Foodmate Chatbot

Corso di Natural Language Processing
Laurea Magistrale in Informatica
Alma Mater Studiorum - Università di Bologna

Gabriele Magazzù

Anno accademico 2023/2024

Indice

1 Introduzione

Foodmate è un chatbot realizzato durante l'anno accademico 2023/2024 come progetto d'esame per l'insegnamento **Natural Language Processing** per il corso di Laurea Magistrale in Informatica, presso l'Università di Bologna.
Lo scopo del progetto verte sulla creazione di un semplice chatbot tramite l'utilizzo di **Google Dialogflow CX**.
Ho, inoltre, deciso di rendere Foodmate fruibile su **Telegram**, tramite la creazione di un bot dedicato, usabile attualmente solo in lingua inglese.

Foodmate, come suggerisce il nome, è stato progettato per l'uso di servizi inerenti al **mondo culinario**.

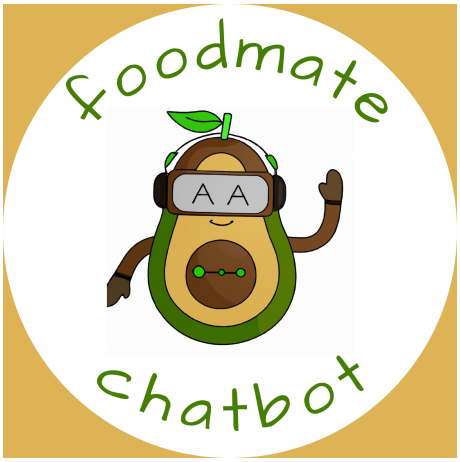

Figura 1: Logo Foodmate-chatbot. Creato su Canva con l'ausilio di DALL·E

Questa relazione ha lo scopo di approfondire tutte le caratteristiche del chatbot. Dapprima analizzerò l'**argomento** scelto come obiettivo, la sua **importanza** e la **soluzione** proposta, quindi la descrizione delle **specifiche** e dei **servizi** che il bot garantisce; continuerò poi approfondendo l'**architettura del progetto**, discutendo delle **tecnologie** utilizzate; procederò dunque con l'analisi delle **scelte implementative**, mostrando estratti di codice e immagini per ottenere una migliore comprensione globale delle implementazioni; riporterò infine le mie personali **considerazioni**, focalizzando la mia attenzione sulle difficoltà incontrate durante la realizzazione e implementazione del progetto e su eventuali **sviluppi futuri** del progetto stesso.

2 Analisi del Problema

Nella mia esperienza da studente universitario fuorisede ho spesso avuto difficoltà a gestire l'**ambito culinario**, dall'**acquisto** dei prodotti necessari fino alla loro **preparazione**. Inoltre, non è stato facile adattarsi a uno stile di vita sano ed equilibrato, che includesse un'alimentazione nutriente e variegata, ma anche gustosa e saporita.
Durante il corso del tempo, ho sperimentato e utilizzato diversi strumenti e metodologie per raggiungere un equilibrio che ottimizzasse anche le **tempistiche** e la **praticità**. Queste caratteristiche sono infatti molto importanti per chi, come uno studente, ha poco tempo da dedicare a tali attività. Purtroppo, non ho mai trovato una soluzione che garantisse completezza ed esaustività: c'erano sempre una o più mancanze.

2.1 Soluzione proposta

Ho pensato che l'implementazione di un **assistente virtuale**, come Foodmate, per la gestione e l'ottimizzazione di alcune mansioni, potesse risultare estremamente utile.
Sono consapevole che non è la prima e non sarà l'ultima soluzione proposta in questo ambito, ma d'altra parte mi sono impegnato per garantire praticità e comodità tramite Foodmate.
Propongo quindi un chatbot fruibile, attualmente solo in lingua inglese, su Telegram: la scelta della lingua è stata una decisione presa per poter garantire l'uso del bot alla maggiore quantità di persone possibile. Tra le idee per gli spunti e sviluppi futuri, che analizzerò nel dettaglio a termine della relazione, è presente la possibilità di interagire con il chatbot in **altre lingue** (e tra tutte, l'italiano si trova in prima posizione).
La fruibilità di Foodmate su più ampia scala è, inoltre, ampliata dalla scelta di renderlo disponibile su **Telegram**, ovvero un'applicazione di messaggistica mobile e desktop basata su cloud, che presenta un'attenzione particolare alla sicurezza e alla velocità.

