

Universidade do Minho

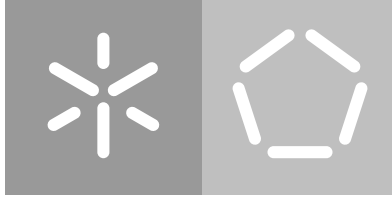
Escola de Engenharia

Departamento de Informática

José Diogo Lago Viana

Monitoring and Real-time Simulation of an Industrial Production Pipeline

December 2020



Universidade do Minho

Escola de Engenharia

Departamento de Informática

José Diogo Lago Viana

Monitoring and Real-time Simulation of an Industrial Production Pipeline

Master dissertation

Master Degree in Integrated Master's in Informatics Engineering

Dissertation supervised by

António Luís Sousa

December 2020

COPYRIGHT NOTICE

This is an academic work that can be used by third parties provided that internationally accepted rules and good practice concerning copyright and related rights are respected. Consequently, this work may be used in accordance with the license [Creative Commons Attribution-NonCommercial 4.0 International \(CC BY-NC 4.0\)](https://creativecommons.org/licenses/by-nc/4.0/) — <https://creativecommons.org/licenses/by-nc/4.0/>.

If one needs permission to make use of the work under conditions not foreseen in the indicated license, the author should be contacted through [RepositóriUM](#) of [Universidade do Minho](#).



ACKNOWLEDGMENTS

To my friends and family.

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 Universidade do Minho.

Braga, 16th November 2020

José Viana

ABSTRACT

There is a shortage of manufacturing management software solutions for businesses with various manual processes, and that offer a wide range of products. Existing solutions can become very expensive for small and medium-sized enterprises, and can discourage them to take the next step towards the *4th Industrial Revolution*.

This dissertation consists of joint work with *Tipoprado, Artes Gráficas*, to develop a package tracking and production performance analysis platform.

The company has a notable number of different clients and offers different types of services. This way, different packages may go through different paths on the production pipeline. Given this, to offer a more close and customized service, *Tipoprado*, wants to develop a package tracking platform. This tracking is not geographical (delivery case), but about the package location over the production pipeline, giving clients the possibility to consult, in real-time, the actual state of their orders.

Apart from this, implementing this platform produces a significant level of data about packages and clients. One of the main goals is to treat, process and analyze this data, to improve production efficiency and be able to help the its managers make crucial decisions about the referred pipeline. Production planning and predictions on delivery dates is the ultimate goal.

This dissertation studies and implements the tracking method that best applies to *Tipoprado* production pipeline, together with data analysis, and prediction options.

The given platform will transport the company to a tech production vision, and kick start its journey through the fourth industrial revolution. It is also expected to increase customer engagement levels, which correlate with a higher number of sales.

Keywords: Web Application, Web Development Architecture, Queuing Theory and Simulations, Production line

RESUMO

Existe uma falta de soluções de software de gestão de manufatura para empresas com vários processos manuais, e que oferecem uma vasta gama de produtos. As soluções existentes podem-se tornar bastante dispendiosas para pequenas e médias empresas, e desencorajar as mesmas para dar o próximo passo em direção à 4^a *Revolução Industrial*.

Esta dissertação consiste num trabalho conjunto com *Tipoprado, Artes Gráficas*, para desenvolver uma plataforma de *tracking* de encomendas e análise de desempenho de produção.

A empresa possui um número notável de clientes diferentes e oferece diferentes tipos de serviços. Dessa forma, encomendas diferentes podem seguir caminhos diferentes na linha de produção. Dado isso, para oferecer um serviço mais próximo e personalizado, a *Tipoprado*, deseja desenvolver uma plataforma de *tracking* de encomendas. Esse *tracking* não é geográfico (caso de entrega), mas sobre a localização da encomenda no *pipeline* de produção, oferecendo aos clientes a possibilidade de consultar, em tempo real, o estado atual dos seus pedidos.

Além disso, a implementação desta plataforma produz um nível significativo de dados sobre encomendas e clientes. Um dos principais objetivos é tratar, processar e analisar esses dados, melhorar a eficiência da produção e ajudar os responsáveis da empresa a tomar decisões cruciais sobre o referido *pipeline*. O planeamento da produção e as previsões nas datas de entrega é o objetivo final.

Esta dissertação estuda e implementa o método de *tracking* que melhor se aplica à linha de produção da *Tipoprado*, junto com a análise de dados e opções de previsão.

A plataforma fornecida transportará a empresa para uma visão de produção tecnológica e iniciará sua jornada na quarta revolução industrial. Também é esperado que aumente os níveis de envolvimento do cliente, que se correlacionam com um número maior de vendas.

Palavras-chave: Aplicação Web, Arquiteturas de desenvolvimento Web, Teoria de Filas e Simulações, Linha de Produção

CONTENTS

1	INTRODUCTION	1
2	STATE OF THE ART	3
2.1	Related Products	5
2.1.1	DHL - Tracking System	5
2.1.2	HiveTracker	6
2.1.3	Productoo	7
2.2	Tracking Methods	8
2.3	Queuing Theory and Predictions on a production pipeline	10
2.4	Technologies	14
2.4.1	Web Development Architectures	15
2.4.2	Web Development Languages/Frameworks	16
2.5	Summary	18
3	THE PROBLEM AND ITS CHALLENGES	21
3.1	Proposed Approach - solution	23
3.1.1	Solution Overview	23
3.1.2	Simulation Algorithm for predicting delivery dates (Sub-module)	24
4	DEVELOPMENT	28
4.1	Decisions	28
4.2	Implementation	29
4.2.1	Database and API	29
4.2.2	Critical Zone and Websockets for real-time interactions	32
4.2.3	Statistics and Metrics for production analysis	37
4.2.4	Delivery dates - Simulation	42
4.3	Outcomes and Summary	53
5	CASE STUDY / EXPERIMENTS	56
5.1	Experiment setup	56
5.2	Results	57
5.3	Discussion and Summary	62
6	CONCLUSION AND FUTURE WORK	63
6.1	Conclusions	63
6.2	Prospect for future work	64
A	SUPPORT MATERIAL	72
A.1	Other relevant Frontend pages	72

A.2 Email Examples

LIST OF FIGURES

Figure 1	Customer engagement process. [1]	4
Figure 2	<i>Productoo - Production Control</i> success story example. [42]	8
Figure 3	<i>Tipoprado's</i> production method.	9
Figure 4	1D vs 2D barcodes. [44]	10
Figure 5	Package flow inside production line example.	13
Figure 6	Simulation Engine Logic. [56]	14
Figure 7	Solution overview scheme.	24
Figure 8	A pipeline workstation inputs and outputs.	25
Figure 9	Database Logic Model.	29
Figure 10	Production Line page for package tracking.	36
Figure 11	Heatmap for mean times in the production line queues.	39
Figure 12	Vertical bar plot for availability, performance and workstation OEE.	41
Figure 13	Calculate approximate paths by Client flow.	44
Figure 14	Filtering paths for simulation.	45
Figure 15	Setup c locks and pop events from list.	47
Figure 16	Workstation updating c locks and informing SimMan of next event.	50
Figure 17	Simulation only uses business hours defined by the platform administrator.	51
Figure 18	Memory distribution of backend application.	54
Figure 19	Heat-map of packages in each workstation.	60
Figure 20	Number of packages at each workstation (paused and active).	60
Figure 21	Package list page. Quick view information or click to open detailed page.	72
Figure 22	New package page.	73
Figure 23	Calendar with weekly view for delivery dates.	73
Figure 24	Package public view page. Information available to the end client.	74
Figure 25	Number of package statistics page.	74
Figure 26	Workstation statistics page.	75
Figure 27	Waiting time in queue statistics page.	76
Figure 28	Ratings received from end clients page.	77
Figure 29	Example of a survey answer by a end client.	78
Figure 30	Automatic report page.	78
Figure 31	New package registered confirmation example email.	79

Figure 32 Package completed warning example email.

80

LIST OF TABLES

Table 1	Asset tag types and applications. [18]	8
Table 2	Web Development Architectures comparison. [2]	16
Table 3	Related products comparison.	18
Table 4	Resources' <i>CRUD</i> operations by user type.	31
Table 5	Email delivery statistics by quarter.	57
Table 6	Mean customer reviews rating by quarter.	58
Table 7	Percentage of packages delivered on time by quarter.	58
Table 8	Company's workstations overall availability and performance by quarter.	59
Table 9	Average waiting times by packages stopped in a queue, by quarter.	59
Table 10	Employees overall opinion on the platform.	61

INTRODUCTION

After some inquiries with the company's managers and some employees, we got to know the primary reason that motivates the need to start this partnership and develop the platform.

Tipoprado offers a wide range of services, from *Business to Client* or *Business to Business* solutions. It is normal in this industry to sub-contract other industries to do part of product processing, due to high product variety. This way, other clients or businesses also have a schedule to follow and need to know how their work will be affected by orders placed on *Tipoprado*.

Almost every day, fifteen to twenty clients call the company's office, to know the state of their packages. Being various the reasons that make them call, it forces the employee (receiving the calls) to spend about 10-15 minutes, searching for the respective package inside the production space. As expected, the company wants to minimize this wasted time to the maximum. So, the need to implement a production pipeline tracking platform became real. The key objective is to allow clients to consult the current and previous locations of each of their packages. Resulting, not only, in reducing time spent in searching for packages across the three pavilions significantly, but also, reducing the number of received calls in the office.

On the other hand, this kind of tracking produces relevant data, that can reveal important information about each client, and the efficiency/bottlenecks of the production pipeline. With this dissertation, one of the objectives is to conclude if the analysis of the data produced, can cause any positive impact on company production.

There are on the market various tracking applications, but according to the company, none enables clients to check the current state/history of the package. The most known, and almost everybody has come across it, at any point in his life, is *DHL* and other logistics/delivery enterprises. However, as explained before, it is a different type of tracking. So, the first step of this dissertation is to search for related software.

The state of the art methods for tracking packages is one of the essential aspects of this work, as it will help us to select the one that applies best to *Tipoprado's* case. We will analyze some of the most known/used methods, and see how they fit with the company's production characteristics and habits.

After this, queuing theory will also be studied for understanding best the referred pipeline. Queuing theory deals with problems that involve queuing (or waiting). As is known, a production line, can be broken into smaller individual subsystems. These subsystems consist of a queue to a specific activity, and the activity itself. With this study, we want to do a system modelling (arrival process, service mechanisms and queue characteristics), and retrieve reliable/accurate conclusions about the company's methods.

The ultimate goal consists in giving an approximate delivery date for a newly registered package or packages already inside the production pipeline. This prediction is, ideally, adaptable to the company's occupation and business hours. In this part, we will make a comparison between some of the most used and known methods/options, to understand each one, and conclude which can respond best to the given problem.

The solution to the challenge, presented by *Tipoprado*, is to join the best of both worlds (tracking system and data analysis) and develop a platform that can help the company reduce the time spent in client support, and improve production efficiency.

During this dissertation we want to expose and discuss some implementation decisions, like the technologies used, the system characteristics and performance problems. In a final phase, we will conclude on the platform usability (inquiring company employees or its clients) and validate its effects.

It is hoped to have a working product with modular characteristics that allow it to evolve, continuing to add new features, new design and also new members to work on it.

STATE OF THE ART

According to a Bernard Marr's [5] article on *Forbes*, "Why Everyone Must Get Ready For The 4th Industrial Revolution", first came steam and water power; then electricity and assembly lines; then computerisation. . . So what happens next? It is based on smart factories in which machines have web connectivity and are connected to a system that can virtually analyse the entire production floor and make decisions on its own.

Founder and Executive Chairman of the *World Economic Forum*, Professor Klaus Schwab, published a book entitled "The Fourth Industrial Revolution" [57], in which he describes how this fourth revolution is different from the previous three, as he claims that those were characterised mainly by advances in technology.

The idea of *Industry 4.0* (or whatever you call it) might seem almost perfect on paper; employees and machines all interconnected in harmony, to increase efficiency and productivity. However, Schwab outlines his concerns that businesses could be unable or unwilling to adapt to these new technologies. Why wouldn't industries leap into tech-based production? [63]

- Economic Challenges;
 - High economic costs;
 - Business model adaptation;
 - Unclear economic benefits/excessive investment.
- Social Challenges.
 - Privacy concerns;
 - Surveillance and distrust;
 - General reluctance to change by stakeholders;
 - Threat of redundancy of the corporate IT department;
 - Loss of many jobs to automatic processes and IT-controlled processes, especially for blue-collar workers.

Many experts are suggesting that this revolution will benefit the rich much more than the poor, especially as low-skill, low-wage jobs disappear in favour of automation.

Schwab said in his book, "The changes are so profound that, from the perspective of human history, there has never been a time of greater promise or potential peril. My concern, however, is that decision-makers are too often caught in traditional, linear (and non-disruptive) thinking or too absorbed by immediate concerns to think strategically about the forces of disruption and innovation shaping our future."

As referred in the quote below, and the present state of change, industries are in current demand for optimisation of their services, and to reach this, they need to go digital and tech-oriented.

"Around the world, and in practically every sector, manufacturing facilities are undergoing a major transformation. Digitisation is changing the way we process materials and make products, and data is becoming the golden key that can open a door to technological possibilities with the power to completely reshape manufacturing.

The traditional manufacturing model is evolving into what is referred to as a "smart factory" or Factory 4.0 – a connected system that links machinery, personnel, maintenance activity, and analytics for a completely integrated approach to factory management." - Seebo, "How factory 4.0 is transforming production" [58].

On the other hand, one of *Tipoprado's* objectives is to increase customer engagement by allowing them to be "involved" with their package orders through the whole process. Firstly, what is customer engagement? It is all about interactions between the customer and the brand. That simple. Why is it important? Finding ways to engage with customers in between purchases/orders strengthens their emotional connection to the brand, helping retain them while sustainably growing the business.



Figure 1: Customer engagement process. [1]

In fact, a study by *Hall and Partners' "Engager"*, described by MaryLou Costa in *Marketing-Week* [34] shows that up to $\frac{2}{3}$ of a brand's profits may rely on effective customer engagement. Another customer engagement study by Ray Wang of Constellation Research [46], in 2016, found that

"... companies who have improved engagement increase cross-sell by 22%, drive up-sell revenue from 13% to 51%, and also increase order sizes from 5% to 85%."

With these values in mind, it's easy to see that there is a direct correlation between how engaged customers are and how profitable the business is.

Being said, the reasons and objectives that drive *Tipoprado* to adopt this new model are valid and relevant.

Introducing the path to the rest of this dissertation, the act of tracking a package inside a production line, requires some sort of system (automatic or not) that registers each process/state. So in this chapter, the goal is to search for related products on the market, analyse their tracking systems, methods, and conclusions retrieved automatically by the respective software. A comparing review between predictions through Queueing Theory, [Machine Learning \(ML\)](#) or Simulation Algorithms is also pretended.

2.1 RELATED PRODUCTS

2.1.1 DHL - Tracking System

As it was mentioned in the introduction, the first related product to be talked about is from the delivery giant *DHL*¹. After a quick analysis on this platform we conclude that it is a Web Application, accessible by everyone on every system with access to the internet and a browser. It also has a homepage where any client can manually search for their packages by a unique code, received previously.

Due to this platform's main features (package tracking) are only accessible to clients that have "active" packages, the rest of this description is based on previous use and public examples made available by *DHL* itself.

One of the key features of this tracking system is that clients do not need to register on the platform. This results in a simple use and less "bureaucracy" that most users don't want to experience, just to know the location of a package. When a client places an order for a product that is shipped and registered under *DHL*, he is notified (generally by email) with a unique reference code to the respective package. With this code, only known by the client, he can search for the package and check its state.

In this platform, there is no relation between client/package, so, if someone has more than one package registered, it has to check for them separately.

Using some of the examples present in this page², we can check some of the "core" features of the platform.

A client can view the package destination, description, and the full history/process through the company system.

¹ <https://www.dhlparcel.pt/en/private-customers/receiving-a-parcel/track-and-trace.html>

² <https://www.logistics.dhl/pt-en/home/tracking/id-labels.html>

Viewing the same example page again is possible to conclude that the tracking method used to register each state, is barcode-based.

2.1.2 *HiveTracker*

This product is still a geographical tracking just like the previous one (it offers real-time product supply chain monitoring at item level from **End to End (E2E)**.) But, it provides new management possibilities, making it similar to one of the main goals of *Tipoprado* - statistics and other efficiency-related problems.

There is not much available about this product, but in *HiveTracker's* website³, are described some of the main features and the goals of the platform. The main objective is to "*Bring light to the dark spots of the supply chain*", to improve efficiency, margins, and sales. Regardless of having a different application, the ultimate purpose is the same as production line tracking for optimisation.

According to *HiveTracker* the benefits of real-time item-level monitoring include; faster release of the product, cost-saving, increased customer satisfaction, the possibility of immediate intervention, unrivalled compliance management, real-time loss or fraud detection, real-time customer behaviour insights and maximised stock control.

HiveTracker's provides a **Platform as a Service (PaaS)** for data visualisation and reporting combined with prescriptive analytics means visibility, agility and unprecedented maximisation of a company (client) supply chain.

Some interesting and related features found on the website:

- All products identified, authenticated and monitored in real-time throughout their life;
- Control and analysis of Product Flows from production to end consumer;
- Real-time monitoring and tracking, anywhere, anytime, fully autonomous with no external human action;
- Reports produced from measured data gives the image of the situation without any interpretation;
- Prescriptive analytics-based reports provided by business intelligence and machine learning indicate in real-time alerts, corrections and predictions of the supply chain for each transport.

Opposing to *DHL* system, *HiveTracker* uses a set of small connected sensors (size of a 2 Euro coin), that measure, log and transmit in real time: global position, temperature, humidity, light exposure and other customized data. Concluding, this is a more *Industry 4.0 (Internet of Things (IoT))* oriented solution.

³ <https://www.hivetracker.io/>

2.1.3 *Productoo*

Provides solutions for a complete digital transformation of production and maintenance planning, execution, tracking, and reporting. Comparing it with the previously presented products, *Productoo* has the most similar use case (a production line). According to their website⁴, *Productoo* main goal is to drive company's **Return on Investment (ROI)** and boost performance. To do so, they offer a series of different products, such as *Maintenance Control*, *Production Control*, and *Factory Cockpit*.

Some of the main advantages of using their products are:

- creating paperless environments for better traceability;
- improving **Overall Equipment Effectiveness (OEE)** by 10-30%;
- optimising planning to meet current demand (demand-driven strategy);
- reducing lead time and changeovers;
- spending less time reporting, making better decisions faster.

Focusing more on *Production Control*, *Productoo* offers:

- production planning and scheduling;
- connection to third party **Enterprise Resource Planning (ERP)**, **Manufacturing Execution System (MES)** and **OEE** software;
- production tracking, live dashboards of the production line and report generation.

It has also available some success cases using this product, as can be seen in one example in Figure 2. The analysis of this case and other examples proves the concept and validates that the process of tracking and monitoring a production a line, has positive impacts on the company efficiency, enabling it to have a **ROI** shortly.

According to specifications, their products are modular and are cloud-based (Web and Mobile Applications).

For *Maintenance Control*, the tracking method of the inventory/machinery is barcode readers, however for *Production Control*, it uses **MES** connectors and operates with the real manufacturing data. This solution is less error-prone and more accurate than manual input or barcode readers. Still, it is necessary for the machinery used to be "high tech", and compatible (more common in highly robotised systems that only produce one kind of material/product, as seen in the example above - automotive company).

⁴ <https://www.productoo.com/>

Customer story
Significant time savings with production digitization



The client an automotive company
 35 manufacturing lines
 11 packaging lines

The challenge an average of 1,200 orders printed daily and distributed to the shopfloor
 total time spent on planning: 15-20 hours weekly
 time-consuming ERP data management
 hard to keep up with current performance (if plan is being met)

The solution digitization of the entire manufacturing process by implementing Production Control (including Digital Workstation)
 SAP integration
 involving and connecting planners, warehouse workers, logistic managers, operators

The results



Figure 2: Productoo - Production Control success story example. [42]

2.2 TRACKING METHODS

As it was lightly introduced before, there are various methods to track assets. In this case, assets correspond to the packages (orders) inside the production pipeline. So, we will explore some possibilities and its viability, such as IoT sensors, Radio Frequency Identification (RFID) and barcodes. The following table, presented by *Hardcat*, represents a quick comparison between the four methods.

