A Big Data System for the delivery of goods in Milan

Alessandro Cortese[1], Matteo Scianna[2]

[1]alessandro.cortese@studenti.unitn.it
[2]matteo.scianna@studenti.unitn.it

Abstract— **During the last few years, many new examples of food and goods delivery services have emerged. Here we provide a system for the delivery of goods in the city of Milan, taking into account the process of orders collecting, scheduling, optimization and analytics providing. All these steps are examined both from a theoretical and a technical point of view, together with all the technologies used for implementation purposes.**

Keywords— **Delivery, Database, Optimization, Assignment Problem**

I. Introduction

The aim of this report is to go through the process of definition and implementation of a big data system for the delivery of goods in the city of Milan. This is a very current topic since many different services of this kind are spreading out in the last few years.

Given a continuous flow of orders and a set of delivery men, the system should be able to first collect and elaborate orders, and eventually to assign them to available delivery men taking into account different factors in order to provide an optimized schedule.

It is important to stress that we decided, following the example of some real delivery systems such as Deliveroo or Glovo, to focus only on the delivery of food by restaurants, bars and similar places. As a consequence, to both calculate the time required for an order to be delivered and to select and graphically represent the best route, the selected means of transport is the bicycle.

The first issue we had to deal with was to provide the system actual data. It has been tried initially to look for historical data regarding real delivery services in order to recreate a situation as similar as possible to an actual delivery system (i.e. the number of delivery men, the number of orders, the frequency with which orders are received).

Since this first research turned out to be unsuccessful, the final decision was to generate simulated orders, taking as eligible different real restaurants in the area of Milan and different randomly generated longitude-latitude couples as delivery points in the area of Milan.

Furthermore, we opted for allowing only one delivery per time per rider, so that the standard routine of a delivery man should be “current position→restaurant position→delivery position” and so on.

II. System Model

A. System architecture

The data processing pipeline of this project is divided into 3 different phases, as shown in Fig. 1.

Data Generation, Ingestion and Storing This first step refers to the process of generating and importing data for immediate use or storage in a database.

When the program is started for the first time, a set of delivery men is inserted into the *deliverymen* collection in a MongoDB database, along with an initial batch of fake food orders. Then, the generator begins to continuously create such fake orders according to a random uniform distribution. These orders, which consist of information about both the clients and the restaurants, along with a timestamp, are published to a MQTT topic and are received by a subscriber. The subscriber, as soon as the messages are received, inserts them into the *pending_orders* collection in the same database.

Computation (Matching) This phase corresponds to the actual matching between orders and delivery men. From the *pending_orders* collections the 70 (due to API limitations) oldest orders are retrieved together with all the currently free delivery men. Then, every free delivery man is matched to a single order.

In order to do so, the program solves what is called a *linear sum assignment problem*: given a set of coordinates of all the free delivery men and another set containing the coordinates of the 70 oldest orders, an origin-destination matrix C is created, where $C_{i,j}$ represents the cost of matching the i-th delivery man with the j-th order (in our case, the time required to complete the delivery). The goal is to find a complete

