Human Computer Interaction a.y. 22-23

Project Report - VoiceFork

Alessio Lucciola - lucciola.1823638@studenti.uniroma1.it
Danilo Corsi - corsi.1742375@studenti.uniroma1.it
Domiziano Scarcelli - scarcelli.1872664@studenti.uniroma1.it

June 28, 2023

Contents

1 Introduction **2**

2 Competitors analysis **2**

3 Need finding **4**

3.1 Questionnaire 4

3.2 Paper Analysis 5

4 Final considerations **6**

5 Alexa Skill: Voicefork **7**

5.1 Implementation details 8

5.2 Backend structure 17

6 Real Scenario Tests **18**

7 Current Limitations and Future Developments **19**

8 Conclusions **20**

1 Introduction

The system we created is called **"Voicefork"**, a **Voice User Interface** that allows users to make reservations at restaurants using their voice. After some research in order to understand how to implement the system, we decided to use **Alexa** which offered the possibility to create a skill through an online console using NodeJS freely. This was made possible by developing a skill specifically for this purpose through which all the information inherent in the booking was collected, making it easier and faster than doing it manually on the smartphone. In order to test the application, we decided to use the "Tripadvisor European Restaurants" [5] (limiting only to the Italian restaurants) that allowed us to test the system as it was released in a real scenario.

2 Competitors analysis

Before starting building the system, we briefly analyzed some apps from competitors to understand how they dealt with the problem:

- **The Fork**: It is an application developed by "LaFourchette" that allows users to book restaurants. The Fork's objective is to help restaurants to be fully booked and help customers to find restaurants that best suit their requirements. The main features available via the reservations management software are:
  - Online Reservations: The online reservation system offered by TheFork enables customers to book a table in a restaurant at any time of day;
  - Online Reviews: Users can write reviews that can help the restaurant to develop an online reputation;
  - Reservation Buttons: The Fork is integrated with other apps so that customers can book a table in a restaurant from TripAdvisor and the social networks Facebook and Instagram. This is a particularly useful feature because today's consumers want immediacy and have no time to lose;
  - Special Offers: It is useful to attract more customers with special offers on quiet days or during off-peak periods. This will encourage customers to visit a TheFork page with a certain offer and reserve to that specific restaurant rather than another;
  - Gamification features: When users make reservations, they earn Yum points that they can use to get discounts and in-app prizes.
- **Quandoo**: It is a restaurant reservation platform that connects customers with restaurants. It's an (mobile/web) application to discover restaurants and book tables. Customers can find restaurants by choosing the city and filtering them by some categories such as open now, type of cuisine and the restaurant category (e.g. breakfast, romantic, branch, ecc..). They can also

search a specific restaurant (regardless of the city) by typing its name in the search bar and they will be offered a list of restaurants with a similar name. Once a restaurant is found, a reservation can be made by specifying the number of people, the date and time of the reservation and some contact info (along with possible special requests - e.g. allergies). All customers can easily edit or cancel their reservation via the Quandoo confirmation email they received after making a booking. Within the email, customers can select to Change or Cancel their reservation. When selected, they will be redirected to the Quandoo website to finish the process. They will finally receive an email confirming their update. There is also a loyalty point program that allows users to earn points by performing some actions like creating an account, making reservations, writing reviews and referring friends. When they reach 1000 points, they can get a cashback.

- **Yelp**: It is a crowd-sourced local business review and social networking site. The site has pages devoted to individual locations, such as restaurants or schools, where Yelp users can submit a review of their products or services using a one to five star rating scale. Businesses can also update contact information, hours, and other basic listing information or add special deals. In addition to writing reviews, users can react to reviews, plan events, or discuss their personal lives. Yelp users can make restaurant reservations in Yelp through Yelp Reservations by searching a restaurant and filling a form with the time and day of the reservation, the number of people and contact info. Some of the Yelp features are integrated with Siri, so users can look for nearby places (including restaurants) and see their details (e.g. the address and the contact info) using the voice assistant (it isn't possible to make online reservations though).

- **Google Duplex**: It is an AI-powered voice dialer that works via Google Assistant. Through Google Assistant, it is possible to make a reservation, and it will take care of the rest. Google Duplex pulls the information from Google Assistant and calls the business for you. Then, using advanced AI and voice emulation, Google Duplex holds a conversation with employees to place the booking. Unlike traditional voice dialing software, Google Duplex is designed to sound as normal as possible, featuring a human-like cadence and intonation. The net effect is it promises a more natural conversation for the person on the other end of the line. Once your booking is complete, Google Assistant will notify you with the details. The pipeline to reserve a restaurant is the following:

  1. Open Google Assistant.
  2. Initiate Google Assistant by typing on the device keyboard, or just say either, "Hey Google," or "OK Google."
  3. Ask Google Assistant to book you a table at the restaurant of your choice.
  4. If the restaurant accepts bookings, Google Assistant will ask you what time you want to visit, and for how many.
  5. Tell Google your preferred time and how many people will be going.

6. Google Assistant will then ask if an alternative time also works, just in case the restaurant doesn't have availability. Simply respond with yes or no, or choose a timeframe from your device screen.
7. Google Assistant will then display your booking details to check everything is correct. If it is, simply say, "OK," or tap Confirm. If there are errors, say or tap Cancel.
8. Once the details are confirmed, Google Assistant will hand everything off to Google Duplex so it can book your table.
9. Google Duplex will dial the restaurant and announce itself as Google Duplex to the call recipient. It will then explain it's calling to book a table for a client.
10. Google Duplex will then discuss the details of the booking with the restaurant worker, including your preferred time and the number of people attending. If the restaurant is unable to seat you at that time, Google Duplex will ask about an alternative slot, based on the timeframe you provided.
11. Once the Google Duplex call is complete, Google Assistant will notify you.
12. If your booking was successful, you'll see the time and details on an information card. If Google Duplex was unable to book your table, Google Assistant will let you know, and offer some alternatives.