Tag Type	Asset value (\$)	Tag cost	Advantage	Disadvantage
Barcode	Low	Low	Minimal cost	No tracking ability
RFID	Medium - High	Medium - High	Basic tracking	Tracking at certain points only
GPS	High	Medium - High	Real-time tracking	Only outside buildings/cover
WIFI	High	Medium - High	Real-time tracking	Only outside buildings/cover

Table 1: Asset tag types and applications. [18]

The use of *GPS* or *WIFI* sensors will be discarded right away since its applications are for high-value assets, and mainly for outside (great distance) tracking. With *RFID* we achieve the same goal with probably fewer costs, for inside monitoring.

RFID is possibly a viable option because of the following characteristics:

- Tags are cheaper to purchase;
- Readers are more powerful with longer read range;
- Fixed and hand-held readers are cheaper to purchase;
- Tags are less sensitive concerning surface placement.

Comparing it with barcodes, many tags can be scanned at once, and the scanner only needs to be within the range of the tag to read it. Also, **RFID** is faster, more durable and more accurate than barcodes.

Nevertheless, **RFID** asset management is not a matter of one size fits all. Its tags and readers depend upon the environment, asset types and the outcome required.

On the other hand, barcode labels are a valuable and viable choice for businesses looking to improve efficiency and reduce overheads. Both cost-effective and reliable, barcodes have the following characteristics:

- Barcodes reduce the possibility of human error;
- A barcode scan is fast and reliable;
- Barcodes are inexpensive to design and print;
- Barcodes are extremely versatile and user-friendly.

For *Tipoprado*, an asset/package can, possibly, be seen as a whole paper sheet pallet or a final product pallet.

Pallets can be easily tracked. However, due to production methods, a pallet does not necessarily identify a package. Analysing Figure 3 that represents how a package is processed through most pipeline places is understandable that a pile of sheets is picked from one pallet, processed and placed on a new pallet on the other side.

This is necessary, because every machinery consumes (raw or not) individual paper sheets on one side, and delivers them on the other. That way, a pallet never follows through a package full "life cycle".

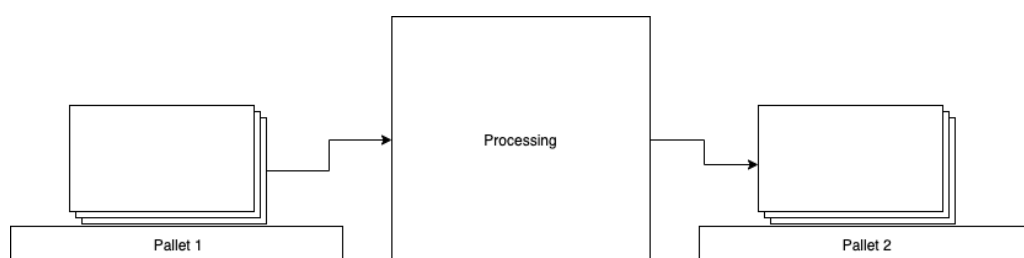


Figure 3: *Tipoprado's* production method.

As pallets cannot identify a package, the only way to do it would be through every single sheet that is processed. Still, this is not possible due to extreme costs marking each one.

Example: an order of 10000 A5 books, needs a minimum of 40000 sheets. A printer machine, like the ones owned by *Tipoprado*, can easily print 5000 sheets/hour. That way, it would be necessary for someone/something to mark each sheet, faster than the printing rate, to not waste valuable time.

Also, a single sheet has varying values, depending on quality; nevertheless, it still has a low value, relative to the cost of marking it.

According to the company responsible, for each package/order placed, it is printed a "description" page that accompanies the respective package for the full path on the pipeline. This page is managed by their ERP and has a barcode referring the package unique number (like a [Universal Product Code \(UPC\)](#)).

For this reason, [RFID](#) is not a viable option for *Tipoprado*, making the use of barcodes the best solution for this case. Barcodes can be *1D*, *2D* (Figure 4). These types depend on product/asset size and data requirements. [36]



Figure 4: 1D vs 2D barcodes. [44]

1D barcodes are typically used for identifying items or assets that are associated with variable information that resides in a database. A good example is an item marked with a [UPC](#) code that is linked to frequently changing information in a pricing database.

2D barcodes can be used to mark very small items. In some cases, these types of codes have been directly etched into auto parts or aeroplane components, or even tiny surgical instruments. While *1D* barcodes were traditionally scanned with laser scanning technology, *2D* codes require an *imager* (which can scan both *1D* and *2D* codes). The cost of imaging scanners has fallen so that they are now price competitive with laser scanners.

2.3 QUEUING THEORY AND PREDICTIONS ON A PRODUCTION PIPELINE

Queuing theory studies the process/act of waiting lines, and it enables the mathematical analysis of several related processes, including arrivals at the queue, waiting in the queue, and being served by the server. Queues are basic to both external and internal business

processes, which include staffing, scheduling and inventory levels. For this reason, businesses often utilise queuing theory as a competitive advantage.

The problem in almost every queuing situation is a trade-off decision. A manager must decide on the added cost of providing a more efficient/quick service (add more checkout places, more staff, etc.) against the inherent cost of waiting in line. If employees spend too much time manually entering data on a platform, a business manager could compare the price of investing in barcode scanners against the benefits of increased productivity. Likewise, if customers are unhappy because of late delivery dates, the business managers could analyse the cost of hiring more staff relative to the value of possibly increasing revenues and ensuring the retainment of customers. Those are the kind of conclusions/decisions that the platform aims to help *Tipoprado* in its production pipeline.

To modulate a queuing system is necessary to know these major components:

1. How packages arrive;
2. How packages are serviced.

Arrivals are divided into two types:

- Constant: the same period between successive arrivals (i.e., machine-controlled);
- Variable: random arrival distributions, which is a much more common form of arrival.

Servicing consists of the line(s) and the available number of servers for that subsystem. To modulate a queue servicing, is necessary to consider, the line length (number of items waiting), number of lines (number of servers) and the queue discipline. An essential feature of the waiting structure is the time the customer spends "inside" the server, also seen as *service time*. It can also be referred to as the *service rate*: the number of items that server will process per period (i.e., ten packages per hour).

Another crucial point on a servicing subsystem is to know the line structure or the overall system. By that, we need to identify if the respective line is *single or multi channel* and *single or multi phase*. The simplest type is the single-channel, single-phase, where there is only one channel for arriving customers and one phase of the service system. An example is the checkout counter at a supermarket, only one step and customers coming only from inside it (the supermarket). On the other side, we have multi-channel, multi-phase. *Tipoprado's* case is this one, but we will enter in more details later.

Some of the performance measures that can be answered by assessing and improving a production line, through the use of Queuing Theory, include:

- Utilization factor; (ρ)
- Number of packages in system; (L_s)

- Number of packages in queue; (L_q)
- Time spent in queue; (W_q)
- Time spent in system; (W_s)
- Service times;
- Efficiency of each workstation.

Yet, such measures can be calculated with the data collected from the tracking on the production pipeline, so there's no direct need to use Little's Law. ($L = \lambda W$) [31]

Like it was introduced before, one of the requirements set by *Tipoprado* is the possibility to predict delivery dates for each package. At the moment ML is the most popular method to make all sorts of predictions. Nonetheless, it raises doubts about being the one that best applies here.

Queuing Theory models try to estimate the behaviour of a queue system given several assumptions. This reveals a big flaw of these models: they only produce valid results if the queue follows the assumed distributions of inter-arrival and servicing times.

Josefin Stintzing and Frederik Norrman made a study about the use of data from queue systems to provide a prediction with artificial neural networks [62]. During this study, we observe the results obtained and how they got to them, using ML algorithms. They compare these results with the real data that the network was trained on, and with results using Queuing Theory methods.

One of their conclusions shows that while Queuing Theory may tell/respond to more specific problems about queue systems, given that the assumptions above are known and approximate, ML algorithms, such as the ones used, can give what rate customers arrive and how long to serve them. Yet, they tell nothing on how to use that information.

This way, if we can accurately modulate the production pipeline into one queuing system (arrival and servicing rates that follow a known distribution), and assuming a "steady-state" system, is possible to give an approximation of "total time inside the system" (delivery date) using Little's Law, referenced above.

However, since *Tipoprado* is looking for a "dynamic" prediction (not "steady-state"), this problem is not **First Come First Served (FCFS)**. Also, company managers are not sure about arrival distribution, meaning that the use of Queuing Theory may not produce an accurate result.

Due to high variety of services that the company provides, a package may take different paths inside the production line, depending on what type of processing it needs. This way, two packages that leave a *Server X* and a *Server Y* can "race for service" in *Server Z*. Checking Figure 5 this is evidenced by green, red, yellow and orange packages. Each one has a defined path and being, apparently, in different stages, they can meet in S4 and/or S5.

This queue state is in constant shifting with different packages arriving at different places with different future paths, making the system a multi-channel/multi-phase one.

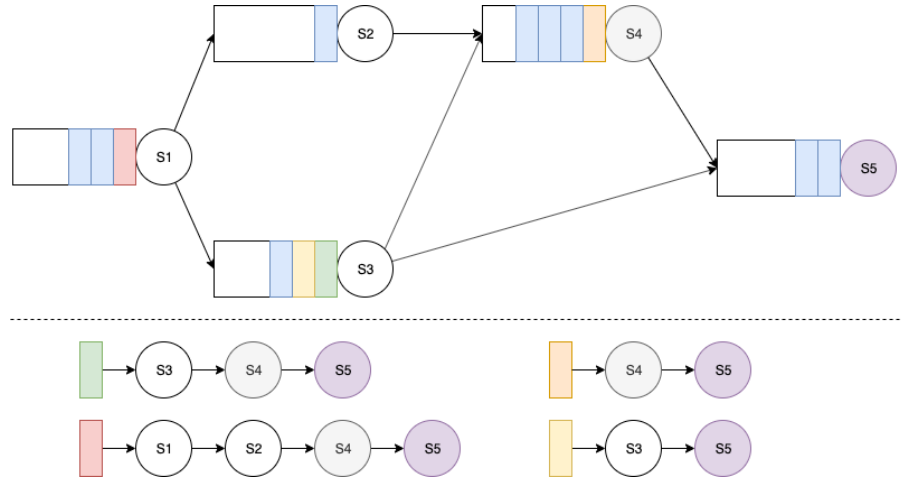


Figure 5: Package flow inside production line example.

So, it is also possible, to develop a simulation algorithm and have a dynamic prediction, with the actual state of the company in mind (real number of packages in each queue and processing times). This way, packages coming from different workstations to the same queue would interfere with each other’s waiting/total times.

There are two different types of simulation, **Discrete Event Simulation (DES)** and **Continuous Simulation (CS)**.

DES models the operation of a system as a (discrete) sequence of events in time [35], having two approach possibilities:

- **Next-event Time Progression:** it is assumed that there are no changes in the system between consecutive events. Thus, the simulation time can jump to the start time of the next event;
- **Fixed-increment Time Progression:** where time is incremented at a fixed rate and the simulation state is updated according to the events happening in that time slice.

Both forms of **DES** contrast with **CS** in which the system state is changed continuously over time, based on a set of differential equations defining the rates of change of state variables. [9]

This way, to create a **DES** (which is the one we need), it is necessary to keep track of:

- **State:** a set of variables that represent the properties of the system to be studied. The state evolution over time can be mathematically represented by a step function whose value can change whenever an event occurs;

- **Clock:** the simulation must keep track of the current simulation time. The units in which it is represented are irrelevant and must be chosen in a suitable way for the system. In *DES*, as opposed to *CS*, time "hops" because events are instantaneous and do not occur in a continuous form. This way, the clock skips to the next event start time, each time an event is completed, as the simulation proceeds;
- **Events list:** the simulation maintains at least one list of simulation events.

It is also necessary to be aware of a **stopping condition** for the simulation to reach an end. In a book by Vladimir Rykov and Dmitry Kozyrev, where Figure 6 is present, it is explained how to model the logic of a *DES* algorithm.

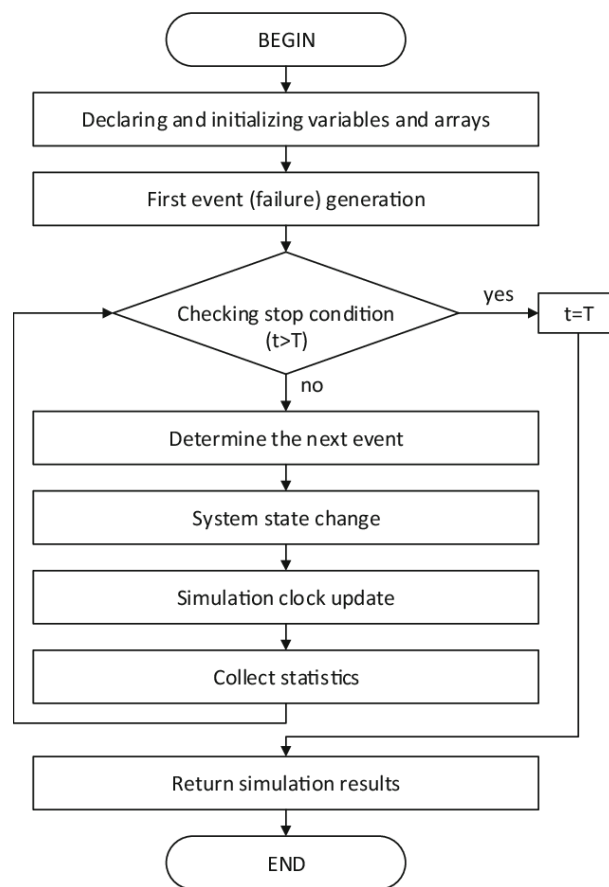


Figure 6: Simulation Engine Logic. [56]

2.4 TECHNOLOGIES

After analysing the following two main topics:

- clients with the possibility to check on their packages' state.

- several employees using different computers as input/access points.

We can conclude that the solution must be cloud oriented. Currently, the most common approach and possibly the one that offers more resources is to develop a Web Application. Related products use the same approach and with it, we can meet all requirements imposed by *Tipoprado*.

In a first phase, we will start by comparing different web development architectures, and later, study some technologies (programming languages/frameworks) to develop a maintainable and scalable platform, with the ability to add new features with ease.

2.4.1 Web Development Architectures

Three "main" architectures are widely used by developers/industries. These consist of **Monolithic** [7] vs **Service-Oriented Architecture (SOA)** [22] vs **Microservices** [6].

The monolithic architecture is seen as the traditional way of building web applications. Usually, it consists of a client interface, server backend and a database (all three main components in one single unit). All functions are managed in one place.

Typically, monolithic applications have one large codebase and lack modularity. For example, if developers want to update or change something, they are always modifying the same code base. This way, every time there is a new update, the whole stack is updated at once. This way, it might be difficult and take time to find and alter the respective code that needs adjusting. As it is all connected in one single module, anything that goes wrong will most likely affect the entire application.

A monolith's code can follow software architecture and code organisation patterns like **Model-View-Controller (MVC)** [61] or **Model-View-View-Model (MVVM)** [27]. Applications that use this type of architecture are mostly used to create **Minimum Viable Products (MVPs)**, also known as proof of concepts. However, when these applications start to grow, they usually become more sophisticated and complex. For this reason, a more modular approach is required for these platforms to be sustainable.

A **SOA** is essentially a collection of services that communicate with each other. Nevertheless, some means of connecting services to each other is needed, such as message protocols.

The microservice architectural style (or "fine-grained **SOA**"), is in short, an approach to develop a single application as a set of various small services, each running in its process and communicating with lightweight mechanisms, often an **Hypertext Transfer Protocol (HTTP)** [45] resource **Application Programming Interface (API)** [51].

These services are independently deployable, and there is a bare minimum of centralised management of these services. Each microservice can be written in different programming languages and use diverse data storage technologies.

According to Martin Fowler and James Lewis, one main reason for using services as components (rather than libraries, as seen in monoliths) is that services are independently deployable ("*Componentization via Services*" [15]). Having an application that consists of multiple libraries in a single process, a change to any individual component results in having to redeploy the entire application. However, if that application is decomposed into various services, you can expect many single service changes to only require that service to be redeployed.

"You want to use *Node.js* to standup a simple reports page? Go for it. C++ for a particularly gnarly near-real-time component? Fine. Do you want to swap in a different flavour of a database that better suits the reading behaviour of one component? We have the technology to rebuild him.

Of course, just because you can do something, doesn't mean you should - but partitioning your system in this way means you have the option." - Fowler and Lewis, *Decentralized Governance* [15].

Checking Table 2, we can observe a "final" comparison on web development architectures. This table will help make a decision, based on *Pros and Cons* of each architecture, on what to use in the platform.

Architectures	Pros	Cons
Monolith	Simpler development and deployment Fewer cross-cutting concerns Better performance	Codebase gets cumbersome over time Difficult to adopt new technologies Limited agility
SOA	Reusability of services Better maintainability Higher reliability Parallel development	Complex management High investment costs Extra overload
Microservices	Easy to develop, test, and deploy Increased agility Ability to scale horizontally	Complexity Security concerns Different programming languages

Table 2: Web Development Architectures comparison. [2]

2.4.2 Web Development Languages/Frameworks

As for Programming Languages/Frameworks used to develop web applications, we can observe some of the most used.

Java [24], by being a versatile programming language, is popular in the enterprise development environment. One of the main advantages is the ability and simplicity to combine and

rely on its native tools and frameworks for creating applications. A great base of documentation and support across the entire spectrum, from simple to more complex applications, makes *Java* very appealing. One of the most used web development framework is *Java Spring* [60].

Python [43] can be used to build server-side web applications. While a web framework is not required, it's rare not use one speed up development. *Django* [8] is a high-level *Python* web framework that encourages rapid development and clean, pragmatic design. It includes dozens of extras to handle common web development tasks. *Django* takes care of user authentication, content administration, site maps, and many more tasks — right out of the box.

PHP [41] is a popular general-purpose scripting language that is especially suited to web development. Has been widely ported and can be deployed on most web servers on almost every operating system and platform.

Node.js [38] relies on an entity-relationship which helps to ensure that the application runs seamlessly, through data systematisation, breaking logic into modules, processing valuable insights from logs and dividing the code. Also helps build scalable web apps.

Ruby on Rails [54] achieves high productivity and makes it a common choice for startups who aim for a running start. Designed to be user-friendly and quite easy to use, with short and readable code. Unfortunately, that sometimes means lower efficiency compared to other programming languages – but it also means higher productivity.

Elixir [11], inspired by **Erlang** [12], a language developed back in the '80s by *Ericsson* for better development and stability of telephony applications, and that was made open source in 1998 [23].

Erlang was made to allow phone switch system to run continuously and supports *hotswapping*: software updates while the software is running. *Elixir*'s author José Valim said that he loved *Erlang*, but noticed some things could use a bit of improvement [37]. This way, all areas addressed in *Erlang* are also present in *Elixir*, together with its package management system, macros, an easy to use build tool and *Unicode* handling.

ReactJS [50] is a *JavaScript* [25] library to develop frontend applications. *React* makes it painless to create interactive **User Interfaces (UIs)**. With the capability to design simple views for each state in an application, *React* will efficiently update and render only the components that need to be updated when data changes. Declarative views make the code more predictable and easier to debug.

It is also possible to build encapsulated components that manage their state, and then compose them to make complex **UIs**. Since component logic is written in *JavaScript* instead

of templates, we can easily pass data through the app and keep the components' state out of the [Document Object Model \(DOM\)](#), [39].

2.5 SUMMARY

Regarding the research on related products, the platform that *Tipoprado* needs, differs from the others as it is visible in Table 3. Tracking systems like *DHL* and other delivery businesses offer the possibility to clients track their packages, but this kind of tracking is not the same. Usually, these products are also custom made for that company and not available to interested parties. *Productoo* focuses on the production line; however, it doesn't enable client tracking and is aimed for different pipelines.

Products	Client Tracking	Analysis Report	Production Pipeline
DHL Tracking	✓	?	✗
HiveTracker	✗	✓	✗
Productoo	✗	✓	✓
Tipoprado	✓	✓	✓

Table 3: Related products comparison.

As for tracking methods, the conclusions are the following:

- There is no "one size fits all" tracking method. Each problem needs to be analysed before making any decisions;
- The one that applies best to this case is barcode scanners.

On Queuing Theory matter, a lot of work has been made about the optimisation of businesses/industries through the use of it. Checking M. Marsudi and H. Shafeek's work, on optimising a company's production line it is possible to understand what kind of measures are retrieved from the business data, and what information each one can provide [32]. This way, it is expected to serve the future platform with some of these vital statistics.

However, this kind of research/work, as many others found, it is mostly a "one timer", that means, a single analysis on the business is made to solve its efficiency problems. In this case, *Tipoprado* wants to have a continuous analysis/status on its product pipeline.