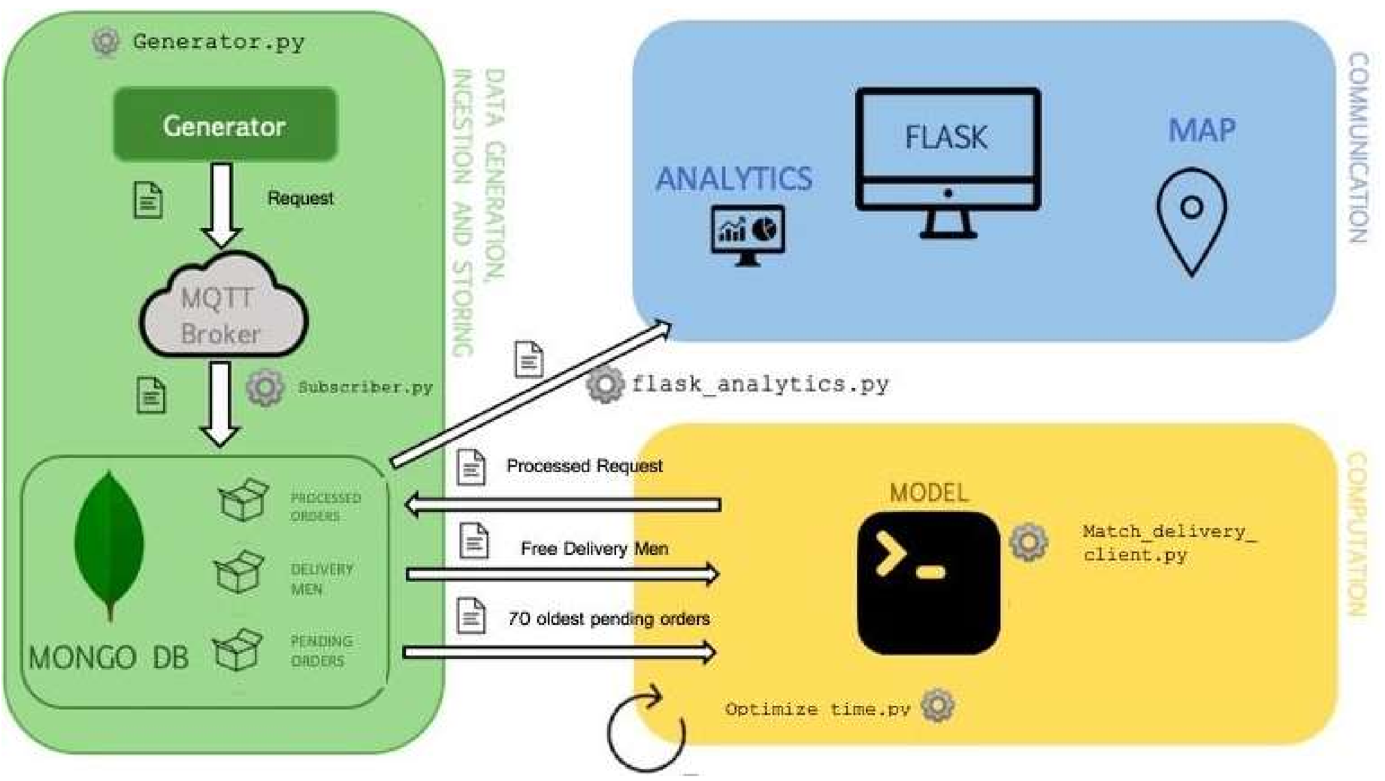


Fig. 1 Data processing pipeline

assignment of delivery men to orders which minimizes the total cost.

Formally, given X a boolean matrix where $X_{i,j} = 1$ iff. delivery man i is assigned to order j, the optimal assignment has cost $min \sum_i \sum_j C_{i,j} X_{i,j}$.

At the end of this process, every free delivery man is coupled with the order that minimizes the delivery time, with respect to all other orders and delivery men.

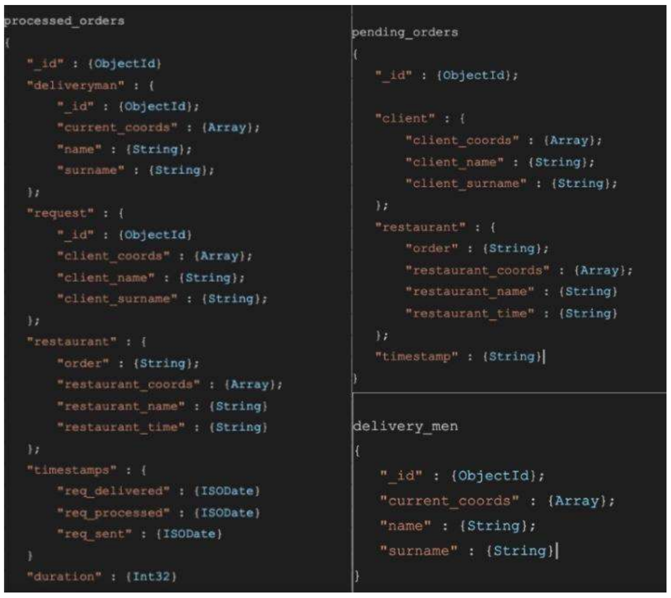
At this point, every collection of the database is updated:

- The orders which get matched are removed from the *pending_orders* collection.
- Inside the *delivery_men* collection, the couple of coordinates that define the current position of a delivery man is updated to the position of the client (*i.e.* the point in which the delivery man will be once he has finished his current job).
- For each processed order, a new element is created inside the *processed_orders* collection, containing all information about the order, the delivery man who took it, the time required etc.