Among the services that we analyzed, Google Duplex was the one that inspired us the most since the task is similar to the one we want to create. For this reason, we will take inspiration from this service in order to understand how to structure the interaction between user and machine. The other services were still useful to understand what is the information needed and the typical pipeline when making a reservation. They may also be helpful to understand how to extend the system with further tasks in a possible future development.

3 Need finding

We analyzed the possible scenarios in which this system could be useful. The main requirement is to make a reservation at a restaurant. Of course, this has many facets, and our goal is to address it in depth in the following sections to determine which details to focus on most. To do this, we decided to administer a questionnaire and analyze some papers to get the information we needed.

3.1 Questionnaire

We designed a questionnaire with the aim of testing, on a large scale, the needs that have been identified. The most interesting considerations that emerged from the questionnaire were that when making a reservation with a voice assistant, people are generally willing to ask for clarifications to ensure that the reservation runs smoothly. They understand that without these clarifications, the booking reservation process may not be possible and they prefer a simplified experience when booking with a voice assistant, ideally they would like to answer no more than 2 or 3 questions. In general, when booking a restaurant, people tend to choose it based on the number of reviews and their ratings. If there are errors or missing parameters during the reservation process, the voice assistant should ask for clarification to gather the necessary information. If the error persists, the voice assistant should provide further guidance and information to help the user solve the problem. In situations where there are multiple errors, it may be necessary to restart the conversation but always provide detailed explanations. To resolve any ambiguity in choosing a restaurant, the voice assistant should present a list of alternatives; this can be sorted by various criteria, such as reviews, convenience, atmosphere, cuisine, and area. The disambiguation process can take into account user characteristics such as booking habits or location to find the most suitable option. The voice assistant should aim to provide clarification using minimal information by allowing users to respond directly with ”yes” or ”no” to quickly disambiguate their preferences.

3.2 Paper Analysis

In addition to the analysis we already described, we also looked at some papers we found online that tried to solve similar problems to ours. Here we make a summary of the information we extracted:

- In case we have a list of restaurants to choose from, it is convenient to directly ask the user to choose one of them in order to leave him in control of the situation (always ask for their confirmation) [1].
- The disambiguation could be based on habits, for example, if the user wants to book at ”Ristorante da Mario”, there are two nearby but he has already booked in one of these then ask if it is the one where he booked last time [1] [2].
- We have established that voice assistants should always ask for clarification when they encounter an ambiguous question, since according to the results of the paper, users seem generally open to responding to such requests.
- Disambiguation could be based on users’ preferences such as cuisine, reviews, nearest restaurants, etc.. [2] [3].
- When we have (dynamic) lists of elements to choose from, a possible method could be to sort it according to the preferences described above (e.g. the nearest restaurant).
- Clarifications should be inserted with as little information as possible (not too complex sentences). Furthermore, if the options are few (e.g. only two places with the same name then it is better to ask immediately ”either that or that” without giving too much additional information) [2].
- Disambiguate only when needed (e.g. if a user asks to reserve a table in Rome, it’s useless to get also the restaurants in nearby cities) [3].

- Yes/No questions are the simplest method of disambiguation, use this approach when one option is more likely than all the others and the cost of being wrong is higher than a simple guess (e.g. if there are two "Ristorante da Mario" close to the user, ask immediately to book of the two based on a certain criterion) [3].
- If the user does not know where the restaurant is but only knows the name then the situation is more complicated. Further disambiguation methods may be needed (e.g. the system can instead filter by the type of cuisine or average rating) [3].
- If an error is detected: A disambiguation occurs and the system should ask questions being sure of not returning again to an error situation which would result in a loop. So we have to carefully evaluate the questions to be asked [4].
- In case of disambiguation, open a branch from the current dialog, ask questions to find a solution and as soon as this is found, return to the initial primary branch [4].
- It is possible to start the conversation in various ways (e.g. book a table for, make a reservation to, I'd like a reservation, get a table, make me a, etc..) and the assistant it should be able to recognize all these modes and start the related task (restaurant reservation). To complete the task, the system must have all the necessary information (e.g. date, time, place and number of people). The user must be able to give this information when and how he wants and if, once the request has been made, all the necessary information is missing (e.g. the user has not said the number of people), the system must understand this and must ask further questions to fill in the missing info and then complete the request to perform the task. There are various cases that need to be managed, especially if you ask to book on special dates (e.g. Easter but Easter changes every year, it's not a specific day like Christmas). Furthermore, it is necessary to understand whether the user means in the morning or in the afternoon (e.g. make a reservation at 9 - am or pm?), and the day (e.g. make a reservation for tuesday - the next or another one?). In general, the system must avoid asking the user for information they have already given. Therefore, to carry out the discourse, a flow that is familiar and logical for the user must be found and if the user does not specify some info, the same flow can still be followed (e.g. if the flow is place $\rightarrow$ date $\rightarrow$ time $\rightarrow$ people) and the user doesn't say date and number of people, follow the same flow and ask first date and then number of people [3].

4 Final considerations

The analysis conducted so far highlights the need to avoid any kind of error during the booking process. It is seen that this aspect is the main reason why a customer decides to use the traditional method rather than the voice assistant method. For this reason, a feedback mechanism from the assistant is important after each key step (e.g., confirmation of various fields such as restaurant name, date, time, and number of people) of the conversation. In this way, the user is always in control and knows exactly what the status of the reservation is and how to proceed in each situation. On the other hand, the conversation should always be smooth and as natural as possible, so the feedback mechanism is used only after a key step. For the same reason, it is impossible for the assistant to list all restaurant names to the user if there is more than one with the same name. So we decided to focus mainly on the latter, trying to minimize disambiguation between restaurants, while providing the most natural guided conversation possible for both experienced and casual users. Among the several issues we described so far, we decided to deal with them in several ways:

- In case we have more restaurants to disambiguate, we will try to make questions in order to minimize the set of restaurants until we reach a final restaurant. At each iteration, the goal is to select a discriminative field to discard as many restaurants as possible so as to minimize the number of questions and avoid bothering the user.
- We will use recent reservations to build a context so that the user can make a faster reservation (e.g. if there are more restaurants to disambiguate and the user already reserved in one of them, that restaurant will be given priority).
- Most errors will be directly solved by Alexa that is delegated to retrieve all the mandatory fields to make the reservations. If it can't get a response from the user, Alexa will reprompt until the session runs out.
- In order to avoid possible confusion, the user will be asked to reply to short sentences in order to clarify something (e.g. reply only with yes/no instead of choosing from a large set of elements that can make the disambiguation process difficult). In general, we prefer a slightly longer conversation that has a higher probability of getting the correct result.
- We will save all the information in variables to avoid to ask several times for the same things (e.g. if a user asks to reserve a table in Rome, we'll keep that information and get the list of restaurants that are only located in Rome)
- Users must keep control over the conversation. For this reason, we'll always ask for confirmation after finalizing the reservation process.
- Users must be able to start the conversation in several ways (e.g. reserve a table, make a reservation, etc..). This will be managed by inserting several user utterances.

5 Alexa Skill: Voicefork

The ultimate goal of the project was to develop a working voice skill. We initially tried to implement this type of interaction with Google and Apple assistants, but we found that they had limitations that did not allow us to achieve a guided and natural conversation at the same time. So we decided to implement it as an Alexa skill. We took the interaction model designed during prototyping and brought it to life. The skill allows the user to make a reservation by indicating the name of the restaurant, the date and time for which they want to reserve a place, and the number of people. It is also possible to improve the user's experience by using their location (in case they want to make a reservation at a restaurant that is in their vicinity), city (in case they want to make a reservation at a distant restaurant), and analyze their habits (e.g., figuring out which restaurants they book at most often), while always offering the ability to summarize the order and allowing users to confirm or decline actions.

5.1 Implementation details

To start the skill, the user must activate it with the keywords ”Voicefork” (e.g., ”Alexa open Voicefork”). From this point, the user is guided step by step by the skill. If the user uses an invalid response during the flow of the conversation, the skill explains to the user how to interact with the skill according to the stage they are in. The entire conversation is based on the use of a few key words from the user that explain the user's intent. Each possible intent in the conversation is handled by a specific lambda function. All handlers (lambda functions) defined in the skill are briefly described below:

- **LaunchRequestHandler:** It captures the invocation phrase that launches the skill.
- **MakeReservationIntent:** It collects the essential parameters to make the reservation, which are: the name of the restaurant, the date, the time and the number of people. With the addition of the user's city or location, if any, as additional parameters to enhance the experience.
- **ReservationContextResponseHandler:** It manages the user's context in order to find the best match between the name of the restaurant provided as input and those where it is possible to make the reservation.

The latter intent is the most important since it manages the reservation process and contains the logic to compute the user's context and to handle the disambiguation process (if the system is uncertain on which restaurant to make the reservation). The following lines describe in detail how the intent works.

The **context reservation system** works this way:

1. The user gives all the details for the restaurant they want to reserve. (`restaurantName`, `date`, `time`, `numPeople`). Let's say the user wants to book the restaurant ”Pizzeria Da Pulcinella”.
2. The system makes a restaurant search based on the query (Pizzeria Da Pulcinella). If the localization is active, the system uses the user's coordinates, otherwise, it uses the city that the

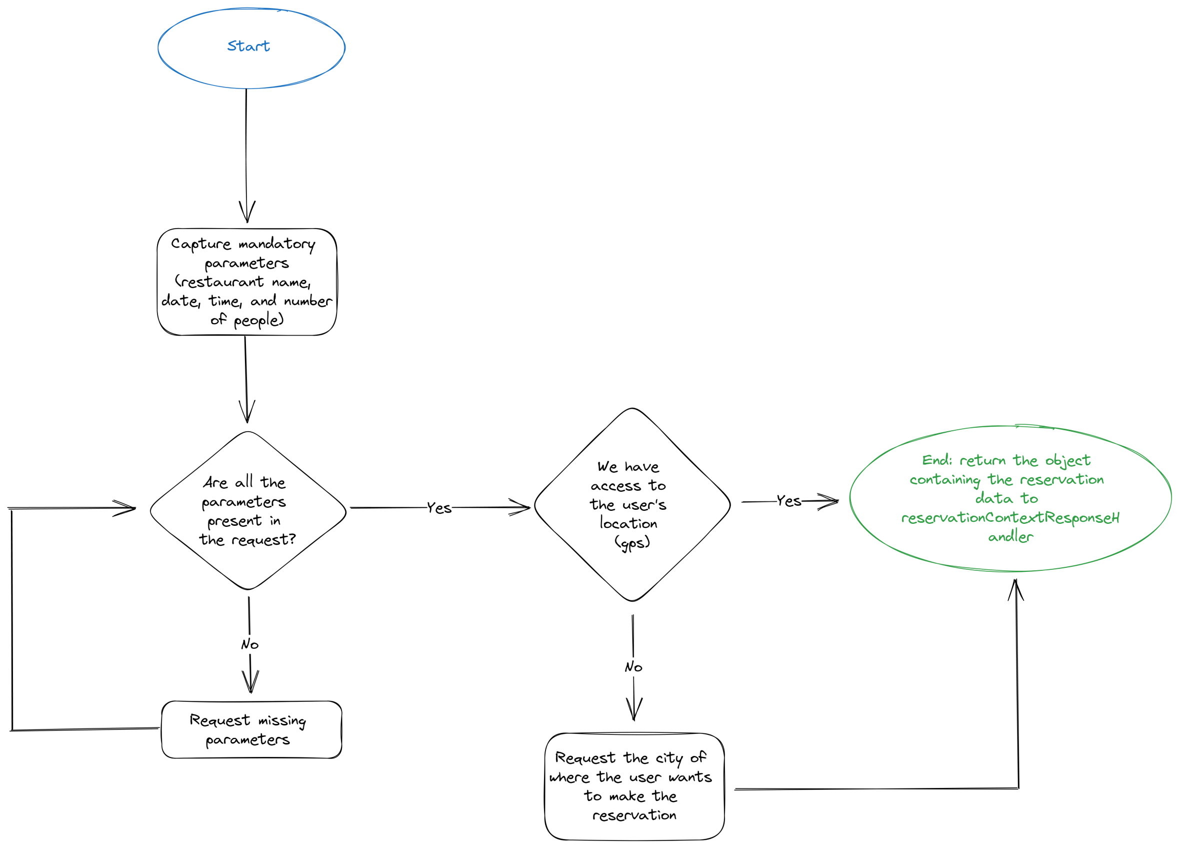
Start
Capture mandatory parameters (restaurant name, date, time, and number of people)
Are all the parameters present in the request?
Yes
No
Request missing parameters
We have access to the user's location (gps)
Yes
No
Request the city of where the user wants to make the reservation
End: return the object containing the reservation data to reservationContextResponseHandler