Concerning the prediction methods observed above, currently, the use of a [DES](#) is the best solution to *Tipoprado's* problem, in detriment of Queuing Theory, using *Little's Law* or any arrival/service rate assumption based method, and [ML](#). A simulation engine can be *single-threaded* or *multi-threaded*. In both cases, there are several problems with synchronisation between current events. Chapter 3 will detail some synchronisation problems that might occur.

The pending event set is typically organised as a priority queue, sorted by event time. That is, we can order events (packages) according to its delivery dates, so regardless of the order that events are added to the event queue, they will always be removed in the same chronological order. Various priority queue implementations have been studied in the context of discrete-event simulation [26].

Finally, addressing the architecture and technologies, having the previous information in mind, and also some of the *Best Practices for Good Web App Architecture*, by Angela Stringfellow [3], such as:

- Solves problems consistently and uniformly;
- Is as simple as possible;
- Offers fast response times;
- Utilises security standards to reduce the chance of malicious penetrations;
- Does not crash;
- Heals itself;
- Does not have a single point of failure;
- Scales out easily;
- Allows for easy creation of known data;
- Errors logged in a user-friendly way;
- Automated deployments.

We decided that this platform will consist of an [API](#) backend, built with *Elixir*, using *Phoenix* framework and a *ReactJS* frontend.

It was opted to separate backend from frontend to follow up the Microservices architecture. With this choice, the application is ready for new modules in the future, such as an [IoT](#) solution for *Tipoprado*, or integration with other [ERP/MES](#), which are present in the similar products previously studied.

Regarding backend, the [API](#) will be *RESTful* [53]. Is based on [Representational State Transfer \(REST\)](#) technology, an architectural style and approach to communications often used in web services development. *RESTful* systems aim for fast performance, reliability, and the ability to grow by reusing components. These components can be updated through time without affecting the system as a whole, even while it is in production. The term [REST](#) was introduced and defined by Roy Thomas Fielding in his doctoral dissertation. [53]

Elixir was chosen because it is a language that runs on the *Erlang* [Virtual Machine \(VM\)](#). This way, it has all of the benefits of the battle-proven system, and it is possible to use the existing *Erlang* libraries with no performance penalty. Another strong point for *Elixir* is concurrency, which, in most languages, is a bit of a pain, not only being dangerous and hard to achieve synchronisation. In *Elixir* it is effortless to create a new process, and it performs very well. It is a core feature of the platform and does not require a separate library. Note that *Erlang/Elixir* processes are not native processes; they are much more lightweight and are scheduled by the *Erlang* *BEAM* virtual machine. [13]

Phoenix [40] is the most popular web framework for *Elixir* right now. One great feature is the real-time capabilities with channels between *Javascript* on the client and *Elixir* on any of the servers in the cluster. Every single visitor to a website can have its process on the server and its real-time connection, achieving a sort of "hybrid" platform. This enables some possibilities that are not present in other common web frameworks.

Virtual [DOM](#) in *ReactJS* makes user experience better and developer's work faster. With this, it updates even minimalist changes applied by the user but does not affect other parts of the interface. (Check *reconciliation* in *ReactJS* documentation, *What is the Virtual DOM?* [48]). Managing updates is easy for developers because all *ReactJS* components are isolated, and a change in one doesn't affect others. This allows for reusing components that do not produce changes in and off themselves to make programming more precise, ergonomic, and comfortable for developers.

It is an open-source library developed by *Facebook* and in constant update. At the moment, *ReactJS* is in 3rd place of *Github's* most starred repositories, with over 141k stars [16]. And more than 1300 open-source contributors are working with the library.

THE PROBLEM AND ITS CHALLENGES

After all the research endured for the *"State of the Art"* Chapter, it's necessary to describe the problem and requirements imposed by *Tipoprado*. Some questions (business requirements) were already explained/addressed before, such as:

- Client interactions;
- Production methods;
- Pipeline characteristics.

About the specific tasks that the platform must respond to, some meetings were arranged with the company responsible in a first approach phase. The refined requirements were separated in different categories (user requirements), to improve development efficiency.

Users:

- **Admin:** has access to all *CRUD*¹ operations and extra features;
- **Employee:** has limited access to *CRUD* operations and limited to the main features of the platform;
- **Client:** only has access to public information such as own packages, and related. (no registered account necessary)

Employee Features:

- Register new packages/orders in the platform;
- View, edit and delete packages;
- Register/Complete/Pause/Resume a package state inside the production line;
- Mark packages as concluded;

¹ *Create, Read, Update, Delete* [33]

- Consult a "production calendar" with package delivery dates, and automatic predicted dates.

Client Features:

- Consult own orders status/history;
- Answer a satisfaction survey after package conclusion.

Admin Features:

- All **Employee** features;
- Consult Statistics/Performance measures about the production pipeline and other business sensitive data;
- Consult reviews given by clients about their orders;
- Platform settings:
 - User management (add/remove employee accounts);
 - Pipeline management (add/edit places of the production line);
 - Client satisfaction survey question management (add/edit questions of the automatic package survey).

Concerning how the system must behave to fulfil the company's expectations (system requirements), we got the following:

- Clients must receive an email when a package is registered or concluded. This email needs to forward them to the package public information page on the platform, or to a satisfaction survey;
- Package state management inside the pipeline must be ready for manual and barcode scanners input. (This process has to be simple and as much little time consuming as possible).

As for the statistics that will be calculated and shown to *Admin* user, consists of a set of performance measures used in queuing theory and other metrics that *Tipoprado* requested. Composing the following list:

- Time evolution of the number of **registered packages**;

- Time evolution of the number of **completed packages**;
- Time evolution of percentage of **on time package deliveries**;
- Overall percentage of **on time package deliveries**;
- Number of **packages active in a place** (workstation);
- Number of **packages stopped after being active in a place** (workstation);
- Heatmap of number of **packages passing on a place**;
- Mean **service time by place**;
- Mean **waiting time by place X to place Y**;
- **Place OEE**;
- Time evolution of **client rating reviews**.

The final goal of this data treatment is to be able to generate automatic reports analysing each one of these measures and inform, as simple as possible, *Tipoprado's* responsible about positive or negative evolution, according to the previously selected time counterpart.

3.1 PROPOSED APPROACH - SOLUTION

3.1.1 Solution Overview

Collecting and analysing all information about the desired platform an architecture scheme is necessary to complete the development roadmap. As said before, this web application will be divided into different modules initially (frontend and backend). This allows an easy affix of separate services such as a [Simple Mail Transfer Protocol \(SMTP\) \[29\]](#) Server for email delivery, or a possible [IoT \[52\]](#) solution in the future (also referenced above).

Since the production pipeline status (adding and completing packages on a workstation), is the "critic zone" with constant data flowing through frontend and backend, *websockets* [21] will be used. The use of *websockets* enables an almost "real-time" feel, ensuring that every endpoint (computer of the company) has the same visual state. The rest of the information will be passed with [HTTP](#) requests through a [REST API](#). This way, an unauthenticated user (**Tipoprado's end client**), can not use *websockets* through the frontend module.

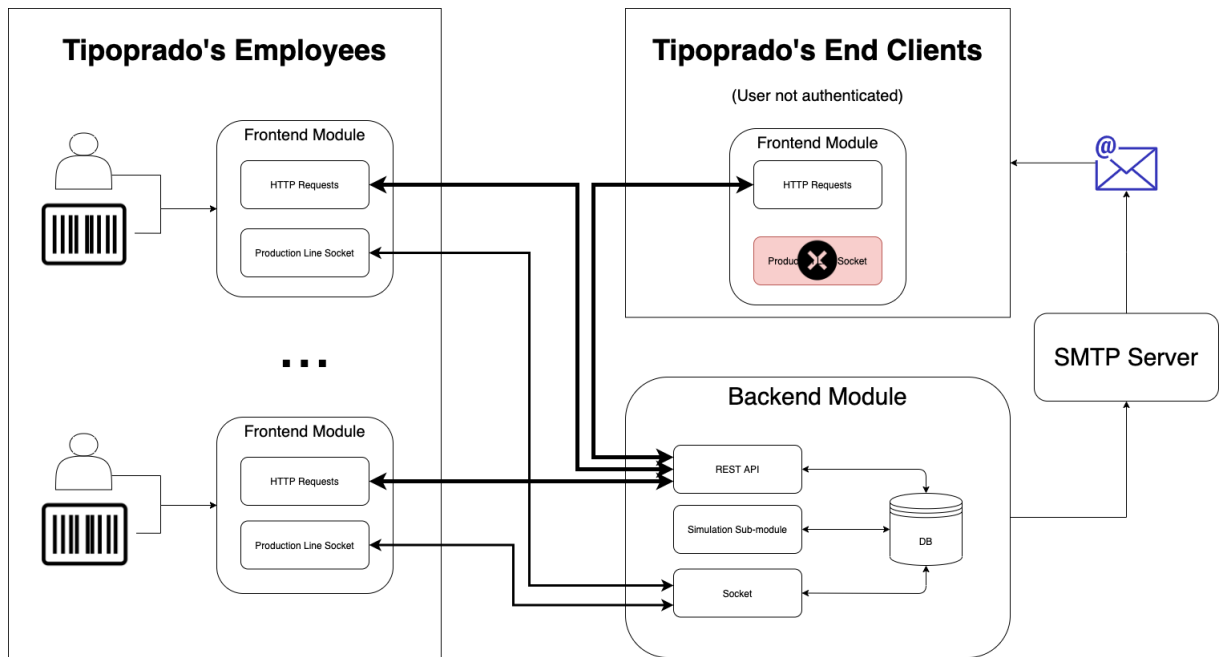


Figure 7: Solution overview scheme.

A *Tipoprado's* employee, authenticated user, can add new information by manual input or barcode scanners. Also, can interact with backend module through [HTTP](#) or *websockets* (for pipeline information).

In Figure 7, we see the different entities that will use the platform inside the squared boxes. The round boxes represent the various modules that compose the application.

3.1.2 Simulation Algorithm for predicting delivery dates (Sub-module)

Considering all the requirements and the architecture defined, the simulation sub-module will constitute the *"the cherry on top of the cake"*, and it deserves a preview of the algorithms/techniques that will be used.

As it was referenced in the *State of the Art*, the algorithm used will be based in Figure 6. This way, it is necessary to define:

- how data used in the simulation will be retrieved (list of packages and places/workstations that compose the pipeline);
- how the simulation engine will keep track of events and data updates

To give the best user experience, each time a user consults the integrated calendar, it should have an overall idea if packages are predicted to be delivered on time or not, and the respective predicted date. This means that this information will be asked for, fairly frequently, corroborating Figure 7 and the necessity for this to be a separate sub-module.

With the possibility of a separate component from the [API](#), instead of calculating each time an [HTTP](#) request is received, a simulation will run periodically (starting with a 1 hour period) and update the respective database table.

Given that a package has a defined path, and it may vary, each workstation also has a different number of "items" to process, receiving from one and delivering to another workstation. It can also be the starting or finishing point of a package. So, we can think of a workstation as an individual actor inside of the simulation engine. (Figure 8).

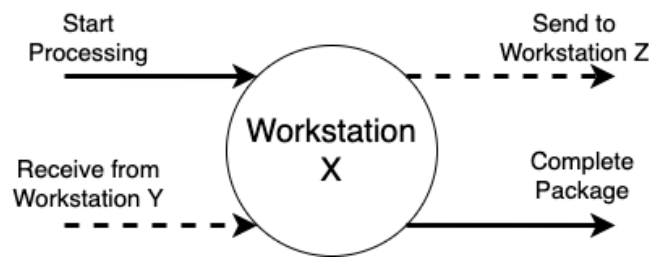


Figure 8: A pipeline workstation inputs and outputs.

Continuing with the same thought, we identify some *Actor Model* [20] programming characteristics. In response to a message it receives, an actor can:

- make local decisions;
- create more actors;
- send more messages;
- determine how to respond to the received message.

Actors may modify their private state, but can only affect each other indirectly through messaging. An actor is also a light-weight entity unlike [Operating System \(OS\)](#) processes or threads, and was introduced in 1973 by Carl Hewitt. [20]

One of the selling points of *Elixir* is its support for concurrency. The concurrency model relies on **Actors**. So, implementing the initial idea, where each workstation is an actor part of the simulation, should not be a problem with the selected language for backend development.

After a small research, we can verify that actor-based programming is present in some simulation problems, and that constitutes a viable solution for this platform. *Citybound* is a city building game that uses microscopic models to vividly simulate the organism of a city arising from the interactions of millions of individuals. This simulation engine is based on actors and message passing. It is implemented in *Rust* [55]. According to Anselm Eickhoff, creator and developer of *Citybound*, the main inspiration for his approach was his previous experience with *Erlang*. [4]

Having this into consideration, the first take on a pseudo algorithm can be the following:

1. Periodically, an *Actor* is spawned. This will be the *Simulation Manager* and is responsible for spawning an *Actor* for each workstation that composes the pipeline;
2. Each of these actors will collect the list of packages in their queue, and order it by the delivery date. Announcing to the manager when the next event will occur. The next event is calculated by the processing time of the first package in queue;
3. When the manager gets all announcements, orders them, and advances clock time to the next event, warning the respective actor where the event will occur;
4. Actor receives a message from the manager and completes "event" (package processed). If the package is concluded, it will update the database value for expected delivery time. Else, it will send a message with the package to the next workstation (actor). It also warns the manager, that needs to wait for a next event update from itself and the next workstation;
5. The intervening workstation actor will recalculate the next event time and announce it to simulation manager (*null* in case that there is no package in the queue), to continue the simulation;
6. The simulation stops when the manager doesn't have more "*next events*".

Now, we can formulate a more fine-grained approach to this algorithm.

The pseudo-algorithm bellow, represents the flow of interactions in an imperative approach, with functions `receiveMessage()` and `sendMessage()` as being blocking functions.

However, this is just a method to better explain how the simulation will work. As said, the actual implementation will be *Actor Model* based, transforming this algorithm in a distributed and asynchronous one, and that's why some of the announcements are needed (intended to keep message order between all actors). This can be seen as a simple sequential consistency problem [30]. We must ensure that the Manager only fast forwards the clock when it has all the updated values from the actors affected by the previous event, keeping "timed events" from overlapping. The existence of a centralised actor (simulation manager) simplifies this problem and potentially increases efficiency due to less metadata and extra messages in a peer-to-peer case.

The portions of code (logic) that will be executed separately from the simulation manager are outlined by "**actor *p* execution**" blocks.

Data: Sim_{Man} as the *Simulation Manager*

begin

$P_{List} \leftarrow getWorkstations()$

for $p \in P_{List}$ **do**

$spawnActor(p)$

actor p **execution**

$Pack_{List} \leftarrow getPackagesInQueue(p)$

$orderByDeliveryDate(Pack_{List})$

$nextEvent \leftarrow getNextEvent(Pack_{List})$

$sendMessage(Sim_{Man}, nextEvent)$

end

end

while $\exists p \notin receivedEvent_{List}$ & $nextEvent \leftarrow receiveMessage()$ **do**

$insertOrdered(receivedEvent_{List}, nextEvent)$

end

while $receivedEvent_{List} \neq []$ **do**

$\{p, time\} \leftarrow popHead(receivedEvent_{List})$

$sendMessage(p, time)$

actor p **execution**

$newUpdate \leftarrow receiveMessage()$

if $isFromManager(newUpdate)$ **then**

$package \leftarrow popHead(Pack_{List})$

if $isCompleted(package)$ **then**

$updateDBDeliveryTime(clockTime)$

$sendMessage(Sim_{Man}, [])$

else

$m \leftarrow getNextWorkstation(pack)$

$sendMessage(Sim_{Man}, [p, m])$

$sendMessage(m, pack)$

end

else

$insertOrdered(Pack_{List}, newUpdate)$

end

$nextEvent \leftarrow getNextEvent(Pack_{List})$

$sendMessage(Sim_{Man}, nextEvent)$

end

$affected_{List} \leftarrow receiveMessage()$

for $p \in affected_{List}$ **do**

$nextEvent \leftarrow receiveMessage()$

$insertOrdered(receivedEvent_{List}, nextEvent)$

end

end

$terminateSimulationActors()$

end

DEVELOPMENT

4.1 DECISIONS

Starting the development phase, it is important to make a summary of the technology decisions, previously referenced.

Regarding Backend, *Elixir* is the selected language, together with *Phoenix* to develop a REST API, and we already went through their great advantages. To simplify development and smooth the process, we will also use *Ecto* [10]. It is a database wrapper and query generator for *Elixir*. *Ecto* provides a standardised API and a set of abstractions for talking to all the different kinds of databases so that *Elixir* developers can query whatever database they're using by employing similar constructs.

As for Frontend, since component development/design is not the goal of this dissertation, the use of *Semantic-UI* [59], together with *React*, takes an important role. It offers a declarative API, controlled components [47], and other features. Looking to the list of components offered by *Semantic-UI* is visible that data visualization components, such as plots and charts are missing. Like it was explained, data analysis by the company managers simply and intuitively is a key feature of the whole platform. This way, after some research, the *React-vis* [49] library was chosen. Developed by *Uber*, *React-vis* is *React*-friendly, high-level, customizable and expressive.

Recalling the tracking method (barcode readers), to update each package state inside the production line, it is crucial to say that one of its main benefits, is that it allows the user to scan barcodes into any application as if the data was being keyed in by the user. It means, that a common barcode reader works like a keyboard. Just scan a barcode and the data will appear wherever the cursor is placed. This way, it's not necessary for any extra library/software to support this feature.

4.2 IMPLEMENTATION

4.2.1 Database and API

Firstly, given the requirements stated in Chapter 3, the database conceptual model, and consequently, the logical model was developed, which resulted in the following Figure 9.

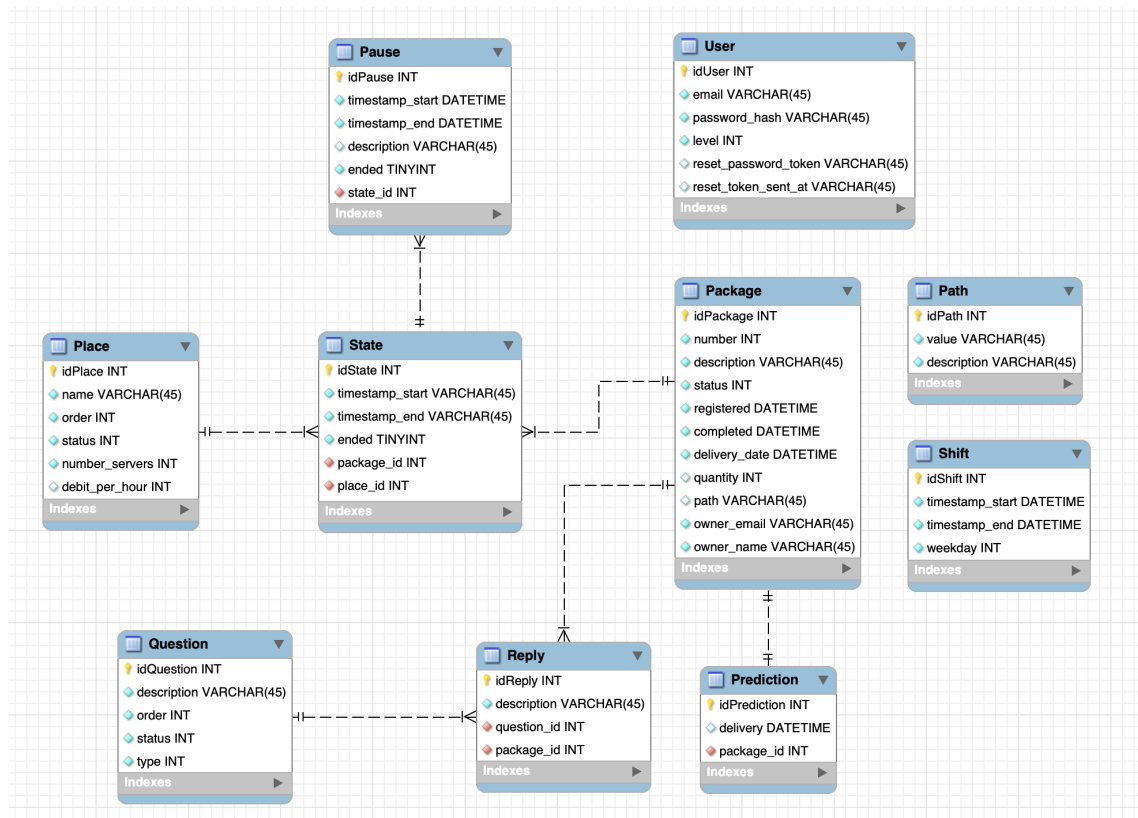


Figure 9: Database Logic Model.

