requirements

To simplify the process of building, running, managing and distributing the service application, all architecture components were deployed using *Docker* (66) stack. Docker is a collection of PaaS

products that make use of OS-level virtualization to distribute software in packages called *containers*. Each component was built inside its own container, independently from others, to keep them separate and avoid potential conflicts. This allows the service to always run in the same environment (local or cloud server) and easy to share with all its dependencies.

By locking the versions' number of the software used we are promoting a more stable development, preventing problems from occurring due to updates of the third-party tools used. This project's architecture was assembled with the following software and hardware specifications:

Table 4.1: Version table of docker images used in RS architecture

Software	Docker Hub (image:version)
Elasticsearch	elasticsearch:7.4.2
Kibana	kibana:7.4.2
Node.js	node:10
MongoDB	mongo:4.2.5
Recommender Engine	python:3.8.2
RabbitMQ	rabbitmq:3.7-management
Redis	redis:5.0-stretch

Table 4.2: Server host machine hardware specifications. Note that the server is hosted in a virtual machine, emulated using *QEMU* (4). *QEMU* allows to run operating systems for any machine, on any supported architecture, with near native performance.

Host machine	Specifications
Operating System	CentOS Linux release 7.7.1908
CPU	8 cores × 2100MHz
Memory (RAM)	8 GB
Swap	3.2 GB
Disk storage	SSD 34 GB
Network	100 GbE

4.2 Product Recommendations

4.2.1 Recommendation Types and Structure

Since there are several different scenarios within the business of an e-commerce platform, there was a need of having a flexible recommendation structure to present different types of recommendations, depending on which fits better on the context of the current web page being displayed to the client. Recommendations are based on data extracted from the platforms, which is related to the three unique *Entities* considered by the recommendation system:

- **Client:** represents customers of e-commerce platforms. These can be regular users (B2C) or companies (B2B) that make purchases on the online stores.
- **Product:** represents the products available, all inventory and stock of items in the online stores.
- **Order-item:** represents the relation between a client and a product. An *order* is a set of one or more order-items, where each order-item corresponds to the link between the order and each different product (item) from that order. For example, if a client purchases three different products in a single order, then that order will originate three order-items. This entity was considered, instead of the entire order, as it is possible to represent information in more detail, facilitating the training of recommendation models by the system's engine and, consequently, increase the accuracy of recommendation results.



Figure 4.2: Recommender System Entities

E-commerce platforms send business information about clients, products and orders' history to the RS, which preserves this data for later analysis and processing by the recommendation engine. In turn, the engine produces and stores recommendations of different types and formats. However, results are sent in the same format to all platforms: a *list of recommended product IDs*, regardless of the type and context of the recommendation.

Distinct web pages of an online store may present different scenarios, hence *four types of recommendation* have been developed to cover various perspectives and functionalities:

Popularity: In this type of recommendation, a list of IDs related to "hot products" is returned. These are the most popular products at the moment and with the highest likelihood of being purchased by clients. The list of IDs is ordered by a *score*, which varies from 0 to 1 (0% -100%), and represents the probability of a product being purchased by a client. This metric is calculated by applying a formula to several attributes of the entities mentioned, such as how many times a certain product was searched and bought, its average quantity in clients' orders and the respective order status (*shipped, canceled, etc.*).

Hybrid: These recommendations are oriented to each client of the e-commerce platform. Given the client ID, the RS returns a list of recommended product IDs computed specifically for that client. As one may conclude by the type's name, these recommendations result from a combination of ML algorithms, following Customer Segmentation, Collaborative and Content-based filtering approaches. In the case of the Collaborative-filtering method, the Singular Value Decomposition (SVD) (67) algorithm is used. On the other hand, Content-based filtering uses the Term Frequency - Inverse Document Frequency (TF-IDF) (68) technique. Customer Segmentation uses K-means algorithm to group clients with similar profiles in clusters. An hybrid model is composed by these different sub-models, which in turn can be configured and balanced in order to complement each other's flaws and obtain more robust recommendations.

Similar Products: Recommendations are oriented to each product of the platform. Given a certain product ID, the RS returns a list of product IDs that are similar to that specific product, sorted by similarity score. In this type of recommendation, the TF-IDF algorithm is also applied to look for similarities between the textual descriptions/characteristics of each product.

Complementary Products: These recommendations serve as suggestions for completing clients' current shopping cart. Given the IDs of product selected by a client, the RS returns a list of product IDs that are usually sold together with the selected items, trying to complement a client's current purchase. Each suggestion produced by the recommendation engine is stored with a *support* value associated, that corresponds to the probability of certain products being purchased together. This support is a metric related to Association Rules with Apriori algorithm (69), which is used to calculate these results and it is very popular to solve this type of problems.

4.2.2 Recommendation Storage

When the service receives data from platforms, duly identified with the type of entity to which they correspond, these are stored in the Elasticsearch database. Subsequently, the Elasticsearch indices are organized by the three entities of the system, per platform,: $\{\text{platform}\}_\text{clients}$, $\{\text{platform}\}_\text{products}$ and $\{\text{platform}\}_\text{order-items}$.

$\{\text{platform}\}$ is the name of the platform which is used as prefix, to distinguish data from different platforms (e.g. `deeply_clients`, `deeply_products`, `deeply_order-items`, where the online store is named `Deeply`).

An index is created for each platform-entity combination, instead of storing all data into a single index, as searching against smaller data sets is faster and it is less limited in mapping structure. If there is a need to change an index for new data, it can keep old data without reindexing and just put a new mapping for the new index.

The platform name is also used as prefix in Mongo collections. Recommendations are stored in a collection identified by the type and the platform: `deeply_popularity`, `deeply_hybrid`, `deeply_similar_products` and `deeply_complementary_products`.

- *Popularity* collections contain just one document as these recommendations are only intended for one type of clients (anonymous clients, i.e., not logged in in the website). Furthermore, each recommended product has associated a *list of categories* to which it belongs. This list of categories will be used by the RS to filter recommendations, depending on the context for which they were requested (section 4.2.3).
- *Hybrid* collections hold several documents and, despite documents following a format similar to the previously described, each document is related to a client, containing the respective recommended products.
- *Similar products* collections contain several documents, one document related to each product of the platform, containing a list of products which are similar to it.
- Finally, *Complementary products* collections store documents which translate the association rules generated by the recommender engine. Each document is identified by a product ID and contains a list of product IDs that are commonly present in clients' orders with the first. In turn, each of these suggested products have their *support* value associated, as previously mentioned.

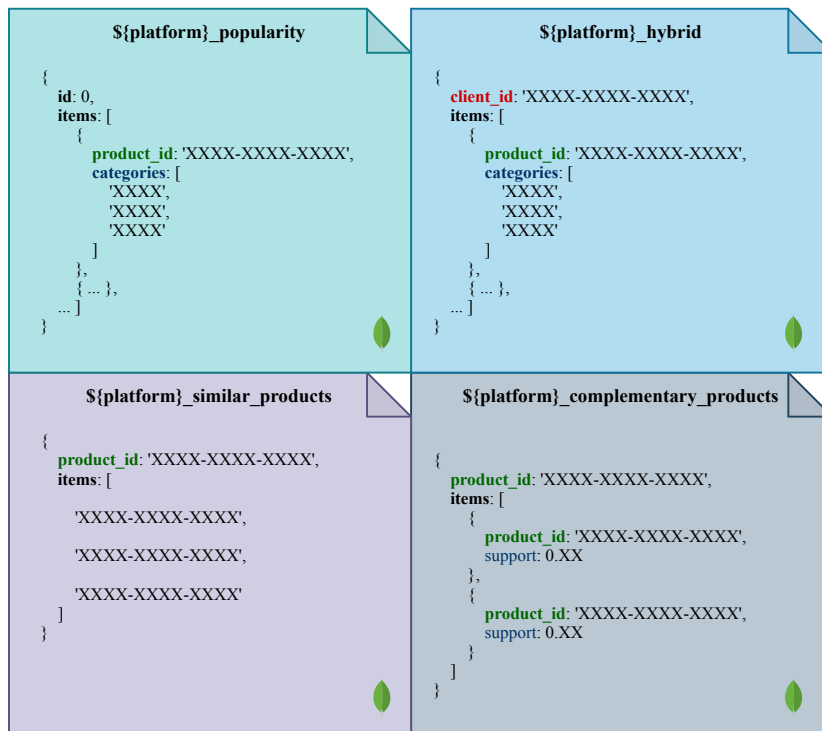


Figure 4.3: MongoDB recommendation documents structure

The process of saving a recommendation, in MongoDB, is instant and consists in replacing the existing recommendation for a recent result, thus not increasing exponentially storage space. Additionally, every document is saved separately so there's no downtime in recommendations' availability. This means that while the recommender engine is updating recommendations, platforms are still able to get results from the RS, because MongoDB database persists the previous results during this process.

4.2.3 Filters

It is evident that the number of recommendations generated is proportional to the number of products available on the online stores. The recommender's API allows a tenant to specify the maximum number of recommended products it expects to receive, as well as the category or set of categories to which those items should belong. If the limit of recommendations is not specified, the default maximum quantity of recommended products to retrieve in the server response is 100 items. This acts as a security measure, if for any reason there is a large number of recommendations for a certain kind of request, preventing the transmission of unnecessary amount of data, since usually when a client performs a search, it doesn't inspect more than 100 results.

On the other hand, when a tenant makes a request indicating the categories to which it expects the recommendations to be related, the system must ensure that all recommended products respect the

filtering rule. As previously mentioned, the filtering process is only possible for recommendations of the type *popularity* or *hybrid* and it is executed according to the following steps:

1. Find the client's recommendation document stored at MongoDB database;
2. Obtain all recommended products which belong to categories from the request's query filter;
3. Order recommendations by *score*, in descending order;
4. Apply limit of recommended items quantity;

If this filtering process did not occur, it was possible that recommended products with a higher *score* would be sent in the response, even if they don't belong to the indicated categories, and may not correspond to the context that the tenant desired. An example of this is the product listing scenario: when a client selects a category on the website, only products within that category should be displayed.

When the tenant sends a request to the recommender, the categories found in the query are individually tested against the list of recommended products. Later, the recommender selects each product that includes the indicated categories. This process is done by MongoDB, which acts as a very powerful and effective search engine, and therefore a good filtering tool. In addition, it is possible to create search indices to increase the performance of MongoDB search.

For *complementary products* recommendations, the filtering process is as followed:

1. Fetch complementary product suggestions, for each product selected by the tenant;
2. Remove products that are already selected from the suggestions;
3. Sort product suggestions by *support* value, in descending order;
4. Get complementary product suggestions IDs and remove duplicates, since different products may be complemented with the same product;
5. Limit the result's length with given *limit* value;
6. Return complementary product suggestions.

In this type of recommendation, as well as in *Similar products*, there is no option to filter by categories, since the contexts in which they are applied do not require filtering by this parameter.

4.2.4 Online and Offline Computation

As mentioned in the Netflix's Architecture Overview section, algorithmic results can be computed either in *online* or *offline* mode.

In **online computation**, recommendations are created in real-time, responding better to recent events and user interactions, whereas in **offline computation** results are calculated and stored in batch mode for later use upon platform's request (*offline jobs*). **Nearline computation** suggests the combination of both approaches, in which it's performed fast and simple computation (online mode), but it's not required to serve in real-time. Instead, results are stored making it asynchronous (offline mode).

Decisions regarding the method and frequency of calculating recommendations were reflected in these concepts. In the end, these decisions can be summarized in two considered options:

1. Training recommendation models in periodic cycles (e.g., every day at 3:00 am, since there is less activity on the e-commerce platforms) for all clients and store recommendations in the database. Thus, when a platform requests recommended products for a given user, the response is immediate, due to the pre-computation performed. Nevertheless, this approach can present several problems, such as the great computational effort on the engine during the period of generating recommendations and after a given moment these are no longer up-to-date, since they are not produced in real-time.
2. Training recommendation models each time a platform requests the list of recommended products for a user. This approach has the advantage of producing recommendations always taking into account the latest data from the platforms. However, if the models are too complex, the training and recommendation process can influence the response time, by increasing it and making it unacceptable. Additionally, the computational effort is proportional to the number of requests the RS receives, presenting a possible bottleneck when the number of requests increases.

Ultimately, to achieve a flexible and efficient system, all approaches should be considered and combined. The general idea is to pre-compute part of a result with an offline process, and using it as a backup, leaving the less costly or more context-sensitive parts of the algorithms for online computation.

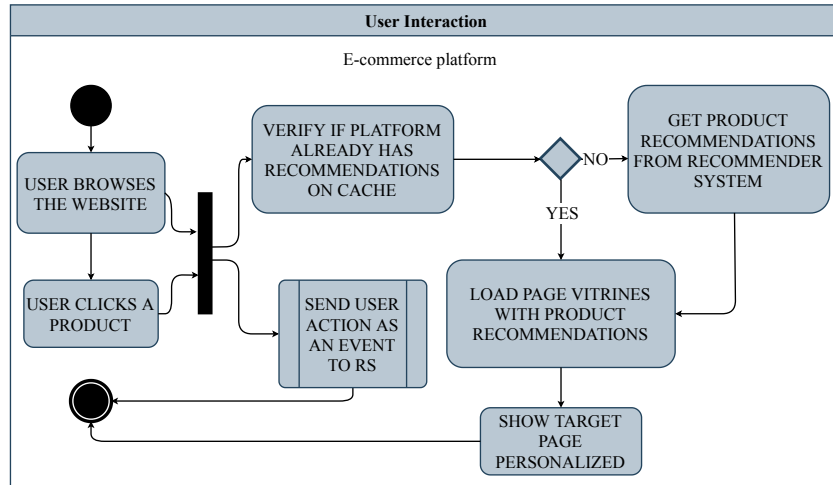


Figure 4.4: User Interaction Activity Diagram

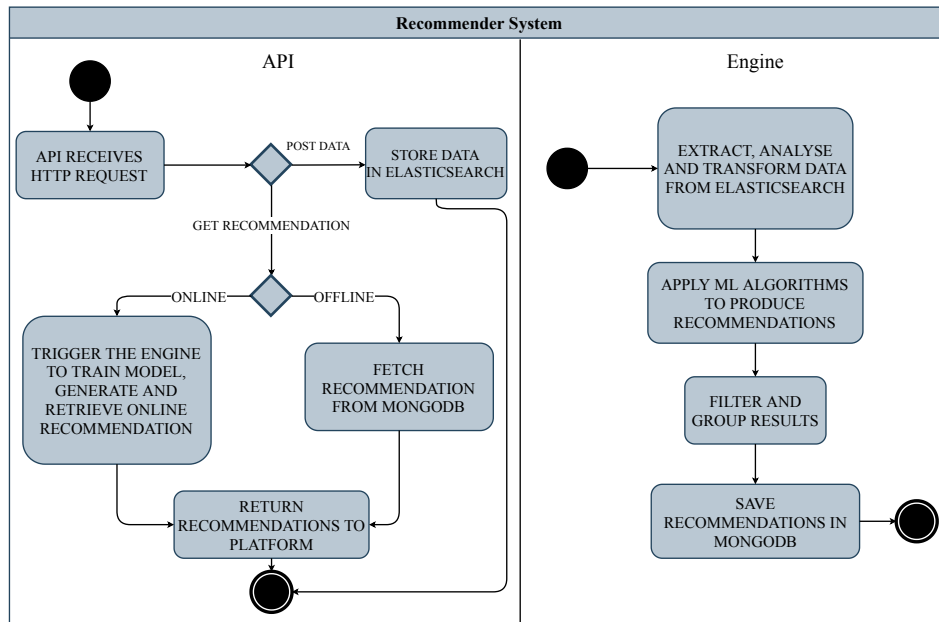


Figure 4.5: Recommender System Activity Diagram

4.2.5 Offline recommendations

Much of the computation of recommendation ML algorithms can be done offline. *Offline* or *schedule jobs* can be defined as tasks executed in background, on the machine, by a *job scheduler* (70). This program monitors and manages batch jobs automatically, allowing users to define and control a schedule to execute those jobs. With this in mind, jobs can be scheduled to run these algorithms periodically and their execution does not need to be synchronous with the request or presentation of recommendation results.

In this project, *Cron* was used as a job scheduler, which is compatible with the operating system of the host machine, to execute *cronjobs* that are applied to model training and recommendations production on the recommender engine. A *crontab* was created to list and configure several jobs; each job runs periodically and executes a python script of the engine, to train models and generate recommendations for each different e-commerce platform. For example, `00 03 * * * python recommend.py` deeply configures a job to run `recommend.py` everyday at 03:00 am, when there's less activity on the website. The `recommend.py` script makes the engine to perform an ETL (Extraction, Transformation & Load (71)) process, extracting data from Elasticsearch, analysing and transforming it into relevant information (EDA), and loading it into data sets to apply ML algorithms, training models and updating recommendation results of Deeply e-commerce platform.

This way, it is possible to use more powerful and complex recommendation techniques, by executing them in the background allowing to produce more accurate results, asynchronously within the system. When the process ends, results are stored in the MongoDB database, being available anytime the platform demands for recommendations.

4.2.6 Online recommendations

Although offline recommendations may provide quite accurate results, the updating rate of these results can be very low. For example, consider the following scenario:

1. A client buys several different shoes;
2. When the recommender engine generates recommendations, it will take this into account and recommend other shoes;
3. Moments later, the client buys various t-shirts;
4. Recommendations remain only for shoes during the period in which the engine stays idle. If the engine runs in cycles of one day, only the next day will the recommendations be updated. In other words, the engine will train the models again taking into account the client's latest purchases, in this case the t-shirts, only on the following cycle.

Summarizing, if a client buys a certain type of product, than only on the next cycle will there be recommendations for him that include products of that type, as models are re-trained and results updated.

A possible solution for this scenario is to make the recommender engine stay put, permanently listening for requests related to online recommendations. The recommender API will receive and handle these requests from platforms and redirect them to the engine, which in turn simply starts the

necessary processes to obtain the demanded recommendations. In offline mode, the engine just takes into account the data loaded at the beginning of the training. If a client has purchased other products in the meantime, the engine will not take these orders into account as this information will only be loaded in the next training session. To obtain recommendations in real-time, a lighter and faster model would have to be developed, even if it is not highly accurate, so as not to increase the response time significantly and to be used only for situations in which it is necessary to have real-time recommendations: results are only produced when requested by the platform, not being needed to store them in the MongoDB database. This model would always take into account the client's latest orders.

However, this solution will depend on the scalability of the system, since it will generate a large number of requests to process, derived from the multiple interactions of users in the platforms. Besides, it is necessary to consider the time for training and formulate new recommendations, as it will influence the response time. In fact, if the recommender is triggered every time a user purchases, to recalculate the recommendations for that user taking into account this new information, a large flow of information will be generated in the system, compromising its capacity to process data, recommend and, consequently, increasing the response time.

4.2.7 Nearline recommendations

In order to keep the performance of the system stable, it was decided to exclude the solution of online recommendations and adopt an intermediate approach: *nearline recommendations*. This method consists of a faster and simpler recommendation model from the ones used in offline mode, which trains in smaller cycles (every 20 minutes, for instance) to produce *near real-time* recommendations, e.g.: when a client buys t-shirts, 20 minutes later he is presented with t-shirts recommendations.

Nearline and online modes provide adaptability to the system, updating recommendations according to the client's recent interactions with the platform. For the nearline mode, it was necessary to create a lighter hybrid model that uses variables more focused on this context, that is, a model that has a reduced execution time to achieve a balance between short updating cycles and a sense of adaptation from the client's perspective. Nevertheless, the disadvantage of a more simplistic recommendation model is reflected in less accurate results. The engine trains this model and generates results more regularly in nearline mode than in offline mode, but the recommendation process is similar on both, saving the results in the MongoDB database at the end, in different collections.

This is, therefore, another very useful recommendation mode, which can be applied in several scenarios, complementing the offline mode, e.g. : a showcase on the platform's Homepage where the first 3 products would be nearline results and the others offline, presenting the most recently

updated recommendations first and giving the feeling of an adaptive system. Yet, offline mode is the most frequently used, since it is more accurate and is always available, allowing an immediate response time and also serving as a backup for the other modes.

4.3 System Communication Process

As previously mentioned, the core of the recommender system's communication focuses on its API. It acts as an intermediary between the platforms and the recommendation engine, making it possible to divide this communication process into two operations: the communication between API and Engine and the communication between RS and e-commerce platforms.

When developing this recommendation system, the **interaction between API and Engine** was first considered, where two options were contemplated to achieve this process:

1. Develop the API *independently* of the recommendation engine. The engine would be considered a *black-box*, ignoring its mode of operation, software and algorithms used. The communication between these two elements would be made through HTTP requests based on REST, that is, the engine would also have to have several endpoints in order to exchange requests with the system's API. The engine's development would be solely focused on the formation of recommendations, disregarding the origin of data or the destination of recommendations. Thus, the API would manage the entire process of storing data and delivering results to platforms. So, whenever the engine needs to train the models and update recommendations, it would reach the recommender API, which in turn would fetch the necessary data from the data lake (Elasticsearch) and return it to the engine. After training and generating the recommendations, the engine sent the results to the API, which would save them in MongoDB, for later platform demands.

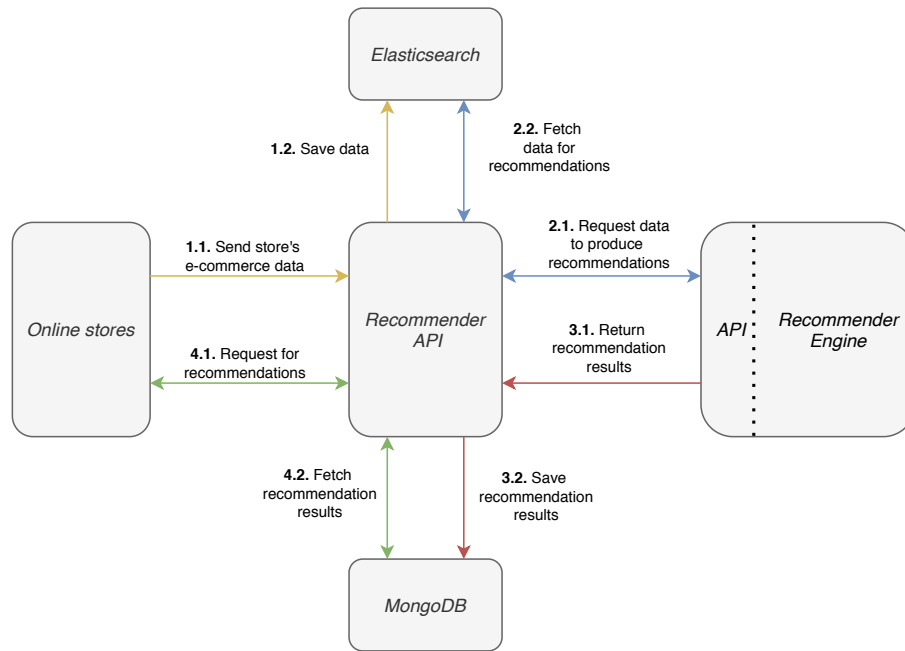


Figure 4.6: Communication process between Recommender's API an Engine - option 1

2. Recognize the functioning of the recommender engine, assuming that it not only has the capacity to access Elasticsearch directly to retrieve the necessary data for model training, but also direct access to the MongoDB database to store the results.