Figure 1: Make Reservation Intent

user has to input (further details below). This returns a list of the restaurant along with a distance $d_n \in [0, \sim 1]$ inside the `nameDistance` field, which represents the distance between the query and the restaurant name.

3. The system filters out the restaurants whose name distance from the query is less than the `DISTANCE_THRESHOLD`, in order to remove from the further computations the restaurants that are very likely to not be the desired ones.

4. Did that, the system generates a `ReservationContext` object with the information about the context the user is in. This object contains information about the current day, time and location, along with the day and time of the reservation the user wants to make. Once done, this `ReservationContext` object is matched with the averaged `ReservationContext` objects of each restaurant in the search results, which will output a `contextDistance` $d_c \in [0, \infty]$, that represents how much the is the context similar to the usual context when the user makes a reservation for that particular restaurant. Note that the context for a restaurant is defined only if the user has a reservation history for that particular restaurant. Also note that usually, an acceptable context distance value is around the range $[0, 3]$. If the user never reserved for the

restaurant r, then `contextDistance = null`.

5. For each restaurant, an aggregated score (that somehow models the probability that the restaurant is the correct restaurant, so the higher the better) is computed from the `nameDistance` value and the `contextDistance` value. First, the `contextDistance` has to be normalized in order to be a value in $[0, 1]$. The normalization follows a `NORMALIZATION_MAP`, that is an object that maps, for each value of `contextDistance`, its normalized value. The values that are not explicitly mapped that are in between two mappings are linearly interpolated, meaning an intermediate value is assigned. This ensures to have a particular distribution of values for the `contextDistance` that is spread for values in $[0, 3]$, and doesn't change much for values in $[3, \infty]$. This is because most of the values will be in $[0, 3]$.

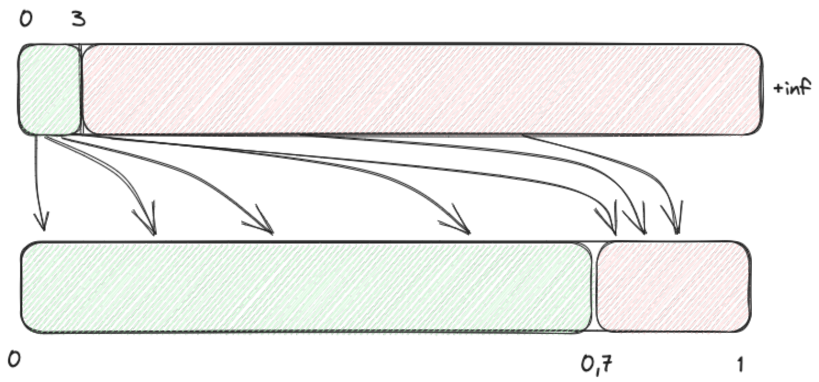

Figure 2: Context distance normalization

Once done that, we have to distinguish three cases: To ease the mathematical expressions, let d_n be the `nameDistance` and d_c the `contextDistance`:

- If the restaurant has both d_n and d_c defined, and $w \in [0, 1]$ is the weight to give to the context, we compute the score as:

$$\text{score} = 1 - ((1 - w) \cdot d_n + w \cdot \text{normalized}(d_c))$$

- If the restaurant has only d_n defined and $d_c = \text{null}$, we compute the score as:

$$\text{score} = 1 - \min(\max(d_n, 0.05)^{0.5}, 1)$$

  The min is used to cap the value to 1, and the max is used to lower bound the minimum to 0.05. The 0.5 exponent is a scaling factor that regulates the importance that is given to the score of the restaurants that have no context, with respect to the ones that have a context defined. The higher, the more importance is given.

- If all the restaurants in the list have $d_c = \text{null}$, we compute the score by just doing:

$$\text{score} = 1 - d_n$$

Location Boost After some trial and error, we found out that probably is a good idea to give a little boost to the score of the restaurants that are near the user location. Even if the restaurant search is restricted to a radius of 50km from the user position, it's possible that if two restaurants have a similar `score`, the one that's near the user may be the correct one. Because of this, when the score is computed we assign a *boost*, which is a factor in the range of $[0, 0.2]$ that is added to the restaurant's score, depending on its proximity to the user. In particular, the *boost* is computed as following:

1. The `locationDistance` (d_l) field, which represents the distance in meters from the user's location to the restaurant's location, is normalized by dividing it by the maximum distance $(50,000m)$

2. A `locationScore` $s_l \in [0, 1]$ is computed from the normalized location distance:

$$\begin{cases} s_l = \min\left(\log_{100}\left(\frac{1}{\text{normalized}(d_l)}\right), 1\right) & \text{if } d_l < \frac{50{,}000}{3} \\ s_l = 0 & \text{otherwise} \end{cases}$$

   The higher s_l, the closer is the restaurant. The choice to not consider restaurants above $\frac{1}{3}$ of the max distance was done in order not to boost each restaurant in the list, but only the closest ones.