Some important notes on the database:

- **User:** The User table is only to represent authenticated users (admins and employees). Public users, such as Tipoprado's end clients, do not need authentication to consult their packages. Only required unique code, generated through Base 64 encoding of number (sent by email only to *Package's* owner_email);
- **Package:** quantity and path can be null. The path attribute is the expected path that the package will undergo in the production line. Used mainly for Simulation purposes. It was chosen to not make them required after talking with Tipoprado's employees who felt this could be too time-consuming and don't have a direct impact in the package tracking flow. The Path table serves only to store frequent or favourite

paths, and associate them with a description, to simplify a package's path definition process. They are not directly referenced in Package table;

- **Place:** `debit_per_hour` can be null. If this is the case it means that the type of work performed by the respective workstation is somewhat manual. Therefore, this attribute will be seen as unquantifiable;
- **Pause:** has a description attribute in case a package is paused due to a reason worth mentioning to the end client. This way, when a client consults a paused package, has the chance to see a reason. Pauses also enable a better understanding of production methods. Will see later why;
- **Question:** questions to appear in the satisfaction survey for each package;
- **Shift:** used to represent the working (business) hours of the company. Used to measure performance, and simulate delivery dates with better result approximations.

Using *Elixir* and *Ecto*, it is very simple to define the table migrations and resources (*maps*) to be used across the whole code. For example the user model (not table migration), bellow:

```

1  schema "users" do
2    field :email, :string
3    field :password_hash, :string
4    field :level, Pipe.EctoEnums.TypesEnum, default: 1
5    field :password, :string, virtual: true
6
7    field :reset_password_token, :string
8    field :reset_token_sent_at, :utc_datetime
9
10   timestamps()
11  end

```

Having explained the basis of the platform database will look into the operations that each user (admin, employee, public) can perform to each resource, according to the requirements specified. This way, Table 4 was generated. With it, it is possible to retrieve the first part of the REST API routes.

Note: Depending on the type of user or type of action, the result can be different. For example, a package *index* for an authenticated user has a different output than an *index* performed by an unauthenticated one. The resource operated over is still the same, but each user sees a treated result for its type.

Resource	Admin	Employee	Public
User	index, create, show, delete	index, show	—
Package	index, create, show, update, delete	Equal to Admin	index, show
Place	index, create, show, update	index, show	—
State	index, create, show, update, delete	Equal to Admin	—
Pause	create, update	Equal to Admin	—
Question	index, create, show, update	index, show	index
Reply	index, show	—	show, create
Shift	index, create, show, update, delete	index, show	—
Path	index, create, delete	Equal to Admin	—

Table 4: Resources' *CRUD* operations by user type.

Looking at the platform requirements again, we can conclude that some of the features are still missing from the basic *CRUD* operations. So, the rest of the available routes, that don't operate over/return a unique database model, and therefore require their own Controllers and Views, can be seen below.

- Production Line, (*Pipeline*) route is somewhat "duplicated", since the main access point is through *websockets*, though it is still possible to request it through the [API](#). Nevertheless this route returns a list of **active** package states, grouped by current workstation;
- *Statistics* receives a query identifier, and it is also possible to pass date periods as filter;
- *Report* generates an automatic summary of a default period of time, or can receive a period as filter like *Statistics*;
- *Calendar* receives year and week as identifiers, in order to show all packages due to deliver that respective week; Week perspective is the only available. Filter to view already completed packages is also an option.

```

1 get "/pipeline", PipelineController, :index
2 get "/calendar/:year/weeks/:week", CalendarController, :show
3 get "/stats/:query", StatController, :show
4 post "/report", ReportController, :create

```

The authentication process is performed through the use of [JSON Web Token \(JWT\)](#) [28], handled by the *Elixir* library *Guardian* [17]. With this library it is very simple to manage and authenticate users/connections, having access to two key functions like:

- `Guardian.Plug.current_resource(conn)`: returns a User resource, through the token present in the connection's headers;

- `Guardian.encode_and_sign(user)`: encodes a `User` resource, and returns the respective `JWT`.

4.2.2 Critical Zone and Websockets for real-time interactions

As it was explained before, there is a "critical zone" in the platform where it's important to ensure that every authenticated user is seeing the same thing (some sort of real-time experience). This page is the one where employees will track packages inside the production line.

- **Backend Perspective**

The problem with using the current `API` routes is that to keep the wanted equal visualization is necessary to keep requesting to the backend server. However, this solution is very inefficient and can cause unnecessary load. Besides, it still is a "pseudo-real-time" interaction.

All of this can be overcome with *websockets*. And *Phoenix* has all the right tools to implement them. `Phoenix.Socket` is a *socket* implementation that multiplexes messages over *channels*. Once connected to a *socket*, incoming and outgoing events are routed to channels, this way, it possible to easily abstract different "lobbies" inside the same *socket*.

Bellow, we can observe the *socket module*, more specific on line 3, events are routed by topic to channels. *Socket* parameters are passed from the client and can be used to verify and authenticate a user. *Socket id's* are topics that allow us to identify all *sockets* for a given user, returning `nil` makes the socket anonymous.

```

1  defmodule PipeWeb.UserSocket do
2    ## Channels
3    channel "pipeline:*", PipeWeb.PipelineChannel
4
5    def connect(_params, socket, _connect_info) do
6      {:ok, socket}
7    end
8
9    def id(_socket), do: nil
10   end

```

In the code snippet above, there is no user authentication. That is done inside the respective *channel*. Every time a user joins a *channel* needs to choose which particular topic

he wants to listen to. The topic is just an identifier, but by convention, it is often made of two parts: `topic:subtopic`. By using a splat (the `*` character) as the last character in the topic, allows matching on all topics starting with a given prefix pattern.

Bellow, there is the `join` function, that receives as an argument, the *channel* the user wants to join and the user's `JWT` token for authentication. If the token is valid, it will be granted access to the *channel*, or else the user will receive an unauthorized reply. (Line 14 of the code snippet).

```

1  def join("pipeline:lobby", %{"token" => token}, socket) do
2    case Socket.authenticate(socket, Pipe.Guardian, token) do
3      {:ok, authed_socket} ->
4        {:ok, authed_socket}
5      {:error, _} ->
6        send(self(), :unauthorized)
7        {:ok, socket}
8    end
9  end
10
11 def handle_info(:unauthorized, socket) do
12   push(socket, "unauthorized", %{})
13   {:noreply, socket}
14 end

```

Seen above, the function `handle_info(msg, socket)` handles received messages from other processes or the *channel* itself (like in line 8 with `send()` to `self()`). Likewise, there is an `handle_in` function that is responsible to handle incoming messages to the *channel* via the *socket*. For example, handle incoming message to request full production line (occurs after successful join).

```

1  def handle_in("full_pipeline", _message, socket) do
2    pipelines = Production.get_states_pipeline()
3    view = PipelineView.render("index.json", pipelines: pipelines)
4    push(socket, "full_pipeline", %{data: view})
5    {:noreply, socket}
6  end

```

Looking at `handle_in`, it takes as arguments a "message event", the "message" itself, and the connected *socket*. In this case, as it isn't necessary to send any message, the "event" name

is sufficient. The *pipeline* is built, transformed into *JSON* and then sent back to the socket through `push()`. This function also needs an "event" name, for the frontend client to be able to treat the given message expectedly.

This way, there are the following "events" to be handled:

- `full_pipeline`: described above;
- `new_state`: receives as message the new state data, equal to `post /states`;
- `delete_state`: receives as message the state id to delete, equal to `delete /states/:id`;
- `complete_state`: receives as message the state id to update its status, equal to `patch /states/:id`;
- `new_pause`: receives as message the new pause data (marks a state as paused in production line), equal to `post /pauses`;
- `resume_pause`: receives as message the pause id to update (marks a state pause as completed - resumed), equal to `patch /pauses/:id`;
- `complete_package`: receives as message the current state id to complete and the package id to also mark as complete. This update is atomic, therefore, if one of this updates fails, the other fails too;
- `default`: default handler that pattern matches all the rest of "message events". It returns an error for invalid event.

Excluding the events `full_pipeline`, `default` and error handling, every backend response is broadcast to all sockets connected. This way, every time a new event occurs, every employee will get that update in a near real-time experience. As it is visible in the code snippet below, this function receives the connected socket, the event message and data necessary for the frontend client to treat it the correct way. These hooks receive a function callback as an argument, to treat the response obtained from the backend.

```
1 broadcast(socket, "<event_message>", %{data: view})
```

- **Frontend Perspective**

On the frontend side, it was used the *Phoenix Channels JavaScript client*. This way, it is possible to keep the same level of abstraction used in the backend, even with two completely different languages, making the overall development and maintenance much easier.

When the page is open and the *React* component mounts, the client tries to connect to the socket with empty parameters. Afterwards, *socket* hooks are defined for handling any error or a successful connection established.

```
1  componentDidMount() {
2    this.socket = new Socket(
3      `${process.env.REACT_APP_SOCKET}/socket`,
4      {},
5    );
6    this.socket.connect();
7    this.socket.onOpen(this.socketRequestInfo);
8    this.socket.onError(this.handleErrorSocket);
9  }
```

In case of a successful connection to the socket, the callback function then tries to enter the defined *channel*, `pipeline:lobby`, passing the `JWT` token as a parameter for authentication, as seen in the backend side. *Channel* hooks are also defined for error handling, in this case, and call a custom defined function `registerSocketMessages` that defines hooks for all valid "events" seen before. A hook registers a handler with a function callback to be executed when a specific "event message" arrives from the other end of the socket, homologous to what happens in the backend.

Then, the "event" `full_pipeline` is pushed to the *channel* and the rest we've seen it already. When the response arrives, the hook has already been registered and the callback will be executed.

```

1  socketRequestInfo() {
2    this.channel = this.socket.channel('pipeline:lobby', { token:
      localStorage.getItem('token') });
3    this.channel.join();
4    this.channel.on('unauthorized', this.handleUnauthorized);
5    this.channel.onError(this.handleErrorChannel);
6    this.registerSocketMessages();
7    this.channel.push('full_pipeline', null);
8  }

```

In Figure 10, it is visible the main layout for this page. The design is a continuous development together with *Tipoprado's* employees, to enable an easy registering and manipulation of package states. Spending the least time possible with inserting new data, was the main concern with this page.

For every action performed, or "event message" there is a confirmation modal, to guaranty some sort of a second step and reducing mistakes caused by neglect. The backend also deals with possible errors that may pass through frontend verification.

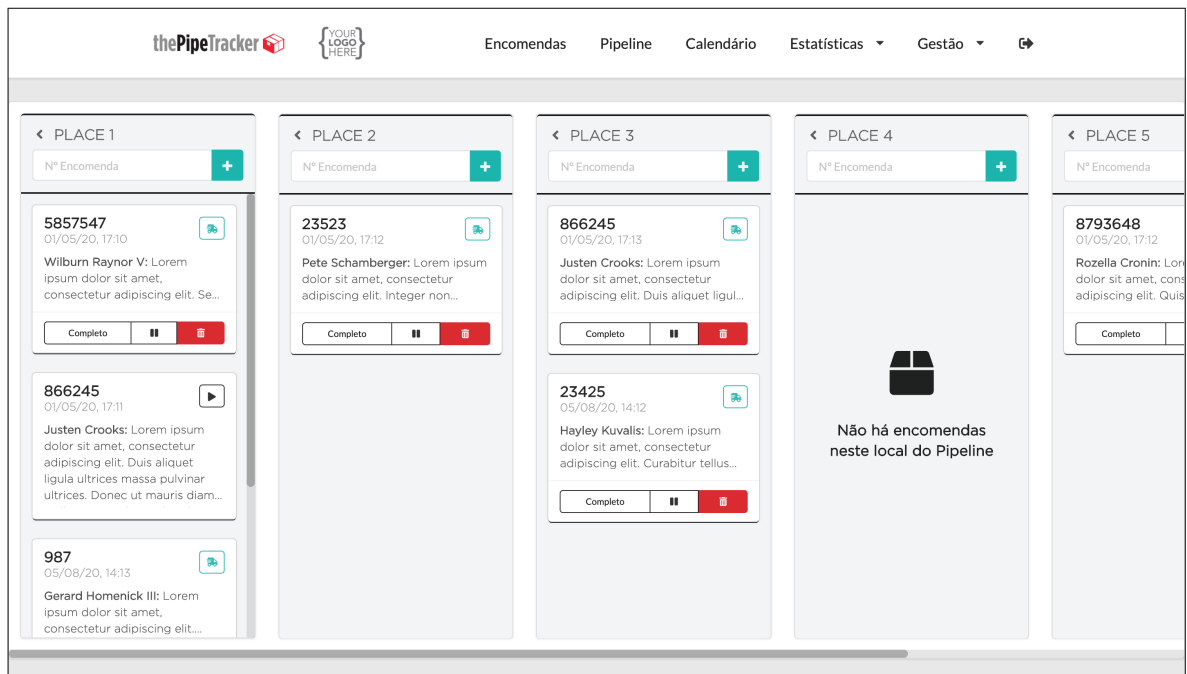


Figure 10: Production Line page for package tracking.

4.2.3 Statistics and Metrics for production analysis

All statistics and metrics, described in Chapter 3, were implemented, however, we will only look in detail to the two most complex and interesting. Final results of every statistics page can be found in Figures 25, 26, 27 and 28 of [Other relevant Frontend pages](#).

- **Mean waiting time in queue**

This statistic demonstrates the mean time a package stays in a queue before starting a given process (enter a new workstation). Due to the fact of variability of products, packages waiting in a workstation queue, can have different predecesing paths (Figure 5). So, it is necessary to take into account all the possible combinations:

$$\alpha \rightarrow \beta, \text{ for } \alpha, \beta \in \text{Workstation_List} \quad (1)$$

To calculate this, it is necessary to get all packages registered between the filter dates and their state list. Each state of the list represents when a package entered and left a given workstation. This way, if we travel through this list and calculate the time passed after a package left a *Workstation A* and started the next process (*Workstation B*), we get the time difference of two workstations for a package (time spent in queue $\text{workstation}_A \rightarrow \text{workstation}_B$).

Doing this for all the packages, we obtain all the possible path combinations that exist in the company's production pipeline.

In the following code snippet, it is visible the function that produces the list of time queue's time of a given package state list.

- `previous` is a tuple that is composed by the workstation order number and the timestamp of when it ended processing in that place;
- `next` will be `previous` in the next recursive call.

The edge case where the time difference is less than zero is taken into account, because sometimes, due to production characteristics, a package could start being processed in a workstation before being completed finished in the previous one. This would generate a negative number, and produce wrong results in the long run.

```

1 defp queue_time_diff([], _previous, times), do: times
2 defp queue_time_diff([state | tail], previous, times) do
3   next = {state.place.order, state.timestamp_end}
4   time_diff =
5     Timex.diff(state.timestamp_start, elem(previous, 1), :seconds)
6     / 3600
7   time_diff = if time_diff < 0 do 0.0001 else time_diff end
8   queue_time_diff(
9     tail,
10    next,
11    [{elem(previous, 0), state.place.order, time_diff} | times]
12  )
13 end

```

The function above is tail-call recursive. Tail-recursive functions are considered better than non-tail-recursive functions as tail-recursion can be optimized by the compiler. The idea used by compilers to optimize tail-recursive functions is simple since the recursive call is the last statement, there is nothing left to do in the current function, so saving the current function's stack frame is of no use. This way, guaranteeing that a stack overflow will never occur.

The output is then grouped by workstation and the mean values are calculated and the resulting output is a list of tuples {previous_workstation, next_workstation, mean_waiting_time}.

On the frontend side, the data received is transformed into a matrix like the one below.

$$\mathbf{M} = \begin{matrix} & \begin{matrix} a & b & \dots & n \end{matrix} \\ \begin{matrix} a \\ b \\ \dots \\ n \end{matrix} & \begin{pmatrix} q_{aa} & q_{ab} & \dots & q_{an} \\ q_{ba} & q_{bb} & \dots & q_{bn} \\ \dots & \dots & \dots & \dots \\ q_{na} & q_{nb} & \dots & q_{nn} \end{pmatrix} \end{matrix}$$

Where:

$$q_{aa} = \text{mean_waiting_time}(a \rightarrow a)$$

$$q_{ab} = \text{mean_waiting_time}(a \rightarrow b)$$

$$q_{nn} = \text{mean_waiting_time}(n \rightarrow n)$$

The first argument of mean_waiting_time represents the previous workstation and the second argument, the next one.

To represent this information, as mentioned at the beginning of this chapter, *React-vis* was used and the *Heatmap Series* the type of plot selected.



Figure 11: Heatmap for mean times in the production line queues.

The color ranges are the following:

$$\begin{aligned}
 [\text{green}; \text{yellow}] &=]0; \text{max} * 0.45] \\
 [\text{yellow}; \text{red}] &=]\text{max} * 0.45; \text{max} * 0.95] \\
 [\text{red}; \text{red}] &=]\text{max} * 0.95; \text{max}]
 \end{aligned}$$

These values represented are in *hours* and truncated to one decimal case.

- **Workstation OEE (Overall Equipment Effectiveness)**

To best understand what this plot shows, it is necessary to know what **OEE** is. It is the gold standard for measuring manufacturing productivity and measuring it is a manufacturing best practice. **OEE** is calculated as a combination of three different metrics.

- **Availability (A):** takes into account unplanned and planned stops. A score of 100% means the process is always running during the planned production time;
- **Performance (P):** takes into account slow cycles and small stops. A score of 100% means that when the process is running it is running as fast as possible;
- **Quality (Q):** takes into account defects. A score of 100% means there are no Defects (only Good Parts are being produced).

This way $OEE = A * P * Q$, and a score of 100% means the workstation is manufacturing only good parts, as fast as possible, with no stop time.

Due to production methods and some platform limitations, the number of defects is not counted and stored. This way, we will only use $A * P$.

The first step is to calculate the availability of a workstation. For this, we need to get the expected production time and the real production time for the selected date period.

- The expected production time is the sum of all business hours (shifts defined by the admin in the platform settings) for the selected period;
- The real production time is the sum of all package states that were active in that workstation during the same period.

After having the two, since the workstation might have more than one server, and therefore should be producing $total_time * n_servers$ we multiply the total time by the number of servers and then obtain the relation of *real/expected*.

```

1 def utilization_factor(min_date, max_date) do
2   total_production = total_production_time(min_date, max_date)
3
4   Stats.real_time_place(min_date, max_date)
5   |> Enum.map(fn x ->
6     {elem(x, 1), elem(x, 0), calc_work_time(elem(x, 2)), elem(x, 4)}
7   end)
8   |> Enum.map(fn x -> real_expected_relation(x, total) end)
9 end
10
11 defp real_expected_relation({id, name, real, n_servers}, total) do
12   if total * n_servers != 0 do
13     {id, name, real / (total * n_servers)}
14   else
15     {id, name, 0}
16   end
17 end

```

In the end, we get a list of tuples identifying each workstation and with respective relation of availability.

As for the performance metrics calculation, a workstation performance is given by:

$$\begin{aligned} \text{real_debit} &= (\text{quantity_prod} / \text{prod_time}) / n_servers \\ \text{performance} &= \text{real_debit} / \text{expected_debit} \end{aligned}$$

The `real_debit` is calculated as an aggregation of all the `n` servers that the workstation offers, and the `expected_debit` is defined by the platform admin as the expected quantity produced per hour per server. However, there are some special cases where there isn't an `expected_debit` defined. This is due to the fact that some workstations represent manual and non-countable production rates since there are a lot of variables that influence this number. So, the platform admin can leave the `expected_debit` for that workstation as unquantifiable.

When this occurs, the approximate debit is calculated with the number of packages processed inside that time period and the `expected_debit` is the mean time duration of a package being processed in that respect workstation. Note that this is not an "official alternative", this is just a way to best approximate the [OEE](#) metric for such variate production methods of this industry that can't be automated or countable.

$$\begin{aligned} \text{approximate_debit} &= (n_packages / \text{prod_time}) / n_servers \\ \text{expected_debit} &= \text{mean_time_duration}(\text{workstation_id}) \\ \text{performance} &= \text{approximate_debit} / \text{expected_debit} \end{aligned}$$

In the end, in a very similar way of the availability metric, we get a list of tuples with the same composition. This allows easy workstation grouping and calculation of the final [OEE](#) values. On the frontend side, this time the *Bar Series* was the selected type of plot provided by *React-vis*, and the final result can be seen below in [Figure 12](#).