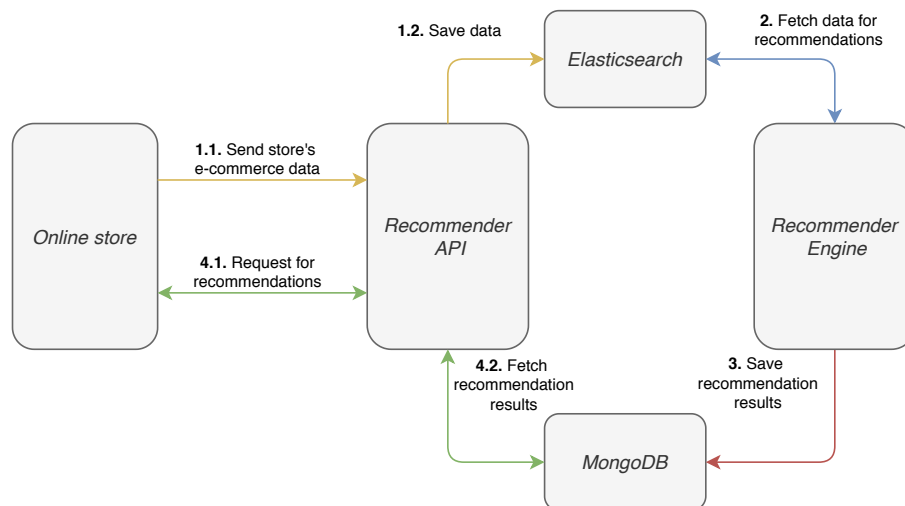


Figure 4.7: Communication process between Recommender's API an Engine - option 2

The first option aimed to produce more generic elements, making the system more flexible. In other words, when implementing a REST communication between the API and the engine, it allows collecting recommendations from several distinct engines, thus having a more robust base of outcomes to be provided to platforms. It would also make it possible to test and replace different engines,

giving the possibility to choose the one most suited to the tenants' goals. However, the complexity of data management in this communication approach does not make the process very efficient, hence adopting the second option. Since the context of this project was properly defined, as well as its objectives, the second approach offers greater performance and effectiveness in the system. So, we assumed the development of a recommendation engine (65) that can connect directly to the databases, with no need to implement REST in the engine for communication by HTTP requests, consequently taking the weight off the system's API of managing too much resources. It is assumed that the engine produces and stores the results in the format that the system architecture requires to communicate with the e-commerce platforms.

The **communication between RS and e-commerce platforms** is made via HTTP requests, through the API based on a REST architecture. This allows the system to be used not only by Beevo, but also by other external entities, i.e., recommendations generated by the recommender can be requested by several different tenants, that may not be related to the company, providing they are duly authenticated in the service. The communication with the RS must be properly authenticated and authorized, as well as validated according to a **Data Contract**. Any entity can communicate with the RS and obtain recommendations if they follow the data contract and this is what offers the system such flexibility to cover all e-commerce platforms.

In the context of the project, the recommendation system was assessed through the interaction with Beevo's platforms, which communicated with RS through the *Beevo's Business Intelligence Application*.

4.4 Beevo's Business Intelligence Application

As stated in the previous section, any entity can communicate with the recommendation service through HTTP requests, as long as it complies with the requirements established in the data contract, regarding formatting and mandatory data that must be included in those requests.

In order to complement the communication process at the company's side, an application called *Business Intelligence Application* or *BI App* was developed, allowing platforms to communicate with this service, transmitting all the information necessary to form data sets, perform EDA, Model Training and obtaining product recommendations. The reason why it was called *Business Intelligence App* is because in the future this application will hold more features, directly linked to the BI area, in addition to this first feature developed - *product recommendations*. The same is reflected in the recommendater server, which will later host other types of services related to the BI area, hence adopting a generic and flexible development.

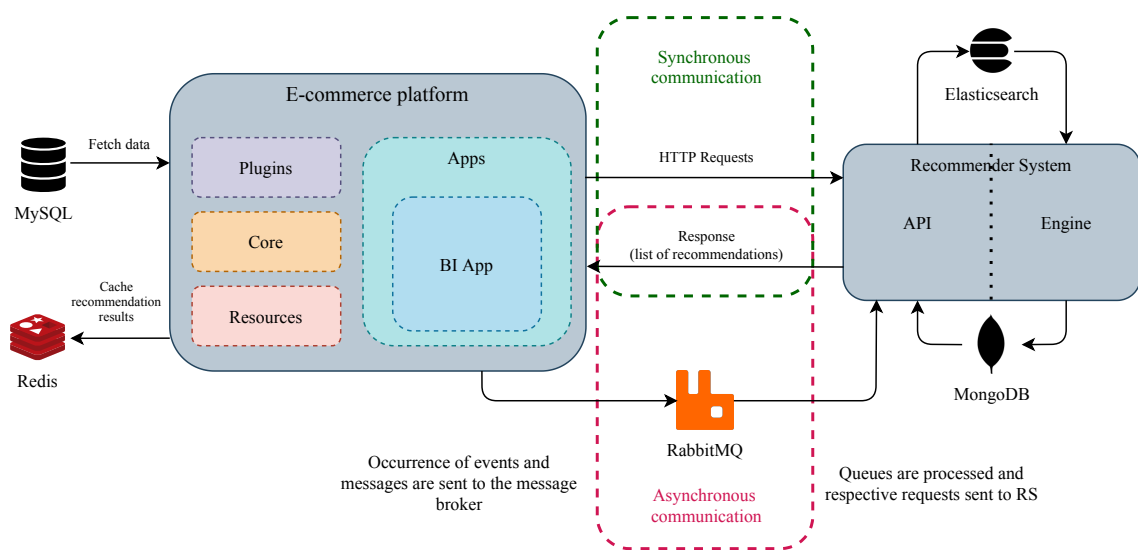


Figure 4.8: Communication process between BI App and Recommender System

Each platform is capable of integrating the *BI App*, earning the ability to interact with the recommendation service and, consequently, obtain product recommendations.

This application was developed in PHP (72), a language specially suited to web development, which is the context where this project is inserted. The following diagram shows the general file structure of the BI App developed. Note that some nuances may not correspond to reality to protect the company's property and privacy:

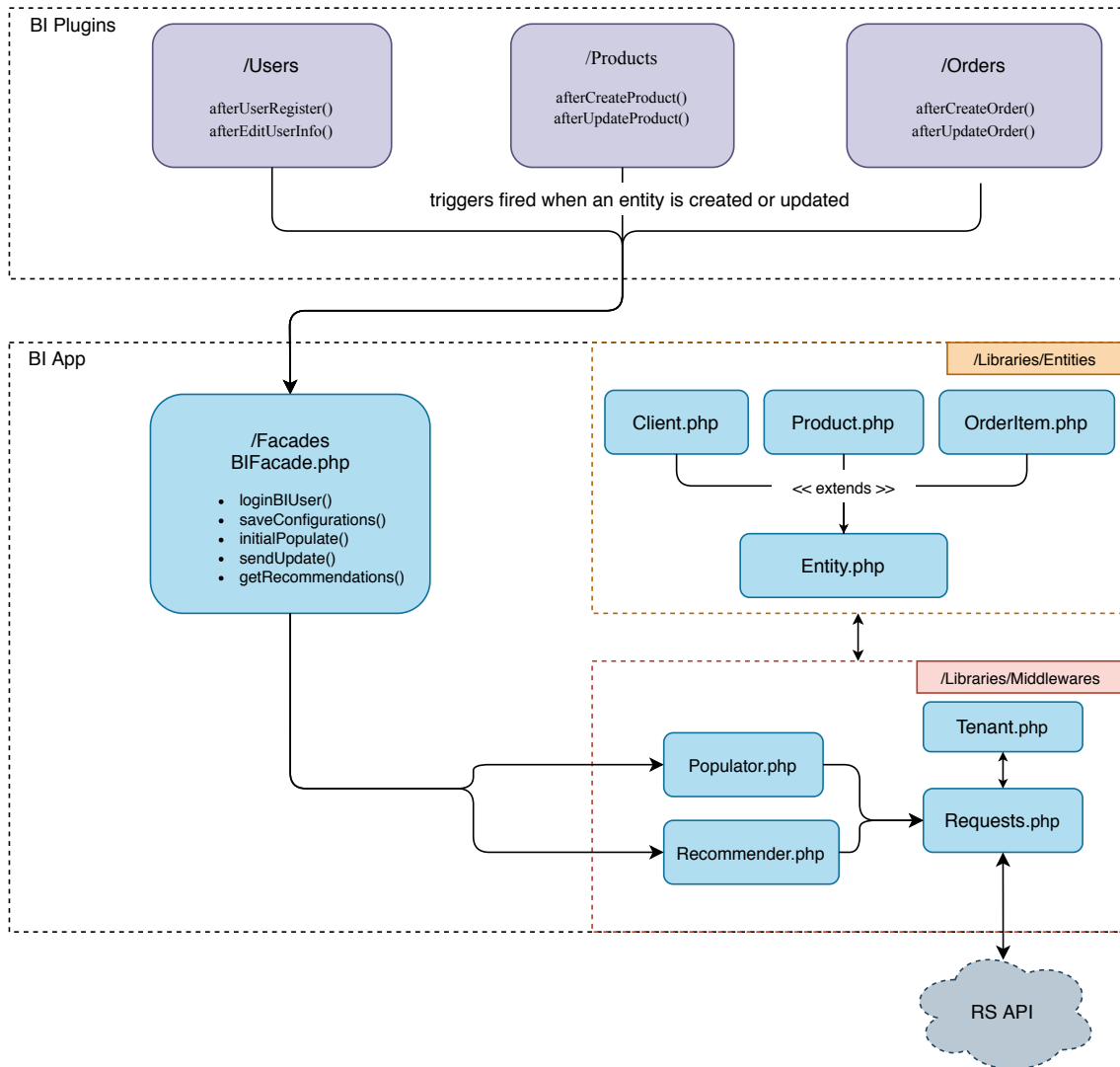


Figure 4.9: Beevos' Business Intelligence Application File Structure

The main directories of the application can be described as:

- **Facades:** The *BI App* presents a facade (`BIFacade.php`) with all available methods that enable Beevo to interact with the Recommender System. It allows each tenant (e-commerce platform) to authenticate itself in the recommendation service, edit settings of the engine's recommendation models, send the necessary data to generate recommendations and, finally, obtain product recommendations.
- **Entities:** It holds the three types of entities considered by the system: Clients, Products and Order-items. The class `Entity` is an abstract PHP class model representation of the three entities, which is then extended to the respective classes. Each of these has specific methods to extract, process and manipulate data from the respective entities, stored

in Beevo's database. Each class also defines a set of default fields, which are used as the foundation for the formation and training of recommendation models. These standard fields are specified in *Variables Selection and Filtering* section.

- **Middlewares:** Here we have two main classes: `Populator.php` and `Recommender.php`. *Populator.php* is responsible for sending information from Beevo to the RS and populate its database, using the existing methods in entities' classes. It handles events that occur in the platform, extracts, filters and sends entities' data to the RS API, which in turn receives and stores it in Elasticsearch to feed the engine. It also sends requests containing the settings that should be used by the engine to generate recommendation models for the respective platform. On the other hand, *Recommender.php* has the role of getting the appropriate recommendation results by requesting it to the RS, depending on the context in the platform.

Both of these classes can send requests to the RS, through the `Request.php` class, which acts as the *HTTP* client of the *BI App*, by implementing a *cURL* client (73), and handles request's authentication and management. The `Tenant.php` class represents the identity of a *RS User*, who can access and interact with the recommender. This class will, therefore, be used to authenticate and validate requests sent by *Requests.php* class. The tenant's authenticity is given by their login credentials (email, password and associated platform) and it is through it that the *BI App* gains legitimate access to the recommender system.

- **Plugins:** *BI Plugins* have the duty of dealing with events that occur on the platform, firing triggers whenever there is a change in the entities' information, useful for model training and generating recommendations. These triggers send messages, whose content is the information of the entity that has been changed at that instant, to *RabbitMQ*. In turn, this message broker will process the messages and send the respective updated information to the RS. The objective of this process is to keep the RS database updated and consistent with the platform's. In section 4.4.3, the operation of these triggers will be explained in more depth.

4.4.1 Application configurations

To integrate the *BI App* on a platform it is required to set some configurations, necessary to establish a connection to the recommendation service and obtain the appropriate engine results. These settings are described in the following table:

Table 4.3: Beevo's business intelligence application configurations

Recommender API	
url	Recommender System API base url (server domain).
Tenant	
platform	Name of the online store to be associated with the tenant. Used to name indices/collections on databases.
email	Email credential to login in the RS.
password	Encrypted password credential to login in the RS.
Entity	
Client	Client entity.
OrderItem	Product entity.
Product	OrderItem entity.
Fields	
Addresses	List of fields to be selected, related to clients' addresses.
Client	List of fields to be selected from the client model.
OrderItem	List of fields to be selected from the order-item model.
Product	List of fields to be selected from the product model.
Attributes	
Product	Set of product attributes to be extracted from the database.
Model	
last_months_orders	Number of months to be considered in order's history when producing recommendations.
min_score	Minimum score of recommendations to save in database.
min_support	Minimum support of complementary product suggestions to save in database.
n_clusters	Number of clusters to use in Customer Segmentation (k-means clustering).
cb_weight	Content-based model weight on Hybrid recommendations.
cf_weight	Collaborative-filtering model weight on Hybrid recommendations.
cl_weight	Clustering model weight on Hybrid recommendations.
bought_products	Include (or not) products already bought by a client, in their recommendations.
evaluate	Configuration to indicate if metrics should be calculated to evaluate models accuracy, whenever they are trained.

These configurations include the domain address of the server where the recommendation service is hosted as well as the tenant credentials necessary to authenticate within the system and send requests. Settings related to the *Entity* parameter consist of *boolean* values that indicate whether the respective entity should be considered in RS database initial population process, or not. In *Fields* parameter, lists of optional fields can be configured, which will be added in the Model Training phase, combining with the standard fields defined in the model class of each entity.

The parameter *Attributes* is only related to the *Product* entity. If no attributes are defined, the attributes used in the selection and filtering of product listing on the online store will be selected, by default.

The remaining parameters are associated with the specific recommendation *Model* of the platform, produced by the Engine. Since these parameters are related to the RS and not the BI App, any external entity, in addition to the BI App, can send a request with these settings to customize the recommendation results they want to obtain. These configurations are stored in the *recommender_configs* collection at MongoDB database so that the engine accesses it to determine how to develop the recommendation models.

The *min_score* value defaults to 0.4 (40%) and only recommendations with a higher score than the minimum score are persisted in the RS database. On the other hand, the *min_support* values defaults to 0.1 (10%) and only complementary products with higher support than the minimum support are saved in the RS database. These default values were established to avoid storing disposable results, whose precision makes the recommendation irrelevant. Thus, there is better management of the storage space of the databases and greater efficiency in the search and obtaining recommendations.

All these configurations allow greater flexibility and customization of the recommendation engine models, by changing their behaviour, adapting them to the context of each platform and, therefore, obtaining better recommendations.

4.4.2 Database population process

As previously mentioned, the class `Populator.php` is responsible for sending data to the RS API which will populate the Elasticsearch database. The process is executed according to the following steps:

1. The first step is to make sure that the tenant is able to send valid requests, so it is necessary to verify if they are authenticated in the recommendation service;
2. Subsequently, the entities whose records must be sent to populate the RS are obtained using the parameter *Entity* of the BI App settings;
3. Next, each of these entities is populated. For that, a set of ids of all the respective entity's records is collected.
4. Finally, this set is divided into 4 different chunks. The reason for dividing this set is to increase the performance and speed of the population process: each chunk will be processed individually in *background*, at the same time, thus making the population process *concurrently*. This is possible to achieve through the message broker RabbitMQ. Messages are sent to it, responsible for carrying out the processes of populating the records that correspond to the ids of each chunk, hence being sent 4 messages in total, by each entity. The messages

are subsequently processed in queues, in parallel, increasing the rate of requests sent and consequently making the population process faster.

5. The method `populateAll()` will send HTTP POST requests to RS, with each request corresponding to an entity's entry record. The various types of requests accepted by the API can be explored in the API Documentation section. According to the activated RS endpoint, data from requests is stored in the Elasticsearch database.

The pseudocode of the population method, present in `Populator.php`, is displayed below:

```
<?php

/**
 * Populator class
 */

// Check if tenant is logged in
if (!$tenant->isAuthenticated()){
    return false;
}

// Get active entities
$entities = $this->getEntities();

// Populate RS database with records of each entity
foreach($entities as $entity) {

    // Get all entry ids of entity
    $ids = $entity->getAllIds();

    // Divide the set of ids into 4 smaller chunks.
    // This way, a message will be sent to RabbitMQ to process each
    // chunk, improving the population process performance
    $ids_chunks = array_chunk($ids, count($ids)/4);

    // Process each chunk
    foreach($ids_chunks as $chunk) {
        // Run population process in the background
        $this->closure(
            function () use ($entity, $chunk) {
                // Send data to recommender API, via HTTP POST requests
                BIFacade::populateAll($entity, $chunk);
            }
        );
    }
}

?>
```

At a first stage, the application must do an initial population of the RS's database, by fetching current data related to the entities mentioned, filtering it by selecting only the relevant fields for recommendations and send it to the RS's server API, which in turn will handle and store this information.

Since there are thousands of records of each entity stored in the company's database, the population process will take a long time, hence being transformed into multiple smaller tasks that run concurrently in the background, to make a better experience for the tenant. This is crucial to create the initial data sets, so that the engine has the required data to form recommendations.

4.4.3 Event triggers

An e-commerce platform is subject to a lot of activity since the intense interaction with users leads to a large number of events, such as the registration of new clients, order placements, browsing, shopping, promotions, database updates, etc. Collecting this constant change and exchange of information is essential to make more accurate recommendations. Hence, the recommender system must be able to listen these events so that its database remains consistent with the platform's, and recommendation results are always up-to-date.

Events' generated by consequent activities in Beevo platforms, such as the creation or update of entities will be communicated to the recommender, so it can be continually up-to-date. This is possible, due to triggers fired whenever one of these events occur. These triggers were implemented in *BI App* plugins which listen for events such as:

- *Clients (/Users)*: A client registers, or edits their profile information, such as gender, birth date, country, etc.
- *Products (/Products)*: A product is created or its attributes changed.
- *Order-Items (/Orders)*: A client places an order or the order's status is changed.

Whenever such events occur, the *BI App* sends updated information about the respective entity to RS, through HTTP requests to its API. This way, it is possible to combat the *cold start problem*, always keeping the RS database consistent and synchronized according with the platform. It guarantees that all entities registered on the platforms are inserted in recommendations, ensuring that there are no gaps of information to provide recommendations to recent clients and include new products in results.

Once again, the *BI App* uses *RabbitMQ* message broker to enable *asynchronous communication* of information sent by the mentioned triggers. This way, *bottleneck* problems are less prone to occur when there's intense activity on platforms, while loading time is not affected. *RabbitMQ* receives

messages from triggers and distributes them across several queues, balancing the load. In turn, these queues process messages asynchronously and sends the information almost in real-time to the RS server - diagram 4.8.

This technique gives RS the possibility to implement *online computation* in the future to create recommendations in real-time. Besides, by keeping the Elasticsearch database always up-to-date, it allows to perform real-time business analysis in Kibana's dashboards.

4.4.4 Storefront widgets

Specific widgets were created to display recommendations to users through vitrines on the platform, communicating with the RS, by using the BI App methods and identifying the client for which it is requesting recommendations, the number of recommended products and the categories to which they should belong, in case these filters are applicable.

These **recommendation widgets** request recommendations according to the type that they were configured, displaying the recommended products that were received through vitrines in the web page. In this project, *four* different features were developed, which use different types of recommendations, and were implemented in several widgets strategically placed through the platform.

In the **Homepage**, "*Recommended Products*" vitrine presents a set of recommended products specific for the user currently on the page, based on *hybrid* or *popularity* recommendations. Popularity recommendations are used everytime the user is anonymous, i.e. not logged in. Otherwise hybrid recommendations are requested. However, if there are no hybrid recommendations for the client, popularity recommendations are used as default as they represent a generic result to use in these situations. This way, the cold start problem can be solved, in the sense that new or recent clients get recommendations, even though the system has no information about them.

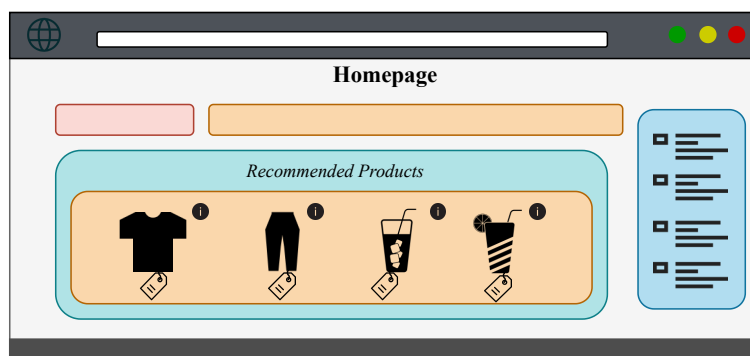


Figure 4.10: Homepage recommendations vitrine generic template. In this example, four recommended products are displayed to the user: two from 'Clothes' category (shirt and pants) and two from 'Drinks' category (iced tea and smoothie). Products are ordered by score, with the product on the left having the highest score, i.e., is more likely to be bought by the user.

Product Details page also presents a recommended products vitrine, visually similar to the one used in Homepage. However, recommendations are from *similar-products* type, meaning that products that are recommended have similarities to the selected product of the page.



Figure 4.11: Product details page recommendations vitrine generic template. In this example, the user selected a shirt, thus be presented products similar to it in the recommendation vitrine, where the product on the left is the most similar to the selected shirt (ordered by score).

As for suggestions to complete the shopping cart, these are available on a **Side Cart**, accessible at any point during the purchase process on the store. Whenever a client adds or removes a product to their cart, the widget reloads and the respective vitrine is updated displaying the appropriate suggestions according to the current cart content. The platform informs the RS of the current products in the cart, which in turn returns the *complementary products suggestions* required by the widget. Furthermore, whenever a client has no products added in their shopping cart, popularity/hybrid recommendations are used instead of complementary products recommendations, due to being impossible to make cart suggestions with an empty cart.

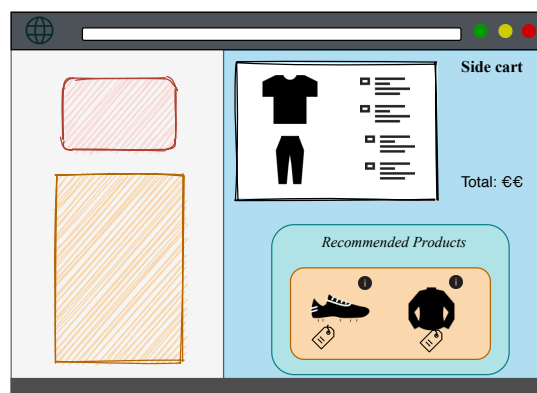


Figure 4.12: Side cart recommendations vitrine generic template. In this example, the user added a shirt and pants to the shopping cart, so the recommendation vitrine displays some products which are commonly bought together with the cart's current content.

Lastly, a new ordering option was added to **Product Listing** on the websites: *"Order by Recommended"*. With this option, products from product listing are ordered according to recommendations given by the RS. Here, once again, recommendations are from types *hybrid* or *popularity*. In this case, two scenarios had to be considered: a *generic* product listing, when the store page is displayed showing all available products, and a *specific* listing, when the user filters the product by category. In this last situation, it was necessary to ensure that all recommended products belonged to the category selected by the client. Since each recommended product of hybrid and popularity types has its category associated, and taking advantage of MongoDB's capacity to be a very efficient search engine, it was possible to filter the recommendations and send only those related to the category of the page.



Figure 4.13: Product listing page recommendations vitrine generic template. In this example, the user browses for products within the *"Drinks"* category and orders them with the *"Recommended"* option. Ergo, products related to *"Drinks"* are listed, ordered by recommendation score, i.e., the products that would appeal the most according to the user's profile are shown first.