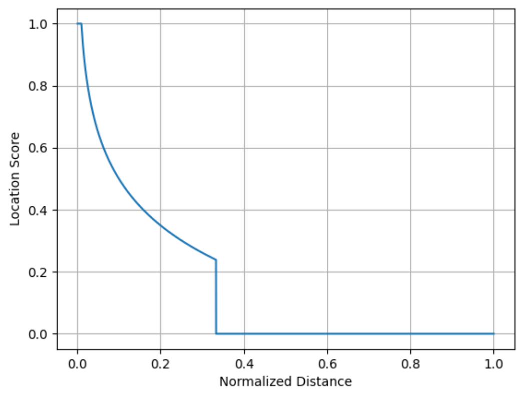

Figure 3: Location score computation

3. The `locationScore` s_l is multiplied by a `LOCATION_BOOST` $\in [0, 1]$ factor, which represents the maximum amount of boost that a score can receive (in our case it's 0.2), and then it's added to the restaurant score, capping the value to 1.

Priority Buckets At the end of this process, we will have several restaurants with a certain score between 0 and 1. Since the score is given with a combination of context information, user's location and name distance, it means that the restaurant that the user wants is probably the one with a high score. This means that we may have a long list of restaurants, but maybe the user is not interested in the ones with lower scores so, in order to avoid a long disambiguation phase, we decided to divide the restaurants into buckets depending on the score:

- High confidence bucket: Restaurants with a score in the range [1, 0.6]
- Medium confidence bucket: Restaurants with a score in the range (0.6, 0.4]
- Low confidence bucket: Restaurants with a score in the range (0.4, 0.1]

So, we only focus on the first bucket with at least one restaurant. This final system was decided after several tests in which we noticed that the restaurant that the user wants is in the bucket that is eventually considered and in which to search for the restaurant. Usually, the desired restaurant is the first bucket (for this reason all the restaurants in the other buckets are ignored) but it may also happen that there are no restaurants in the first bucket (because all restaurants have a lower score) and they reside in the second or third bucket. This usually happens when the user specifies a name that is a slightly different from the real name of the restaurant (e.g. the name of the restaurant is 'Pizzeria da Mario' but the user says 'Pizzeria da Marione'). Still, we noticed that the desired restaurant is still in the bucket that is analyzed. Of course if the user says a name that differs a lot from the real one, they will never be able to reserve a table at that restaurant but this is an unsolvable problem.

Once we select the bucket, we still may have several restaurants. This may be because those restaurants have a similar name, a similar context and both in a similar distance from the user (e.g. We may think of a scenario in which the user goes to different restaurants called 'Pizzeria Da Mario' and 'Pizzeria Da Marione'. Both restaurants have more or less the same name, they are located at the same distance and have a similar context because the user books a week at the first restaurant and the following week at the second, always for dinner on Sunday). Since those restaurants have a similar score, we'll be likely to be put in the same bucket but still the system may have some doubts as to which restaurant to choose. For this reason we have to apply disambiguation and ask the user to confirm which is the right restaurant.

The disambiguation pipeline is the following:

- If there is only 1 restaurant, that will be the final restaurant to which to make the reservation.
- If there are 2 restaurants to disambiguate, take the one with the highest score and ask the user if that's the desired restaurant. In order to help the user in this decisional process, they will be given some other useful information like the address of the restaurant. If the user confirms then it is the desired restaurant, otherwise it will be discarded and the remaining one will be considered.

- If there are more than 2 restaurants a more complicated approach has to be used in order to iteratively remove unwanted restaurants and get to the desired one.

Regarding the third case, we selected several fields that could help the user to decide what is the right restaurant and eventually allow the system to finalize the reservation process. These fields are:

- LatLng: The position of the restaurants.
- City: The city (also the neighborhood) of the restaurant.
- Cuisines: The type of cuisines of the restaurant (e.g. Italian, Thai, Japanese).
- AvgRating: The average rating given by other users.

At each step, we select the field that helps to discriminate more meaning that we want to reach the final decision as fast as possible in order to avoid bothering the user with many questions.

In order to know which is the field that is able to discriminate the most, we:

1. Take the first non-empty bucket with the highest priority.
2. For each field inside a subset of chosen fields that are able to disambiguate, compute the variance across all the restaurants in the bucket.
3. Take the field that has the highest variance since it's the one that variates the most inside of the bucket, and so it's the one that allows to best disambiguate.

There are three types of fields, and the variance has to be computed differently for each one of them:

- Numerical values: We compute the variance by just considering the set of numerical values;
- Categorical values: We associate each categorical value with a numerical value, and then we compute the variance as above. In this way the more different categorical values we have, the higher will the variance be;
- Geographical values (coordinates): We first compute the centroid from the list of coordinates, then we convert the list of coordinates into the list of distances from the centroid, and we compute the variance from this list of numerical values. In this way, the more spread the restaurants are around their centroid, the higher will the variance be.

Once the best field is selected, the system will try to disambiguate depending on that field:

- **LatLng**: The default disambiguation field. It simply takes the location of the restaurant with the highest score and asks the user if it is that one showing some other information like its address. This is the easiest disambiguation approach and it is often used if the others don't allow the system to make the disambiguation.

- **City**: We have restaurants to disambiguate that are in different cities. The system takes the city of the restaurant with the highest score and asks the user to confirm that the desired restaurant is in that city. If the user replies with 'yes', the restaurants in all the other cities are discarded (so we remain with restaurants in one city only). If the user replies with 'no', the restaurants in the same city as the best restaurant are discarded (this is one of the cases in which the restaurant with the highest score isn't the desired one). For example, think of the case in which we have 3 restaurants, one in Rome and the other two in Tivoli. The restaurant with the highest score is in Rome. The system asks the user if the restaurant is in Rome and if the user replies with 'yes', the system immediately finds out that the desired city is the one in Rome. This approach is valid only if we have restaurants in different cities. If all restaurants to disambiguate are in the same city, the situation is more difficult to solve. For this reason we decided to use the neighborhoods. The approach we used is the same as before and it can be useful if the user knows the zone in which the restaurant is. For example, think of the case in which we have 3 restaurants, one in the Prati neighborhood in Rome and the other two in Garbatella. The restaurant with the highest score is in Garbatella. The system asks the user if the restaurant is in the Garbatella neighborhood and if the user replies with 'yes', the system immediately discards the restaurant in Prati and keeps on disambiguating the remaining two restaurants.