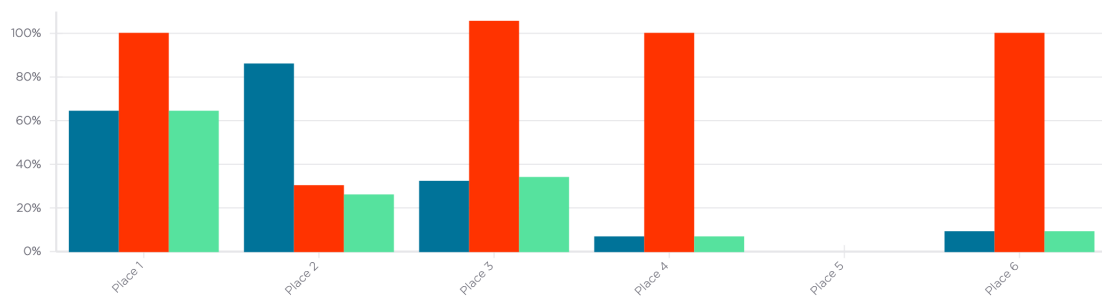


Figure 12: Vertical bar plot for availability, performance and workstation OEE.

4.2.4 Delivery dates - Simulation

Remembering Chapter 3 and Figure 8 together with the suggested algorithm, the first step for this implementation was to create the base structure of actors that intervene in the simulation. Therefore, it starts with the creation of the simulation manager and workstation actors.

Both simulation manager and workstations are `GenServers`¹. They are lightweight processes that can execute code, keep states and send messages to others, asynchronously.

`SimMan` (the simulation manager) starts with the following state, defined in the `SimMan.start_link/0` function:

```

1  pids: active_places,
2  clock: Timex.now(),
3  events: [],
4  awaiting: []

```

Where:

- `pids`: is a map which the keys are the company's workstation ids, and the values are the pids returned when an *actor* is started with `GenServer`;
- `clock`: will synchronise all workstation actors and keep track of each timestamp when a package advances in the simulation;
- `events`: the list of events (packages' new states) that each workstation actor will "report" to the simulation manager. The simulation ends when this list is empty. However, starting empty, the verification is post-initialization;
- `awaiting`: together with `clock` has the purpose to synchronise all of the simulation processes. It's a list of workstations ids that the manager is awaiting answer from. While this list is not different from empty, the simulation will not advance;

Also when calling `GenServer.start_link/3`, the option name is passed as an argument. This will register the manager process under the name of `SimMan`, and other processes (workstation actors) will be able to send messages to the manager through the registered name, without knowing its pid.

After the simulation manager and all of the workstation processes are started, the manager sends each workstation 3 messages with the following contents:

¹ <https://hexdocs.pm/elixir/GenServer.html>

1. Info about other workstation pids. Before, a workstation process only knows about itself, but to implement the algorithm described before, it has to know the others also. It was chosen to not register named processes for the workstations, so it is necessary to pass the extra data structure (pids map) between all of them;
2. Each workstation will receive its own mean processing time of a package, calculated by the manager previously. This is needed to give an approximation of an event timestamp for packages that don't possess info about quantities or workstations that don't have quantifiable output performance. (This was referenced before in **Workstation OEE**);
3. Active packages at the moment, in any state of processing (not started yet or mid processing). The complete list is sent to all workstation processes, and then each one will filter the packages that will start/are active in their own place. The best solution here would be that each process access the database and retrieve only the respective packages. But having about 15 processes (for this case) consulting the database, plus normal users, would most likely exceed the maximum number of database connections available in the plan that will be used in *Heroku* (20) [19] when deployed. Causing some delays and/or unexpected errors. So, this way all the work is done in memory with only one process (simulation manager) accessing the database.

With the *Erlang* and *GenServer* characteristics, a process has a **First In First Out (FIFO)** "mailbox", ensuring the ordering of the messages received by each workstation.

To calculate the mean processing time for each workstation, are used the same functions described in the **OEE** plot. Reusing code/functions simplifies all of this process.

The third message, active packages in the production line is the one to focus now. As it was referenced before at the beginning of this chapter, the same way quantity is not required when registering a new package in the platform, the production path it will go through can also be `nil`. This creates the following problem: "How to simulate all factory production if we don't know packages will be processed?".

The answer to this question is to obtain an approximate path for packages that don't have one defined. After some inquiries with the company's responsible and employees, to discover the best way to approximate those paths, the following pattern was discovered:

Recurring clients order packages with the same processing path most of the times. So if **Client A** requests a book/magazine, most likely, next time will order something similar but with different contents. However, processing methods remain the same.

This way, tracing a pattern of each client associated with a package will give us an approximated path for all those packages that don't have one defined and are entering

the simulation. Verifying this with some plots when in production would be interesting. Though, that won't be done for now.

```

1  defp send_active_packages(pids) do
2    packages = Models.list_all_active_packages()
3    grouped = Enum.group_by(packages, & !is_nil(&1.path))
4
5    approximate_paths = approximate_paths(packages)
6    with_path = Enum.map(grouped.true, & filter_rest_places/1)
7
8    without_path =
9      Enum.map(grouped.false,
10         & filter_rest_places(&1, approximate_paths))
11
12    total = with_path ++ without_path
13    # for packages that don't have enough info (path or client history)
14    filtered = Enum.reject(total, & is_nil/1)
15    Enum.each(pids,
16      fn {_place, pid} -> send(pid, {:packages, filtered}) end)
17  end

```

In the code snippet above, we start from obtaining all active packages at the moment (packages that are completed), and then separate them in two different groups: the ones with path defined (`grouped.true`) and the ones without path (`grouped.false`). To trace a client pattern and approximate a path for a package the following flow is done:

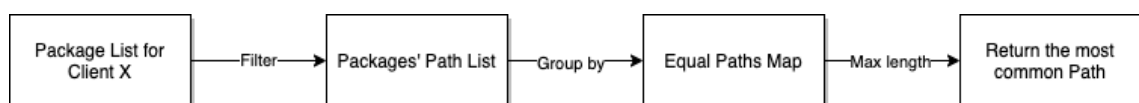


Figure 13: Calculate approximate paths by Client flow.

After calculating every approximate paths for every client with active packages (function `approximate_paths/1`), all paths are filtered. In other words, the simulation will take into account previous states that have already been processed, removing the past ones from the full defined path.

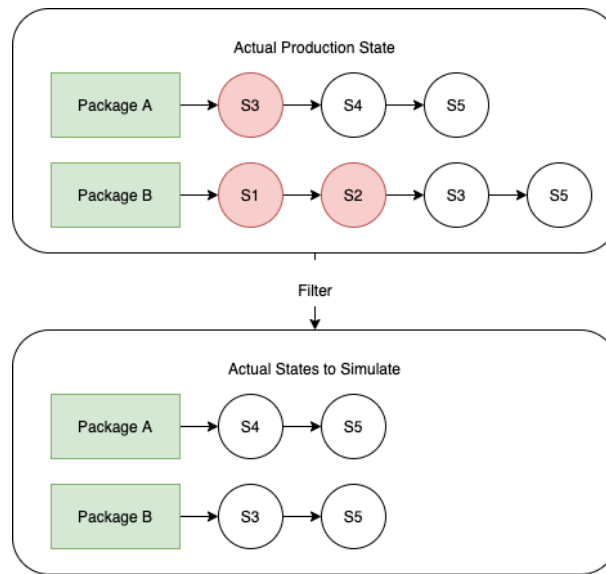


Figure 14: Filtering paths for simulation.

The awaiting list in the simulation manager state is set with all the workstation ids. This way, for the simulation to proceed, the manager has to receive a response from every workstation process, indicating the next event to simulate or nil in case there are no packages in that workstation at the moment.

On the workstation process side (SimPlace), each process starts with the following state:

```

1  self_info: {name, id, order, debit, servers}
2  pids: %{},
3  clocks: [],
4  events: [],
5  mean_time: 0

```

Where,

- `self_info`: is a tuple with the information that identifies that workstation process. Every process has its own information since there is only one process for each workstation;
- `pids`: info about the other workstation's pids;
- `clocks`: each workstation process has a list of clocks. A clock will identify the timestamp for the next events, and a process has as many clocks as the number of servers. With this approach, we can easily simulate various packages being processed in the same workstation, synchronously;

- `events`: the list of packages in the queue to be processed in this workstation. When this list empty, the process is listening for new events coming from other processes. Packages are removed from this list and placed in `clocks` together with a next event timestamp;
- `mean_time`: this represents the mean processing time for this workstation. It doesn't belong in the `self_info` tuple, because this information is sent separately by the simulation manager.

After state initialization, each process receives the messages sent by the simulation manager (referenced before) and updates the state with the new information. When the workstation process receives the last message from `SimMan` (referring package list), the process filters this list, keeping only packages that are in the queue for it. Using the example in Figure 14, workstation S4 would only keep Package A and S3 keep Package B, because those are the next places in production line for that package.

In the previous example, the final list has only one element, but in real cases, there's a great chance of existing more than one package in the queue for a certain workstation. So, that resulting list of packages is ordered by `deliver_date`, with packages that have a closer delivery date having priority over the others.

The next step is to set up the workstation `clocks`. As said before, a workstation process will have as many `clocks` as the number of servers, so we can simulate processing several packages at the same time. Assuming a workstation S1 with 2 servers, will have 2 `clocks`, that identify a package and the expected completion timestamp. (Will look into how to calculate the expected timestamp later.) After this setup, the events (packages) that make it into processing, are popped from the `events` list, because they are no longer in the queue for that workstation.

Finally, the `SimPlace` process calculates which package, in `clocks` will finish first, and sends a message to `SimMan` with the next event information.

Check Figure 15 bellow for a visual example of how this is handled. In that case, `SimMan` would receive a message with `{Package B, 2020-07-20 14:34:57.981960Z}`, since this is first active in process package, to be completed.

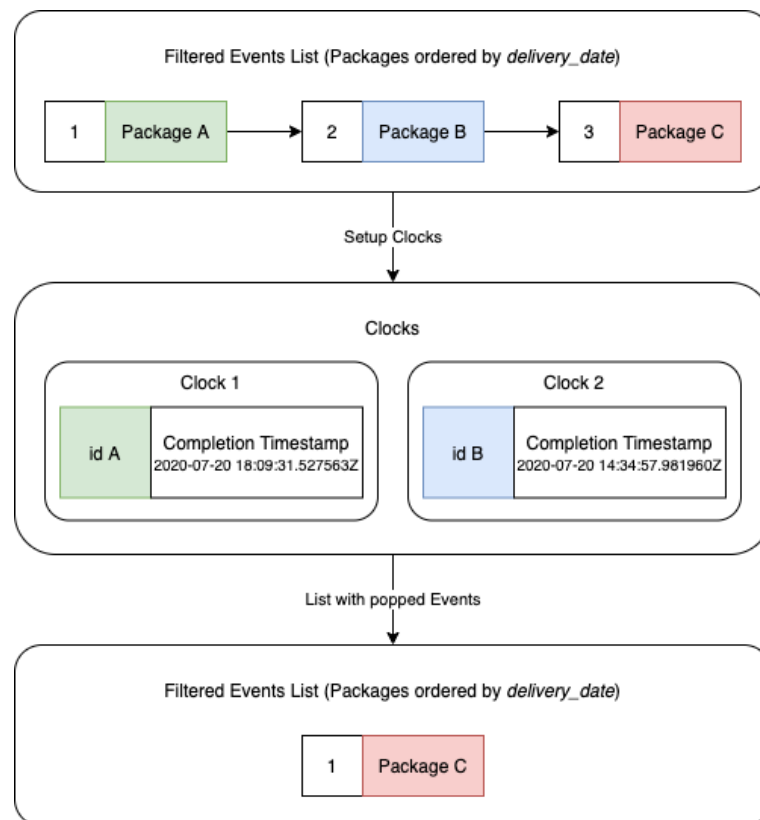


Figure 15: Setup clocks and pop events from list.

Every time the simulation manager receives a `:next_event` from a workstation process, it adds this event to its events list. The insertion is ordered, meaning that it is ordered by the timestamp received. When it has received all answers, it pops the first event from the list and warns the respective workstation to "*advance the clock*". In the following code snippet, it is visible how the simulation manager handles these `:next_event` messages.

First, it checks if the message received is from a valid workstation (its `id` is member of the awaiting list). This being true, the event received is inserted, ordered by timestamp of completion, and the `id` of the workstation is removed from the list.

```

1 def handle_info({:next_event, event, sender}, state) do
2   if Enum.any?(state.awaiting, fn place -> place == sender end) do
3     ordered = insert_event_ordered(state.events, event, sender)
4     awaiting = state.awaiting -- [sender]
5     {rest_events, new_awaiting} = advance_clock(state.pids, ordered,
6       awaiting)
7   end

```

In `advance_clock/3` it is checked if all workstations have reported their next event, in other words if `awaiting` is empty. If this is the case, the first event in line is popped and the respective workstation holding that event is warned. The clock has "ticked" and can advance the simulation. This side will be explained in more detail. If not all workstations have sent their message to the simulation manager, no one is warned to advance, because we can't ensure synchrony between all processes, otherwise.

If `advance_clock/3` returns that there are no missing messages from workstation processes and there are no events in the list also, it means the simulation has ended. There's no more work to do, so `SimMan` orders every process to terminate sending a `:finish` message.

```

1 if rest_events == [] && new_awaiting == [] do
2   Logger.info("Finished Simulation")
3   Enum.each(state.pids, fn {_place, pid} -> send(pid, :finish) end)
4   {:stop, :normal, []}
5 end

```

Now, on `SimPlace` side, when a `:advance` message is received, given the conditions above, it retrieves the `clock` information about the package (event) to advance in simulation. With this package's information, the workstation process verifies if there are more workstations in the rest of the path, for this package to go through. If the `rest_path` list is empty, it means this package is now fully completed and can be removed from the simulation. A database line is created or updated with this new delivery date prediction for this package. If there are more workstations to process, the current one gets the `id` of the next in line and warns the simulation manager to wait for both their `:next_event` messages, also sending the package information with updated simulation clocks and updated remaining path to the next workstation respective process, with a `:next_step` message.

With the extra message sent to the simulation manager, `:wait_for` we ensure that the next event to be simulated is, in reality, the next event. Meaning that a workstation with a free server to process a new package can, for example, calculate that its next event is at timestamp X . If the first event in line in `SimMan` is at timestamp Y and $X < Y$, then the simulation must take into account this new value. So, no wrong results with overlapping clocks will be produced as the manager only advances the simulation when both intervening workstations in an event, reply. Also, if there are no more events in the list, the respective `SimPlace` sends a `:next_event` message as `nil` to `SimMan`. This message is discarded since it doesn't change the simulation current status.

```

1 case tl(rest_path) do
2   [] ->
3     # Update DB Value
4     Models.create_or_update_prediction(
5       %{package_id: event.package, delivery: at})
6
7   path ->
8     next_place = List.first(path)
9     new_event = Map.put(event, :rest_path, path)
10    send(SimMan, {:wait_for, [self_id, next_place]})
11    send(pids[next_place], {:next_step, %{at: at, event: new_event}})
12 end

```

As explained before, a workstation process is also listening to incoming messages from other workstations with information for a new package to process or place in the queue. It is possible to look at this the same way as an employee moving the package from one place to another in the production pipeline.

When a workstation process receives a `:next_step` message with package information for a new event, the first thing it does is to check if there are available clocks. By this, it means that if not all workstation servers are taken, the new arrival can occupy a place in the clocks list. If there are no free servers (clocks list positions all occupied), then this new package will have to wait in line.

It is possible to see below how the process deals with the message if there are no available servers. In this case, as there are no spots free, the *next event* for this process won't change, so it only adds the event to the full list and warns the simulation manager that its `next_event` is `nil`. This way, `SimMan` will just ignore this message, remove this workstation from the awaiting list and proceed with the simulation if possible.

```

1 index = available_server(state.clocks)
2
3 if is_nil(index) do
4   updated_events = insert_event_ordered(state.events, new_event)
5   send(SimMan, {:next_event, nil, elem(state.self_info, 1)})
6   new_state = Map.put(state, :events, updated_events)
7   {:noreply, new_state}
8 end

```

If there is a clock available (server) for the new package to start processing, the workstation process has to calculate the prediction of completion and verify if there is any change to the last *next event* announced to the simulation manager. To keep all processes synched, the process has to send a message to SimMan, even if it is just `nil` for it to ignore and proceed. This flow is presented below in Figure 16. In case of the new arrival has the smaller timestamp, SimMan is warned that this workstation *next event* has changed.

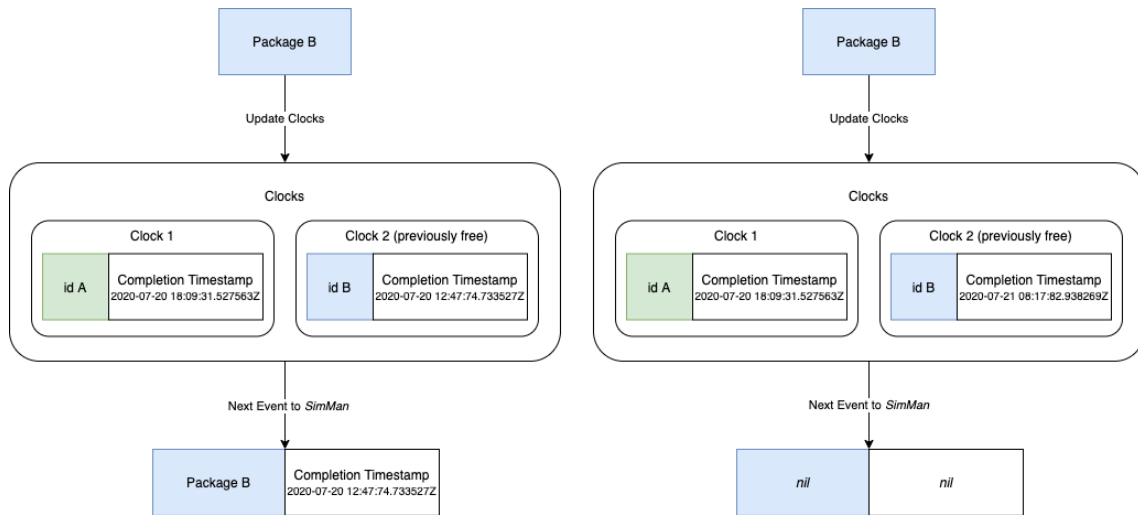


Figure 16: Workstation updating clocks and informing SimMan of next event.

Thanks to the GenServer characteristics and message ordering from *Erlang*, referenced before, we have guarantees that the SimMan always receives `:next_event` messages after receiving a `:wait_for` one.

- **Calculating the workstation process completion timestamp**

Having a way to calculate when a package will be completed inside a workstation that gives the most approximate value is more important than all the process described before, because, without a good estimation, this simulation is irrelevant.

The first step is to calculate how many hours of work is needed to process a given package in a given workstation. The following equation takes care of that:

$$total_time = quantity / workstation_debit$$

As said before, it is possible that `quantity` and/or `debit` to be `nil`, so when this happens, its when the mean time production time is used to approximate *total_time* value.

One of the strongest points in calculating the processing time for a package in a workstation is that not only total production time is accounted for, but also the real business/working

hours of the company, and that added a bit of complexity to the simulation itself. In Figure 17 we have a visual representation of what was said. The simulation engine must behave like a real production pipeline. So, for example, if the company's shifts are 9h-13h and 14h-18h, and *Package A* starts processing at 16h, with a predicted total time of 10h, the completion timestamp should be at 18h of the next day. If this wasn't the way deliveries were calculated, the simulation engine would simply predict a conclusion at 24h of the same day, making all possible predictions invalid.

In summary, first, it is calculated the total processing time for a certain workstation but then it is necessary to transform that value in how many shifts that correspond.