Recommendation widgets use the *Redis* component to cache results received from the RS. By saving recommendations, for a certain time, the application no longer needs to send repeated requests demanding the same recommendations, therefore reducing the number of calls to the server and, consequently, the load on the communication flow, increasing overall system performance. A user's session on a Beevo platform typically takes half an hour, with the possibility of the user returning within the next hour after that visit. Thus, when a widget obtains recommendations from the RS, it saves those results in cache. When the user accesses a page whose context requires recommendations that have been previously loaded, the widgets will fetch the results from cache, allowing the page to load faster and improving the user experience.

The format of recommendation requests is specified in section 5.3.

4.5 Security

In order to assemble the security mechanisms of the system, potential threats were raised first according to the **STRIDE** (74) methodology: *Spoofing, Tampering, Repudiation, Information disclosure, Denial of service* and *Elevation of privilege*. Consequently, it was ensured that the system has properties such as authenticity, integrity, non-repudiability, confidentiality, availability and authorization.

Thus, it was decided to develop an *Access Control List (ACL)*, which dictates a hierarchy of user *roles*, and their respective **permissions**. The ACL establishes the various *policies* of the system, indicating which resources exist, which operations can be done in each of them and who can execute them, thus constituting the RS authorization mechanism.

There are three types of users, i.e. roles, with access to the recommender system:

- **System Admin:** The user is an administrator of the system. The administrator has global access to the recommendation system and is allowed to create different types of users for each platform (Tenant Admin and Tenant User). It also can access and operate on any platform and respective resources.

Each platform is referred as a tenant and has its own Tenant Admins and Tenant Users.

- **Tenant Admin:** The user is associated to a platform and acts as an administrator for the resources of that platform. They cannot make CRUD operations on other platforms' resources, just manage the ones of their own. Tenant Admin users are allowed to create common users (Tenant Users), which are consequently associated with the respective platform.
- **Tenant User:** These are common users that can only make *GET* requests to the resources of the platform with which they are associated. They cannot make CRUD operations in ACL nor access the other platforms.

System users and the ACL are stored and managed in the MongoDB database, supported on the npm ACL package ² - *Node ACL - Access Control Lists for Node*. This module offers a minimalist ACL implementation, providing methods that help to create roles and permissions. The following table describes the system policies:

²<https://www.npmjs.com/package/acl>

Table 4.4: RS Access Control List: existing roles, resources and permissions. As it can be observed, Tenant Users are not allowed to access Users and ACL resources. On the other hand, Tenant Admins are not allowed to edit system users information nor create or remove ACL elements (roles, resources and permissions). The System Admins are able to see the roles and permissions of all users. They have full access to the ACL, which allows them to manage all system's policies.

<i>Resources/Roles</i>	System Admin	Tenant Admin	Tenant User
/recommender	*	GET/POST PUT/DELETE	-
/users	*	GET/POST DELETE	-
/acl	*	GET PUT	-
/clients	*	GET/POST PUT/DELETE	GET
/products	*	GET/POST PUT/DELETE	GET
/order-items	*	GET/POST PUT/DELETE	GET
/hybrid	*	GET/POST PUT/DELETE	GET
/popularity	*	GET/POST PUT/DELETE	GET
/similar-products	*	GET/POST PUT/DELETE	GET
/complementary-products	*	GET/POST PUT/DELETE	GET

System Admin operates at the `/api/resource` level, where the platform is not specified and the default is assumed. *Tenant Admin* and *Tenant User* operate with endpoints in the `/api/{platform}/resource` format, where it is necessary to specify the platform to which they are trying to access, having only access to the one they are associated with. The existing resources are listed in the API Documentation section. Note that only System and Tenant administrators are allowed to access platform engine recommendations configurations (`/recommender`), as Tenant Users are not.

In practice, this table is translated into JSON documents and later stored in the MongoDB collections, thus dictating which operations are allowed for each role to perform on each system resource - *policies*. All system users are stored in a single collection, with a *composite primary key* formed by the user's email and platform. Thus, a user can associate their email on more than one platform with different roles.

Taking into account the user hierarchy, we can draw the following process for the creation of users:

1. *System Admin* is registered directly in the database;
2. *System Admin* logs in to the service and registers another user with one of the remaining roles;
3. *Tenant Admin* logs in and has the possibility to register a *Tenant User*.

To develop the login feature, an authentication mechanism was created using JWT (*JSON Web Tokens* (57)) to transmit information in a secure way between entities as a JSON object, since it can be verified and trusted because it is digitally signed. This was implemented through the npm *jsonwebtoken*³ package to exchange information about users in requests between the system and tenants.

Whenever a tenant logs in, and after verifying the login credentials (email and password), the system creates an authentication token to return as the response. In the token's payload, it is stored the user's identifier, their role and platform. At the end of this process, that token is signed with the server's secret key and encrypted. To increase security, the token is defined with a 2 hours expiration time, preventing the abuse of its use indefinitely in case a third party can access a valid token. The properties of authentication, integrity, non-repudiation and confidentiality are therefore maintained.

Both authentication and authorization mechanisms were incorporated into middlewares installed on the system's server to filter requests received from tenants. Access to platforms, according to roles, is validated in the authentication phase, while access to the respective resources is verified in the authorization phase.

The **authentication** middleware sets the system endpoints as private, so that only authenticated users can reach them. It verifies if a user has access to an endpoint, by validating the token received from requests. The middleware fetches the authorization token from request's cookies or authorization header, verifies and decodes it according to the server's secret key and, after this validation, it stores the user data from the token into a session, to pass it to the next middleware (authorization).

In turn, the **authorization** middleware protects the resources of the system, by checking the user role permissions; If the user has the *System Admin* role, then they have access to all resources of the system. Otherwise, they only have access to the resources of the platform they are associated with. It takes the user identifier from the session and checks their permissions according to the ACL, by taking into account the resource they are trying to access and the method of the request. This way, authorization property is established.

³<https://www.npmjs.com/package/jsonwebtoken>

As for the rest of the architecture's components, they already have security tools integrated, thus being sufficient creating users with secure login credentials for direct access to these elements: Elasticsearch, Kibana and MongoDB. As for the availability property, this is also ensured by the existing mechanisms of high availability of these databases.

Furthermore, as an extra security measure, alternative TCP ports were opened, instead of the default ports, for the system's docker containers on the host machine in order to make the resources available through the Recommender API in a more secure way.

4.6 Summary

Having identified which technologies to use, which best fit the context of the project and requirements of the host machine, we proceeded to the planning and design of the types and structure of product recommendations. Therefore, four types of recommendation were defined - *Popularity*, *Hybrid*, *Similar-products* and *Complementary-products* - with three entities being assumed throughout the recommendation system: *Client*, *Product* and *Order-item*. These recommendations will be produced in *offline* mode, which means that the recommendation engine will generate results, in the background and in cycles, always keeping recommendations relatively up-to-date and available for online stores. To communicate with the recommendation service, Beevo online stores integrate an application, called *BI App*, which connects to the RS API and send the necessary information so that the engine can produce recommendations, as well as receive these results to show products to clients. During the project, four scenarios were contemplated to show recommendations on the web pages: *Homepage* with popular products, *Product Listing* with a new 'Recommended' ordering option, *Product Details* page showing products similar to the selected one and *Side cart* suggestions that displays products suited to complete the customer's purchase. The *BI App* also allows customizing the recommendation models of the system's engine, while all this communication process is ensured by the authentication and authorization mechanisms of the service.

5. METHODS

The process of recommending can be understood as a *Data Mining* (75) process, in which correlations between clients and products are discovered, supporting the recommendation of an item to a user. Although this process is of recommendation engine's responsibility, it is important to contemplate the first phase of Data Mining, which consists of *Data Analysis and Preprocessing*: obtaining data, processing it and forming data sets that will be the source of information for engine consumption. As mentioned, obtaining the necessary data for recommendations is of most importance, thus the architectural elements being crucial for the existence of communication. However, not all data can be relevant, so there is a need for filtering to get only significant information, discarding those considered outliers and that can negatively influence the results.

The flow of retrieving data from the platforms, filtering and subsequent storage in the databases can be described based on the ETL (71) process (*Extraction, Transformation and Load*).

5.1 Data Extraction Strategy

Before deciding that data would be received via API, the possibilities of adopting other approaches were considered for the first step - *Extraction* - such as using *Logstash* or *Debezium* for CDC. In the next sections, the conclusions drawn after exploring these alternatives are explained, as well as the decisions that led to adopting an Application Programming Interface.

5.1.1 Logstash

Like Kibana, Logstash (76) is another tool from the vast arsenal developed by Elastic, which serves to complement the use of the Elasticsearch database.

Logstash is a server-side, lightweight and open-source data processing pipeline that allows to collect

data from different sources, transform it in real-time and insert it to any database, in the desired format. Due to its close integration with Elasticsearch, powerful log processing features and multiple plug-ins that can help an application to adapt to most data sources, we tried to use Logstash as a channel for data linking e-commerce platforms to Elasticsearch, rather than using an API.

Logstash has a wide range of plugins, which allows an application to consume different types of database, providing great versatility in adapting and extracting information. The streams used in Logstash are better than conventional ETL processes, because they can extract data from multiple different sources simultaneously, faster and efficiently. Instead of this process being divided into 3 phases (extraction, transformation and load), the tool's streams and filters can execute everything in one step and on-the-fly.

All of these features have great potential for the construction of generic architectures. However, to be possible to extract data from Logstash, there must be at least one of the following two cases: either e-commerce platforms grant access credentials to the system, specifically Logstash, to reach its database or provide an API for Logstash to request data through HTTP requests. Furthermore, plugins would have to be configured according to the different types of databases from inside and outside Beevo's domain and this strategy could lead to some security breaches. Since both scenarios imply dependencies and developments on the part of the platforms and do not present a generic way of obtaining data, this technique was abandoned.

Besides, this tool was developed not to receive data, but to collect it within a certain time interval, constantly monitoring the target databases in cycles. However, considering the event-driven side of our architecture, it was declared that it would be more efficient to build a structure that listens to requests, receiving and processing changes only when they occur (triggers). Additionally, in each cycle Logstash always runs through all existing records rather than only new records, making the process more costly and time-consuming. Hence the choice of using an API, in a more economical perspective in terms of computational process and resources consumption.

5.1.2 Debezium (Change Data Capture)

While continuing to deepen the hypothesis of applying streams in the data extraction process, the implementation of Debezium (77) tool was studied. This tool, sponsored by Red Hat (78), is a distributed platform capable of transforming platforms' databases into event streams, allowing the recommender to immediately detect and respond to changes in each entry in the database tables. Debezium is built on Apache Kafka (63) and provides Kafka Connect compatible connectors that monitor database management systems. In addition, it records the history of data changes in the Kafka logs, from where the application consumes them. This makes it possible for the RS to easily consume all events correctly and completely. Even if there is a problem causing the system to crash, when restarted it will begin to consume the events from where it left off, without losing any piece of

information.

Hence, Debezium is basically a modern, open-source *Change Data Capture* platform that will eventually support the monitoring of a variety of database systems. Change Data Capture, or CDC, is an older term for a system that monitors and captures changes in data so that other software can respond to those changes.

Through this tool, it would be possible to cover various types of data sources, as it can be implemented for most e-commerce platforms. Yet, with the condition that a user needs to be created with access to the database with certain privileges and that the database must be enabled for *bin-log*. These limitations, coupled with the fact that this tool is still at an early stage of development, determined that it was too risky to design an architecture based on it. With Debezium, e-commerce platforms would have the responsibility to create specific tables with the necessary information for RS consumption, which implies altering the infrastructure and schema of the existing database.

With this option, Debezium would be used for data extraction, but the communication and return of recommendations to the tenants would still have to be done through the API. Thus, it was decided to focus the entire communication process on a single element.

5.2 Exploratory Data Analysis (EDA)

As mentioned throughout this document, the quality of the data used will be reflected in the accuracy of the recommendations produced, which reinforces the aphorism of this dissertation - *information is power*.

In order to obtain quality data of the entities considered (Clients, Products and Order-items), there must be a refinement process, resulting from previous analysis and filtering. However, this process must be delicate so that the filtering rules don't compromise the amount of data extracted: cleaning data can cause the removal of data that appears useless and, consequently, losing potential information. On the other hand, a large amount of data can lead to the existence of wide variations due to unusual values (outliers) and this can prejudice the final results. That is why it is important to do exploration and analysis of the data, as the first step, to ensure a balance between quality and quantity.

According to the creator of this process, John W. Tukey, he describes his perspective of EDA, in the book *Exploratory Data Analysis (EDA)* launched in 1977, combining statistical reasoning with the processes of data transformation and exploration:

"Exploratory data analysis can never be the whole story, but nothing else can serve as the foundation stone." - John W. Tukey (79)

Lyle V. Jones summarizes the EDA process in three steps, in his work titled *The Collected Works of John W. Tukey: Philosophy and Principles of Data Analysis (1965-1986)*:

”Three of the main strategies of data analysis are:

1. graphical presentation.
2. provision of flexibility in viewpoint and in facilities
3. intensive search for parsimony and simplicity.”

- Jones, 1986, Vol. IV, p. 558 (80)

The recommendation service offers the possibility not only to developers but also to tenants themselves to carry out the Exploratory Data Analysis process through its Kibana component.

Kibana provides several tools to explore and learn about data stored in Elasticsearch indexes, the construction of graphical representations and other forms of analysis.

One of these tools is *Data Visualizer*, which belongs to the *Elastic Machine Learning* area and lets users view data and gain an understanding of the metrics and fields associated with the respective indices. With this tool, it is possible to take a sample of data (or even the entire data set) and build basic views for all fields according to an index pattern. This area also features other tools such as the *Anomaly Explorer* which by applying unsupervised machine learning helps to find patterns in the received data, stored in the Elasticsearch database. By using time series modelling, it is possible to detect anomalies in current data and predict trends based on historical data, allowing for quick and effective corrections when an error occurs, keeping the database always valid. The *Analytics* functionality enables users to build data analysis and outlier detection structures.

Additionally, Kibana’s *Discover* interface is the tool traditionally used for data exploration. This interface also provides several ways to learn about data, sorting all documents of an index according to a pre-defined timestamp. This facilitates the visualization and analysis of the structure and content of each document, in a simple way, in addition to various filtering options or even the execution of KQL (Kibana Query Language) or Lucene search queries.

Finally, another useful section of Kibana is the *Visualize* area, which supports the development of graphical representations of various types from the data in Elasticsearch, in order to obtain new knowledge and to translate the raw data into relevant information, in a visual manner. Afterwards, the graphics created can be grouped into dashboards, giving valuable Business Intelligence insights to users.

It appears, therefore, that through Kibana it is possible to fulfil all the steps mentioned by Jones, in the previous quote. A practical example of applying this process through Kibana is described in Kibana Data Analysis section.

5.2.1 Variables Selection and Filtering

Following the concept of product recommendations and the context in which the system was developed, there was a need to consider the extraction of data that is suitable not only for the formation of the said recommendations but also for other applications such as the Business Intelligence processes mentioned: analyzes, dashboards, etc.

The selection of variables to be extracted was developed around the notion of e-commerce, in which these are typically used in this type of commerce and, therefore, common among platforms of this area. Thus, the base variables for the formation of recommendations in this system were defined, since all e-commerce platforms are usually able to obtain and send them to the RS. Evidently, different additional variables can be used depending on the platform, as is the case with product attributes, e.g., a store that sells clothing will have different attributes from a store that sells food. This is possible given the system's generic and flexible structure and the dynamic mapping of the databases, as already discussed.

Since each tenant is responsible for sending data to the system, the basic and obligatory variables for the proper functioning of the service must be tangible to all and available on the platforms.

In order to define and filter the fields to be used by each Entity, a pre-selection took place, to choose the variables that presented the minimum of potential to form recommendations but taking care not to discard data that could offer unknown knowledge and important information.

The following mappings describe which variables are considered for each Entity and that must be sent by default to the system, in order to be consumed by the engine and used by the recommendation models:

Table 5.1: Fields selection for Client entity

Field	Description
<i>client_id</i>	Client identifier
<i>gender</i>	Client gender
<i>birth_date</i>	Client birthdate
<i>created_at</i>	Record creation date
<i>updated_at</i>	Last record update date
<i>isNew</i>	Indicates if client is new to the online store
<i>country</i>	Client country prefix (ISO 3166-1 alpha-2 code)
<i>address_country</i>	Shipping address country prefix (ISO 3166-1 alpha-2 code)
<i>address_1</i>	Shipping address
<i>address_2</i>	Optional second shipping address
<i>locality</i>	Shipping locality
<i>city</i>	Shipping city
<i>zip</i>	Shipping zip
<i>district</i>	Shipping district

Table 5.2: Fields selection for Product entity

Field	Description
<i>product_id</i>	Product identifier
<i>parent_id</i>	Product parent identifier
<i>name</i>	Product name
<i>sku</i>	Product sku code
<i>aggregator_id</i>	Product aggregator identifier
<i>stock</i>	Product stock amount
<i>ordering</i>	Product ordering for sorting options
<i>status</i>	Product status
<i>buyable</i>	Indicates if the product can be bought
<i>published</i>	Indicates if product should be available in the store
<i>created_at</i>	Record creation date
<i>updated_at</i>	Last record update date
<i>hits</i>	Number of hits, i.e., number of visits to its product detail page
<i>manufacturer</i>	Product manufacturer
<i>parent_published</i>	Product parent 'published' status
<i>categories</i>	List of product categories
<i>has_discount</i>	Indicates if product price has discount
<i>attributes</i>	Product attributes, with key-value format, as in { "attribute_name" : "attribute_value" , ... }

Table 5.3: Fields selection for Order-item entity

Field	Description
<i>order_item_id</i>	Order-item identifier
<i>order_id</i>	Order identifier
<i>product_id</i>	Order-item's product identifier
<i>product_name</i>	Product name
<i>product_category</i>	Product category name
<i>quantity</i>	Product quantity
<i>product_total_with_tax</i>	Price per item, with tax
<i>item_total_with_tax</i>	Sum of prices of total items in order, with tax
<i>status</i>	Order status
<i>created_at</i>	Record creation date
<i>updated_at</i>	Last record update date
<i>client_id</i>	Order's client identifier
<i>ip_address</i>	Order's user ip address
<i>ship_city</i>	Shipping city
<i>ship_locality</i>	Shipping locality
<i>ship_country</i>	Shipping country prefix (ISO 3166-1 alpha-2 code)
<i>manufacturer</i>	Product manufacturer
<i>hits_count</i>	Number of hits on that product by the client
<i>product_attributes</i>	Product attributes, with key-value format, as in { "attribute_name" : "attribute_value" , ... }

In the case of the project's company, these fields are defined in the PHP classes of each of the entities of the *BI App*, thus being selected by default from Beevo platforms' databases and sent to the RS. It should be noted that it is possible to define extra fields, to be selected in addition to the default ones, in the parameters of the application's configurations, thus allowing to train the models more specifically for the platform context and thus achieving better results. The rules for variables selection are translated into *data contracts* between the RS and tenants. A practical example is described in section 6.1.1.

As it can be observed in the *Client* fields table, the data relating to the personal area (name, email, phone number, etc.) are not selected, as they are not necessary for the recommendation process. Each client is identified by its `client_id`, promoting information security, as their privacy is protected.

As for the *Product* fields, `product_id` and `parent_id` fields should be mentioned, where the first refers to the id of the product itself and the last to the id of its parent product. These fields were developed due to Beevo's product logic where parent products represent the generic model of a product that can later be extended into several child products, each varying in different attributes such as color or size, but maintaining the general characteristics of the product. This logic is later applied in the engine, training models using detailed information from *child products* and recommending *parent products* in a more generic way. However, platforms that do not possess the `parent_id` field may ignore it, since the engine will then adapt to each tenant's conditions. Moreover, note the statuses `buyable` and `published` in product extraction; there was the possibility of accepting only those products that were available and buyable in the online store, since only those were relevant to recommend. It wouldn't make sense to recommend a product that is not in these conditions. However, when discarding this data, one could be losing knowledge, which is why it was decided to maintain a more profitable Data Mining process, discovering new information and helping the recommendation algorithms to create connections and correlations between the characteristics of various products, thus increasing the accuracy of results. Subsequently, it is the engine's responsibility to manage this logic and only recommend valid products.

The `attributes` field was design to create a generic structure in which each tenant could fill in their data in a simple way: an object in a key-value format, e.g., `{"Color": "Red", "Size": "XL"}`, which can be satisfied with any type of attributes, regardless of the type of product considered.

The API is only responsible for receiving the necessary data for engine consumption, i.e., all data received is used in the recommendation process, without "waste" (excessive data). Null fields are not saved, optimizing storage in Elasticsearch, but empty data is. It is up to the tenant to respect the rules of the data contract, yet, not all of this data may come standardized (some may be empty or in invalid formats) so, if necessary, the data transformation process can occur on the engine side according to its needs and the context of the recommendation models. The engine has the ability to

process data from Elasticsearch, transform it and make it valid for application in its ML algorithms. One can say, therefore, that this phase resembles to the *Transformation* and *Load* stages of the ETL process.

5.3 API Documentation

The documentation of a RESTful API is a controversial subject among current software developers, according to the article *Best Practices for the Design of RESTful Web Services* (17):

"A documentation for Web APIs is a debatable topic in the context of RESTful web services since it represents an out-of-band information, which should be prevented according to Fielding: "Any effort spent describing what method to use on what URIs of interest should be entirely defined within the scope of the processing rules for a media type""

The description of a REST API violates two of the mentioned constraints of REST in section 2.1: messages must be self-describing and hypermedia must be the engine of application state (55). However, it is justified that although this API is not full-REST, it was created *based* on the REST principles, since it belongs to a service to be used by developers and not normal users, hence its documentation being necessary.

Swagger (81) framework was used to compose the API documentation, for description, consumption and visualization of the service. This tool allows the documentation to evolve at the same rate as the implementation since it can be generated automatically based on annotations of the code. Documentation is generated according to the standard, programming language-agnostic interface description for REST APIs defined by the *OpenAPI Specification (OAS)* (82) which allows humans and machines to understand the capabilities of a service without requiring access to source code, additional documentation, or inspection of network traffic.

The API documentation describes, therefore, all the existing resources in the system, established endpoints, possible methods of making requests for each corresponding route and which parameters are necessary for these requests to be successful.

The documentation composition can be described by dividing it into the following schemas:

- **Client, Products and Order-Items:** These schemas describe the parameters needed to insert a record of one of these entities, following the data contract established. To insert, obtain, replace or delete a record, a request must be made using the HTTP methods POST, GET, PUT and DELETE, respectively, making it possible to make CRUD operations under the system database.

- **Recommender:** This schema represents the recommendation engine configurations, described in table 4.3, providing several endpoints to obtain and update these parameters in order to customize the engine recommendations.
- **User:** Describes the endpoints and respective operations that can be performed under the system's User model, such as registration, login and various CRUD operations.
- **Recommendation:** It is an abstract schema, used for the different types of recommendations. Each type has a different endpoint associated, thus separating the multiple resources related to each recommendation.
- **ACL:** Describes all endpoints that allow the configuration of the system's Access Control List and respective policies, managing the existing roles, their assignment to users and permissions of each role.

This documentation was shared among all RS and platforms' developers, so that teams were aware of the system's general functioning and how to communicate with it to develop certain features. In appendix A.2 it is possible to visualize the appearance of the documentation made with Swagger. It should be noted that the technical description of the system was also published in the company's documentation.

5.4 Summary

In this chapter, different options for data extraction were addressed, in order to populate the recommendation system. This is an important phase since, without quality control of these data, it would be improbable to form good recommendations by the engine. It was decided that the best approach would be the API, instead of extracting data in cycles or using change data capture techniques, as it would be the most efficient way of the service to react to platforms' events and consequent changes in the databases. Before developing the extraction process, a pre-selection took place to elect the fields that would be used in the formation of recommendations. The standard fields to be extracted were defined, as they are common to most e-commerce platforms, leaving space for additional fields that can be used to personalize the recommendation models and obtain more accurate results, in line with the needs of the online store. Finally, the API documentation was made available to all developers, spreading the service's functioning mode throughout the company.