The detailed data model for each collection is shown in Fig. 2.

The last point of this stage is to define the time to wait until a new run of the matching program.
At the beginning, we opted for a static time period (*i.e.* every 2 minutes). Nonetheless, we decided to also implement a function to make this time dynamic, according to the variation in the number of free delivery men.

In particular, our reasoning was that it could be useful in some scenarios to wait a slightly longer time in order to dispose of a higher number of free delivery men, which can be better coupled with new orders.
The formal procedure is explained below.


```
processed_orders
{
   "_id" : {ObjectId}
   "deliveryman" : {
       "_id" : {ObjectId};
       "current_coords" : {Array};
       "name" : {String};
       "surname" : {String};
   };
   "request" : {
       "_id" : {ObjectId}
       "client_coords" : {Array};
       "client_name" : {String};
       "client_surname" : {String};
   };
   "restaurant" : {
       "order" : {String};
       "restaurant_coords" : {Array};
       "restaurant_name" : {String}
       "restaurant_time" : {String}
   };
   "timestamps" : {
       "req_delivered" : {ISODate}
       "req_processed" : {ISODate}
       "req_sent" : {ISODate}
   }
   "duration" : {Int32}
```

```
pending_orders
{
   "_id" : {ObjectId};

   "client" : {
       "client_coords" : {Array};
       "client_name" : {String};
       "client_surname" : {String};
   };
   "restaurant" : {
       "order" : {String};
       "restaurant_coords" : {Array};
       "restaurant_name" : {String}
       "restaurant_time" : {String}
   };
   "timestamp" : {String}
}
```

```
delivery_men
{
   "_id" : {ObjectId};
   "current_coords" : {Array};
   "name" : {String};
   "surname" : {String}
}
```

Fig. 2 Data Model for each collection

Given t_i the time at which the delivery man i concludes his current job and i the number of delivery men available at time t_i, we defined as (t_i, i), $i \in \{1, .., n\}$ with n total number of delivery men, the set of couples ordered according to t_i. The optimal time is defined as the time subsequent to the highest increment of free delivery men: $t_{opt} = argmax_{t_i}(i/t_i)$.

Furthermore, in order to avoid waiting too long before a new run, if t_{opt} results being higher than a given percentile of t_i, we set t_{opt} as such percentile.

Communication and Presentation The final step of the program is to define and communicate the output.

Since we identified the users of our system as a generic company which manages the delivery of food in the city of Milan, the output is set as a dashboard containing a map and a series of potentially useful analytics and information to monitor the ongoing situation of the delivery system.

B. *Technologies*

For the realization of this project we relied on many different technologies. The aim of this paragraph is to describe them, justifying their use and mapping them inside our workflow.

First of all, MQTT serves as a bridge between the generation of our data and their insertion into the *pending_orders* collection. We opted for the use of a queue system in order to decouple the speed with which orders are created from that by which they are inserted into the database. Since we are dealing with a constant flow of data potentially very close in time one to another, and since databases are not optimized for very fast writing, this system ensures that we don't end up missing some data or more generally run into some errors during this phase of data collection.

Since we immediately thought of our data as having a nested structure and since we wanted to allow changes in the data structure over time, we opted to work using JSON format. Hence the choice of a noSQL database such as MongoDB, which has the advantages of being easy to use, scalable and most importantly to revolve around such a format.

One other technology that was initially part of our toolbox but that we eventually decided to discard is Redis. We firstly thought about using it to keep track of the occupied delivery men. Since one useful possibility of Redis is to give keys an expiration time, we exploited such functionality to create a key inside Redis for each occupied delivery man, setting its expiration time equal to the time required to complete the delivery. Doing so, every time the optimization process is run, checking for the free delivery men is equivalent to looking for the delivery men ids which are not present in Redis.