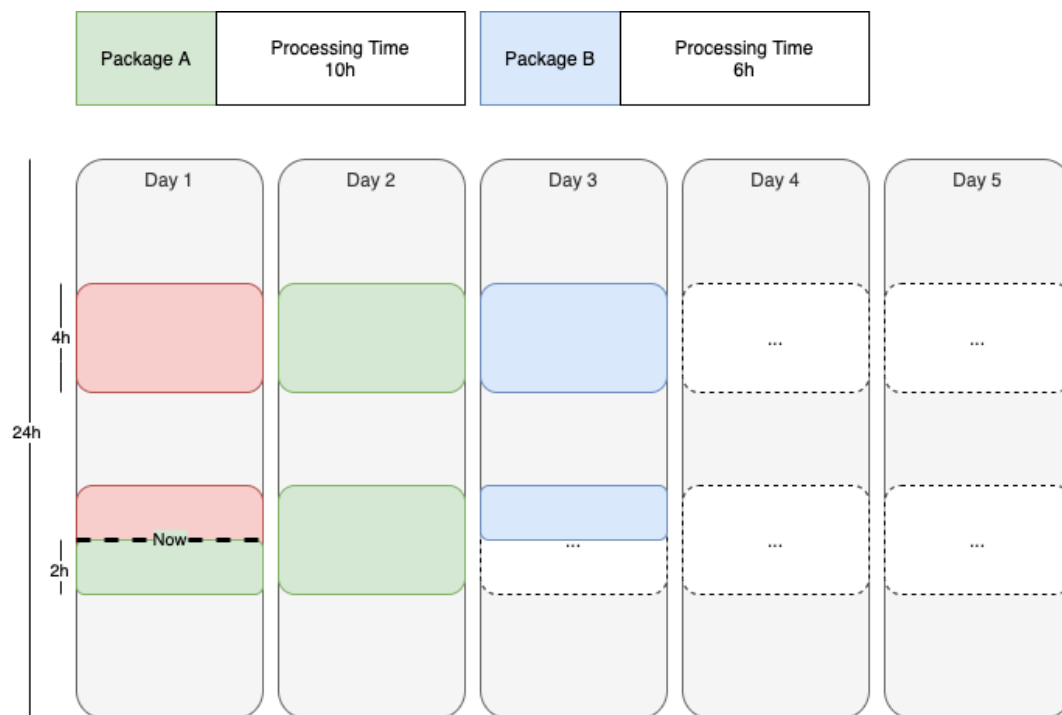


Figure 17: Simulation only uses business hours defined by the platform administrator.

In code snippet below it is possible to see how this process is done. It is a recursive function that receives `timestamp` and `hours` as arguments. `timestamp` represents the moment the simulation and `hours` is the total amount of processing time. As the calculation goes on, `timestamp` increases, with total hours decreasing. When the current shift has more available hours than the ones remaining, the function returns with the final `timestamp` value, with the rest of hours added.

If there are no sufficient remaining hours in the current shift, this function gets the next one (same day or not), and shifts `timestamp` to the start of it. Timezone is taken into account here because every `datetime` timestamp in the platform is in [Universal Time Coordinated \(UTC\)](#). However, platform users should see predictions according to their timezone.

The recursive call, subtracts the remaining hours of the current shift are to hours, like the processing time was complete, passing timestamp also.

```

1 def total_hours_for_work(timestamp, hours) do
2   rest_of_shift = rest_hours_shift(timestamp)
3
4   if rest_of_shift >= hours do
5     Timex.shift(timestamp, seconds: hours)
6   else
7     next_shift = get_next_shift(timestamp)
8     new_hour = next_shift.timestamp_start.hour
9     new_minute = next_shift.timestamp_start.minute
10
11    update_shift_day(timestamp, next_shift)
12    |> Timex.set(hour: new_hour, minute: new_minute)
13    |> Timex.shift(seconds: -timezone_offset())
14    |> total_hours_for_work(hours - rest_of_shift)
15  end
16 end

```

With this, all of the simulation engine development is now detailed and explained.

- **Scheduling hourly simulations**

Since simulations need to run every hour, it is necessary to implement/use a scheduler that can handle that. For that, *Quantum²* was the solution chosen. With this package we can define recurring tasks as they were *cronjobs*. For example, the configuration for it to run the hourly simulation:

```

1 config :pipe, Pipe.Scheduler,
2   overlap: false,
3   jobs: [
4     delivery_simulation: [
5       schedule: "0 * * * *",
6       task: {Pipe.RecurringTasks, :delivery_simulation, []}
7     ]
8   ]

```

² <https://hexdocs.pm/quantum/readme.html#content>

It is easy to understand that it is possible to add more jobs to the list, completely independent from each other. The crontab format is *minute, hour, day of month, month and day of week*, being possible to also use *second* in special cases. The configuration above runs the function `delivery_simulation()` of the `Pipe.RecurringTasks` module every hour. Not running during nights or weekends could also be a good approach, and save some resources during that time. Despite being simple to configure, it was decided to run at every hour.

The function that is called every time, is as simple as starting the simulation manager (`SimMan`), and the rest we have already detailed. The duration of the simulation is logged for debug and future analysis purposes.

```

1  def delivery_simulation do
2    time =
3      fn -> SimMan.start_link() end
4      |> :timer.tc()
5      |> elem(0)
6      |> Kernel./(1_000_000)
7
8    Logger.info("Simulation Duration: #{time}s")
9  end

```

On the `Pipe.Application` module `Pipe.Scheduler` was added to the children list that will be passed as argument to `Supervisor.start_link(children, opts)`. This means that the scheduler has the *Application* as supervisor and it will be restarted if some error occurs.

4.3 OUTCOMES AND SUMMARY

After implementing and developing all proposed features, (pages and features not referenced in this chapter can be found in *Support Material Other relevant Frontend pages*), some conclusions can be retrieved and analysed.

One of the major goals was to develop a platform that could be modular and with the capability to add new features in the future. This was made possible by a separation of the backend and frontend, which can be developed and modified in parallel. Integration with external platforms such as the email server, database or deployment are generic and don't depend directly on the ones that are currently used.

On the other hand, heavier requests to the [API](#) were optimized through data processing whenever possible. These optimizations don't reflect directly on response times but on preventing some associated problems with functional programming such as stack overflows. This problem was addressed previously in this chapter with the use of tail-recursive calls, but

sometimes streams were also used to process data. Due to their laziness, streams are useful when working with large (or even infinite) collections. Functions in the `Stream` module work in linear time. This means that the time it takes to operate grows at the same rate as the length of the list, just like the homologous eager versions found in the `Enum` module.

All of the points presented here contribute to a lightweight code with fast response times and low memory usage (even with the hourly simulations) with the value normally below 80MB for the complete backend application.

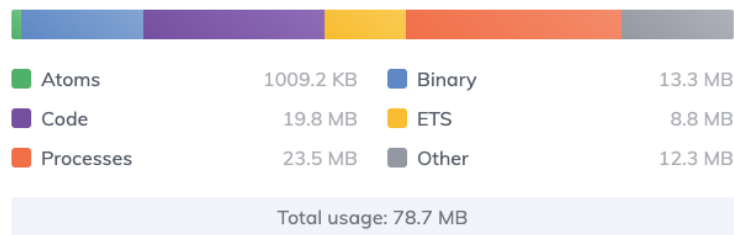


Figure 18: Memory distribution of backend application.

Some other metrics will be referenced in the next chapter, as part of the case study analysis.

The selected distributed approach for the delivery simulations account also for low memory usage, since *Erlang* processes are very lightweight, and fast calculation times for some relevant numbers of active packages inside the production line.

Choosing a distributed approach for the delivery simulation, can introduce some extra complexity and errors to the codebase. Nevertheless, the *Erlang* philosophy of “*let it crash*” was used here. Since every `SimPlace` is started with a `link`³ to `SimMan`, when one of these processes crashes, the `link` to `SimMan` is broken and the simulation manager process will also stop, propagating to the rest of workstation processes, halting every single process related to the simulation. *Erlang* has a limit of simultaneously alive Erlang processes of 32768, by default. But this limit can be raised to at most 268435456 processes [14]. So, in the worst-case scenario, memory leaks with badly terminated/blocked simulation processes need about $32768/(n + 1)$ hours until memory shortage. n being the number of workstations of the pipeline plus 1 for the simulation manager, giving plenty of time to resolve the problem before the system becomes completely unusable.

This is a valid approach since the simulation runs every hour and the process that manages this scheduling is independent of the simulation ones. It is also supervised by the *Application* and if something very wrong happens, the scheduler is restarted. Users will still have the possibility to consult the last updated prediction value until the next simulation runs.

Some extra problems not accounted in the proposed algorithm for the simulation engine, due to the distributed approach, add more complexity to the platform and the engine itself.

³ https://hexdocs.pm/elixir/GenServer.html#start_link/3

For this, and as it is known in other distributed projects, the extra complexity in the code can introduce additional challenges on future changes or understanding of why some things are done like that.

There are, also, some limitations such as not being able to count defected products and use a more "automatic", less error-prone, tracking method. Integration with the company's [ERP](#) is also a problem. Office employees need to register the order twice (on the [ERP](#) on and this platform). Nonetheless, the current one used by *Tipoprado* does not offer a cloud version and also doesn't provide a public [API](#) for integration.

CASE STUDY / EXPERIMENTS

As expected, the case study for this dissertation is the implementation of the platform in *Tipoprado's* production line. In this Chapter, we will explain and detail the following:

- how the platform was put into use;
- the impact of using the platform by the company's employees;
- the impact of the platform in the production line;
- the impact of the platform with customer engagement and relationship.

5.1 EXPERIMENT SETUP

The platform was deployed on *Heroku*¹ that simplifies a lot of the process. The server used has 512MB of **Random-access memory (RAM)** and 10 process types. For database *PostgreSQL*² was the selected one, and the database used in production has a limit of 20 simultaneous connections.

As for mail server, *Mailgun*³ is the solution opted and offers a lot features to analyse email delivery and user interaction that will have a major role in the final conclusions of the platform.

Note that these options can be changed with small configuration changes, and the developed platform is not bound to use any of them in particular.

Started collecting some initial data in September 2019, with a limited prototype of the platform. The goal was to obtain the most data as early as possible, so in the end, could be possible to analyse the evolution with as much information as possible.

The configuration for the production line set by *Tipoprado's* managers, has a total of 13 workstations with a total of 16 computers spread across the office and production floor,

¹ <https://www.heroku.com/home>

² <https://www.postgresql.org/>

³ <https://www.mailgun.com/>

using the platform (excluding external clients accessing it). 4 of those computers have a barcode reader to introduce data more quickly and less error-prone than manual input.

5.2 RESULTS

Before starting reviewing the real impact in the company, we will examine how the business clients adhered to this new platform and method of communication/engagement. For this, let's scrutinize email information given by *Mailgun*, where it is possible to see how many emails were sent and how the receivers (end clients) interacted with them. The whole period from which we have data (ranging from 1 September 2019 to 30 September 2020), was broken into quarters and the values are presented in Table 5.

	Sep-Dec (2019)	Jan-Mar (2020)	Apr-Jun (2020)	Jul-Sep (2020)
Sent	2616	2177	1166	1735
Delivered	99%	99%	100%	99%
Opened	227%	240%	231%	186%
Clicked	66%	45%	37%	33%

Table 5: Email delivery statistics by quarter.

Analysing Table 5 we can conclude that there was a big adhesion by end clients and the emails sent, with emails being opened more than once (as the rates above 100% show), and a mean of about 50% of them being clicked. By clicking an email, as defined in the Table 5, it means that the client clicked in a button or link that redirected him/her to the web application. A client can click in a link to consult the package current state or to answer a satisfaction survey, however, these two types of clicks are not distinguished here.

Lower values after the second quarter can be mostly explained by the *COVID-19* pandemic that affected the company's production in about 70% during March, April and May, with slow recovery after June. With less production, fewer packages were registered and consequently fewer emails sent, and clients were less interactive.

Clients can also consult and access the platform directly from the link provided, after accessing the first time or searching for the package using the provided unique code, in the application homepage. Although looking at the logs these two methods were not very used compared to simply clicking buttons in emails.

Taking a look at the data produced by the platform and also using some of the developed plots together with a company's employees survey, to support conclusions, let's analyse some of the following topics:

- **Customer engagement and satisfaction**

As part of customer engagement, according to *Tipoprado's* managers, during the year previous to start using the platform, (September 2018 to September 2019) about 50 reviews were received from end clients. And these were personally requested to them. Since the beginning they received 141, resulting in a 282% increase compared to the homologous period.

Observing Table 6 we can check on the evolution of the mean ratings received from clients. In these satisfaction surveys, clients were questioned about price, quality, and delivery schedule, and others, in a 1-5 rating.

	Sep-Dec (2019)	Jan-Mar (2020)	Apr-Jun (2020)	Jul-Sep (2020)
Overall Rating	4.53	4.40	4.45	4.67

Table 6: Mean customer reviews rating by quarter.

Just looking at the previous table we can't conclude if the platform had any positive impact on those specific topics since the variations are small. Also, we don't possess meaningful answers before the platform implementation, to compare these results with.

Through this automatic survey end clients were also enquired about what they thought about this tracking platform, and 4.8 is the mean rating.

- **Packages delivered on time**

Another key value to present here in the results topic is how the percentage of packages delivered on time evolved through the year. Similarly to the customer reviews, these values are analysed by quarter. Observing the next table, although we don't have relevant data for the first quarter, there is a clear positive evolution of the percentage of packages delivered on time.

	Sep-Dec (2019)	Jan-Mar (2020)	Apr-Jun (2020)	Jul-Sep (2020)
Delivered on time	—	53%	88%	96%

Table 7: Percentage of packages delivered on time by quarter.

At first sight, this can be correlated with higher levels of organization within the company and production. However, the reduced number of new orders during the global pandemics from March to June might have a major role here. It is expected that with more information, scrutinized next, will be possible to have a more concrete explanation.

- **Performance and Efficiency (OEE)**

These metrics were one of the mentioned in the implementation Chapter and have a direct relation with how a production line performs. The objective is to maximize the OEE, as $OEE = A * P$ (in this case), and observing Table 8 there is a clear negative evolution through the year.

	Sep-Dec (2019)	Jan-Mar (2020)	Apr-Jun (2020)	Jul-Sep (2020)
Availability	68%	56%	36%	40%
Performance	—	108%	109%	108%
OEE	—	61%	40%	46%

Table 8: Company's workstations overall availability and performance by quarter.

Again, we don't have sufficient data to calculate *Performance* and *OEE* for the first 3 months, but the noticeable decrease can be easily explained by the reduced values of *Availability*, caused by the pandemics. Workstations had less active time, but *Performance* kept stable and slightly above expected values through that period. Note that as there are many manual processes, expected performance values (maximum debit) were approximated by the company's managers. These values can be adjusted in the platform.

- **Queue waiting times**

The other measure referenced in the implementation Chapter was the waiting time in queue for a package to start being processed in a certain workstation. When a company can reduce these "dead times", it's a great step towards increased production levels.

In order to keep it simple and easy to understand the values presented in the matrix provided by the platform's plots page, were treated as a whole, instead of a $A \rightarrow B$ one. After this, the following approximations were achieved:

	Sep-Dec (2019)	Jan-Mar (2020)	Apr-Jun (2020)	Jul-Sep (2020)
Waiting Time (days)	3.7	2.2	1.7	1.35

Table 9: Average waiting times by packages stopped in a queue, by quarter.

- **Production pattern and bottlenecks**

Being able to identify a production pattern and bottleneck is also a good achievement of this platform. Looking at Figure 19, which represents a heat-map of where packages pass through their production processes, and it is easy to understand the workstations that are more required and the ones that might compensate in investing more or not.

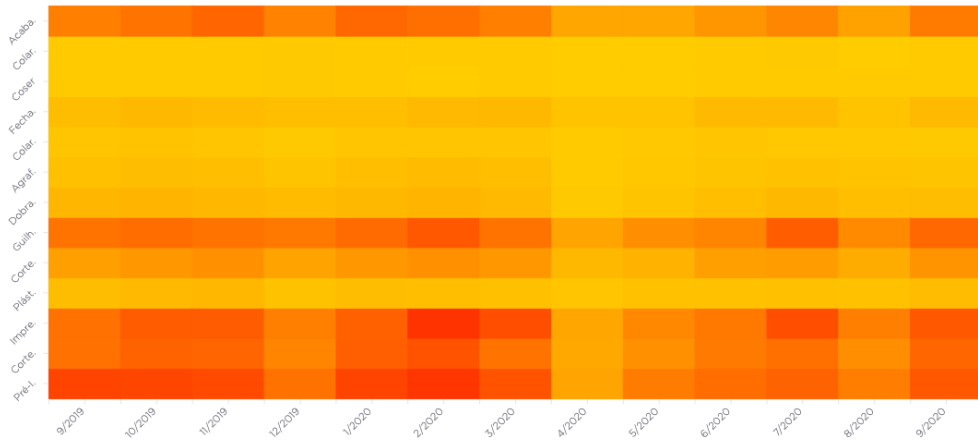


Figure 19: Heat-map of packages in each workstation.

Figure 19 is complemented by 20. In the figure below, red bars represent packages that have left that workstation and are waiting in line to start the next process, and blue represent the ones that are active at that moment in each workstation. The first column, as the name states, are the ones that haven't been picked up yet.

Here, the first 3 workstations present a clear bottleneck with about 80% of packages accumulating in this area. Figure 19 confirms this, as these workstations are the most required ones.

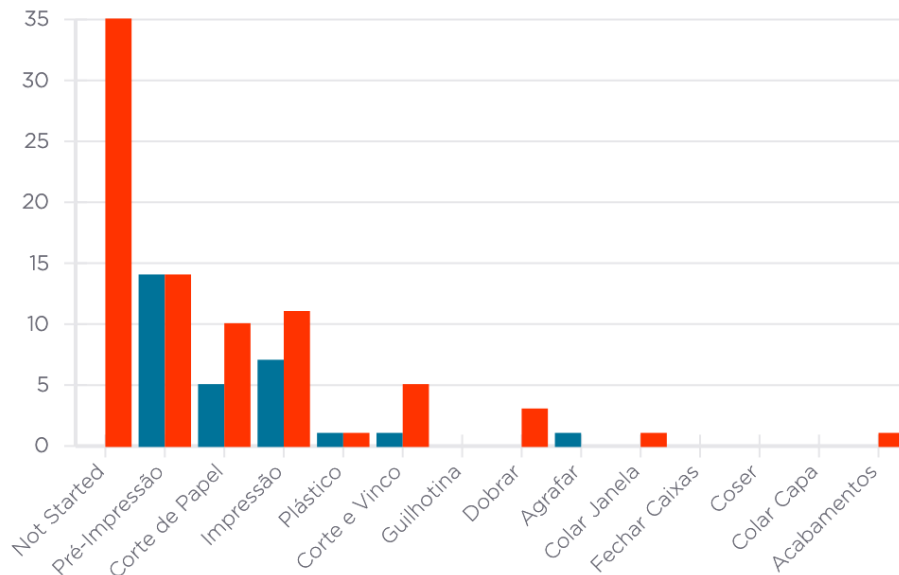


Figure 20: Number of packages at each workstation (paused and active).

Note: both Figure 19 and 20 are screenshots from the platform implemented at *Tipoprado*.

- **Delivery dates simulation**

Looking at the delivery dates simulation it's not possible to access the proximity of the expected values provided by the simulation engine. These predictions are dynamic and changing every hour, as said before, so the values at one moment are merely indicative to *Tipoprado's* managers and employees.

The analysis that can be made is on how the engine behaves in a real-world situation, and with values provided by logs, we get an average of 0,00384975s with a total of about 160 packages being simulated through the 13 different workstations, on the day measured (04/08/2020). Meaning that these simulations are almost instantaneous and with almost none to little effect on the platform's overall performance.

- **Employees impact and opinions**

Since *Tipoprado's* employees have great implications if the validity of the platform developed, it was decided to enquire all of them on its impact on the business. A total of 20 workers were part of an anonymous survey, with six questions to rate 1-5. These were the following:

- **P1:** The use of this platform had a positive impact on the general organization of the company;
- **P2:** The use of this platform had a positive impact on the production and efficiency of the company;
- **P3:** The use of this platform had a positive impact on the relationship with clients;
- **P4:** The extra time spent on registering data in the platform makes up in the long run;
- **P5:** The ease of use of the platform;
- **P6:** Overall analysis of the platform.

Below, the average value of the answers provided enables to retrieve a fairly positive opinion on the platform by the people that are using it every day, for the last year (since September 2019).

	P1	P2	P3	P4	P5	P6
Average	4	3.9	3.7	4	4.5	4.2

Table 10: Employees overall opinion on the platform.

Analysing **P3** in particular, since it's based on the employees' idea of how this platform impacted end clients, with the values provided by email information and the overall rating of the platform answered by these clients, it is safe to say that one of the major goals of this dissertation was accomplished.

5.3 DISCUSSION AND SUMMARY

With all this detailed information retrieved from the platform itself and *Tipoprado's* employees, it's possible to reach some conclusions.