6. CASE STUDIES / EXPERIMENTS

In order to prove this project's concept, the recommendation system was applied in practice on Deeply's online store, a platform developed by the company. The RS was implemented in a *staging* environment of the platform to carry out several tests and evaluations so its development and deployment in *production* environments can be justified, in the future.

In this chapter, a practical experience made on Deeply e-commerce platform is described, exploring the extracted data from the store and multiple analysis techniques possible with Kibana, as well as the widgets to display recommended products to clients.

Finally, the obtained results are explained and discussed, reflecting the overall performance of the system according to the architectural component and highlighting the potential of this data in the Business Intelligence area.

6.1 Experiment setup

Deeply is an online store that sells clothing collections and surf equipment. The steps to install the recommendation system on an e-commerce platform can be summarized as follows:

1. Register the tenant in the recommendation service through the web pages directly provided by the system. A login page is presented, which after authenticating, redirects the user to a registration page - to create new users in the system. Only *System Admins* and *Tenant Admins* can authenticate themselves on the login page, as they are the only ones allowed to create other users. Note that when a *System Admin* creates a user, he can indicate the platform and role (*Tenant Admin* or *Tenant User*) that he wants to associate with the new user, while a *Tenant Admin* can only create new *Tenant Users* associated with its own platform, as defined by the system's policies - section 4.5. The appearance of these web pages are attached in

appendix A.4.

2. After registration, the tenant will be able to get an authentication token, by sending a request to the system's login endpoint, as described in the API documentation, and insert it in the header of its consequent requests, making them valid and accepted by the service.

```
{  
  "userId": "5e4bf76b5be7e4151aeb99ae",  
  "userRole": "SystemAdmin",  
  "userPlatform": "deeply",  
  "iat": 1582036952  
}
```

Listing 1: Example of a JWT decoded JSON object

In the case of Beevo's online stores, this is possible through the platform's *Back-Office* (management area for store administrators), in a specific section for the interaction with the RS, through the *BI App*. This section presents a tab to enter the tenant's credentials that will be used to authenticate in the recommendation service.

BI User Login Configurations Populate BBIS

BI User Association

Please, Insert the login credentials for an existing BBIS account. The respective BI user will be associated and used to access the BBIS API.
If you do not have an account, please contact the BBIS developers to grant you access to the service.

Email address	Password	Platform name
test@email.com	*****	deeply

Associate Cancel

Figure 6.1: Deeply Back-Office - RS user association to tenant

3. Afterwards, it is necessary to configure the recommendation engine, again through the *BI App*, in the section entitled for that. Here, all *model* parameters mentioned in section 4.3 can be changed and adapted with the desired behavior for the engine.

BI User Login Configurations Populate BBIS

Recommendation Model Configurations

On this form, recommendation model parameters can be configured to adapt the behavior and results (recommendations) of the BI engine.

Minimum score

Minimum support

Last months of orders

Number of clusters

Show 'Recommended' ordering Yes No

Recommend products already bought Yes No

Evaluate model Yes No

Content-based model weight

Collaborative-filtering model weight

Clustering model weight

Figure 6.2: Deeply Back-Office - Recommender engine configurations

4. Ultimately, it is necessary to perform an initial population of the Elasticsearch database of the RS, extracting the data previously described from the entities *Client*, *Product* and *Order-item*, so that the engine has the data sets it needs to produce recommendations. The population can be general, cloning information from all entities, or selected ones. In this tab, it is also possible to indicate the optional fields to be extracted from those entities, in addition to the fields selected by default, or even additional attributes for the *Product* entity - section 5.2.1.

BI User Login Configurations Populate BBIS

(optional) Attributes Selection by BI App

On this form, optional fields can be added to be selected when BI App extracts information from Beevo's database.

PRODUCT ATTRIBUTES	PRODUCT FIELDS	CLIENT FIELDS	CLIENT'S ADDRESS FIELDS	ORDER-ITEM FIELDS
<input type="checkbox"/> size	<input type="checkbox"/> description	<input type="checkbox"/> name	<input type="checkbox"/> phone_1	<input type="checkbox"/> vendor_id
	<input type="checkbox"/> aggregator_id			

BI Initial Population

The BI Application transfers data from Beevo databases to the BI System. It allows an initial population of the BI's database and sending further events generated by consequent activities in Beevo's platforms.

At the first stage, the BI App must do an **initial population** of the BI's database, by fetching current data from Beevo's DB, filtering it by selecting only the relevant fields for recommendations and send it to the BI's API, which in turn will handle and store this information.

BI Entity Population

It is also possible to populate entities individually, by selecting only those to be sent to BBIS.

Select entities to populate:

Client Orderitem Product

Figure 6.3: Deeply Back-Office - Attribute selection and Recommender Population

- At the time of execution defined by a *cron job*, the engine will load, process and analyze data from these data sets, train recommendation models according to previously defined configurations and generate recommendations for Deeply's clients, storing those results in MongoDB at the end of the process. These results will then be available via API.

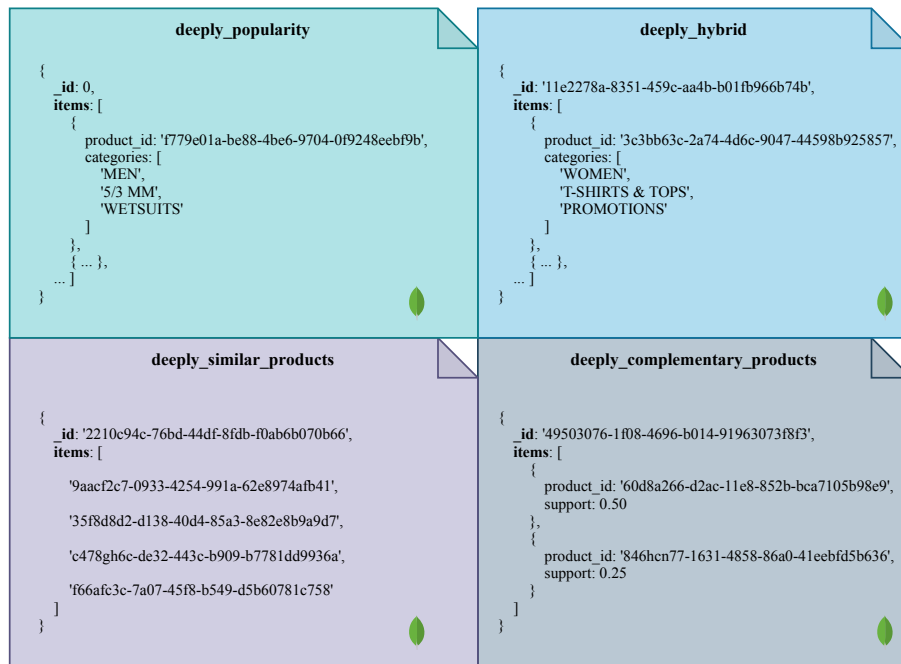


Figure 6.4: Deeply MongoDB recommendation documents examples

Once the recommendation process has taken place, it is time to show the results to the store's clients, through the storefront web pages. As described in section 4.4.4, the recommended products are displayed in vitrines, coordinated by widgets developed for this purpose, according to the type of recommendation that is intended to be used on the web page. When a client visits the web page, widgets request recommendations to the service, using the methods available in *Facade.php* of the *BI App*, and after receiving the list of recommended product ids, it fetches the information of those items in the store's database and presents them to the client. These results are cached for about 2 hours, so when the client visits the same page it is not necessary to request the same results to the service again and avoid a great page loading time and resource usage. Below are some examples of these vitrines:

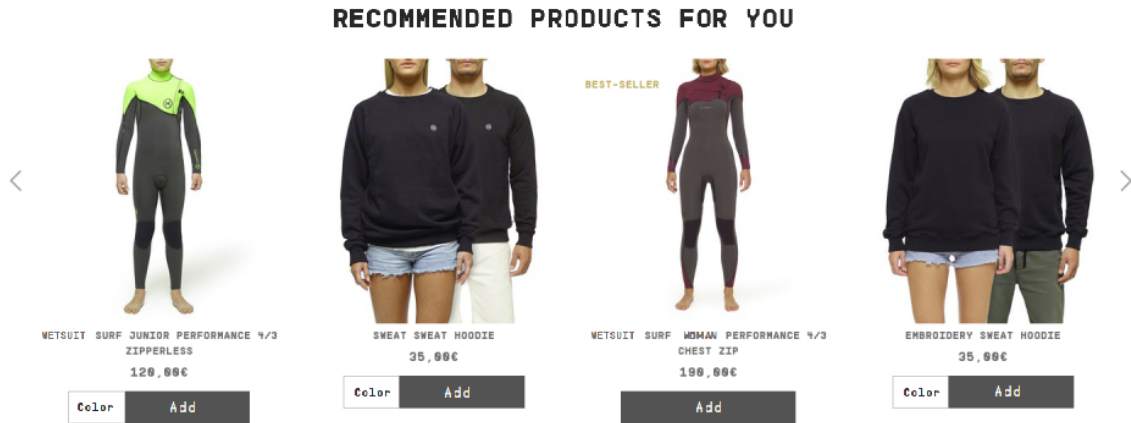


Figure 6.5: Deeply Homepage - Popularity recommendations example

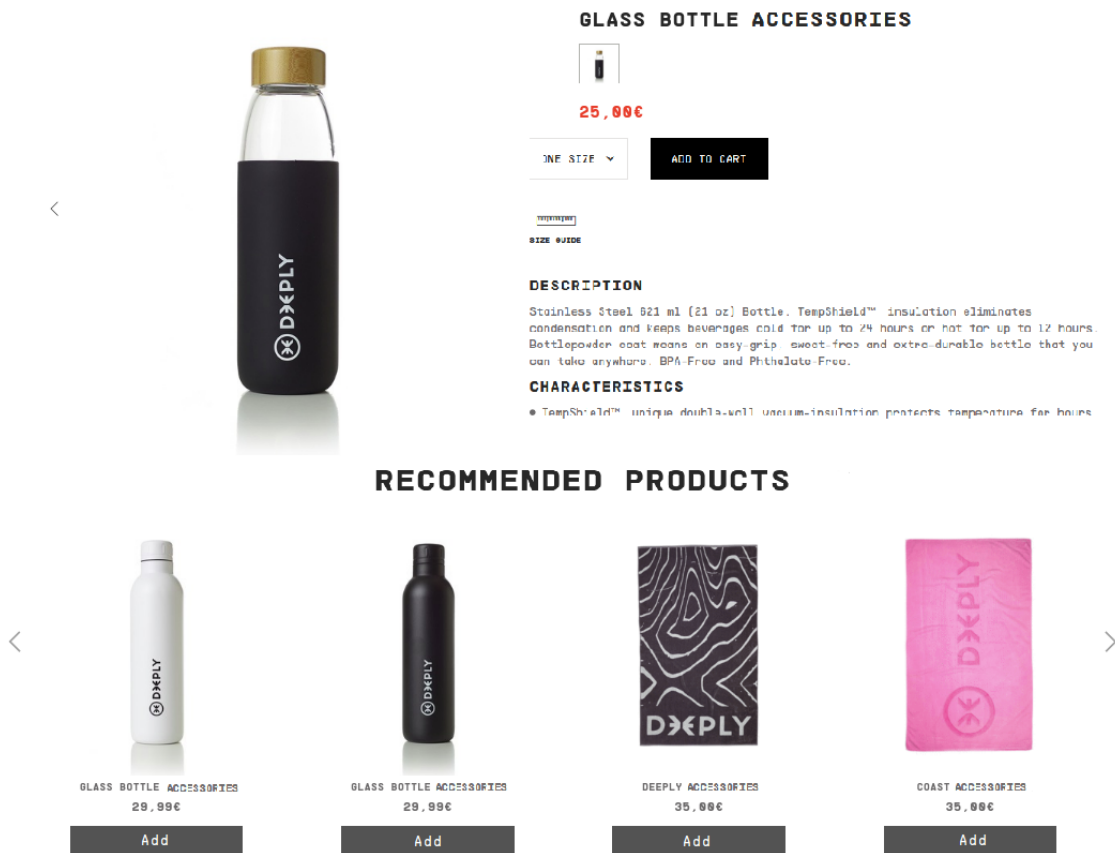


Figure 6.6: Deeply Product Details - Similar-products recommendations example

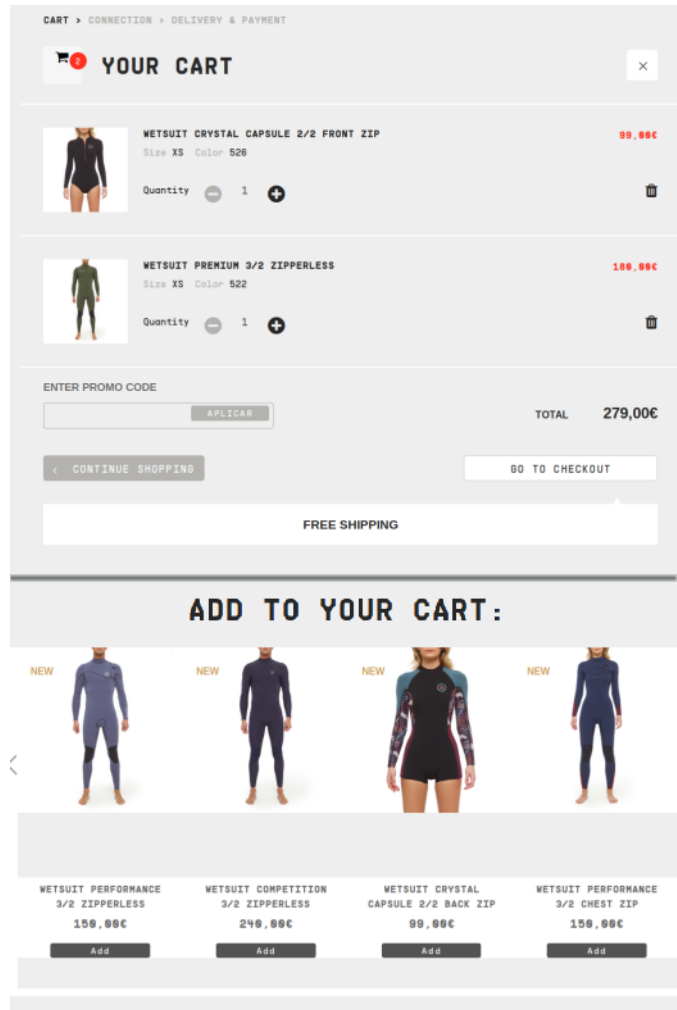


Figure 6.7: Deeply Side-Cart - Complementary-products suggestions example

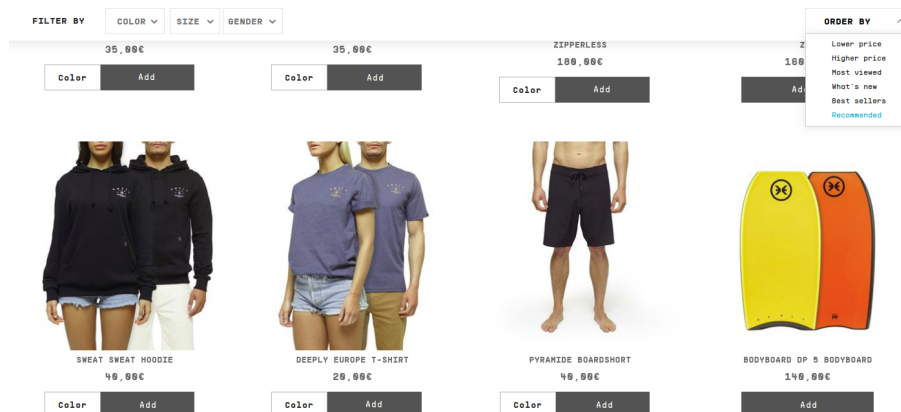


Figure 6.8: Deeply Product Listing - Hybrid recommendations example

6.1.1 Data Contract

To ensure the steady functioning of the system, it is necessary to ensure good communication between it and the platforms. This is achieved through a *data contract*, where it is agreed between tenants and the service which and in what format data should be exchanged between them, guaranteeing that the system receives the necessary information to produce recommendations and that tenants can obtain and use these results. This is a way to "oblige" both parties to maintain a coherent and flawless communication.

The data restrained in the contract reflects the variables pre-selection made in section 5.2.1, considering that, when selecting and filtering data in the extraction process, all *string* variables are in the same language, as this will affect recommendations made by the engine since some recommendation algorithms are based on textual terms.

The data contract between the recommendation service and Deeply is available in appendix A.6.

6.1.2 Kibana Data Analysis

With the Elasticsearch database filled in, Kibana can be used to explore its content. Elasticsearch has two core data types that can store string data: *text* and *keyword*. *Text* data type is useful when it comes to product descriptions: if a product description has "t-shirt made of 100% cotton" and the user searches for the string "*cotton*", that product will appear as a result. On the other hand, the data type *keyword* is used for exact matches, where the results must precisely match the search terms: in the case of a product being a "blue t-shirt", if the user searches only for the "*t-shirt*" keyword it is likely that this product will not be returned as a result of that search. This is what makes Elasticsearch an excellent full-text search engine, a feature that may complement this recommendation system in the future.

Through the *Data Visualizer* area we can obtain statistical information regarding data extracted from Deeply. Deeply platform has a *Portuguese* database, hence some data values can be presented in that language as they do not have an English translation stored. Appendices A.8, A.10 and A.12 show the information that Kibana offers from Client, Product and Order-item entities's data, respectively¹. Each document corresponds to an entity record.

¹The Deeply database used is deprecated and was only used for testing purposes, thus some data may have poor values or even the lack of them. Nonetheless, the engine adapts the recommendation models according to the needs of the online platform.

Clients

Regarding the *Client* entity, it was possible to verify that there are 23,889 documents stored in Elasticsearch, which means that there was the same total of client records on the platform, at the time. Only 5,265 documents (22.04% of clients) contain the **gender** field, while in other records this field remains null. These values are in Portuguese, so it can be assumed that *sr/dr* values refer to the *male* gender and *sra/dra* to the *female* gender:

Table 6.1: Deeply clients' gender distribution

Male	sr	62.1%	65.4%
	dr	3.3%	
Female	sra	23.2%	23.7%
	dra	0.6%	
Undefined	<i>(empty)</i>	10.9%	10.9%

About 14.761 (61.79%) of clients have their **country** associated with their record, allowing to understand in which countries the store has more affluence:

Table 6.2: Deeply top 5 country values

Country (ISO 3166)	Percentage
PT	55.6%
ES	34%
FR	5.7%
DE	1.5%
IT	0.9%

From the translated values in the tables, we can conclude that most of the store's clients are Portuguese and Spanish men.

Note also that several values are not normalized. For example, the **locality** field is filled in manually by the client at the registration page, so different clients, even belonging to the same locality, can type its designation differently.

Due to this type of flaws and inconsistencies, it is necessary that the engine executes a *data treatment* before model training and production of recommendations.

Products

In Deeply platform there are 8,860 products in total, so far, but only 22.1% of them are available (**published**) in the online store. When analyzing the data through Kibana, we can see the following top of values of the **categories** field, **color** and **size** attributes:

Table 6.3: Top 5 values of colors, sizes and categories of Deeply products

Color	Percentage	Size	Percentage	Categories	Percentage
526	30.5%	L	16.9%	PROMOTIONS	13.3%
504	10.5%	S	15.5%	MEN	12.8%
pink	7.2%	M	15.3%	CLOTHING	6.8%
508	6.7%	XL	11.2%	WOMEN	6.7%
510	5.6%	XS	8.9%	JUNIOR	4.3%

Exposing the data in this table, one can easily notice that the store produces mostly black products (color 526) for men between sizes from S to L (agreeing with the previous data analysis), and many of them were in promotions at the time this analysis was made.

Through this examination, it is also possible to check the different **manufacturers** associated with the various products of the store, however in this experimental case there is only one: Deeply.

Finally, the **parent_id** field stands out, in which 29.8% has a value of 0, indicating the number of parent products in the store, that is, the number of products available for recommendation - section 5.2.1.

Order-Item

As for order-items, there are a total of 37,148 records, but the number of orders placed is obviously smaller, since several order-items can belong to the same order. Hence, it is necessary to have this notion into consideration when analyzing order data. In this case, the use of the aggregation functionality of Kibana was decisive to assist in the analysis of orders, exploited in Business Intelligence Dashboards.

However, in the analysis of the individual order-item instances, one can realise that the median price for each product in an order-item (**product_total_with_tax**) is about 98.75, which is similar to the total price of the products in an order (**item_total_with_tax**), justified by the fact that the **quantity** of products in an order-item record being 1, in about 98.7% of cases.

With this data, we can already obtain information on the **color**, **size** and **category** fields, that are present in most of the ordered products:

Table 6.4: Top 5 values of colors, sizes and categories of ordered products

Color	Percentage	Size	Percentage	Categories	Percentage
526	47.8%	M	25%	PROMOTIONS	31.2%
grey	8.1%	L	14.4%	MEN	21.4%
504	7.2%	S	14.3%	WETSUITS	4%
dark grey	3.8%	MT	8%	4/3 MM	3.8%
508	3.7%	MS	7.3%	ACCESSORIES	3.2%

Translating these values into a table for better visualization, it is easy to understand that most of the products sold are products whose color is black (526 code) for men, with the size between S and L, belonging to the promotions group. Thus, there is an evident correlation with the previous table, being logical that the store invests more in the stock of products that are most sold. About 42.4% of order-items belong to orders addressed to Portugal, 36% to Spain and 12.3% to France. In total, 61.2% were shipped and 28.6% cancelled (**status**).

As can be seen from the examples of the values that Kibana presents for the fields mentioned above, these are not normalized (in the same format), meeting the issue highlighted throughout this section: the recommendation engine is responsible for handling data in such a way that it satisfies the requirements for the application of the ML algorithms and consequent production of recommendations. Besides, as the engine recommends for textual terms, it does not matter whether the fields' values (such as colors or sizes) are represented by their number or name, as long as consistency is maintained. It is this capacity for abstraction that makes the system so flexible.

Furthermore, the information presented in *Data Visualizer* is updated in real-time, as Elasticsearch receives new data.

From a Business Intelligence perspective, the variables previously discussed are quite interesting to analyze in order to obtain relevant information for *Targeted Marketing*, for example. This potential is demonstrated in the Business Intelligence Dashboards section.

6.2 Results

After integration and configuration phases of the recommendation system on the e-commerce platform, several tests were carried out to assess the overall performance of the architecture idealized in this project. These tests were used to identify possible flaws that may exist either in the general functioning of the service, or in its communication with tenants, to raise performance improvements and, finally, to support the proof of concept so that it can advance to production environments. Thus, in this section, the results of the evaluations regarding the interactions with the recommendation

service are compared, justified and, subsequently, several graphs inherent to the Business Intelligence area are exposed and grouped in dashboards, made with Kibana. Finally, all decisions made throughout the project are reflected in a discussion section describing potential improvements regarding both API and the machine of the recommendation system, its scalability and a global perspective on the entire project.

6.2.1 Performance Tests

To assess the performance of the proposed architecture, *load tests* were developed collecting several evaluation metrics from the server.

These tests were created using *Locust* tool (83), a distributed and scalable load testing framework, which allows writing user test scenarios in Python to test any system.

The general flow for performing a load test consists of three concepts: *Workload*, *System Under Test (SUT)* and *Metrics*.