- **Cuisines**: We have restaurants to disambiguate with different cuisines. The system takes the cuisines of the best restaurant and tries to find out what is the most discriminative one. Once that cuisine type is found the system asks if the restaurant that is looking for has that type of cuisine. If the user replies with 'yes', the restaurants that don't have that type of cuisine are discarded. If the user replies with 'no', the restaurants that have that type of cuisine are discarded (so also the best restaurant with the highest score). For example, let's think of the case in which the most discriminative cuisine type taken from the restaurant with the highest score is 'Thai'. All the restaurants to disambiguate are Italian restaurants except for the best one. If the user states that the desired restaurant has Thai cuisine, the system immediately understands what is the right restaurant and discards the other ones. Notice that we may have the case in which the best restaurant doesn't have a discriminative cuisine (e.g. the most discriminate cuisine is 'Italian' but also the other restaurants have that type of cuisine). In this case we try to understand if there is another category in the other restaurants that still allows the system to remove as many restaurants as possible. So it takes the most discriminative cuisine in the other restaurants and performs the same operation as before. We may also have the case in which we don't have a discriminative cuisine at all (in all restaurants): in this case we simply skip the cuisine disambiguation and try the other ones.

- **AvgRating**: We have restaurants to disambiguate with the average of their reviews. The system sorts the list of restaurants by this value (the higher it is, the more this indicates to us that the restaurant has an extremely positive average rating). Starting with the restaurant with the highest value, the system asks if it is the restaurant the user is looking for. If the answer is 'yes' then the restaurant has been found. If the answer is 'no' the current restaurant is discarded and the next one on the list is moved on. We chose to use this field only if the score between the restaurants is very similar to each other. This because in case the system chooses to disambiguate using this field, we would be in the special case when there is an extreme uncertainty such that with the other fields it was not possible to disambiguate (e.g. where several restaurants with the same name are all very close to each others in terms of location). It can also occur the case in which one or more restaurants do not have an average rating (thus equal to 0): in this case they will be at the bottom of the list giving more importance to those with higher values.

So, the disambiguation phase is an iterative approach in which the system can understand which restaurant is the desired restaurant in case of uncertainty. At each iteration the system asks the user some information to cut the set of restaurants to disambiguate and eventually selecting one. Finally, the user is asked to confirm the details of the reservation and if they reply with 'yes' the reservation is successful.

Before going on, we should describe briefly how the system retrieves the location. A location (city, coordinates) is mandatory in order to shrink the set of results and better understand what is the restaurant the user wants to reverse to. We have 3 cases:

- **We have the coordinates**: If the user grants the system for their coordinates then restaurants will be searched using those coordinates. In particular the system will find all the nearby restaurants matching the query within `MAX_DISTANCE` km from the user. `MAX_DISTANCE` is a constant that we set to 50 km. This limit is necessary in order to avoid disambiguating for restaurants that are too far away from the user. Usually, when the user wants to reserve a restaurant without specifying another location it is because they want to reserve a restaurant nearby.
- **We have the coordinates but the user specifies another location**: If the user specifies another location then the system will find all the nearby restaurants matching the query within `MAX_DISTANCE` km from that location. For example, if a user asks to reserve a table at 'Pizzeria da Mario' in Milan (but the user is in Rome) then the system will find all the restaurants matching that query in Milan (and not in Rome).
- **We don't have the coordinates**: If the system doesn't have the user's coordinates, it will force the user to specify a location. If they do it then the process is exactly like in the previous case. Otherwise the reservation can't go on.