Unfortunately, the current global pandemics affected the company's production pipeline, as it was explained before. And that made it more difficult to identify some positive points. However, merging information coming from end-clients, the platform and the company's employees, we can state that:

- End clients had a good interaction with the platform, having rated it with a very positive value (4.6);
- The number of calls received in the office decreased drastically. About 2-5 per day, compared with the previous 15-20 daily;
- The company is effectively more organized and efficient, accordingly to what its employees feel. Despite values produced from the platform's statistics pages not showing that directly, since these values were "distorted" by the pandemics;
- The final result of the Web Application is simple to use and that reflects on the users' opinion;
- Production patterns and production line's bottlenecks were identified and will help the business managers taking future decisions when involving where to invest or not.

CONCLUSION AND FUTURE WORK

6.1 CONCLUSIONS

Having completed this dissertation, it can be concluded that companies embracing *Industry 4.0* are the future. The tech-based production and planning is a great plus and a differentiation point from the competition. This, alongside improving customer engagement is set to have a great impact on *Tipoprado's*.

Research for related products also proved that this platform is a different take on optimizing production efficiency. It has a client-package orientation (queue theory performance measures and expected delivery dates simulations), opposing to a product orientation (constant production of just one product/raw material), used in similar services.

The choice of tracking methods was based not only on the cost/effect relation but also on *Tipoprado's* production characteristics, for this application to be less intrusive as possible for employees. However, the selected one keeps an open door for future implementation of new tracking methods, since this one (barcode readers) still requires manual use and is more error-prone than a more IoT approach. The fact that the platform was built with micro-services in mind, it is also prepared to receive these different approaches. More features can be added to the core ones, as building blocks.

This platform, as hinted before throughout the document, has a lot of customizable features and can be configured to adapt to the evolution of *Tipoprado's* business. However, these same characteristics still apply to similar industries with various products and manual processes. Workstations, employee accounts, satisfaction survey, statistics, and many others can change at a distance of a few clicks, so a possible next step is to validate it in other businesses.

After developing the entire platform and put it into use in the company's production floor and offices, data was collected for almost a year. With this information, relevant conclusions were reached. Both employees and clients made a very positive evaluation of the platform in different aspects. With the second ones having a major role in the platform's use, reducing significantly the number of calls to the office asking how their orders were.

Goals of helping the business managers taking future decisions based on production patterns and flows can also be checked off, thanks to an insightful statistics page with broad analysis.

As part of the simulation for delivery dates, despite the values being merely representative, they provide a good amount of knowledge on production, as well.

6.2 PROSPECT FOR FUTURE WORK

Two features can, possibly, be developed in the future. These were suggested by different employees after chatting with some of them and would have a big impact on the organization and less time expended.

1. Integration with the company's [ERP](#). Resulting in a unique order registry instead of two (a separate registry for this platform only);
2. Implementing stock management. This feature is already present in a lot of *ERPs* or [MES](#) and would be a great addition to the web application developed.

Registering which employee started/completed a given state is also something to take into account since it can provide more important information. Initially, this wasn't part of the platform's objective, but it can unroll this way in the future.

A lot of new things can appear too since this is an area that it is still in growth, and will be watchful to how things evolve.

BIBLIOGRAPHY

- [1] Alex McEachern. Why you should care about customer engagement. <https://blog.smile.io/what-is-customer-engagement-and-why-is-it-important>, 2019. Accessed: 2019-11-26.
- [2] Anastasia D. Best Architecture for an MVP: Monolith, SOA, Microservices, or Serverless? <https://rubygarage.org/blog/monolith-soa-microservices-serverless>, 2019. Accessed: 2019-12-19.
- [3] Angela Stringfellow. What is Web Application Architecture? How It Works, Trends, Best Practices and More. <https://stackify.com/web-application-architecture/>, 2017. Accessed: 2019-12-19.
- [4] Anselm Eickhoff. Citybound - Engine Technology. <https://aeplay.org/citybound>. Accessed: 2020-01-02.
- [5] Bernard Marr. Why everyone must get ready for the 4th Industrial Revolution. <https://www.forbes.com/sites/bernardmarr/2016/04/05/why-everyone-must-get-ready-for-4th-industrial-revolution>. Accessed: 2019-11-26.
- [6] Chris Richardson. Pattern: Microservice Architecture. <https://microservices.io/patterns/microservices.html>. Accessed: 2020-10-25.
- [7] Chris Richardson. Pattern: Monolithic Architecture. <https://microservices.io/patterns/monolithic.html>. Accessed: 2020-10-25.
- [8] Django. Why Django? <https://www.djangoproject.com/start/overview/>. Accessed: 2020-10-26.
- [9] Wouter Duivestijn. *Continuous Simulation*. 2006.
- [10] Ecto Docs. Getting Started. <https://hexdocs.pm/ecto/getting-started.html#content>. Accessed: 2019-01-10.
- [11] Elixir. The Elixir programming language. <https://elixir-lang.org/>. Accessed: 2020-10-25.
- [12] Erlang. Getting Started with Erlang User's Guide. https://erlang.org/doc/getting_started/users_guide.html. Accessed: 2020-10-25.

- [13] Erlang - Ericsson AB. Getting Started with Erlang - Concurrent Programming. https://erlang.org/doc/getting_started/conc_prog.html. Accessed: 2019-12-19.
- [14] Erlang - Ericsson AB. System limits - Processes. http://erlang.org/documentation/doc-5.8.4/doc/efficiency_guide/advanced.html. Accessed: 2020-07-28.
- [15] Martin Fowler and James Lewis. Characteristics of a microservice architecture. 2014.
- [16] Github. Github's most starred Repositories. <https://github.com/search?q=stars%3A%3E1&s=stars&type=Repositories>. Accessed: 2019-12-19.
- [17] Guardian Docs. Create implementation module. <https://hexdocs.pm/guardian/1.2.1/tutorial-start.html#create-implementation-module>. Accessed: 2019-01-11.
- [18] Hardcat - Asset Management Software. RFID asset management - expectations vs reality. <https://www.hardcat.com/2019/02/07/rfid-asset-management>. Accessed: 2019-10-18.
- [19] Heroku Postgres. Plans Pricing. <https://elements.heroku.com/addons/heroku-postgresql>. Accessed: 2020-10-24.
- [20] Carl Hewitt, Peter Bohler Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI*, pages 235–236, 1973.
- [21] A. Melnikov Iode Ltd. I. Fette, Google Inc. The WebSocket Protocol. <https://tools.ietf.org/html/rfc6455>. Accessed: 2019-10-17.
- [22] IBM Cloud Education. SOA (Service-Oriented Architecture). <https://www.ibm.com/cloud/learn/soa>. Accessed: 2020-10-25.
- [23] Jane Walerud. Erlang/OTP Released as Open Source™. <https://web.archive.org/web/19991009002753/http://www.erlang.se/onlinenews/ErlangOTPos.shtml>, 1998. Accessed: 2019-12-19.
- [24] Java - Oracle. JDK 15 Documentation. <https://docs.oracle.com/en/java/javase/15/>. Accessed: 2020-10-25.
- [25] Javascript - Pluralsight. Learn Javascript. <https://www.javascript.com/learn>. Accessed: 2020-10-25.
- [26] Douglas W. Jones. An Empirical Comparison of Priority-queue and Event-set Implementations. pages 300–311, 1986.
- [27] Josh Smit. Patterns - WPF Apps With The Model-View-ViewModel Design Pattern. <https://docs.microsoft.com/en-us/archive/msdn-magazine/2009/february/>

- [patterns-wpf-apps-with-the-model-view-viewmodel-design-pattern](#), 2009. Accessed: 2020-10-25.
- [28] JWT Docs. Introduction to JSON Web Tokens. <https://jwt.io/introduction/>. Accessed: 2019-01-11.
- [29] J. Klensin. Simple Mail Transfer Protocol. <https://tools.ietf.org/html/rfc5321>, 2008. Accessed: 2020-10-25.
- [30] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, pages 690–691, 1979.
- [31] Little, J. D. C. A Proof for the Queuing Formula: $L = \lambda W$. *Operations Research*, pages 383 – 387, 1961.
- [32] M. Marsudi and H. Shafeek. The application of queuing theory in multi-stage production line. 2014.
- [33] J. Martin. *Managing the Data Base Environment*. A James Martin book. Pearson Education, Limited, 1983.
- [34] MaryLou Costa. Customer engagement improves brand profits. <https://www.marketingweek.com/customer-engagement-improves-brand-profits>, 2010. Accessed: 2019-11-26.
- [35] Norm Matloff. Introduction to Discrete-Event Simulation and the SimPy Language. pages 3–6, 2008.
- [36] Miles Data Technologies. 1D vs. 2D Barcodes - When and Why to Use Each Type. <https://www.milesdata.com/blog/1d-vs-2d-barcodes-when-and-why-to-use-each-type>. Accessed: 2019-10-18.
- [37] Nihal Sahu. An Interview with Elixir Creator José Valim. <https://www.sitepoint.com/an-interview-with-elixir-creator-jose-valim/>, 2015. Accessed: 2019-12-19.
- [38] Node.js. About Node.js. <https://nodejs.org/en/about/>. Accessed: 2020-10-25.
- [39] W3C Philippe Le Hégarret. The W3C Document Object Model (DOM). <https://www.w3.org/2002/07/26-dom-article>, 2002. Accessed: 2020-01-02.
- [40] Phoenix Docs. Guides - Overview. <https://hexdocs.pm/phoenix/overview.html>. Accessed: 2019-12-19.
- [41] PHP. History of PHP and Related Projects. <https://www.php.net/manual/en/history.php>. Accessed: 2020-10-26.

- [42] Productoo. Boost your factory performance with Productoo. <https://www.slideshare.net/PavlaDolealov1/boost-your-factoryperformance8roidrivers>. Accessed: 2019-10-17. page 14.
- [43] Python. About. <https://www.python.org/about/>. Accessed: 2020-10-26.
- [44] QRcodesHowTo.com. 2d Barcode versus 1d Barcode. <http://www.qrcodeshowto.com/what-is-a-qr-code/2d-barcode-versus-1d-barcode-with-pictures/>. Accessed: 2019-10-17.
- [45] R. Fielding, UC Irvine, J. Gettys, Compaq/W₃C, J. Mogul, Compaq, H. Frystyk, W₃C/MIT, L. Masinter, Xerox, P. Leach, Microsoft, T. Berners-Lee . Hypertext Transfer Protocol – HTTP/1.1. <https://www.ietf.org/rfc/rfc2616.txt>, 1999.
- [46] Ray Wang. *Why Live Engagement Marketing Supercharges Event Marketing*. 2016.
- [47] React. Controlled Components. <https://reactjs.org/docs/forms.html#controlled-components>. Accessed: 2019-01-02.
- [48] React Docs. Virtual DOM and Internals. <https://reactjs.org/docs/faq-internals.html>. Accessed: 2020-01-02.
- [49] React-vis. React visualization library. <https://uber.github.io/react-vis/>. Accessed: 2019-01-14.
- [50] ReactJS. React - A JavaScript library for building user interfaces. <https://reactjs.org/>. Accessed: 2020-10-25.
- [51] Red Hat. What is an API? <https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces>. Accessed: 2020-10-25.
- [52] Rouse, Margaret. internet of things (iot).
- [53] Roy Thomas Fielding. Architectural Styles and the Design of Network-based Software Architectures. pages 76–105, 2000.
- [54] Ruby on Rails. Imagine what you could build if you learned Ruby on Rails. . . . <https://rubyonrails.org/>. Accessed: 2020-10-25.
- [55] Rust. Learn Rust. <https://www.rust-lang.org/learn>. Accessed: 2020-10-24.
- [56] Vladimir Rykov and Dmitry Kozyrev. *On Sensitivity of Steady-State Probabilities of a Cold Redundant System to the Shapes of Life and Repair Time Distributions of Its Elements*. 2018.
- [57] Klaus Schwab. *The Fourth Industrial Revolution*. Currency, 2017. ISBN-10: 9781524758868.

- [58] Seebo. How Factory 4.0 is transforming production. <https://www.seebo.com/factory-4-0>. Accessed: 2019-10-14.
- [59] Semantic-UI. Semantic-UI React. <https://react.semantic-ui.com/>. Accessed: 2019-01-10.
- [60] Spring. Why Spring? <https://spring.io/why-spring>. Accessed: 2020-10-26.
- [61] Steve Burbeck. Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC). <https://web.archive.org/web/20120729161926/http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html>, 1992. Accessed: 2020-10-25.
- [62] Josefin Stintzing and Frederik Norrman. *Prediction of queuing behaviour through the use of artificial neural networks*. 2017.
- [63] Marc Ingo Wolter, Anke Mönnig, Markus Hummel, Christian Schneemann, Enzo Weber, Gerd Zika, Robert Helmrich, Tobias Maier, and Caroline Neuber-Pohl. *Industrie 4.0 und die Folgen für Arbeitsmarkt und Wirtschaft*. 2015. ISSN 2195-2655.

GLOSSARY

- API** Application Programming Interface. 15, 19, 23, 25, 28, 30–32, 53, 55
- CS** Continuous Simulation. 13, 14
- DES** Discrete Event Simulation. 13, 14, 18
- DOM** Document Object Model. 18, 20
- E2E** End to End. 6
- ERP** Enterprise Resource Planning. 7, 10, 19, 55, 64
- FCFS** First Come First Served. 12
- FIFO** First In First Out. 43
- HTTP** Hypertext Transfer Protocol. 15, 23–25
- IoT** Internet of Things. 6, 8, 19, 23, 63
- JWT** JSON Web Token. 31–33, 35
- MES** Manufacturing Execution System. 7, 19, 64
- ML** Machine Learning. 5, 12, 18
- MVC** Model-View-Controller. 15
- MVPs** Minimum Viable Products. 15
- MVVM** Model-View-View-Model. 15
- OEE** Overall Equipment Effectiveness. 7, 23, 39, 41, 43, 59
- OS** Operating System. 25
- PaaS** Platform as a Service. 6
- RAM** Random-access memory. 56
- REST** Representational State Transfer. 19, 23, 28, 30
- RFID** Radio Frequency Identification. 8–10
- ROI** Return on Investment. 7
- SMTP** Simple Mail Transfer Protocol. 23
- SOA** Service-Oriented Architecture. 15
- UIs** User Interfaces. 17

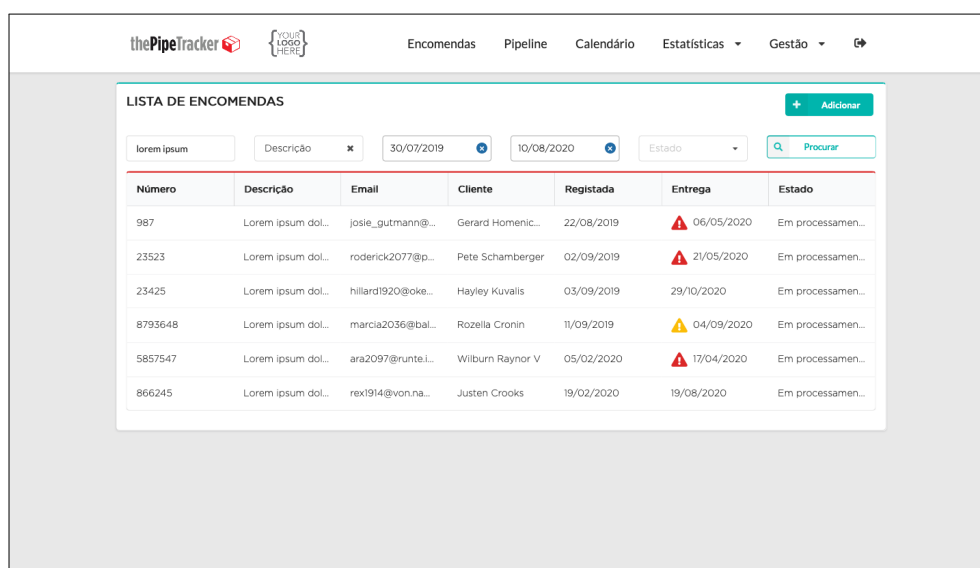
UPC Universal Product Code. 10

UTC Universal Time Coordinated. 51

VM Virtual Machine. 20

SUPPORT MATERIAL

A.1 OTHER RELEVANT FRONTEND PAGES



thePipeTracker {YOUR LOGO HERE} Encomendas Pipeline Calendário Estatísticas Gestão

LISTA DE ENCOMENDAS + Adicionar

lorem ipsum Descrição x 30/07/2019 10/08/2020 Estado Procurar

Número	Descrição	Email	Cliente	Registada	Entrega	Estado
987	Lorem ipsum dol...	josie_gutmann@...	Gerard Homenic...	22/08/2019	06/05/2020	Em processamen...
23523	Lorem ipsum dol...	roderick2077@p...	Pete Schamberger	02/09/2019	21/05/2020	Em processamen...
23425	Lorem ipsum dol...	hillard1920@oke...	Hayley Kuvalis	03/09/2019	29/10/2020	Em processamen...
8793648	Lorem ipsum dol...	marcia2036@bal...	Rozella Cronin	11/09/2019	04/09/2020	Em processamen...
5857547	Lorem ipsum dol...	ara2097@runte.l...	Wilburn Raynor V	05/02/2020	17/04/2020	Em processamen...
866245	Lorem ipsum dol...	rex1914@von.na...	Justen Crooks	19/02/2020	19/08/2020	Em processamen...

Figure 21: Package list page. Quick view information or click to open detailed page.

ADICIONAR NOVA ENCOMENDA

Número *

Descrição (Limite 250 char) *

Quantidade

Caminho de Produção

Data de Entrega* (Valor default para 3 semanas)

26/08/2020

INFORMAÇÃO DE CLIENTE

Nome Cliente *

Email Cliente *

- ASDFGHJKL teste@teste.pt
- CLIE1 cli@cli.cenas
- CLIENTE 1 teste@teste.pt
- CLIENTE 12

+ Adicionar

Figure 22: New package page.

agosto 2020

Hoje

Seg 31	Ter 1	Qua 2	Qui 3	Sex 4	Sab 5	Dom 6
				<p>8793648</p> <ul style="list-style-type: none"> Place 1 <p>Rozella Cronin: Lorem ipsum dolor sit amet, consectetur adipiscing elit. Quisque in luctu...</p>		

Figure 23: Calendar with weekly view for delivery dates.

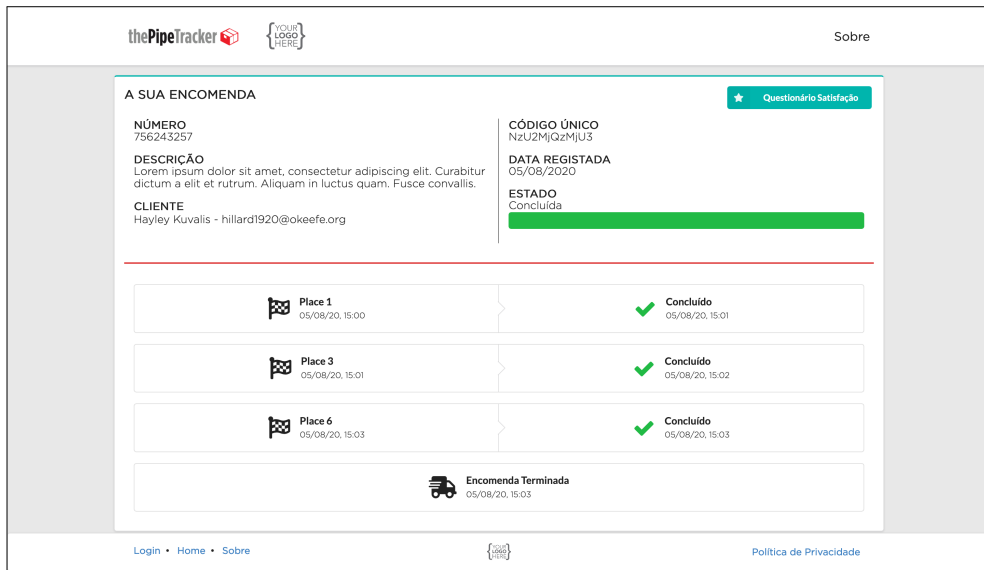


Figure 24: Package public view page. Information available to the end client.