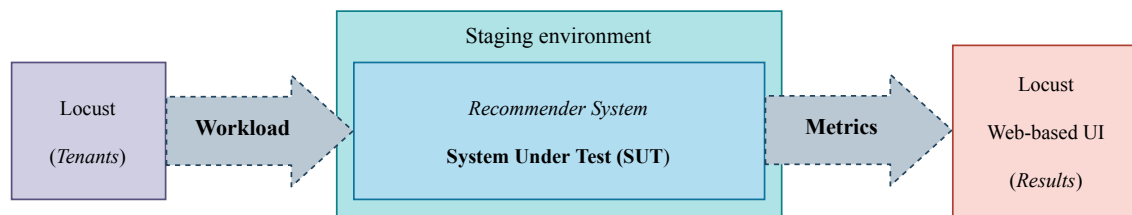


Figure 6.9: Execution of load tests: Workload, System Under Test (SUT) and Metrics.

Locust simulates the interaction of several tenants with the system, i.e., mimics the browsing behavior of multiple clients on the web pages of different e-commerce platforms, triggering events and sending several requests to the recommendation system, pushing its performance to the limit. According to the responses that the RS returns, Locust is able to collect various metrics and make the results available through its web interface, showing the progress of the load tests in real-time.

The workload consisted of requests generated synthetically and randomly, either in time and payload, intending to simulate real users and prevent the software used from caching data, after receiving the same requests in the same order for some time.

About 1500 users were simulated in total, at a rate of 2 users created per second, of which 375 (25%) represent *logged users* and 1125 (75%) correspond to *anonymous users* (who are not authenticated on the store), since users of this last type are more frequent in online stores. Logged users sent requests between 5 to 10 seconds, whereas anonymous users send at a rate of 7 to 14 seconds, this last simulating limited knowledge about the store and a longer search for products.

Table 6.5: Simulated users in the load tests

Total Users	Authenticated Users	Anonymous Users
1500 (100%)	375 (25%)	1125 (75%)
2 Users created/second	Sends request every 5~10 seconds	Sends request every 7~14 seconds

Regarding NodeJS, it is known that this run environment is *single-threaded* by default, using a single core of a machine while others remain idle. However, it is possible to implement a *clustering* module, using **Process Manager 2 (PM2)** (84), which enables an automatic usage of *Node's Cluster API* (85) and a built-in load balancer, giving the application the capacity to run in multiple processes. With this module, the parent process can be forked into several child processes, all sharing server ports and handling a large volume of requests concurrently.

Two different scenarios were tested: the first was a simple environment test, without the installation of PM2, while the second scenario implements this module, using all 8 CPUs of the host machine simultaneously, each raising an instance of the application, allowing a better distribution in the processing of requests.

An important note to take into account is that in these test scenarios, cache on the tenant-side was not considered; in this case, caching results in Redis via *BI App*.

The following table refers to the overall results of the system's performance in the two mentioned scenarios. It should be noted that each load test has a duration of 15 minutes and, in order to collect metrics with reliable values, the server machine's warm-up and cool-down were taken into account.

Table 6.6: System's server load test performance results

	Throughput (requests/second)	Reliability (failures/second)	Median response time (ms)	Average response time (ms)	95 percentile response time² (ms)
Single-core	205.8	5.8	90	230	280
Multi-core (8×CPU)	239.4	6.5	69	107	170

Examining the values in table 6.6, one can verify that the response time is shorter when **clustering** is implemented.

The *average response time* is influenced by momentary peaks due to some connection errors, hence the most reliable value is the *median response time*, presenting a value below 100 ms, which does not produce a significant impact on the loading time of web pages nor in the user experience on the platform. In addition to some connection errors, failures include responses with HTTP

²95% of the requests are served before this time

404 code, regarding requests for recommendations that the system does not possess, due to the recommender engine not having produced results for some products (e.g. some complementary products may not be found for some items of the store).

Furthermore, a clustering approach increases the system capacity to scale, allowing it to serve a greater number of requests while maintaining a reduced response time, making it more robust. However, it is necessary to have a healthy management in the assignment of CPUs to the Node.js component, since a large consumption of resources can affect the performance of the other docker containers. Still, in this case, since Node.js is the most active element in the system, it is the one that is entitled to more dedicated CPUs for its execution.

In this second scenario, a load test was set up in Locust, according to the following structure:

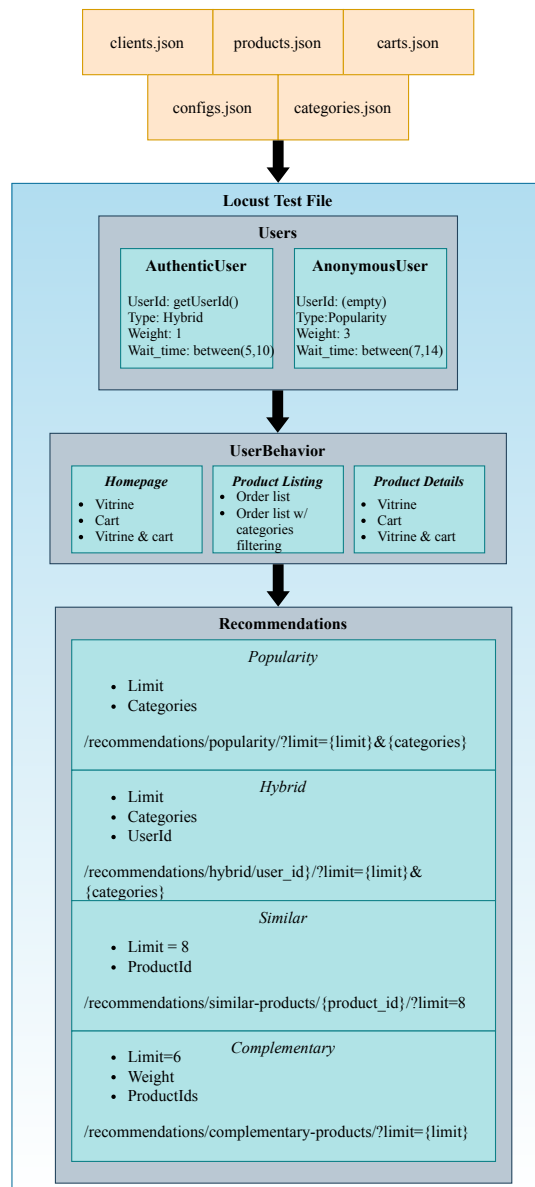


Figure 6.10: *Locust* load test structure.

Several JSON files were created, to load the test program with samples of client and product ids, from Deeply's database, and various configurations related to the recommender's server.

Two different classes of users were created - *authenticated* and *non-authenticated* users - each configured according to the information referred in table 6.5. Each user class is supported by another class, named *UserBehavior*, which defines the behavior of each type of user and simulates various actions as if users were browsing the online store. This class requests recommendations from the RS, choosing randomly several product and client ids in each execution cycle, according to the configurations previously made. Only the scenarios in which users browse web pages covered by the recommendation widgets were considered: Homepage, Product listing, Product Details and Side-Cart.

As for recommendations, each type is specified in the class *Recommendations*, in which *hybrid* and *popularity* recommendations, specific for authenticated and non-authenticated users respectively, receive as arguments the maximum number of recommended items that the service should return, as well as the list of categories to use in filtering results. These arguments are placed in the query parameters of the requests. As for *complementary products*, it is worth mentioning that the *weight* field represents the likelihood of a shopping cart being empty: most users, when browsing an online store, explore a lot without adding a single product to their cart.

The following table presents detailed information of the load test that took place in this second scenario. In addition, it is complemented with images of graphs referring to the total number of users, the rate of requests per second and the response time value, over time:

Table 6.7: Locust table that translates the values of metrics collected during the load test, discriminated by each of the routes/endpoints provided by the Recommender API

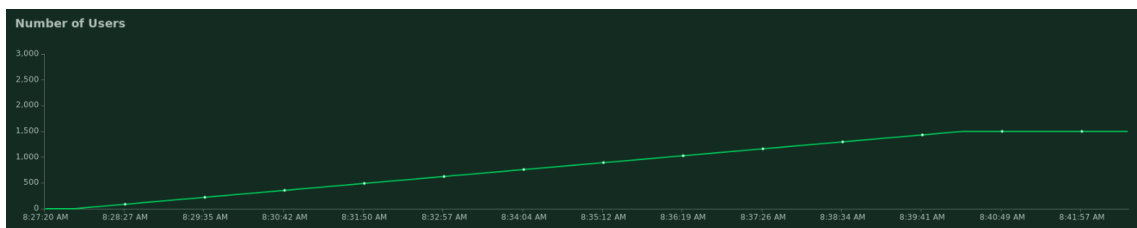
Method	Endpoint	Request Count	Failure Count	Median Response Time (ms)	Average Response Time (ms)	Min Response Time (ms)	Max Response Time (ms)	Average Content Size (bytes)	Requests per second	Failures per second
POST	<i>/api/deeply/recommendations/complementary-products/</i>	9	9	19	38	4	87	0	0.01	0.01
POST	<i>/api/deeply/recommendations/complementary-products/?limit=6</i>	25974	0	73	114	22	16571	220	28.86	0.00
GET	<i>/recommendations/hybrid/clientID/?limit=10</i>	5265	45	60	105	5	16352	390	5.85	0.05
GET	<i>/recommendations/hybrid/clientID/?limit=100</i>	1116	27	70	108	4	16343	3894	1.24	0.03
GET	<i>/recommendations/hybrid/clientID?categories=[categories]/?limit=100</i>	3186	1935	75	119	6	16377	501	3.54	2.15
GET	<i>/recommendations/popularity//?limit=10</i>	18477	0	50	93	21	16483	391	20.53	0.00
GET	<i>/recommendations/popularity//?limit=100</i>	5679	0	70	98	22	16480	3901	6.31	0.00
GET	<i>/recommendations/popularity//?limit=6</i>	71712	63	41	95	4	16529	234	79.68	0.07
GET	<i>/recommendations/popularity/?categories=[categories]/?limit=100</i>	11970	3672	72	115	23	16381	521	13.3	4.08
GET	<i>/recommendations/similar-products/productID/?limit=8</i>	64467	99	69	121	7	16590	252	71.63	0.11
None	Aggregated	207855	5850	69	107	4	17081	429	230.45	6.5

All routes made available by the API to obtain recommendations are listed in the table, each with the respective metrics collected during the test.

When carrying out load tests it was noticed that most of the simulated requests addressed to *popularity* recommendations. This can be justified by the fact that there are more non-authenticated users and the Homepage has a greater number of hits. Thus, and since these recommendations are the same for all users, it was proposed to cache them in the server, using Redis, in an attempt to improve the performance of the system. However, when implementing the Redis component on the server-side, there were no changes in response time. On the other hand, the load that was submitted to MongoDB was reduced, allowing it to have more capacity to respond to the remaining requests. This can be a solution, if there is a large number of requests that require multiple and distinct recommendations and compromise MongoDB's performance. Otherwise, the implementation of a Redis server-side component does not pay off.

As for the other routes, it should be mentioned that the different limits in the query parameters represent distinct vitrines, on different pages and that the first *complementary-products* route refers to an empty cart, hence all requests for this route to fail, as there are no products to recommend (HTTP 404). However, the weight for this action to occur is quite low, so it can be ignored since it is rare to request product suggestions for items that are not present in the cart.

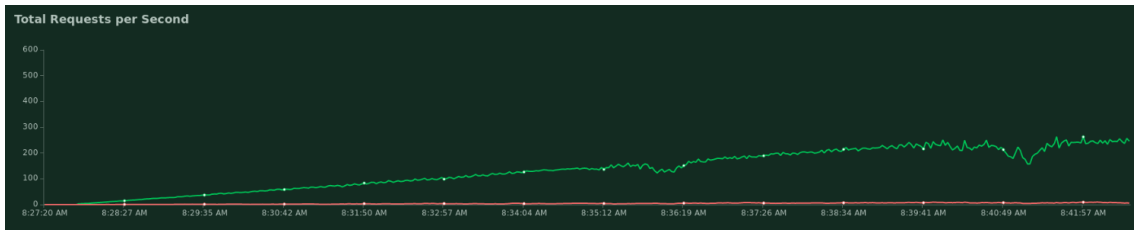
It is also worth mentioning that the response time for the authentication route (login) is about 154 ms, on average, which can be considered quite slow. However, this is due to the authentication password validation process, which uses the *bcrypt*³ library that was designed to be slow in order to avoid brute-force attacks.



Source: Locust web-based UI

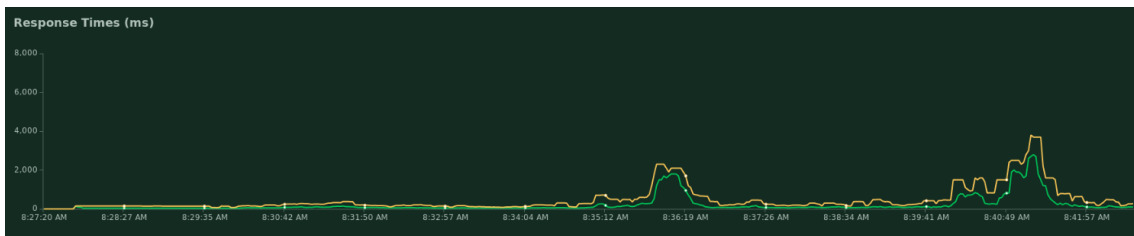
Figure 6.11: Number of Users over time

³<https://www.npmjs.com/package/bcrypt>



Source: Locust web-based UI

Figure 6.12: Rate of total requests per second over time



Source: Locust web-based UI

Figure 6.13: Response time value over time

Comparing the 3 graphs, we can identify that there are some peaks where the response time is higher and, consequently, the number of requests handled is lower, as the number of users increases. This allows to identify several points where bottleneck problems may occur, caused by the immense load made by test users - the limit of connections to the server is sometimes exceeded. This is visible in the table, where a *max response time* of 17 seconds was recorded. In contrast, a 4 ms *min response time* was also recorded, which may have resulted from similar and consecutive requests.

These failures are accounted for in the previous table, adding to the HTTP 404 - Not Found server-side errors when recommendations are requested for which the recommender has no answer: e.g., product listing with a certain combination of categories or complementary products for a certain item.

Obviously, in practical cases, the tolerance to these bottleneck concerns is greater, since after sending a request, the response is cached on the platform side, by the *BI App*, so the number of requests will be much lower, for the same number of users.

6.2.2 Business Intelligence Dashboards

As it has been reinforced throughout this dissertation, it is important to take full advantage of the collected data to obtain knowledge and, thus, allow the definition of strategies to give an economic boost to companies. Hence, the data is not only used to produce recommendations but can also be explored and give much more information about the online store's general commerce. With Kibana, this can be arranged in various visual representations (such as graphs, tables, etc) and grouped in dashboards, to support business decisions.

In this case, a dashboard was created based on the Deeply platform with some of the following graphs:

Total Orders by Country

From a Marketing perspective, it is interesting and useful to calculate the total number of orders by country, since it is possible to identify and invest in various forms of marketing, such as advertising strategies or promotions, appropriate to the context of each country, and consequently maintain or even increase the number of sales.

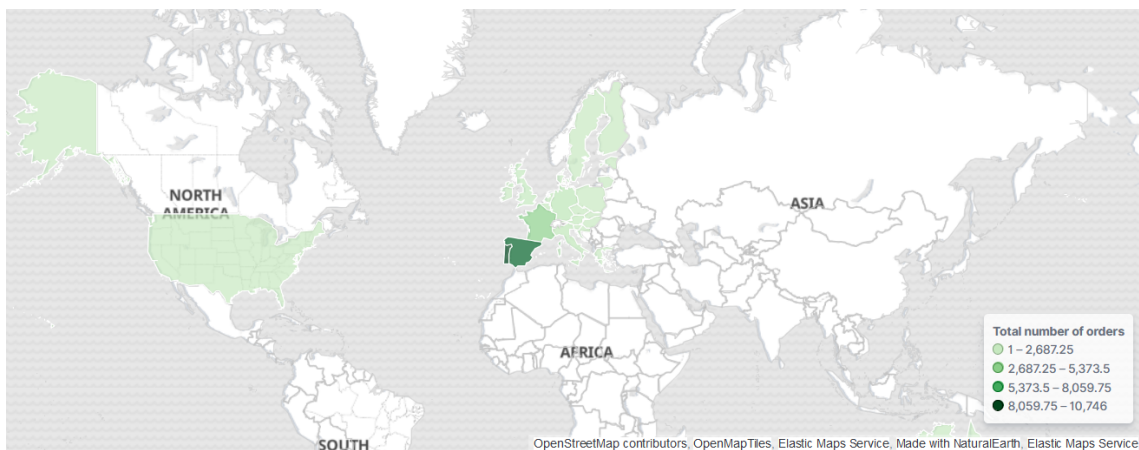


Figure 6.14: Total Orders by Country - Map

Kibana allows to make various types of graphs, it is a matter of deciding which style best suits the information one wants to know. Seeing countries on the map, for example, is more visually appealing. However, a bars graph allows a more objective and effective analysis. As we can see, these values correspond to the analysis made in section 6.1.2.

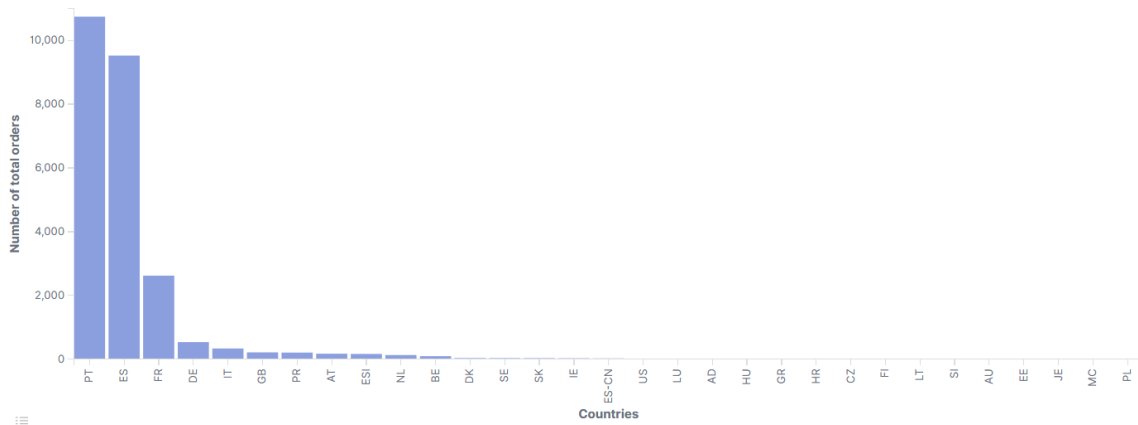


Figure 6.15: Total Orders by Country - Bars graph

Average spend by Country

In turn, when calculating the average spend by country, we must have in mind that a country that has higher revenue than another does not necessarily mean that the first has more orders. Nevertheless, knowing how much is spent on average for each order by country can show a glance on the buying behavior of clients in that country, and the company can adjust both prices and invest more in product promotions that match the average price spent by each client, in an attempt to increase the number of sales.

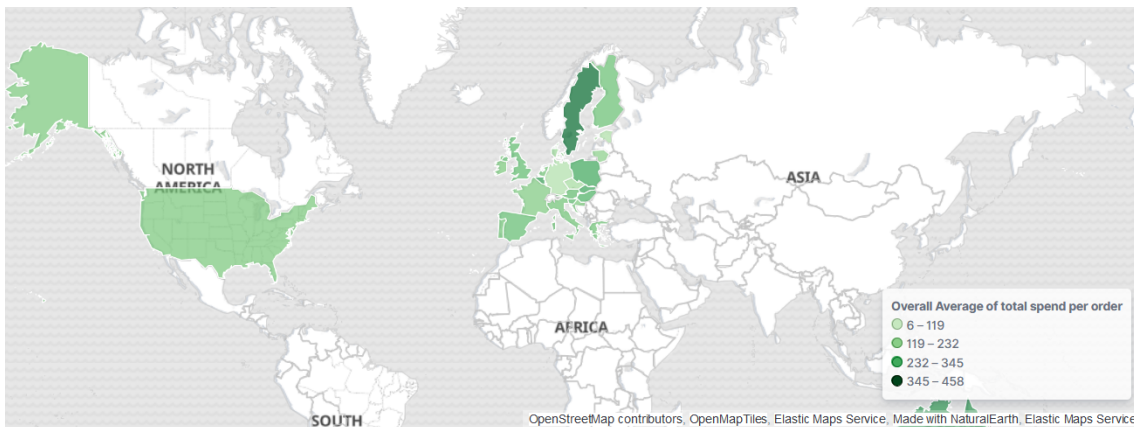


Figure 6.16: Average spend by Country

Average quantity and spend per order

It is also important to have graphs that show a global perspective of the stores's sales, such as the average price or average quantity of items per order, giving feedback to users on how the online store trade is going and if it corresponds to vendor's expectations and the company's objectives.

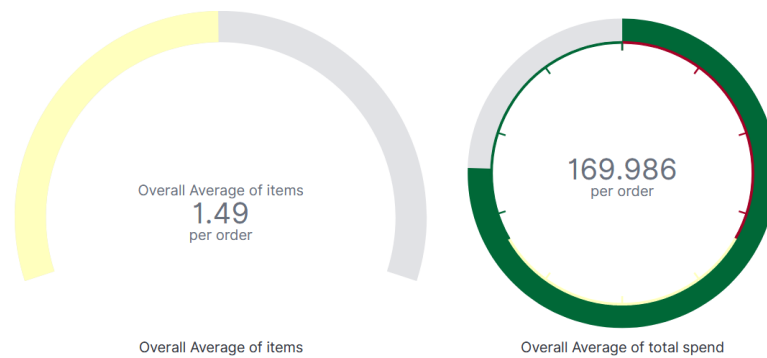


Figure 6.17: Average quantity and spend per order

Client Genders

The Client values previously analyzed in section 6.1.2 can be reflected in a pie chart for better visualization. With this knowledge, companies can make certain decisions according to the gender of their clients, applying this information to assume the right direction in their business and increase profit, combining the BI sector with Targeted Marketing.

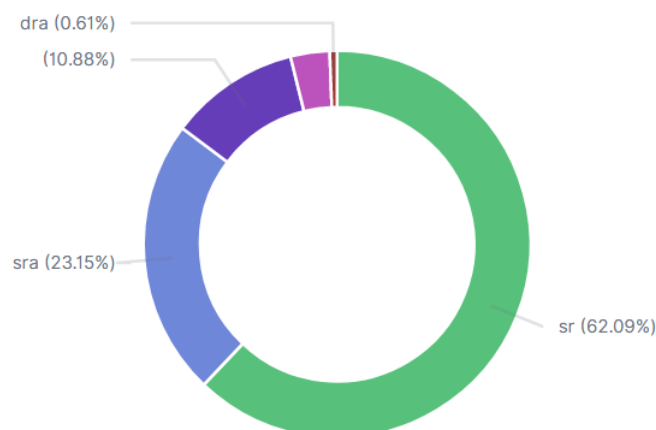


Figure 6.18: Client genders

Promotion Tracking

Through the Time Series Data Visualizer (TSDV) feature, it is possible in Kibana to identify possible moments with the potential to carry out promotions. As a hypothetical case, in this test scenario, an alert was defined to notify the owner of the online store that there is a potential chance of making a discount on clients' purchases whenever the total price of the order (`item_total_with_tax`) is greater than 400€ for each product of a different category. For example, whenever the store sells 400€ or more of a certain product with X category ($X \in \{\text{MEN, WETSUIT, SURF ACCESSORIES, CLOTHING}\}$), a promotion is applied. This chart indicates and maintains the history of when promotions would be most efficient and in what context.