In the end, however, we realized that the same result could be achieved simply by adding a timestamp corresponding to the delivery time of the order (*i.e.* the timestamp of when the order is matched plus the duration of the corresponding trip) to the data inserted into the *processed_orders* collection and then querying the database for the various orders whose delivered timestamp is greater than the current time. Therefore, we thought it would be wiser to keep the system simpler without adding another technology on which it should rely.

As far as the communication phase is concerned, the framework we used was Flask. This library enabled the creation of a web server hosting a demo dashboard with real-time analytics and a graphical representation of the output for the user.

Finally, in order to make the whole system replicable on different machines, we dockerized the code.

III. Implementation

The project implementation can be found on GitHub at the following link: https://github.com/alescortes/food-delivery_bdt2022.

The repository contains a *docker* folder, which contains five subfolders, corresponding to the four services which are run and the Mongo database. Each subfolder contains the Dockerfile relative to the specific service, along with a *.env* file specifying environment variables.

Inside the *src* folder, the scripts called by the services are presented along with additional files, such as a *requirements.txt* file to download the necessary packages or the *functions.py* file containing a series of classes used to organize different functions called by the services.

In the *dashboard* subfolder, the code to run and render the Flask interface can be found. In the *handlers* subfolder, a series of functions to interact with the database is provided in the *mongo_handlers.py* file.

When the program is started, through the command *-sudo docker-compose -up*, four different services are run simultaneously:

1) *generator.py:* the first time such service is launched, the function *generate_restaurant_list()* makes a call to the Openrouteservice Points of Interest API and, given a couple of coordinates (chosen by us roughly at the center of Milan), collects all Points of Interest which belong to five different categories of food establishments within a 2km radius, parses them and creates a .csv file that will be used to create simulations of orders regarding real restaurants. However, such a file is already provided in the repository under the name *restaurants.csv*, in order to avoid potential API errors which could break the whole architecture.

Then, if the *delivery_men* collection in MongoDB is empty, the function *initialize_deliverymen()* is called. Such a function randomly generates and inserts into the corresponding collection 50 delivery men, each one with a random initial position inside a bounding box within the city of Milan.

In the same fashion, if the *pending_orders* collection is also empty at the beginning, the function *initialize_pending()* randomly generates 70 orders and inserts them into their collection. This step has been done so that the simulation can start straight away with a decent number of orders, without having to wait several minutes for the collection to fill.

At this point, *generator.py* starts a loop, generating random orders. The time between two new different orders is random and follows a uniform distribution between 0 and 90 seconds, in order to reproduce a more realistic situation.

Whenever a new order is generated, it is encoded in JSON format and published through a MQTT client on the *bdt2022/new_orders* topic.

2) *subscriber.py*: this piece of code connects to the MQTT client and subscribes to the same topic. When a message (*i.e.* an order) is received, it is inserted into the *pending_orders* collection.

3) *match_delivery_client.py*: this is where the scheduling takes place. The program retrieves from the database a list of currently free delivery men and a list of the oldest 70 pending orders. In case either of those two lists are empty, the process restarts after 2 minutes. The function *scheduler* is then called. This sends two requests to the Openrouteservice Matrix API. The first one is needed to compute the origin-destination matrix for the first leg of the trip (delivery men → restaurants) and solve the assignment problem. The second one is linked to the second leg of the trip (restaurants → clients) and is needed to compute the total duration of each trip. The database is then updated and the next run of the scheduling process is computed through the *optimize_time()* function as described in II.A.

4) *flask_analytics.py*: this program is used to run the dashboard on 127.0.0.1:5000. It runs a series of queries to the database in order to compute relevant analytics and renders the map through a call to *create_map()*.

IV. Results

In terms of final outcome, the results of the program is a dashboard containing several pieces of useful information. First of all, a map with the initial position of each delivery man, the location of the restaurant and of the client, along with the route among these points is provided, as shown in Fig. 3.