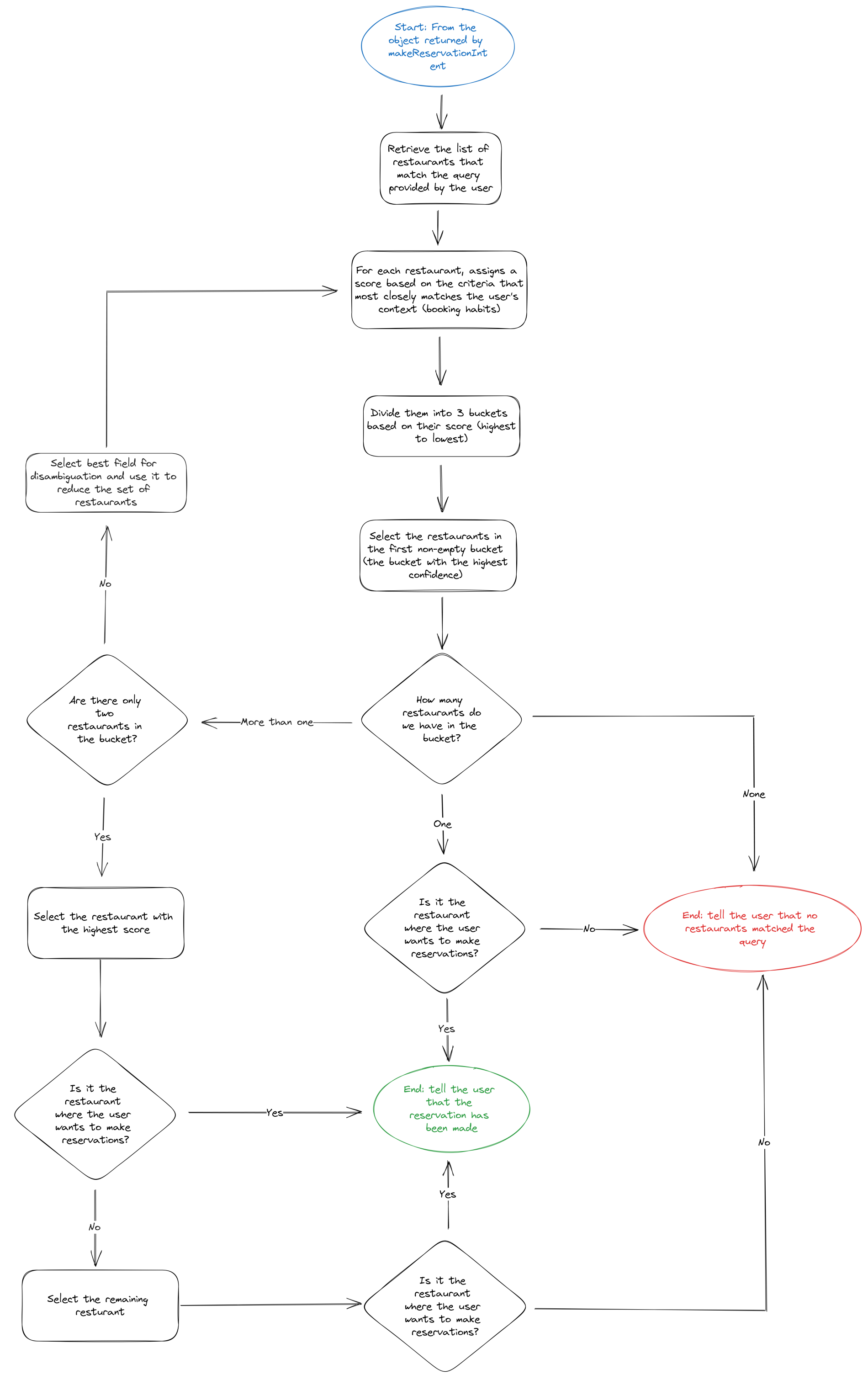
Start: From the object returned by makeReservationIntent
Retrieve the list of restaurants that match the query provided by the user
For each restaurant, assigns a score based on the criteria that most closely matches the user's context (booking habits)
Divide them into 3 buckets based on their score (highest to lowest)
Select the restaurants in the first non-empty bucket (the bucket with the highest confidence)
How many restaurants do we have in the bucket?
More than one
Are there only two restaurants in the bucket?
No
Select best field for disambiguation and use it to reduce the set of restaurants
None
One
Yes
Select the restaurant with the highest score
Is it the restaurant where the user wants to make reservations?
No
End: tell the user that no restaurants matched the query
Yes
Is it the restaurant where the user wants to make reservations?
Yes
End: tell the user that the reservation has been made
No
Yes
No
Select the remaining resturant
Is it the restaurant where the user wants to make reservations?

Figure 4: Reservation Context Response Handler

5.2 Backend structure

We structured the backend in different microservices where each of them has its own purpose, and it's as independent as possible with respect to the other microservices, in order to reduce the overall coupling of the system. Each microservice has its own independent database.

To invoke the various features offered by the system, we will use APIs for single, well-defined tasks. The APIs have been built using REST technology.

In the local development environment, we defined a `docker-compose.yml` file in order to start all the containers that are needed to get the system up and running.

There are four main microservices:

Reservations Its purpose is the management of reservations. It exposes APIs that allow to add a new reservation for a certain user at a certain restaurant, for a given day and time. The data is stored inside of a MySQL database.

Restaurants Its purpose is to manage everything that concerns a restaurant. It handles the API calls for searching a restaurant with a given name, inside of a given city, nearby a certain set of coordinates. Regarding the database, we used PostgreSQL. Differently from the other microservices that use MySQL, we opted for PostgreSQL mainly because the geographic queries support through the *Postgis* plugin, which allows querying restaurants that are in the radius of a certain set of coordinates in an efficient way.

Embeddings The *Embeddings* service serves a fundamental role in the restaurant search operation. In particular, it exposes an API that, given a query, returns the top k restaurants whose name is the most similar from the query. We leveraged the performances of the *Faiss* library in order to achieve extreme efficiency at performing vector L_2 similarity calculations. We built the indexed collection of restaurant name embeddings (i.e. the *Faiss* index) by pre-computing the embeddings for all the restaurants in the database using the *Universal Sentence Encoder* model (downloaded from *tensorflow hub*).

At inference time, the query embedding is computed on the fly with the same model, and the result is cached using *Redis*, an in-memory key-value storage that allows the computation for future requests with the same query to be even faster.

The final distance value is then averaged with the *Levenshtein* distance between the query and the restaurant name, due to the empirically better results we obtain with respect to using only the embedding L_2 distance.

Nginx Gateway

In order to connect all the microservices under the same URL, we used an Nginx gateway. This allows to define a single entry point, from which to access all the different microservices, by just appending the microservice's name.

Let's say the public URL is `https://voicefork-api.com/`, we can reach the `restaurant` microservice by making requests to the `https://voicefork-api.com/restaurants/` URL, which will redirect the requests to the `restaurant` microservice endpoints.