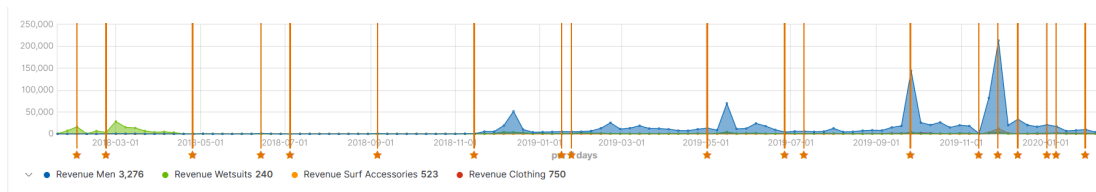


Figure 6.19: Promotion Tracking

The complete dashboard is attached in appendix A.14, with the remaining graphs and tables:

- Sales by Category;
- Sold products per Week;
- Total Revenue;
- Top Selling Products;
- Clients table;
- Products table;
- Order-items table;

Most of these visual representations are based on orders from the online store, even though this does not correspond to any entity considered in this project. Nevertheless, it is possible to make observations of orders through the *Order-item* entity by grouping them using the aggregation feature provided by Kibana. Thus, the obtained results were produced by aggregating order-items forming several "buckets" in which each one was identified by the `order_id` of the order-item, thus considering orders as a whole. This brings great advantages since we can have generalized information about the orders from the online store, keeping the detailed data of each order-item.

Another feature that makes Kibana such a powerful tool is the ability to update in real-time, as Elasticsearch receives and stores data. Since the Elasticsearch database is synchronized with the database of the e-commerce platform, through the developed triggers, Kibana is able to update the dashboard graphs in real-time.

The registration tables inserted at the end of the dashboard, obtained from Kibana's *Discover* area, operate as logs of the Elasticsearch database, indicating which data was received and when it was received, thus maintaining a history and providing a preview of each stored document.

To obtain dashboards with more reliable results, the data must be as complete and normalized as possible, being from tenants' responsibility since they are the data providers.

6.3 Discussion

A microservice approach was adopted instead of a monolithic architecture, since a single application can present data congestion problems (bottleneck) when faced with a scenario where e-commerce platforms are hit by countless users, generating a large number of events/requests. A microservice architecture allowed to build a robust and flexible system, due to the independence of each component, capable of handling requests and managing resources efficiently. This way, the service can serve multiple tenants - multi-tenancy - instead of assigning a monolith instance (copy) to serve each platform - single-tenancy.

It should be noted that all requirements raised in section 3.2 are satisfied by the system:

1. The RS returns the results of recommendations in the form of a list of product IDs, in JSON format, as required by Beevo's e-commerce platforms;
2. It is possible to control the information that comes in recommendation results, indicating the quantity and category of recommended products that should be included in the response of the service. If no limit is imposed, the service returns a maximum of 100 recommended items;
3. RS provides a Kibana component, thus offering several data analysis and business intelligence features;
4. The company guarantees the maintenance of the server machine for at least 361 of the 365 days a year;
5. From the results of the load tests, we can verify that the system fulfils the requirement of responding under 100 ms to most requests, never exceeding 150 ms, even in moments of intense activity;

6. The system has authentication and authorization mechanisms, supported by JWT and an ACL, protecting all data involved in the recommendation process;
7. Supports multi-tenancy;
8. Allows the tenant to manipulate the recommendation models of the system's recommendation engine.

All recommendations produced were based on three entities (client, product and order-item), forming four types of recommendation (popularity, hybrid, similar and complementary products), having considered four different scenarios on the web pages (homepage, product listing, product details and side-cart) that meet the system requirements. These results were computed in offline mode, which means that the recommendation system calculated recommendations in the background, saving the results in MongoDB to be made available whenever required by the online stores. In turn, online stores formed data sets in the system's Elasticsearch, by sending data about the entities, from their database to the service, through the BI App. With this application it was also possible for tenants to manipulate the service, changing the configurations and data used by the recommendation engine.

It was decided that this would be the best way to extract data from e-commerce platforms, as an API allows the recommendation system to receive only data suitable for recommendations and to react to events only when necessary. Before the extraction process took place, there was an investigation and pre-selection of which fields should be extracted from the stores' databases, regarding the entities considered.

The accuracy of recommendations depends on the quality of the collected data and there is a need to adapt the models of the recommendation engine according to the available data. Better results could be obtained if there were more suitable variables for producing recommendations, such as:

- Sales per Time (SPT): Ratio between the number of sales and the time the product is on sale
- Sales per View (SPV): Ratio between the number of times a product has been sold and the number of times the product has been viewed by users on the product listing page
- Product Buy Path: Products visited by the user, until his next purchase, since the beginning of a session;
- Ratings and Reviews: product ratings and reviews given by users;
- Weather conditions: weather conditions and temperature at the time of the purchase at the user's location

These and many other variables could exponentially increase the accuracy of the results or even give a greater capacity of adaptation to the service, extending the scope to real-time recommendations.

However, since Beevo's e-commerce platforms have not yet developed the necessary mechanisms to obtain these variables, it was not possible to apply them. One can also notice that product prices are not considered as a default field. This is due to Beevo's product pricing logic complexity, in which price values change according to the type of client that is being addressed. Nevertheless, it is possible to add this value in the optional extraction fields in the BI App.

The architecture developed in this project, as well as the metrics and results obtained from its performance, are exposed and analyzed in my paper "*Improving Performance of Recommendation System Architecture*" (Appendix I - Publications).

There are several suggestions for improving the overall performance of the architecture, such as converting the protocol used in requests from HTTP/1 to HTTP/2 (86): HTTP/2 is more efficient and faster than the first due to multiplexing, header compression and binary formatting capabilities. As for scalability, the company opted for *horizontal scaling*, from an economic point of view, by adding more machines to the resource pool, instead of adding more power (CPU, RAM, ...) to the existing machine (*vertical scaling*). This way, it is possible to better manage the resources of each component and even open the possibility to implement a *Nginx* (87) element to serve as a load balancer and security mechanism in the future.

7. CONCLUSION

Conclusions

Given the growing number of online offers, recommendation systems appear as an effective strategy to combat the multiple decisions and divergent options that users face in online stores. The project's RS works as a PaaS, containing all the necessary infrastructure and computation for the production of product recommendations. The e-commerce platforms that use this service, on the other hand, only need to worry about communicating with it to receive such recommendations, through their respective applications.

Although the system does not include real-time recommendations, the trigger mechanisms implemented support the use of *online computing* in the architecture, giving the recommendation engine the possibility to generate recommendations in real-time, if it has the capacity to do so, and consequently allow platforms to present results in real-time, in response to user activities or other events.

This means that the progress of both architectural and engine components are interconnected, with the structure depending on the complexity of the engine. The more features the recommendation engine has, more endpoints and management will be required by the architecture to support these features.

On the other hand, it is possible to analyze and obtain statistics in real-time from the data collected, through Kibana dashboards, allowing greater control over data management. Thus, data stored by the system not only is used to produce recommendations, but also helps to form new perspectives on the market and support in business decisions - *Business Intelligence*.

So we may conclude that all the objectives proposed for this project have been achieved. Although the system is not yet a Beevo's final product, many of its customers have already shown interest in subscribing and incorporating the recommendation service in their online stores. The developed architecture allowed the integration of a recommendation engine in online stores, obtaining and

analysis of data from its clients and products, which will fulfil the company's objective of collecting more information for the application of various marketing strategies and expanding its Business Intelligence sector. Therefore, there is an expectation of increasing the number of users on the platforms, the number of sales and, consequently, obtaining a greater profit, leading the company to stand out in its market.

The technological components used in the architecture allow to combat all the problems foreseen in section 3.1.1, such as:

1. Personalized data sets: The Elasticsearch component allowed the storage of personalized data sets that contain various types of data from different e-commerce entities, thanks to the flexibility provided by the structure of the JSON documents used;
2. Data analysis: Complementing the previous component, Kibana allowed the analysis of the personalized data sets, in order to extract important information either for the application of marketing strategies or in the creation of dashboards to support business decisions;
3. Information up-to-date: Thanks to the triggers developed and implemented on the tenants' side, it was possible to keep the data sets always up to date and synchronized with the databases of the online stores;
4. Cold start problem: It was also possible, along with the algorithms used by the recommendation engine based on Collaborative and Content-based filtering, to combat the cold start problem by always including the most recent information in the recommendation process;
5. Availability and scalability: The MongoDB component guaranteed, to the system, a database with high availability and scalability necessary to satisfy the requirements raised at the start of the project, so that online stores could always have access to recommendation results;
6. Multi-tenancy and security: The approach of a microservices architecture gave the system the capacity to serve multiple stores simultaneously (multi-tenancy), always maintaining the communication flow protected by JSON Web Tokens and an Access Control List.

All the main points of this project are exposed and explained in the article *Improving Performance of Recommendation System Architecture* (Appendix I - Publications), where it emphasizes the development of the communication process between the RS and e-commerce platforms, the structure of recommendations and analysis the performance results.

Future improvements

In a futuristic perspective, a Javascript library could be developed to accommodate all the logic of communication with the recommendation system: this library would contain all the required methods for e-commerce platforms to communicate with the recommendation service. Online stores only needed to import and implement this library on their side, integrating it in the code of their widgets of the respective web pages, and the methods of recommendation would be responsible for communicating with the RS API to obtain results. Thus, facilitating the integration of the recommendation service on any platform.

As for the management of the system's host machine, a better alternative for the future would be to host the service in a cloud (e.g. Google Cloud or Amazon Web Services), freeing the company from the problems of resource management and scalability when the system evolves.

Finally, some ideas for future functionalities and types of recommendations were conceived, according to the current technology of the architecture. One of them will be *intelligent search*, which consists of search suggestions in the search bar of online stores. This would certainly be powered by the Elasticsearch full-text term search capabilities of the service, comparing the keywords entered by the user with the terms of names, attributes and categories of products, for example. Another improvement would be to make more direct recommendations, using child products instead of parents, as these are more detailed, which would provide clients with an even more personalized experience. However, Beevo does not yet support this feature. And lastly, in the Business Intelligence area, other types of provisional mechanisms could be developed by Kibana's Machine Learning feature, to predict stock replenishment or detect anomalies in sales.

Information is going to continue growing over the next years, and more and larger sources of data will appear. As personalization algorithms keep improving and data keep growing, recommendation system architectures must improve together, with opportunities and lessons to be learned.

REFERENCES

- [1] I. Amazon Web Services, “Amazon web services.” Url: <https://aws.amazon.com/>, 3 2006. Subsidiary of Amazon that provides on-demand cloud computing platforms and APIs.
- [2] I. Netflix, “Netflix.” Url: <http://www.netflix.com>, 8 1997. American media-services provider and production company.
- [3] S. Quinn, “Bulletproof node.js project architecture.” Url: <https://softwareontheroad.com/ideal-nodejs-project-structure/>, april 2019. [Accessed: 2020-08-18].
- [4] F. Bellard, “Qemu,” 2019. QEMU is a generic and open source machine emulator and virtualizer.
- [5] Y. A. Nanehkaran, “An introduction to electronic commerce,” *International Journal of Scientific & Technology Research*, vol. 2, pp. 190–193, 04 2013.
- [6] Beevo, “Beevo - business ecommerce evolution.” Url: <https://www.beevo.com/>, 2015. The Business eCommerce Evolution for medium-sized and large companies.
- [7] I. Amazon.com, “Amazon.” Url: <http://www.amazon.com>, 7 1994. American electronic commerce and cloud computing company.
- [8] G. Adomavicius and A. Tuzhilin, “Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions,” *Knowledge and Data Engineering, IEEE Transactions on*, vol. 17, pp. 734–749, 07 2005.
- [9] S. K. Singh, V. Prasad, S. Tanwar, and S. Tyagi, *Cloud Computing*, pp. 1–50. 08 2019.
- [10] S. K. Singh, V. Prasad, S. Tanwar, and S. Tyagi, *Software as a Service*, pp. 95–118. 08 2019.
- [11] D. Pascual, P. Daponte, and U. Kumar, *Handbook of Industry 4.0 and SMART Systems*. 09 2019.
- [12] I. Red Hat, “What is an api?.” Url: <https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces>, 2019. American multinational software company.
- [13] S. K. Singh, V. Prasad, S. Tanwar, and S. Tyagi, *Platform as a Service*, pp. 119–150. 08 2019.
- [14] S. Carey, “What’s the difference between iaas, saas and paas?,” 7 2019. [Accessed: 2019-12-23].
- [15] L. MuleSoft, “Microservices vs monolithic architecture.” Url: <https://www.mulesoft.com/resources/api/microservices-vs-monolithic>, 2019. [Accessed: 2019-12-09].

- [16] A. Kharenko, "Monolithic vs. microservices architecture." Url: <http://www.antonkharenko.com/2015/09/monolithic-vs-microservices-architecture.html>, 9 2015. [Accessed: 2019-12-09].
- [17] P. Giessler, M. Gebhart, D. Sarancin, R. Steinegger, and S. Abeck, "Best practices for the design of restful web services," *Proceedings - International Conference on Software Engineering*, vol. 10, pp. 392–, 11 2015.
- [18] T. Johnson, "What is a rest api?." Url: https://idratherbewriting.com/learnapidoc/docapis_what_is_a_rest_api.html#what-is-an-api, 2017. [Accessed: 2019-12-12].
- [19] A. Monus, "Soap vs rest vs json comparison [2019]." Url: <https://raygun.com/blog/soap-vs-rest-vs-json/>, 1 2019. [Accessed: 2019-12-12].
- [20] R. Meteren, "Using content-based filtering for recommendation," 06 2000.
- [21] B. Schafer, B. J. D. Frankowski, Dan, Herlocker, Jon, Shilad, and S. Sen, "Collaborative filtering recommender systems," 01 2007.
- [22] M. Hasan, S. Ahmed, M. Malik, and S. Ahmed, "A comprehensive approach towards user-based collaborative filtering recommender system," pp. 159–164, 12 2016.
- [23] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl, "Item-based collaborative filtering recommendation algorithms," *Proceedings of ACM World Wide Web Conference*, vol. 1, 08 2001.
- [24] Y. Deng and Q. Gao, "A study on e-commerce customer segmentation management based on improved k-means algorithm," *Information Systems and e-Business Management*, 12 2018.
- [25] NIST/SEMATECH, "Nist/sematech e-handbook of statistical methods." Url: <https://www.itl.nist.gov/div898/handbook/eda/section1/eda11.htm>, 10 2013. [Accessed: 2019-12-02].
- [26] a. S. M. G. c. KPI.org, "What is a key performance indicator (kpi)?." Url: <https://kpi.org/KPI-Basics>, 2019. [Accessed: 2019-12-03].
- [27] K. D. Foote, "A brief history of business intelligence." Url: <https://www.dataversity.net/brief-history-business-intelligence/>, 9 2017. [Accessed: 2019-12-13].
- [28] S. Negash, P. Gray, F. Burstein, and C. Holsapple, *Business Intelligence*, pp. 175–193. 01 2008.
- [29] L. Google, "Youtube." Url: <http://www.youtube.com>, 2 2005. American video-sharing platform.
- [30] S. Spotify Technology, "Spotify." Url: <http://www.spotify.com>, 4 2006. Audio streaming platform.

- [31] I. Facebook, "Facebook." Url: <http://www.facebook.com>, 2 2004. American online social media and social networking service.
- [32] e. I. Pierre Omidyar, "ebay." Url: <https://www.ebayinc.com/>, 8 1995. American multinational e-commerce corporation.
- [33] J. Mangalindan, "Amazon's recommendation secret." Url: <https://fortune.com/2012/07/30/amazons-recommendation-secret/>, 7 2012. [Accessed: 2019-12-06].
- [34] S. N. Ian MacKenzie, Chris Meyer, "How retailers can keep up with consumers." Url: <https://www.mckinsey.com/industries/retail/our-insights/how-retailers-can-keep-up-with-consumers>, 10 2013. [Accessed: 2019-12-06].
- [35] S. Chhabra, "Netflix says 80 percent of watched content is based on algorithmic recommendations." Url: <https://mobilesyrup.com/2017/08/22/80-percent-netflix-shows-discovered-recommendation/>, 8 2017. [Accessed: 2019-12-06].
- [36] C. A. Gomez-Uribe and N. Hunt, "The netflix recommender system: Algorithms, business value, and innovation," *ACM Trans. Manage. Inf. Syst.*, vol. 6, pp. 13:1–13:19, Dec. 2015.
- [37] X. Amatriain and J. Basilico, "System architectures for personalization and recommendation." Url: <https://medium.com/netflix-techblog/system-architectures-for-personalization-and-recommendation-e081aa94b5d8>, 3 2013. [Accessed: 2019-12-24].
- [38] X. Amatriain, "Big personal: Data and models behind netflix recommendations," pp. 1–6, 08 2013.
- [39] J. Katukuri, R. Mukherjee, and T. Könik, "Large-scale recommendations in a dynamic marketplace," 10 2013.
- [40] Apache Software Foundation, "Hadoop." Url: <https://hadoop.apache.org>.
- [41] Y. Afify, I. Moawad, N. Badr, and M. Tolba, "A personalized recommender system for saas services: A personalized recommender system for saas services," *Concurrency and Computation: Practice and Experience*, vol. 29, 01 2016.
- [42] G. Guleria, "How recommendation systems work in ecommerce." Url: <https://blog.gluelabs.com/how-recommendation-systems-work-in-ecommerce-6cc30b56b401>, 9 2018. [Accessed: 2019-12-06].
- [43] G. R. . D. Ltd, "Yusp - personalization engine." Url: <https://www.yusp.com/>, 2017. [Accessed: 2020-04-12].

- [44] K. Hafner, "Netflix prize still awaits a movie seer." Url: <https://www.nytimes.com/2007/06/04/technology/04netflix.html>, 06 2007. [Accessed: 2020-04-12].
- [45] Strands, "Strands." Url: <http://retail.strands.com/>, 2004. FinTech software company.
- [46] i. salesforce.com, "Commerce cloud einstein implementation." Url: <https://trailhead.salesforce.com/en/content/learn/modules/cc-einstein-plan-and-implement>, 2019. [Accessed: 2019-12-13].
- [47] i. salesforce.com, "Einstein product recommendations for commerce cloud." Url: <https://trailhead.salesforce.com/en/content/learn/modules/cc-einstein-product-recommendations>, 2019. [Accessed: 2019-12-13].
- [48] I. Amazon Web Services, "Amazon personalize." Url: <https://aws.amazon.com/personalize/>, 2019.
- [49] P. Basford, "Creating a recommendation engine using amazon personalize." Url: <https://aws.amazon.com/blogs/machine-learning/creating-a-recommendation-engine-using-amazon-personalize/>, 6 2019. AWS Machine Learning Blog.
- [50] I. Amazon Web Services, "Amazon personalize developer guide." Url: <https://aws.amazon.com/personalize/resources/>, 2019.
- [51] I. Ltd, "Inawisdom." Url: <https://www.inawisdom.com>, 2016.
- [52] T. F. E. Wikipedia, "Multitenancy." Url: <https://en.wikipedia.org/wiki/Multitenancy>, 12 2019. [Accessed: 2019-12-27].
- [53] F. Consulting, "Akamai reveals 2 seconds as the new threshold of acceptability for ecommerce web page response times." Url: <https://www.akamai.com/uk/en/about/news/press/2009-press/akamai-reveals-2-seconds-as-the-new-threshold-of-acceptability-for-ecommerce-web-page-response-times.jsp>, note = Akamai Technologies, Inc., Cambridge, MA, 10 2009.
- [54] R. T. Fielding, "Rest apis must be hypertext-driven." Url: <https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven>, 10 2008. [Accessed: 2019-12-26].
- [55] S. Allamaraju, "Describing restful applications." Url: <https://www.infoq.com/articles/subbu-allamaraju-rest/>, 12 2008. [Accessed: 2019-12-26].
- [56] A. Prando and S. N. A. Souza, "Modular architecture for recommender systems applied in a brazilian e-commerce," *Journal of Software*, vol. 11, pp. 912–923, 09 2016.
- [57] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)." RFC 7519, May 2015.

- [58] N. Elastic, "Elasticsearch," 2019. Elasticsearch - The heart of the Elastic Stack.
- [59] I. MongoDB, "Mongodb," 2019. The database for modern applications.
- [60] Joyent, "Node.js," 2020. Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine.
- [61] StrongLoop, "Express.js," 2020. Fast, unopinionated, minimalist web framework for Node.js.
- [62] K. Gamage, "Separation of concerns for web engineering projects," 07 2017.
- [63] A. S. Foundation, "Apache kafka," 2020. Apache Kafka is an open-source distributed event streaming platform used by thousands of companies for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications.
- [64] N. Elastic, "Kibana," 2019. Your window into the Elastic Stack.
- [65] V. Peixoto, H. Peixoto, and J. Machado, "Integrating a data mining engine into recommender systems," in *Intelligent Data Engineering and Automated Learning – IDEAL 2020* (C. Analide, P. Novais, D. Camacho, and H. Yin, eds.), (Cham), pp. 209–220, Springer International Publishing, 2020.
- [66] I. Docker, "Docker," 2019. Docker: Empowering App Development for Developers.
- [67] Sheng Zhang, Weihong Wang, J. Ford, F. Makedon, and J. Pearlman, "Using singular value decomposition approximation for collaborative filtering," in *Seventh IEEE International Conference on E-Commerce Technology (CEC'05)*, pp. 257–264, 2005.
- [68] C. Sammut and G. I. Webb, eds., *TF-IDF*, pp. 986–987. Boston, MA: Springer US, 2010.
- [69] S. Gupta and R. Mamtara, "A survey on association rule mining in market basket analysis," *International Journal of Information and Computation Technology*, vol. 4, no. 4, pp. 409–414, 2014.
- [70] K. Aida, "Effect of job size characteristics on job scheduling performance," vol. 1911, 05 2000.
- [71] T. Costello and L. Blackshear, *What Is ETL?*, pp. 1–3. 01 2020.
- [72] R. Lerdorf, "Php: Hypertext preprocessor," 1995. PHP is a popular general-purpose scripting language that is especially suited to web development.
- [73] D. Stenberg, "Php: Hypertext preprocessor," 1997. Command line tool and library for transferring data with Urls.
- [74] R. Khan, K. McLaughlin, D. Laverty, and S. Sezer, "Stride-based threat modeling for cyber-physical systems," in *2017 IEEE PES Innovative Smart Grid Technologies Conference Europe (ISGT-Europe)*, pp. 1–6, 2017.

- [75] X. Amatriain, A. Jaimes, N. Oliver, and J. Pujol, *Data Mining Methods for Recommender Systems*, pp. 39–71. 10 2011.
- [76] N. Elastic, “Logstash,” 2020. Centralize, transform stash your data.
- [77] D. Community, “Debezium,” 2020. Stream changes from your database.
- [78] M. E. Bob Young, “Red hat,” 2020. Red Hat, Inc. is an American multinational software company that provides open source software products to enterprises.
- [79] J. W. Tukey, *Exploratory data analysis*, vol. 2. Reading, MA, 1977.
- [80] J. W. Tukey, *The collected works of John W. Tukey*, vol. 1. Taylor & Francis, 1984.
- [81] S. Software, “Swagger,” 2020.
- [82] S. Software, “Openapi specification,” 2020.
- [83] J. H. J. H. L. H. Carl Byström, Hugo Heyman, “Locust,” 2020. An open source load testing tool.
- [84] A. Strzelewicz, “Process manager 2.” Url: <https://pm2.keymetrics.io/>, 2013. [Accessed: 2020-06-05].
- [85] O. Foundation, “Node’s cluster api.” Url: <https://nodejs.org/api/cluster.html>, 2020. [Accessed: 2020-06-05].
- [86] N. Ramadan and I. Abdelwahab, “Impact of implementing http/2 in web services,” *International Journal of Computer Applications*, vol. 147, pp. 27–32, August 2016.
- [87] I. Sysoev, “Nginx,” 2005.
- [88] G. Cunha, H. Peixoto, and J. Machado, “Improving performance of recommendation system architecture,” in *Intelligent Data Engineering and Automated Learning – IDEAL 2020* (C. Analide, P. Novais, D. Camacho, and H. Yin, eds.), (Cham), pp. 495–506, Springer International Publishing, 2020.