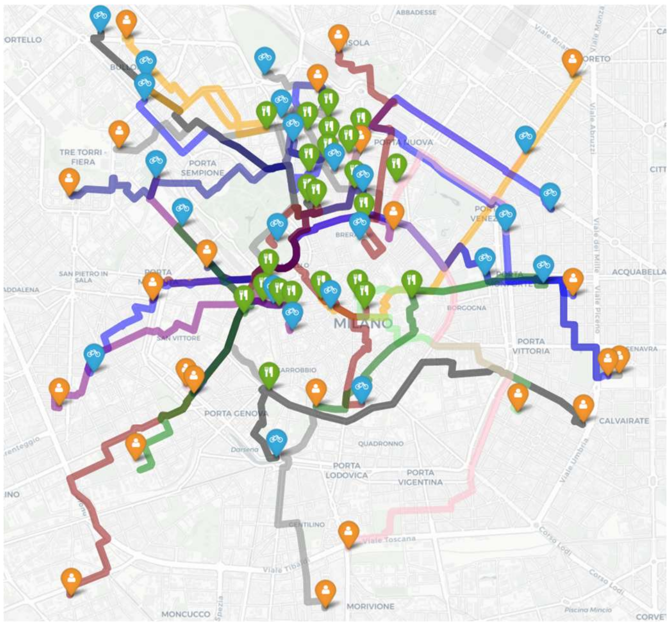


Fig. 3 Flask interface (Map)

Below the map, various analytics regarding the orders are shown (Fig.4):

- Average waiting time, *i.e.* the average time that goes between an order collection and its delivery, a useful parameter in order to keep track of the performance of the model;
- The total number of pending orders;
- The total number of orders which have been delivered;
- A ranked list of the most common ordered items, together with their absolute frequencies;
- The total number of occupied delivery men, along with their names and current destination.

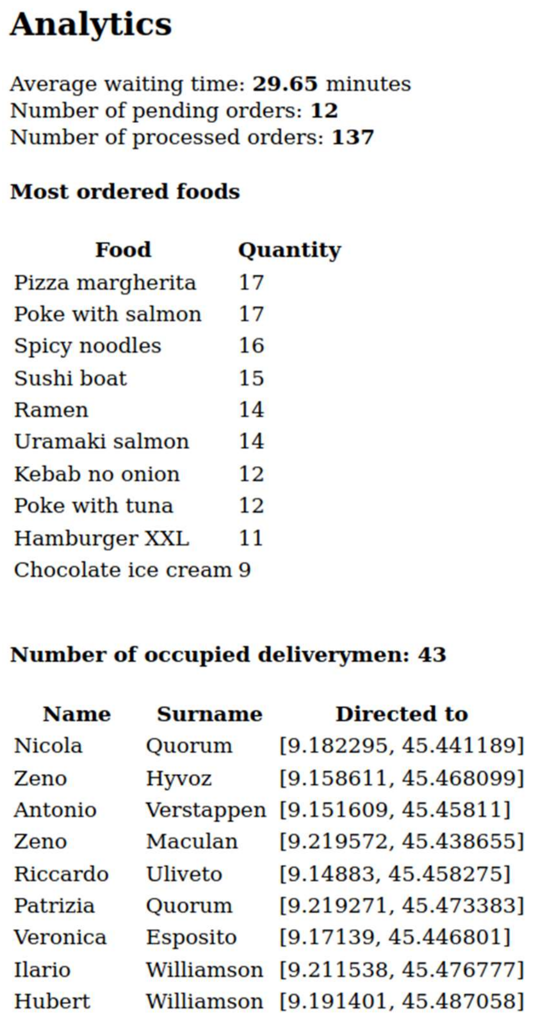
In order to test the goodness of the model, we analyzed three scenarios corresponding to different definitions of the time to wait before each new run of the scheduling.

TABLE I

AVERAGE WAITING TIMES ACCORDING TO DIFFERENT SCHEDULING TIME CONFIGURATIONS

Time until new scheduler run	**Average waiting time (in minutes) after 90 minutes**
Static (every 2 min.)	25.21
Optimized with 25th percentile	33.24
Optimized with 50th percentile	35.43

As shown in Table 1, with the function *optimize_time()* mentioned above, after one hour and a half the average waiting time is 33.24 and 35.43 minutes using the 25th and 50th percentile respectively. This value improves quite significantly by setting a fixed time of 2 minutes before a new match (25.2 minutes), therefore not providing any kind of such optimization. We finally understood that despite our initial intuition, our proposed dynamic selection of the scheduling time is not very effective in this situation.