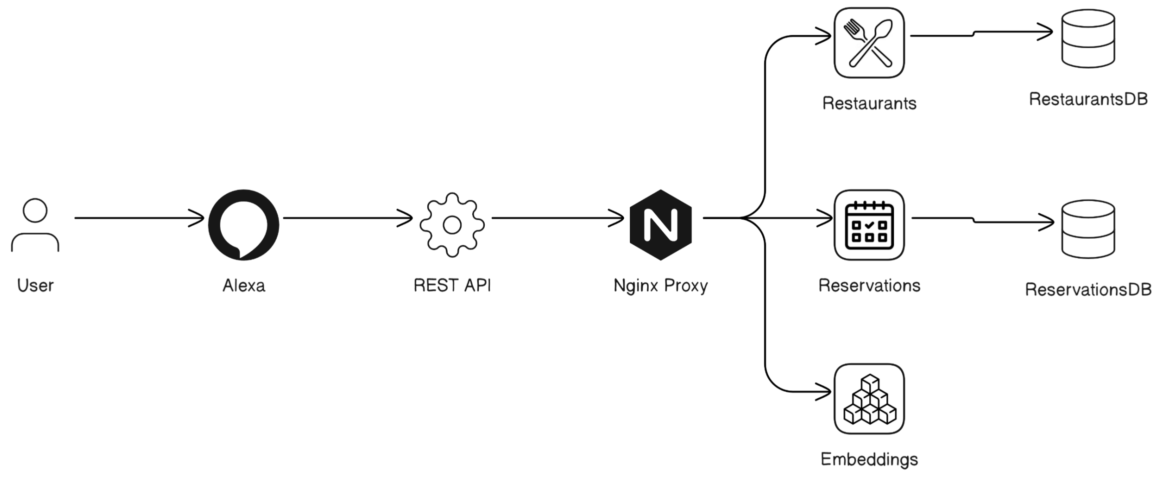

Figure 5: Architecture diagram

6 Real Scenario Tests

In order to make the application operate with a dataset close to reality, we decided to use the "**Tripadvisor European Restaurants**" [5], limiting only to the Italian restaurants. The original dataset has 1M+ instances. After the filtering, we are left with a dataset of over **200K instances**.

To test our skill, we asked people to make multiple reservations in various ways (e.g. try saying all parameters of the reservation in one sentence, or say only the name of the restaurant, etc.) or "free" reservation phrases to see how the system can process different requests. Tests were carried out using the **Think Aloud** method so having users comment on everything they thought and did while interacting with the skill without help from the administrator (unless there were obvious difficulties in continuing the test). At the end, if necessary, supplementary questions were asked based on the progress of the test. The phrase being used to anticipate the tests was "Imagine you want to use Alexa's voice assistant to make a reservation at your favorite restaurant". We made several iterations and which we did tests, analyzed possible mistakes, fixed them and start the round again until we reached a stable version of the system.

Here we briefly summarize the results of the tests:

- **Version 1**: The application was already built and allowed users to make a reservation. This version allowed users to disambiguate restaurants based on its location and city and also through the user's context (based on their previous reservations). So far, we tested the application on well-defined examples without exploring other restaurants. For these reasons we made some tests with several users and found out a good amount of bugs that made it really difficult to get to the final reservation. For this reason we stopped the tests in advance in order to solve the issues we found out.
- **Version 2**: We solved most of the errors we found in the previous version and did some tests again to see how the system performed and finally, most of the users managed to make the reservation successfully. Unfortunately, some bugs still remained but they were linked to how Alexa fills the missing information (the reservation part worked smoothly). Some users claimed that the reservation process took too long because of the disambiguation part. Others said that too much information (name of the restaurant, address, city, neighborhood, data, time and number of people) are given when confirming the reservation making the process difficult to understand.
- **Version 3**: We tried to limit the Alexa errors when filling the missing information by adding more user utterances, until we reached a satisfying result. We added cuisine and average rating score disambiguation to try to make the disambiguation process even faster (in some cases they help remove many restaurants in one step). We added the Italian language (along with the English one) in order to reach a wider audience. We slightly reduced the number of information given by the Alexa responses. We noticed that we couldn't remove more due to the fact that the disambiguation process still need to give some information to the user in order to make him understand which restaurant the system is trying to analyze, so responses are still quite long but users managed to reserve smoothly and didn't seem to complain contrarily to the previous version. We were satisfied with the tests so we can assume the third to be the last version.

In the attachment, it is possible to find the results of the tests in detail and a demo showing how the application (Version 3) works.

7 Current Limitations and Future Developments

The system's current state allows a simple interaction between the user and the virtual assistant, through which the user can reserve a restaurant and the assistant can make some questions to disambiguate the restaurant. Said that, the system has some limitations, which are advanced features that go beyond the purpose of this project, but that can be implemented in the future to improve the user experience and the system's reliability.

Let's analyze which are those limitations:

- The assistant can only understand pieces of information if the sentence is phrased according to a well-defined pattern, which can be slightly modified but not completely rephrased by the user, otherwise the VUI doesn't know where the information is inside of the sentence.
- The user cannot specify a more precise location of the place they want to reserve, neither as the neighborhood name or the street name, but only the city name (or implicitly by the user coordinates). This would be useful if the user knows in advance which is the geographical zone where the restaurant is, in order to limit the amount of disambiguation that the system has to do.
- If the restaurant name is very different from what the user says (maybe because the user usually calls that place in another way), then the desired restaurant might not appear in the search results, and so the disambiguation will never be successful. This is due to the limitation of the restaurant search algorithm.

8 Conclusions

In this project we tried to build a **Voice user interface** to allow users **making reservations to restaurants**. After having tried several approaches, we opted to implement this task using **Amazon Alexa**. What came out is that reserving a restaurant using the voice is not an easy task because of the high amount of information needed to accomplish the operation. Moreover, the system might not be able to understand what restaurant the user wants to reserve to. In order to ease this task, we implemented a **context aware system** that obtains some information on the past reservations, the date and the place of the reservation to understand if the user is trying to reserve a restaurant that was already reserved in the past so that the application is able to adapt to their habits. This can make the disambiguation process a lot faster. If no context is available, we implement an iterative method to get to the desired restaurant by making questions to the user. Tests showed that in most cases the user managed to successfully book the desired restaurants with few interactions with the assistant. In other cases some other questions were needed to reach the final results.

Some future works might be extending the system with other features such as deleting an already made reservation or retrieving the list of reservations. We may also deploy the backend on Amazon AWS to try to publish our system to Amazon Alexa (tests were done locally using a simulator provided by Amazon).

References

[1] Disambiguation of Entity References Using Related Entities

[2] Toward Voice Query Clarification

[3] Resolving Incomplete Requests Through Disambiguation

[4] A Pattern Language for Error Management in Voice User Interfaces

[5] Tripadvisor European Restaurants Dataset