APPENDICES

Appendix I - Publications

Improving Performance of Recommendation System Architecture (88)

Authors: Cunha G., Peixoto H., Machado J.

Editors: Analide C., Novais P., Camacho D., Yin H.

Title: Improving Performance of Recommendation System Architecture

Book: Intelligent Data Engineering and Automated Learning – IDEAL 2020

Year: 2020

Publisher: Springer International Publishing

Pages: 495–506

ISBN: 978-3-030-62365-4

Abstract: The exponential appearance of online stores has implied higher market competitiveness and, consequently, companies need to adopt certain strategies to obtain greater prominence and gain clientele. This paper explores an architectural approach to incorporate a recommendation system in online stores, in order to offer a solution to achieve those goals. Developing the recommendation system infrastructure with NodeJS, based on a REST API, and according to microservices architecture concepts, has proven to be very efficient when it comes to managing great volumes of requests and data, and be capable to serve multiple tenants within a short response time. Clustering techniques were also implemented to increase the system's performance and capability of handling requests.

Appendix II - NodeJS Application Structure

```
.env                # Environment variables
ecosystem.config.js # PM2 configurations
package.json       # NPM dependencies
docs               # API swagger documentation
src                # Source code
├─ app.js          # Application loader
├─ config          # Environment variables and configurations processor
│  └─ index.js
├─ controllers    # Request managers
│  └─ acl.js
│  └─ clients.js
│  └─ order-items.js
│  └─ products.js
│  └─ recommendations.js
│  └─ recommender.js
│  └─ users.js
├─ loaders        # Load services at startup
│  └─ elasticsearch.js
│  └─ express.js
│  └─ index.js
│  └─ mongodb.js
│  └─ routes.js
│  └─ services.js
├─ middlewares    # Request middlewares
│  └─ authentication.js
│  └─ authorization.js
├─ models         # MongoDB Models
│  └─ ComplementaryProduct.js
│  └─ Item.js
│  └─ Recommendation.js
│  └─ Recommender.js
│  └─ SimilarProducts.js
│  └─ Suggestion.js
│  └─ User.js
├─ public         # Public files
│  └─ css
│  │  └─ login.css
│  │  └─ register.css
│  └─ js
│  │  └─ register.js
├─ routes         # Defines API endpoints
│  └─ acl.js
│  └─ complementary-product.js
│  └─ clients.js
│  └─ hybrid.js
│  └─ order-items.js
│  └─ popularity.js
│  └─ products.js
│  └─ recommender.js
│  └─ similar-products.js
│  └─ users.js
├─ server.js      # Server startup
├─ services       # Business Logic
│  └─ AclService.js
│  └─ ClientService.js
│  └─ EntityService.js
│  └─ OrderItemService.js
│  └─ ProductService.js
│  └─ RecommendationService.js
│  └─ RecommenderService.js
│  └─ UserService.js
├─ views          # HTML Pages
│  └─ home.ejs
│  └─ login.ejs
│  └─ register.ejs
```

Figure A.1: NodeJS Application Structure

Appendix III - API Documentation with Swagger

Beevo Business Intelligence Service API 6.1 OAS3

BBI System documentation can be found [here](#).
Contact the [developer](#)
Apache 2.0

Servers
 Authorize

Recommender

- GET `/{platform}/recommender/configurations`
- PUT `/{platform}/recommender/configurations`
- DELETE `/{platform}/recommender/configurations`

Clients

- GET `/{platform}/clients`
- POST `/{platform}/clients`
- GET `/{platform}/clients/{clientID}`
- DELETE `/{platform}/clients/{clientID}`
- PUT `/{platform}/clients/{clientID}`

Products

- GET `/{platform}/products`
- POST `/{platform}/products`
- GET `/{platform}/products/{productID}`
- DELETE `/{platform}/products/{productID}`
- PUT `/{platform}/products/{productID}`

OrderItems

- GET `/{platform}/orders`
- POST `/{platform}/orders`
- GET `/{platform}/orders/{orderID}`
- DELETE `/{platform}/orders/{orderID}`
- PUT `/{platform}/orders/{orderID}`

Recommendations

- GET `/{platform}/recommendations/similar-products/{productID}`
- GET `/{platform}/recommendations/popularity`
- GET `/{platform}/recommendations/hybrid/{clientID}`
- POST `/{platform}/recommendations/complementary-products/`

Figure A.2: API Documentation with Swagger - part 1

Users ▼

- GET /{platform}/users 🔒
- DELETE /{platform}/users 🔒
- POST /{platform}/users/register 🔒
- GET /{platform}/users/login.{format} 🔒
- POST /{platform}/users/login.{format} 🔒

ACL ▼

- GET /acl 🔒

Permissions ▼

- POST /acl/permissions 🔒
- DELETE /acl/permissions 🔒
- GET /acl/permissions/users/{userID} 🔒

Roles ▼

- GET /acl/roles 🔒
- POST /acl/roles 🔒
- DELETE /acl/roles 🔒
- GET /acl/roles/{userID} 🔒
- GET /acl/roles/users/{role} 🔒
- POST /acl/roles/parents 🔒
- DELETE /acl/roles/parents 🔒
- DELETE /acl/roles/{role} 🔒

Resources ▼

- GET /acl/roles/resources/{role} 🔒
- DELETE /acl/resources/{resource} 🔒

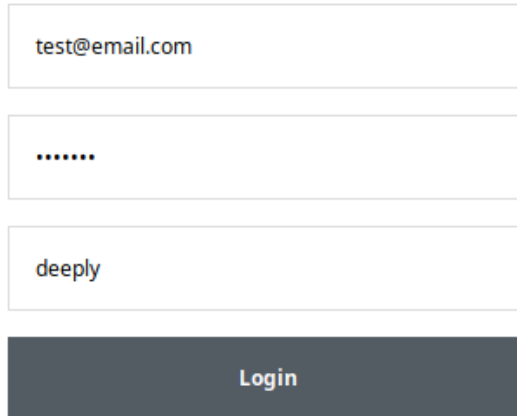
Schemas ▼

- Client >
- Product >
- Orderitem >
- Recommender >
- User >
- SimilarProducts >
- Recommendation >
- ComplementaryProduct >
- Item >
- Suggestion >

Figure A.3: API Documentation with Swagger - part 2

Appendix IV - User login page of the Recommender System

BBIS User Login Page

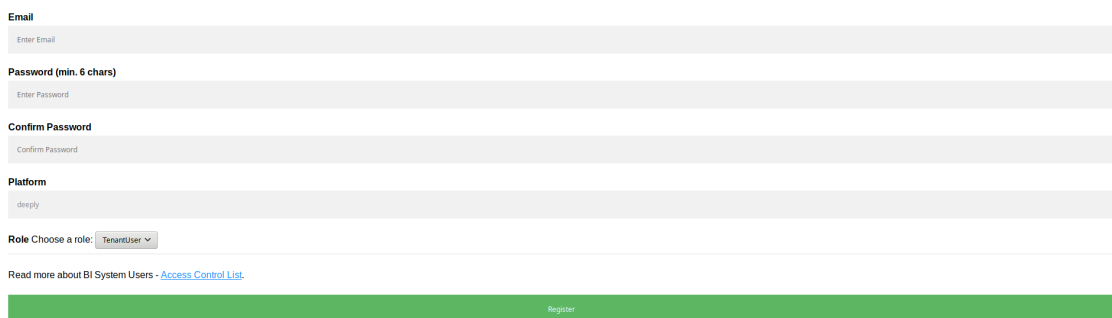


The image shows a user login page with three input fields and a login button. The first field contains the email address 'test@email.com'. The second field contains a password represented by six dots. The third field contains the name 'deeply'. Below the fields is a dark grey button labeled 'Login'.

Figure A.4: User login page of the Recommender System

BBIS User Registration Page

Please fill in this form to create a BBIS account.



The image shows a user registration page with several input fields and a registration button. The fields are labeled 'Email', 'Password (min. 6 chars)', 'Confirm Password', and 'Platform'. The 'Platform' field contains the value 'deeply'. Below the fields is a dropdown menu for 'Role' with the selected option 'TenantUser'. At the bottom, there is a green button labeled 'Register'.

Figure A.5: User registration page of the Recommender System

Appendix V - Beevo Business Intelligence Data Contract

Object	Field BI	Field BEEVO	Format	Description
Client	client_id	client_id	STRING	Client identifier
Client	gender	gender	STRING	Client gender
Client	birth_date	birth_date	DATE	Client birthdate
Client	created_at	created_at	DATE	Client creation date
Client	updated_at	updated_at	DATE	Last update date
Client	isNew	-	BOOLEAN	Indicates if client is new to the platform/website
Client	address_country	-	STRING	Shipping address country prefix
Client	country	-	STRING	Client country prefix
Addresses	address_1	address_1	STRING	Shipping address 1
Addresses	address_2	address_2	STRING	Shipping address 2
Addresses	locality	locality	STRING	Shipping locality
Addresses	city	city	STRING	Shipping city
Addresses	zip	zip	STRING	Shipping zip
Addresses	district	district	STRING	Shipping district
Product	product_id	product_id	STRING	Product identifier
Product	parent_id	parent_id	STRING	Product parent identifier
Product	name	name	STRING	Product name
Product	sku	sku	STRING	Product sku
Product	aggregator_id	aggregator_id	STRING	Product aggregator identifier
Product	stock	stock	INTEGER	Product stock amount
Product	ordering	ordering	INTEGER	Product ordering for sorting options
Product	status	status	INTEGER	Product status
Product	buyable	buyable	INTEGER	Product 'buyable' status
Product	published	published	INTEGER	Product 'published' status
Product	created_at	created_at	DATE	Product creation date
Product	updated_at	updated_at	DATE	Last update date
Product	hits	-	INTEGER	Number of hits
Product	manufacturer	-	STRING	Product manufacturer
Product	parent_published	-	INTEGER	Product parent 'published' status
Product	categories	-	ARRAY	List of product categories
Product	has_discount	-	BOOLEAN	Indicates if product price has discount
Product	attributes	-	OBJECT	Product attributes, with key-value format, as in {'attribute_name': 'attribute_value', ...}
OrderItem	order_item_id	order_item_id	INTEGER	Order-item identifier
OrderItem	order_id	order_id	INTEGER	Order identifier

Figure A.6: Beevo Business Intelligence Data Contract - part 1

Object	Field BI	Field BEEVO	Format	Description
OrderItem	product_id	product_id	INTEGER	Order's product identifier
OrderItem	product_name	order_item_name	STRING	Product name
OrderItem	product_category	product_category_name	STRING	Product category name
OrderItem	quantity	quantity	INTEGER	Product quantity
OrderItem	product_total_with_tax	product_total_with_tax	FLOAT	Price per item, with tax
OrderItem	item_total_with_tax	item_total_with_tax	FLOAT	Sum of prices of total items in order, with tax
OrderItem	status	order_status->name	STRING	Order status
OrderItem	created_at	created_at	DATE	OrderItem creation date
OrderItem	updated_at	updated_at	DATE	Last update date
OrderItem	client_id	(order) entity_id	STRING	Order's client identifier
OrderItem	ip_address	(order) ip_address	STRING	Order's user ip address
OrderItem	ship_city	(order_headers) ship_city	STRING	Shipping city
OrderItem	ship_locality	(order_headers) ship_locality	STRING	Shipping locality
OrderItem	ship_country	-	STRING	Shipping country code
OrderItem	manufacturer	-	STRING	Product manufacturer
OrderItem	hits_count	-	INTEGER	Number of hits on that product by order's client
OrderItem	product_attributes	-	OBJECT	Product attributes, with key-value format, as in { "attribute_name": "attribute_value", ... }

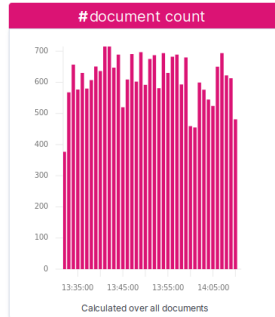
Figure A.7: Bevo Business Intelligence Data Contract - part 2

Appendix VI - Kibana's data analysis on Clients data

Metrics

1 field (1 exists in documents)

Sample documents per shard from a total of **23889** documents



Fields

25 fields (25 exist in documents)

All field types

filter

<p>address_1</p> <p>examples</p> <p>Avenida Pedro Alvares Cabral, 5, 4ºEsq Largo Hintze Ribeiro , nº2, 1ºDt Calle Manuel de Falla 32, 6ª Calle Severo Ochoa nº4 5º Ci Largo de Sao Sebastiao 10 Rua António Nobre nº25 A avenida fernandez latorre nº53 / 10 derecha paseo del parque 100, sotogrande Travessa de São Jorge 52 2, rue Président Carnot</p>	<p>taddress_1.keyword</p> <p>21,025 documents (88.01%) 20162 distinct values</p> <p>top values</p> <table border="1"> <tr><td>rua das piscina:</td><td>0.9%</td></tr> <tr><td>R BARTOLOME</td><td>0.2%</td></tr> <tr><td>Rua</td><td>< 0.1%</td></tr> <tr><td>teste</td><td>< 0.1%</td></tr> <tr><td>Rua Sao Miguel</td><td>< 0.1%</td></tr> <tr><td>asd</td><td>< 0.1%</td></tr> <tr><td>Rua Duque Salk</td><td>< 0.1%</td></tr> <tr><td>Rua Carlos Maç</td><td>< 0.1%</td></tr> <tr><td>Rua X</td><td>< 0.1%</td></tr> <tr><td>dsadsa</td><td>< 0.1%</td></tr> </table>	rua das piscina:	0.9%	R BARTOLOME	0.2%	Rua	< 0.1%	teste	< 0.1%	Rua Sao Miguel	< 0.1%	asd	< 0.1%	Rua Duque Salk	< 0.1%	Rua Carlos Maç	< 0.1%	Rua X	< 0.1%	dsadsa	< 0.1%	<p>address_2</p> <p>examples</p> <p>Largo de Sao Sebastiao 10 Praceta Dr. Clementino de Brito Pin Guadlario 50 5 Izquierda 3 1a Carrer del Xiprer 19 La Lomada, Calle Cascante 12 Praceta João Anastácio Rosa N1 2A rua das piscinas Cee</p>	<p>taddress_2.keyword</p> <p>20,901 documents (87.49%) 2921 distinct values</p> <p>top values</p> <table border="1"> <tr><td>rua das piscina:</td><td>85.5%</td></tr> <tr><td>11</td><td>< 0.3%</td></tr> <tr><td>Escalera 2 sota:</td><td>< 0.1%</td></tr> <tr><td>Lisboa</td><td>< 0.1%</td></tr> <tr><td>2B</td><td>< 0.1%</td></tr> <tr><td>5D</td><td>< 0.1%</td></tr> <tr><td>Caldas de São .</td><td>< 0.1%</td></tr> <tr><td>Centro Comerci</td><td>< 0.1%</td></tr> <tr><td>Estoril</td><td>< 0.1%</td></tr> </table>	rua das piscina:	85.5%	11	< 0.3%	Escalera 2 sota:	< 0.1%	Lisboa	< 0.1%	2B	< 0.1%	5D	< 0.1%	Caldas de São .	< 0.1%	Centro Comerci	< 0.1%	Estoril	< 0.1%
rua das piscina:	0.9%																																								
R BARTOLOME	0.2%																																								
Rua	< 0.1%																																								
teste	< 0.1%																																								
Rua Sao Miguel	< 0.1%																																								
asd	< 0.1%																																								
Rua Duque Salk	< 0.1%																																								
Rua Carlos Maç	< 0.1%																																								
Rua X	< 0.1%																																								
dsadsa	< 0.1%																																								
rua das piscina:	85.5%																																								
11	< 0.3%																																								
Escalera 2 sota:	< 0.1%																																								
Lisboa	< 0.1%																																								
2B	< 0.1%																																								
5D	< 0.1%																																								
Caldas de São .	< 0.1%																																								
Centro Comerci	< 0.1%																																								
Estoril	< 0.1%																																								
<p>address_country</p> <p>examples</p> <p>PT ES FR DE DK SK IT GB NL</p>	<p>taddress_country.keyword</p> <p>20,943 documents (87.67%) 24 distinct values</p> <p>top values</p> <table border="1"> <tr><td>PT</td><td>51.9%</td></tr> <tr><td>ES</td><td>35.2%</td></tr> <tr><td>FR</td><td>7.7%</td></tr> <tr><td>DE</td><td>1.6%</td></tr> <tr><td>IT</td><td>1.1%</td></tr> <tr><td>GB</td><td>0.7%</td></tr> <tr><td>AT</td><td>0.4%</td></tr> <tr><td>BE</td><td>0.4%</td></tr> <tr><td>NL</td><td>0.3%</td></tr> <tr><td>SK</td><td>0.2%</td></tr> </table>	PT	51.9%	ES	35.2%	FR	7.7%	DE	1.6%	IT	1.1%	GB	0.7%	AT	0.4%	BE	0.4%	NL	0.3%	SK	0.2%	<p>birth_date</p> <p>12,063 documents (50.5%)</p> <p>earliest Jan 2 1900, 23:23:15.000 latest Jul 5 2020, 01:00:00.000</p>	<p>city</p> <p>examples</p> <p>Braga</p>																		
PT	51.9%																																								
ES	35.2%																																								
FR	7.7%																																								
DE	1.6%																																								
IT	1.1%																																								
GB	0.7%																																								
AT	0.4%																																								
BE	0.4%																																								
NL	0.3%																																								
SK	0.2%																																								
<p>tcity.keyword</p> <p>49 documents (0.21%) 2 distinct values</p> <p>top values</p> <table border="1"> <tr><td>Braga</td><td>95.9%</td></tr> <tr><td></td><td>4.1%</td></tr> </table>	Braga	95.9%		4.1%	<p>client_id</p> <p>examples</p> <p>00226847-f2a6-4584-a190-d861cb462c15 0008d6bf-d4a0-4b71-98ad-395b0177d833 0019c543-1e92-4dd4-a09c-4d1c6fete7f6 006a95bc-7060-4044-8068-a5031bb32e78 001dd91e-c606-4111-a292-ed1d7fd28ef4 000d85fd-06b9-4cc2-9f1e-9402f52e63d3 00788ea5-b542-4014-ba74-286cef7ee8c9 000f79ee-922a-4578-927e-4f51bde316d4 006fc7a6-09d6-41ba-8dfo-163ff8a386f0 007bd9f9-c0e0-4ea6-968e-82c9b3a4f274</p>	<p>tclient_id.keyword</p> <p>23,889 documents (100%) 24081 distinct values</p> <p>top values</p> <table border="1"> <tr><td>000368d39ef4</td><td>< 0.1%</td></tr> <tr><td>00038bb3-52e</td><td>< 0.1%</td></tr> <tr><td>0006d6bf-d4a</td><td>< 0.1%</td></tr> <tr><td>00078fcc-596f</td><td>< 0.1%</td></tr> <tr><td>000d85fd-06b</td><td>< 0.1%</td></tr> <tr><td>000ef5f9-976a</td><td>< 0.1%</td></tr> <tr><td>000f79ee-922e</td><td>< 0.1%</td></tr> <tr><td>00194c56385t</td><td>< 0.1%</td></tr> <tr><td>0019c543-1e9;</td><td>< 0.1%</td></tr> <tr><td>001bd984ac89</td><td>< 0.1%</td></tr> </table>	000368d39ef4	< 0.1%	00038bb3-52e	< 0.1%	0006d6bf-d4a	< 0.1%	00078fcc-596f	< 0.1%	000d85fd-06b	< 0.1%	000ef5f9-976a	< 0.1%	000f79ee-922e	< 0.1%	00194c56385t	< 0.1%	0019c543-1e9;	< 0.1%	001bd984ac89	< 0.1%	<p>country</p> <p>examples</p> <p>FR PT ES AT IT DE BE GB</p>														
Braga	95.9%																																								
	4.1%																																								
000368d39ef4	< 0.1%																																								
00038bb3-52e	< 0.1%																																								
0006d6bf-d4a	< 0.1%																																								
00078fcc-596f	< 0.1%																																								
000d85fd-06b	< 0.1%																																								
000ef5f9-976a	< 0.1%																																								
000f79ee-922e	< 0.1%																																								
00194c56385t	< 0.1%																																								
0019c543-1e9;	< 0.1%																																								
001bd984ac89	< 0.1%																																								