Analytics

Average waiting time: **29.65** minutes
Number of pending orders: **12**
Number of processed orders: **137**

Most ordered foods

Food	Quantity
Pizza margherita	17
Poke with salmon	17
Spicy noodles	16
Sushi boat	15
Ramen	14
Uramaki salmon	14
Kebab no onion	12
Poke with tuna	12
Hamburger XXL	11
Chocolate ice cream	9

Number of occupied deliverymen: 43

Name	Surname	Directed to
Nicola	Quorum	[9.182295, 45.441189]
Zeno	Hyvoz	[9.158611, 45.468099]
Antonio	Verstappen	[9.151609, 45.45811]
Zeno	Maculan	[9.219572, 45.438655]
Riccardo	Uliveto	[9.14883, 45.458275]
Patrizia	Quorum	[9.219271, 45.473383]
Veronica	Esposito	[9.17139, 45.446801]
Ilario	Williamson	[9.211538, 45.476777]
Hubert	Williamson	[9.191401, 45.487058]

Fig. 4 Flask interface (Analytics and statistics)

V. Conclusions

While proceeding implementing this system, we identified several limitations, the most evident stemming from the fact that the program is a simulated model of a food delivery system and therefore lacks a set of features present in real-world applications (such as the handling of a monetary transaction, the possibility for delivery men to refuse orders or to arrive late, etc.).

Other limitations are linked to the Openrouteservice API. For example, the origin-destination matrix computed through the Matrix API can contain at most 3500 cells. This leads to a strong restriction on the total number of delivery men and orders that can be matched during a single process and explains why such parameters were set to respectively 50 and 70.

Furthermore, the Direction API used to plot the delivery men's routes on the dashboard map can be called at most 40 times per minute.

Another issue we encountered had to do with the random coordinates used for the orders and the starting locations of the delivery men. Initially, we defined a function to generate them inside the administrative perimeter of the city of Milan. Such a choice led to a series of frequent API errors, since it could happen that coordinates defined in this way would be too far from an actual road (*i.e.* in the middle of a field or of an industrial complex) and so it was impossible to compute a trip for the delivery man. As a consequence, we reduced the perimeter by generating coordinates inside a smaller bounding box.

For all these reasons, as of now this system cannot be scaled to perform correctly with a very high number of delivery men and requests.

There are also many potential improvements that can be made on the system itself.

First of all, the orders chosen to be matched are selected exclusively by the time they were generated (*i.e.* oldest first). One possible enhancement could be to consider other factors together with this one, such as the distance between each restaurant and each delivery man, in order to compute a priority score for each order.

Furthermore, up to now the only temporal features that we considered during the process were the time required to complete the route "initial position→restaurant→delivery point". One additional factor to be considered could be the time required for the actual food preparation, adding a new important variable into the optimization process.

Finally, our choice to use MQTT was motivated by its simplicity of use and its efficiency. Indeed, in a real world scenario with a higher frequency of orders, it could be reasonable to use more sophisticated technologies such as Apache Kafka, which enables better throughput, built-in partitioning and fault tolerance.

References

[1] Burkard, Rainer E., and Ulrich Derigs. "The linear sum assignment problem." *Assignment and Matching Problems: Solution Methods with FORTRAN-Programs*. Springer, Berlin, Heidelberg, 1980. 1-15.

[2] Toth, Paolo, and Daniele Vigo, eds. *The vehicle routing problem.* Society for Industrial and Applied Mathematics, 2002.

[3] Rad, Babak Bashari, Harrison John Bhatti, and Mohammad Ahmadi. "An introduction to docker and analysis of its performance." *International Journal of Computer Science and Network Security (IJCSNS)* 17.3 (2017): 228.

[4] Soni, Dipa, and Ashwin Makwana. "A survey on mqtt: a protocol of internet of things (iot)." *International conference on telecommunication, power analysis and computing techniques (ICTPACT-2017)*. Vol. 20. 2017.

[5] Chauhan, Anjali. "A Review on Various Aspects of MongoDb Databases." *International Journal of Engineering Research & Technology (IJERT)* 8.5 (2019).