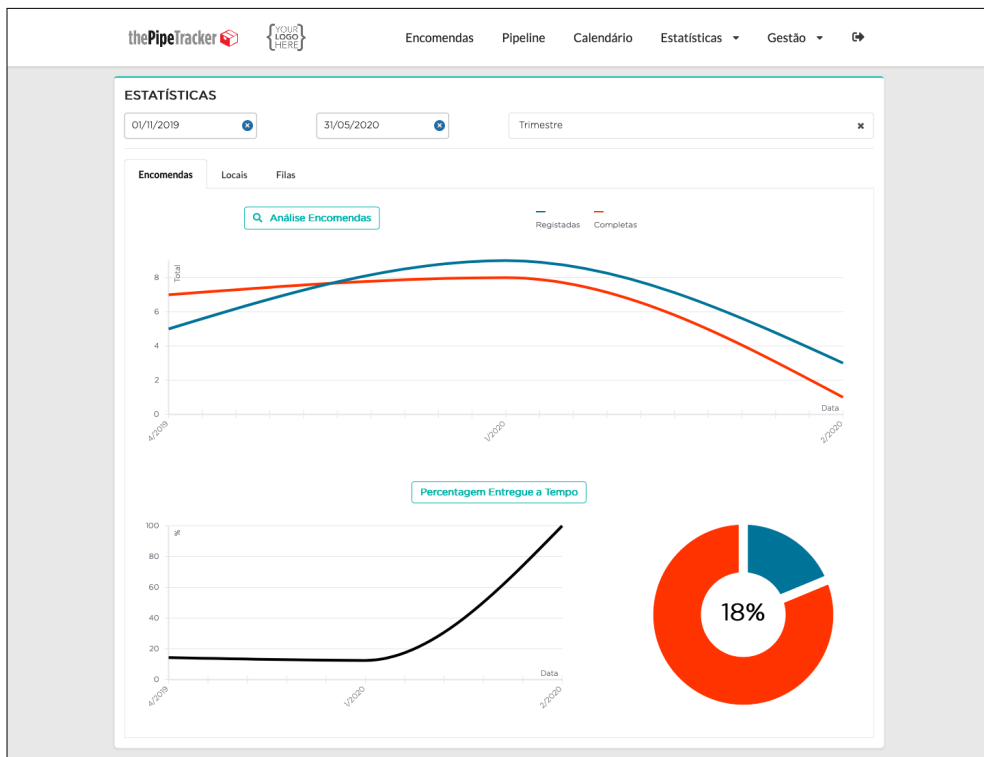


Figure 25: Number of package statistics page.

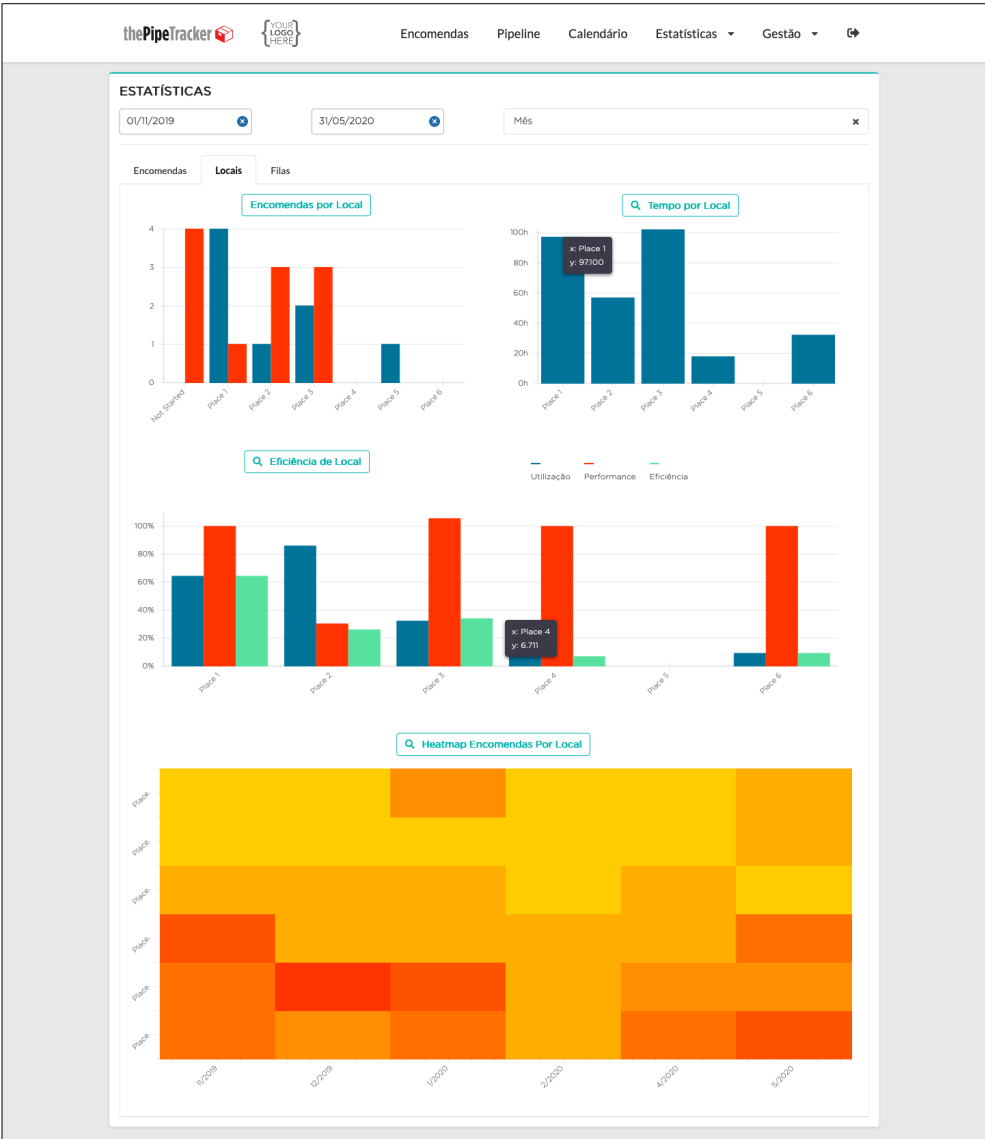


Figure 26: Workstation statistics page.

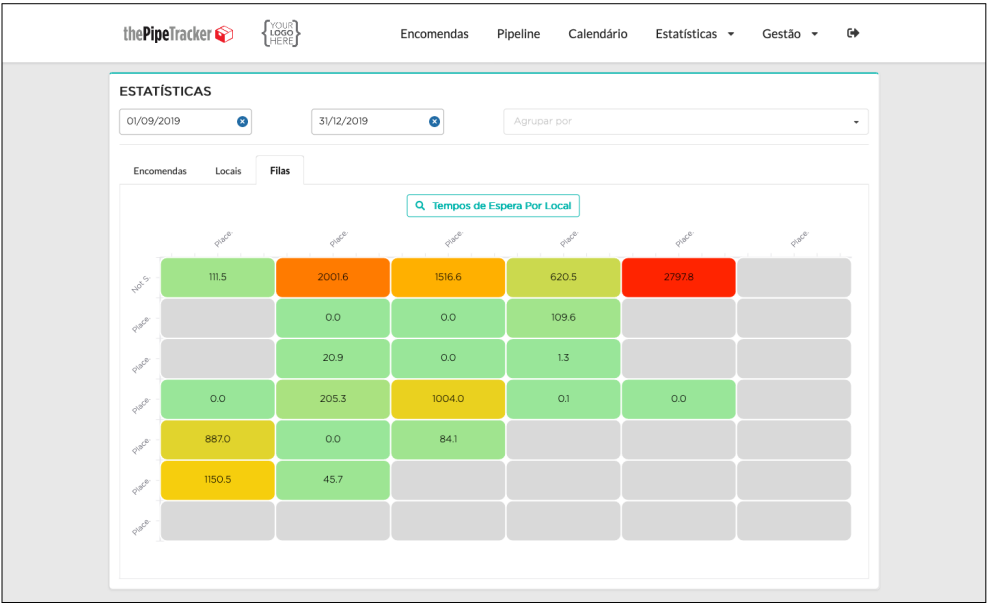


Figure 27: Waiting time in queue statistics page.

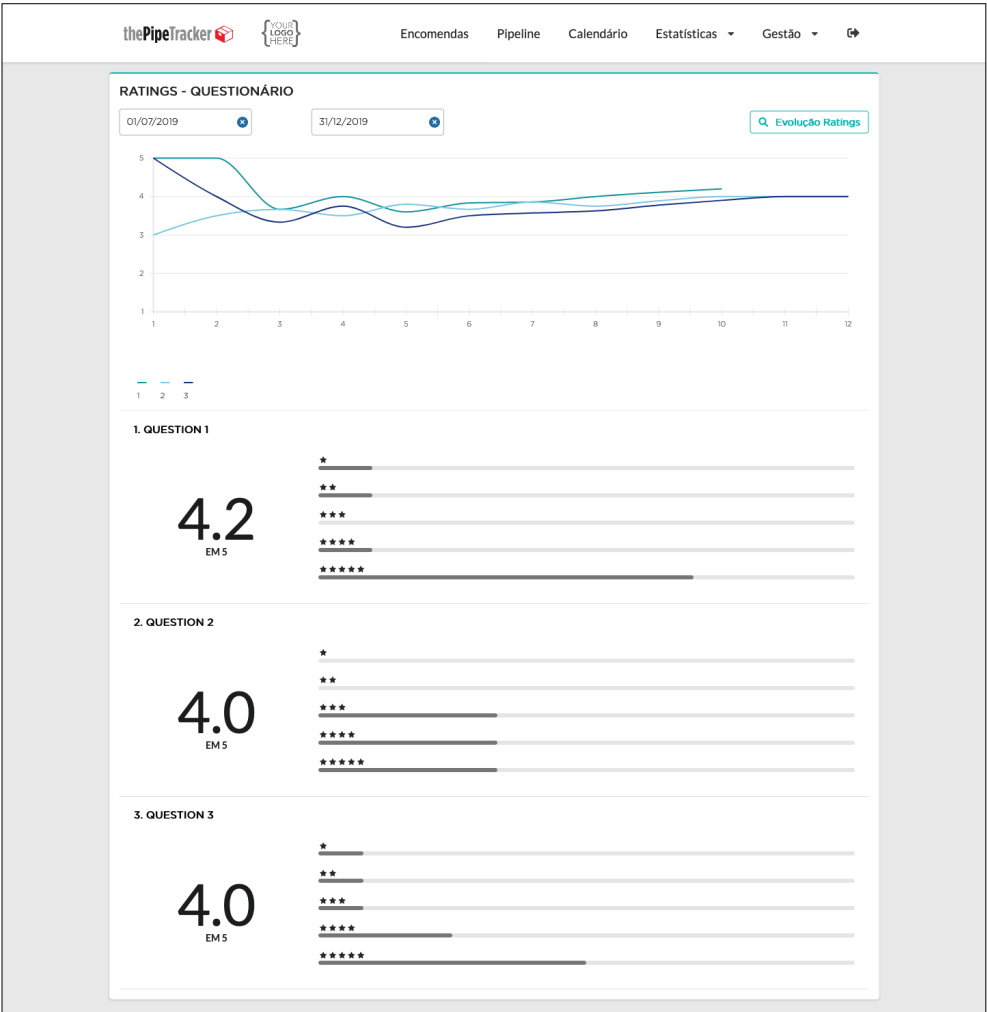


Figure 28: Ratings received from end clients page.

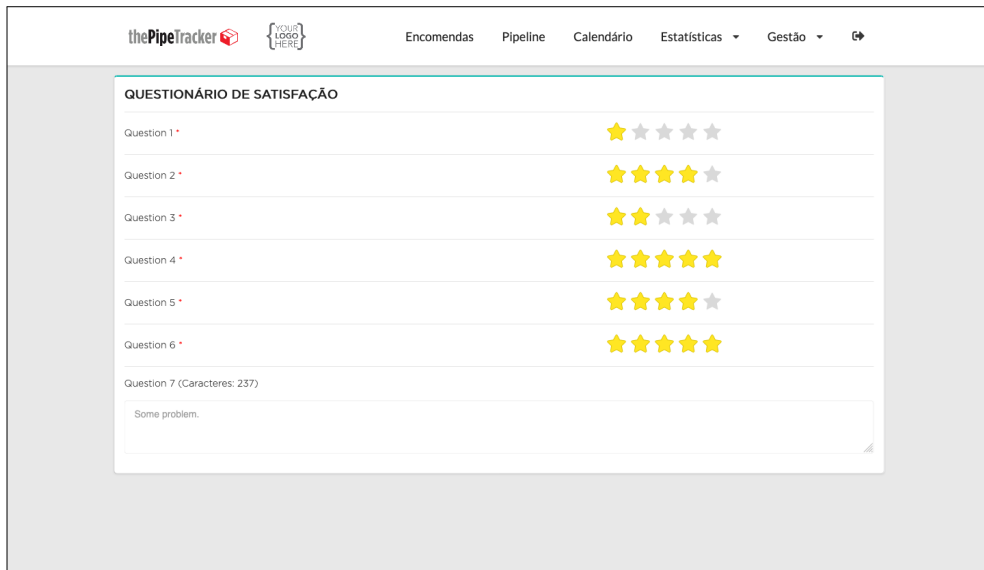


Figure 29: Example of a survey answer by a end client.

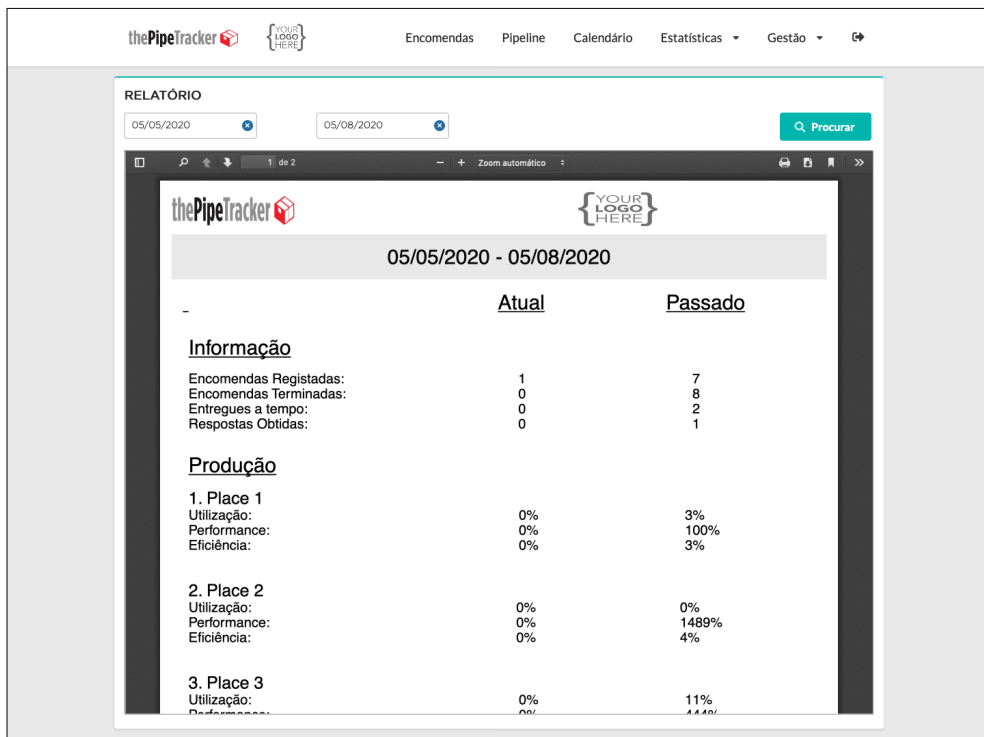


Figure 30: Automatic report page.

A.2 EMAIL EXAMPLES

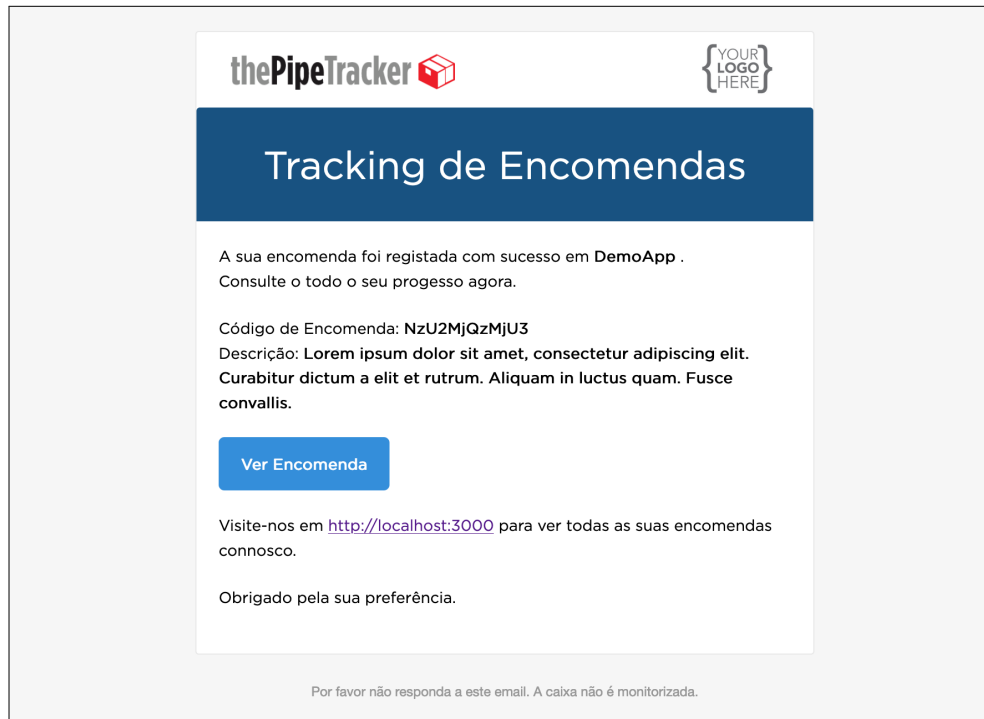


Figure 31: New package registered confirmation example email.

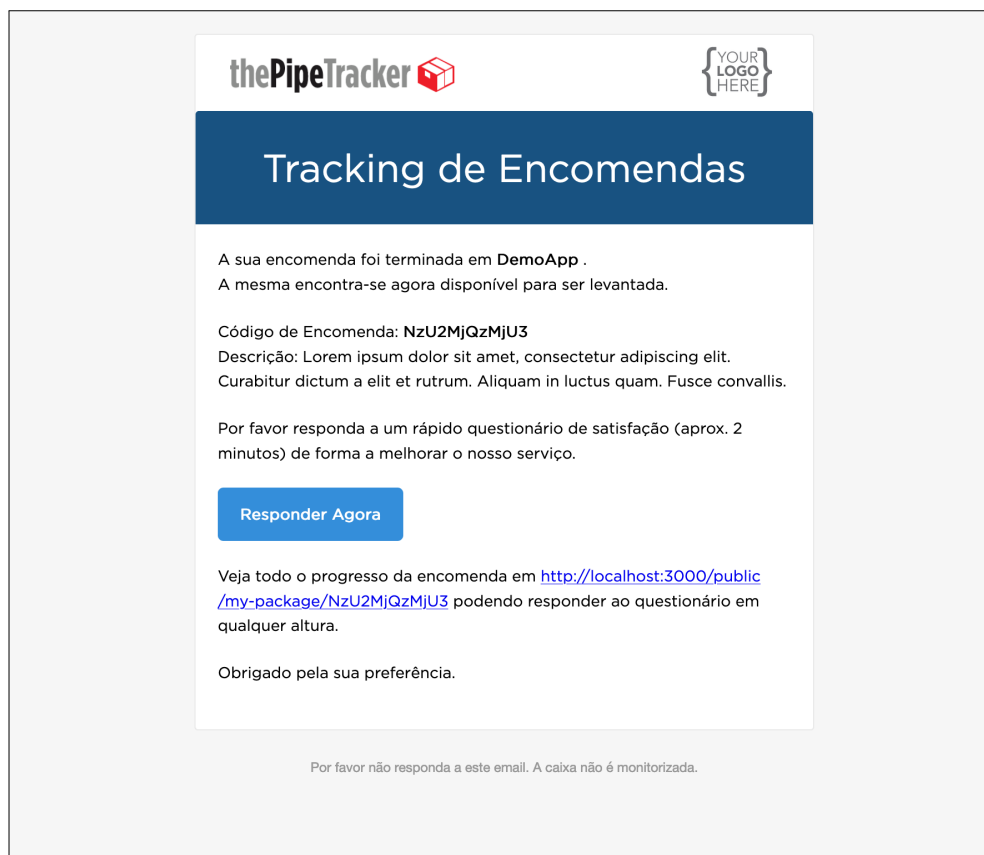


Figure 32: Package completed warning example email.