Figure A.8: Kibana's data analysis on Clients data - part 1

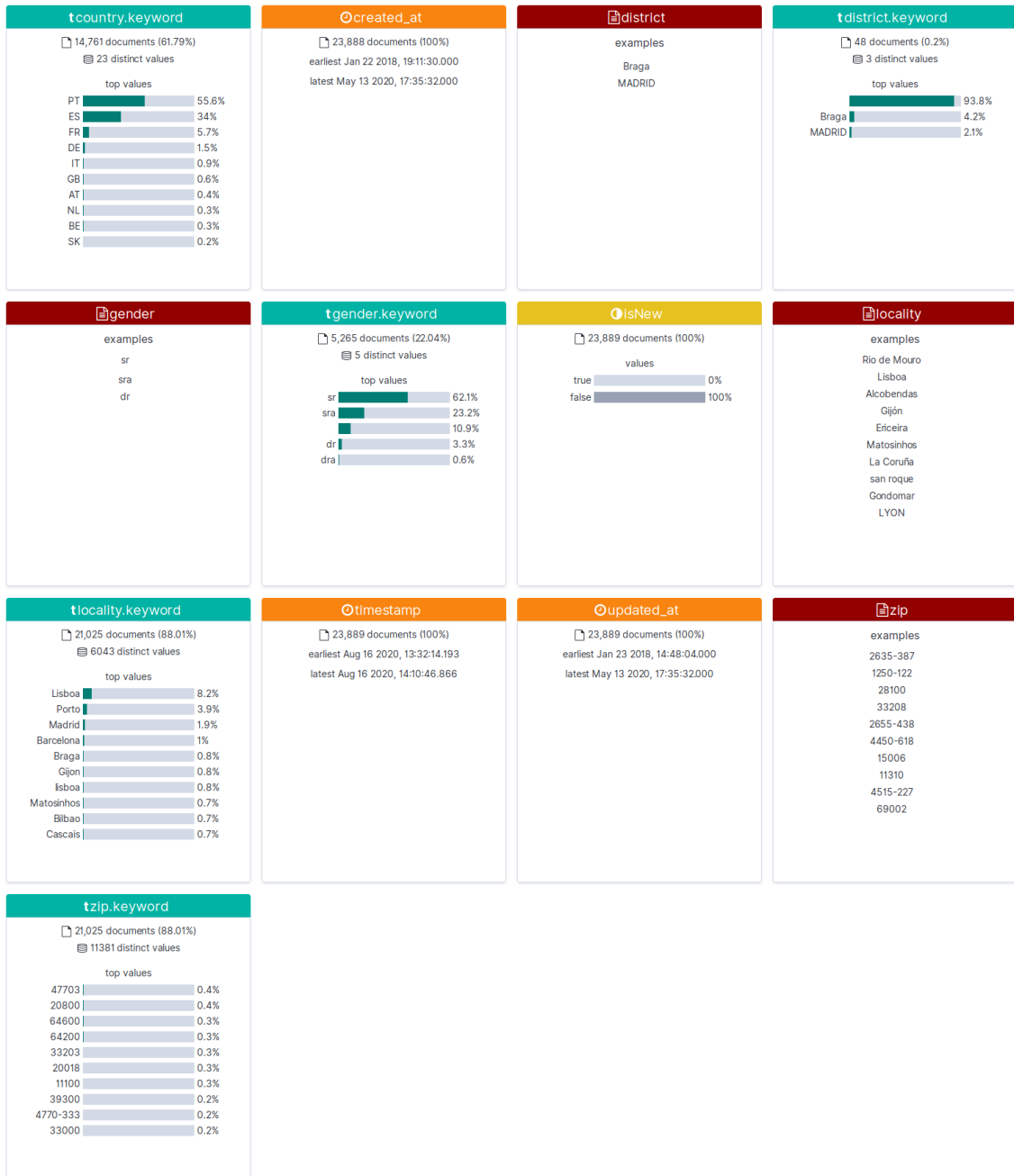


Figure A.9: Kibana's data analysis on Clients data - part 2

Appendix VII - Kibana's data analysis on Products data

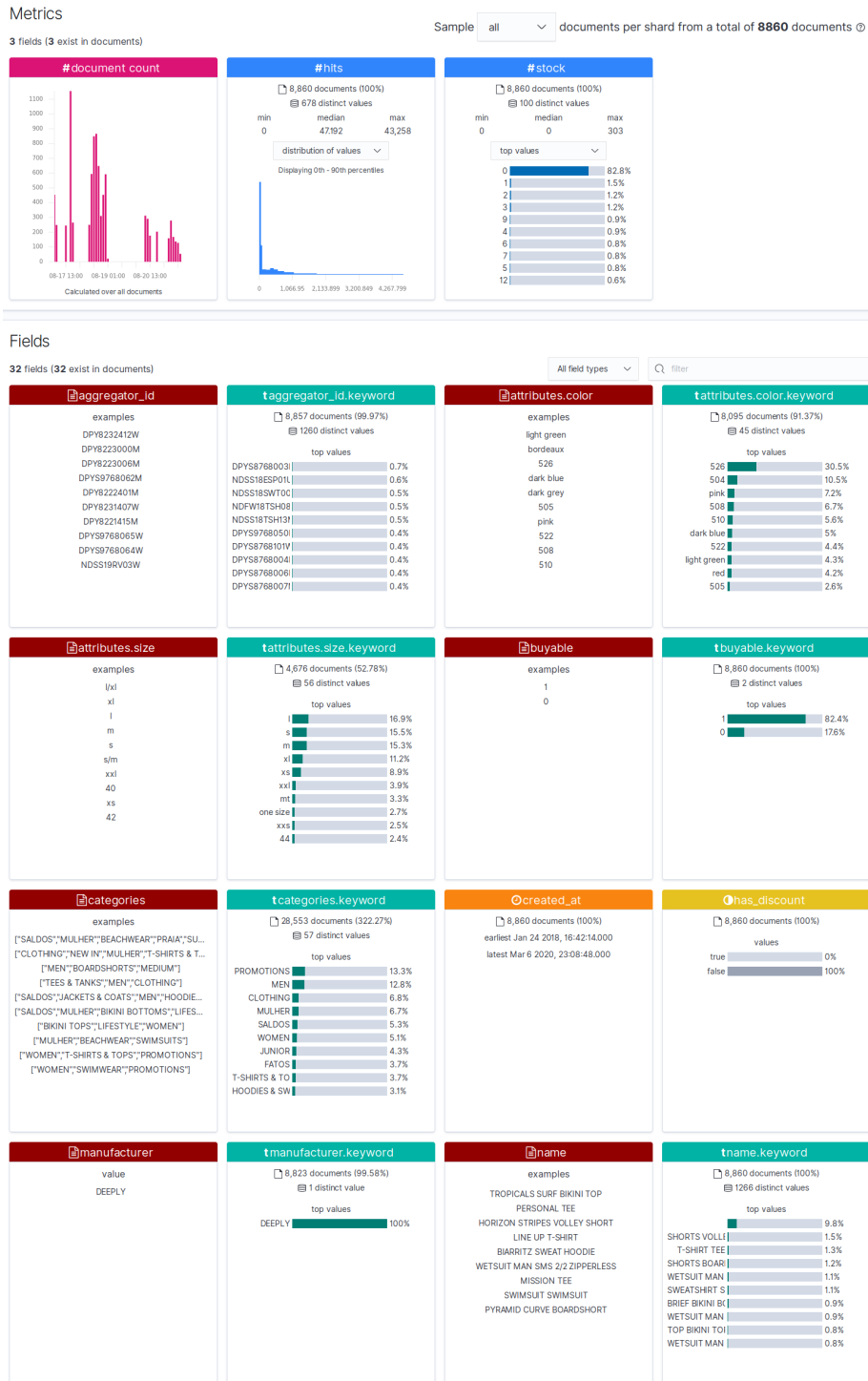


Figure A.10: Kibana's data analysis on Products data - part 1

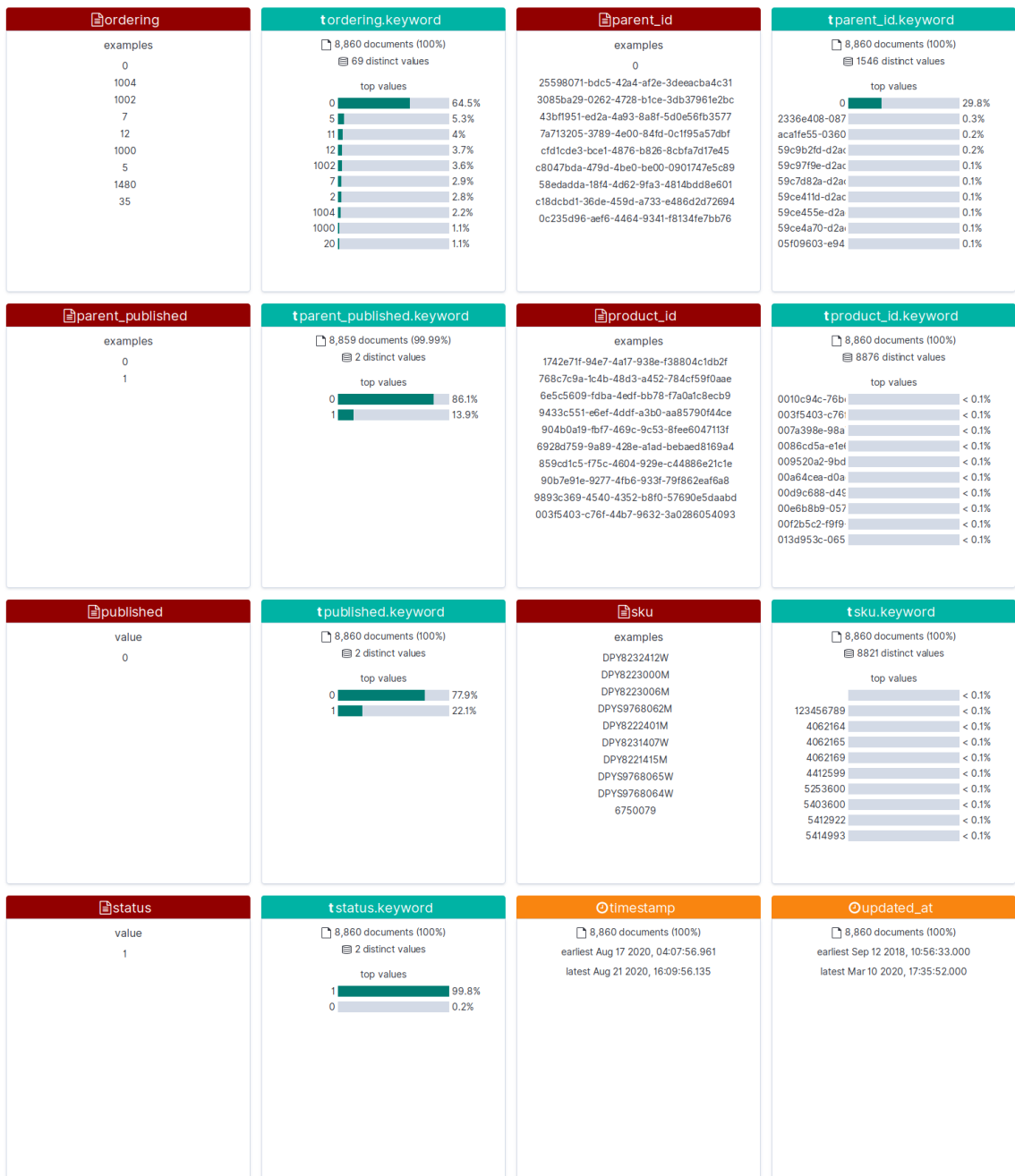


Figure A.11: Kibana's data analysis on Products data - part 2

Appendix VIII - Kibana's data analysis on Order-items data

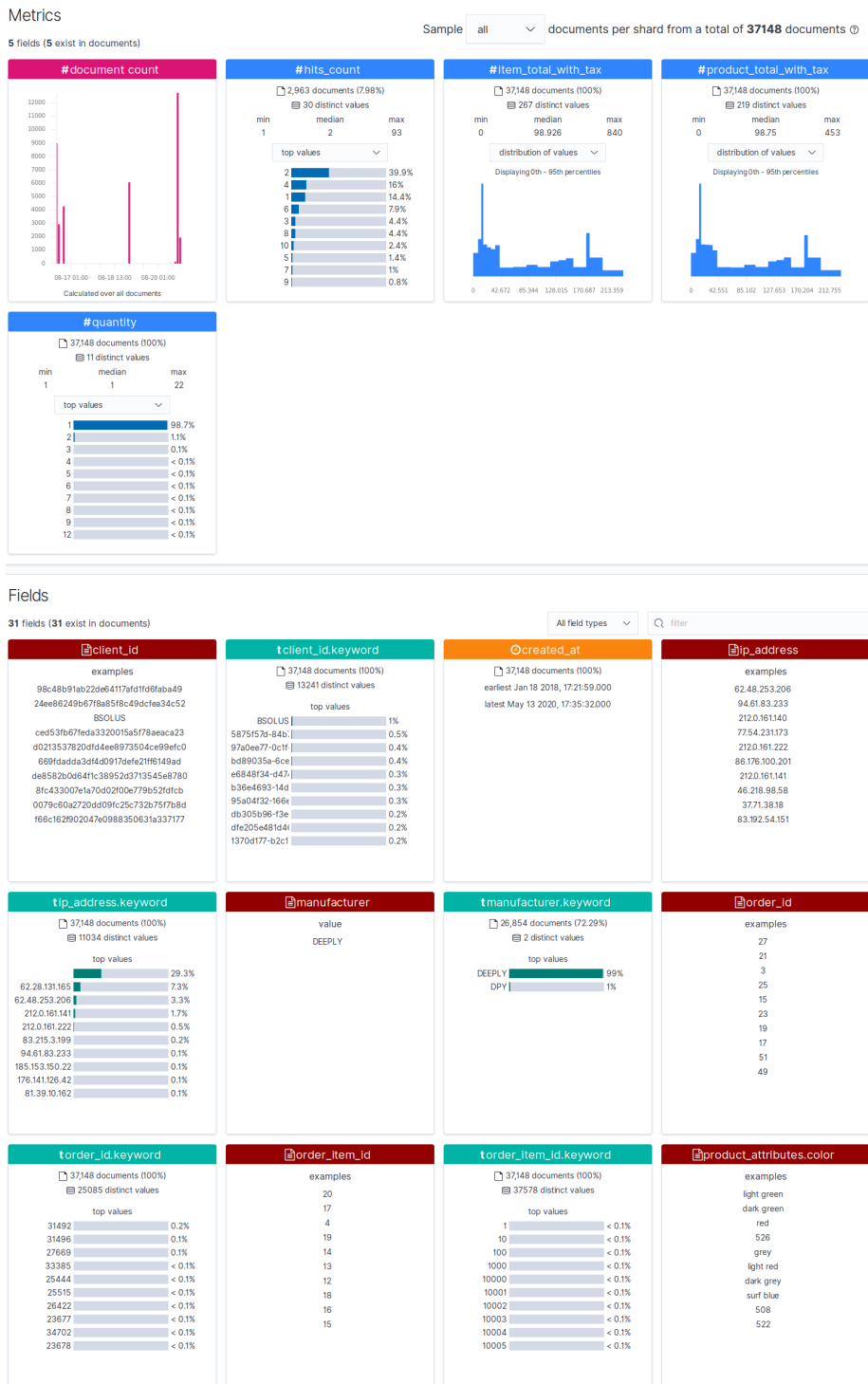


Figure A.12: Kibana's data analysis on Order-items data - part 1

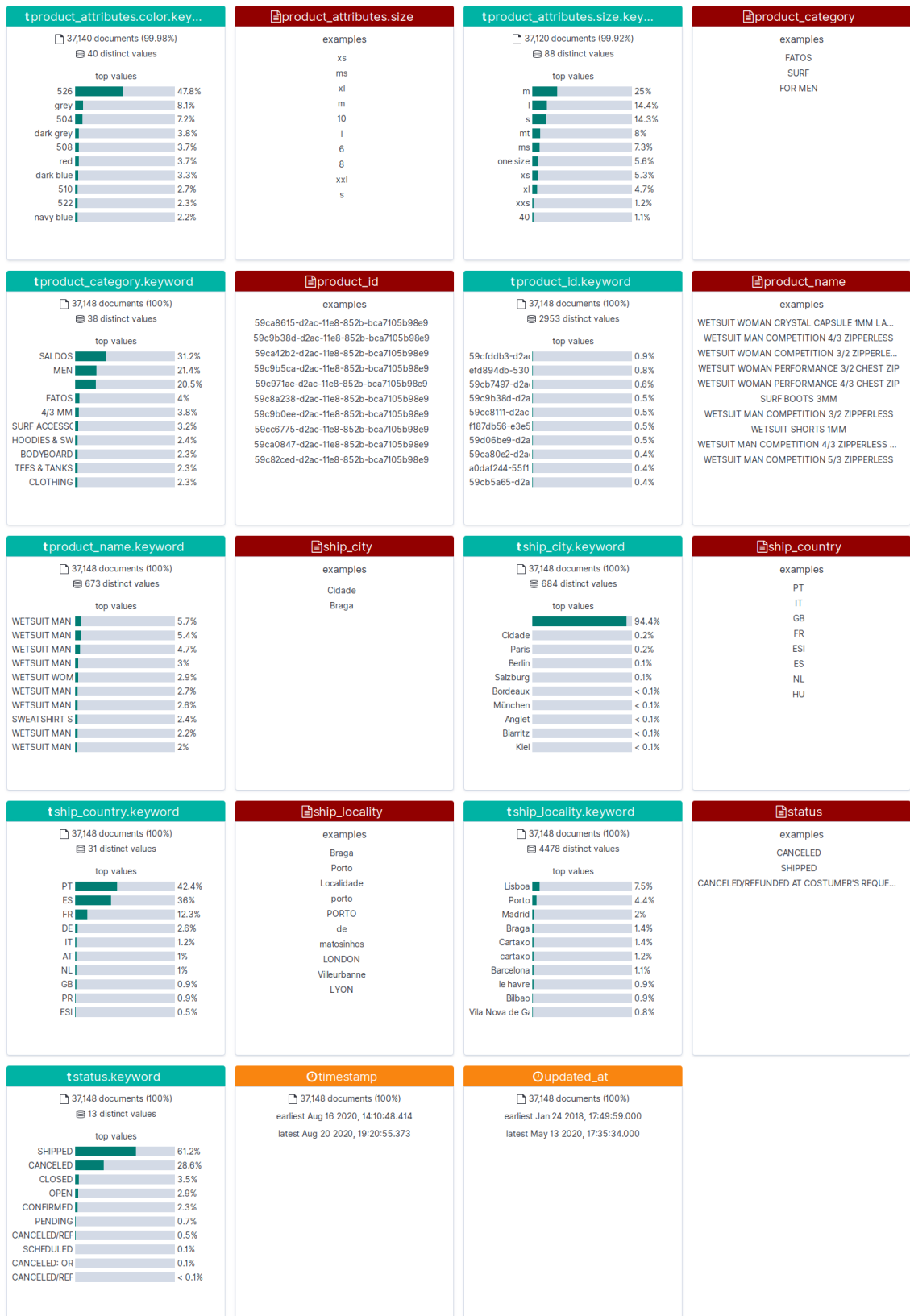


Figure A.13: Kibana's data analysis on Order-items data - part 2

Appendix IX - Kibana's Business Intelligence Dashboards

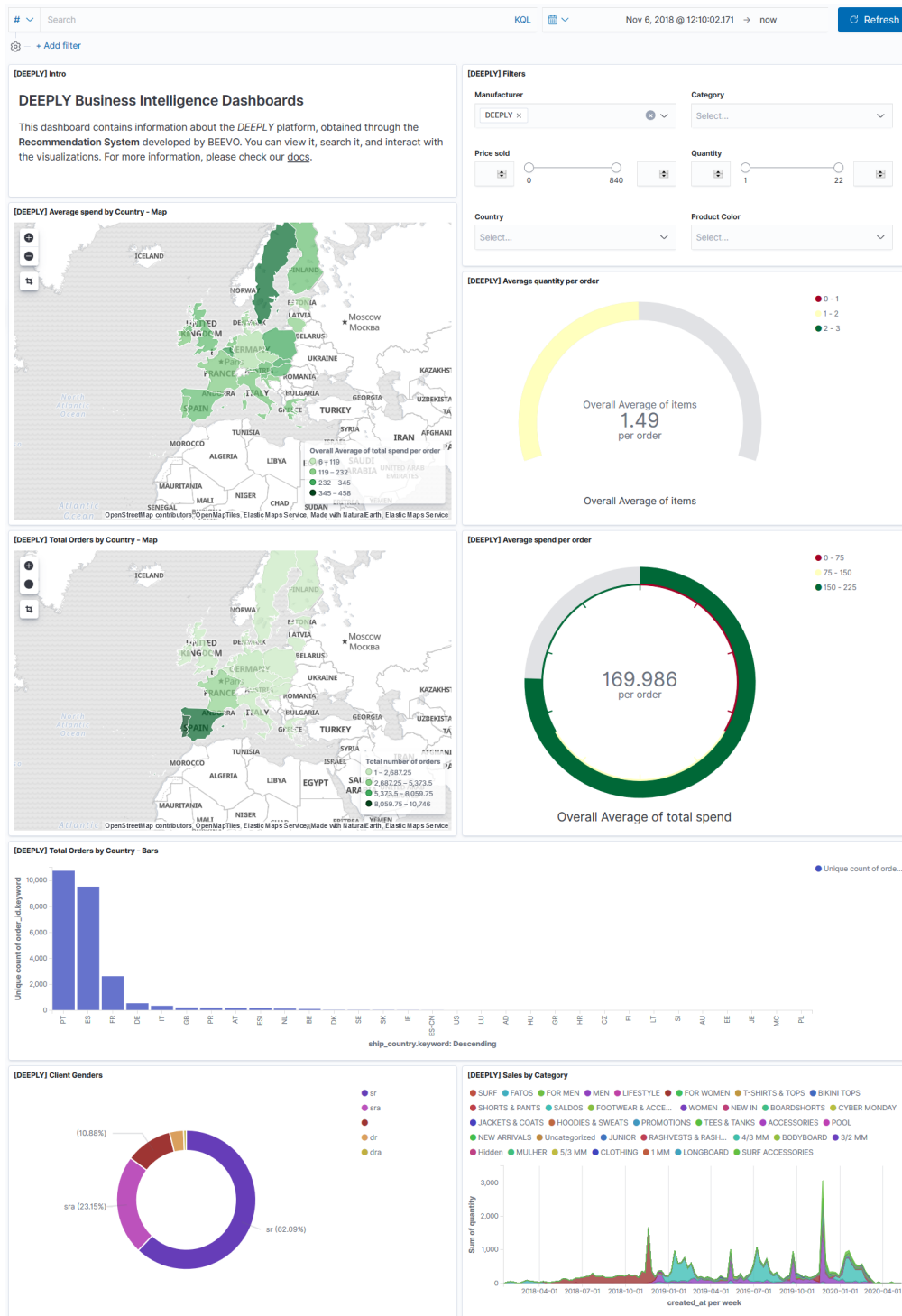


Figure A.14: Kibana's Business Intelligence Dashboards - part 1

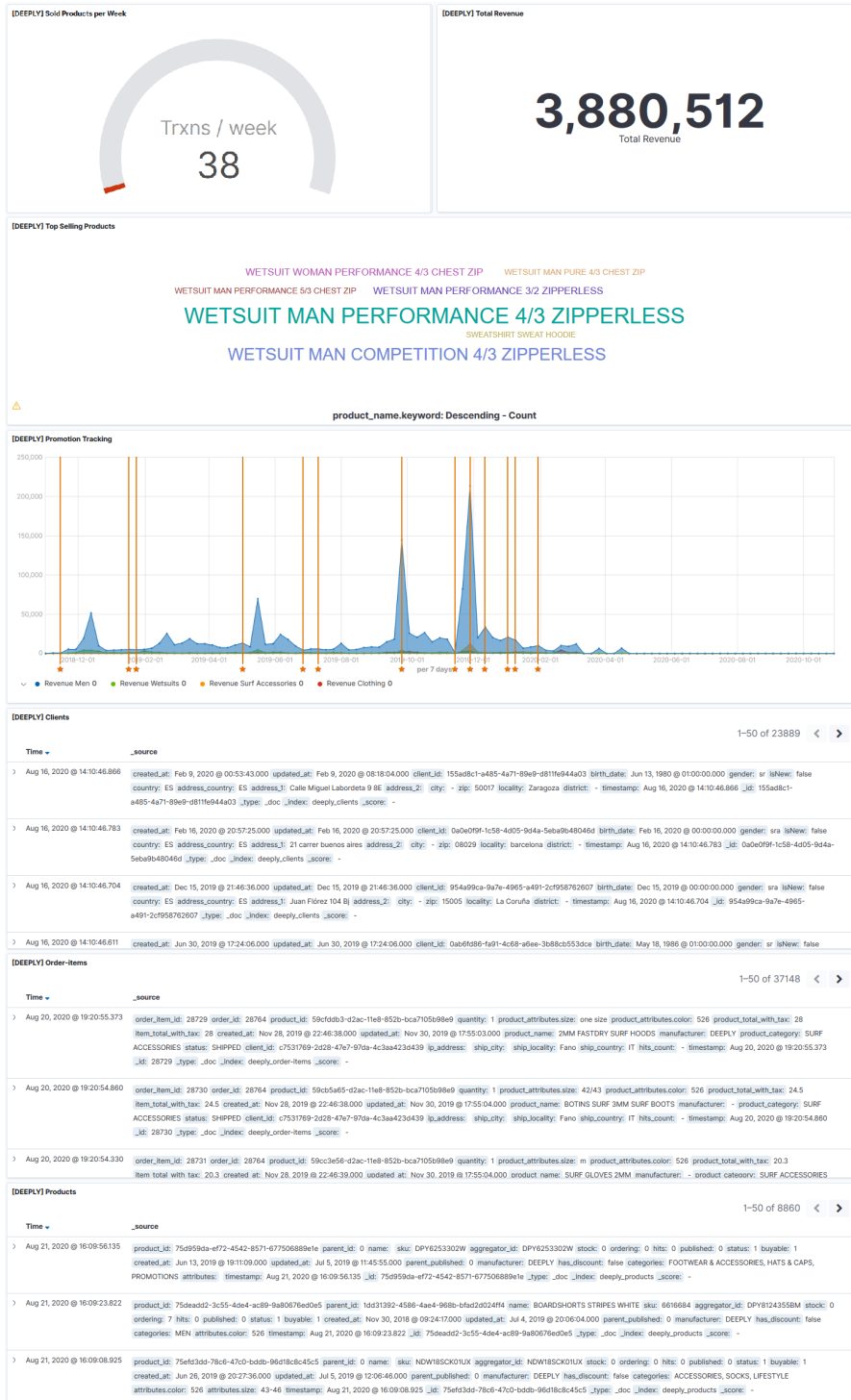


Figure A.15: Kibana's Business Intelligence Dashboards - part 2