Foodmate è, quindi, una proposta semplice, ma pratica; affidabile, ma eventualmente adattabile. Foodmate è un progetto per una migliore gestione delle attività relative all'ambito culinario.

2.2 Servizi garantiti

Foodmate offre i seguenti servizi:

1. **Grocery list service**: servizio di gestione della lista della spesa.
2. **Recipe search service**: servizio di ricerca di ricette basato su una parola chiave (ingrediente) data in input.
3. **Nutritional analysis service**: servizio di analisi nutrizionale basato su una parola chiave (ingrediente) data in input.

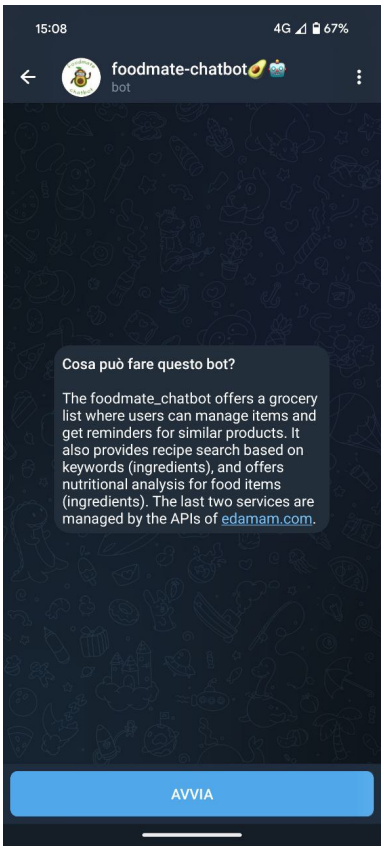

Figura 2: Schermata iniziale del chatbot

Scrivere una lista della spesa è spesso un compito **noioso**, soggetto a dimenticanze ed errori. Con Foodmate, è possibile **aggiungere** e **rimuovere** elementi in modo semplice in una lista della spesa salvata su un server cloud. Inoltre, è possibile **visualizzare** la lista della spesa, che viene automaticamente suddivisa in sezioni o reparti del supermercato. In questo modo, ogni utente che utilizza il servizio non dovrà scorrere tutta la lista per sapere se vi è un altro prodotto o elemento appartenente alla stessa sezione.

Per persone poco esperte, come potrebbe essere uno studente fuorisede alla prima esperienza lontano da casa, ma anche per persone più abili in cucina, avere un servizio come quello che Foodmate offre, basato sulla **ricerca di ricette** tramite una parola chiave (un ingrediente), può risultare estremamente utile per ottenere nuove idee o spunti con cui iniziare a cucinare.

Riuscire a equilibrare la propria alimentazione per condurre uno stile di vita sano non è semplice, e conoscere i valori nutrizionali di un alimento è fondamentale in questo ambito. Tramite il servizio di **analisi nutrizionale** di Foodmate,

è possibile ricercare i valori di un determinato alimento inserendo una parola chiave (ingrediente). In questo modo, è possibile decidere e selezionare quali prodotti o alimenti scartare in base alle proprie esigenze.

La parte di **categorizzazione** dei prodotti al momento della visualizzazione della lista della spesa (opzione garantita dal primo servizio del chatbot) è gestita tramite una chiamata API a un servizio esterno (**Gemini**).

Gli ultimi due servizi, che analizzerò più avanti nel dettaglio, sono forniti e implementati tramite delle chiamate API a un servizio esterno (**Edamam**), il quale mette a disposizione un piano gratuito (seppur limitato) per l'uso dei suoi dati e delle sue funzioni. Un utilizzo combinato di questi due servizi potrebbe risultare utile per condurre un'alimentazione sana e nutriente **senza rinunciare** a gusto e varietà delle ricette.

3 Architettura del progetto

Ho sviluppato Foodmate utilizzando diverse tecnologie: in primo luogo, ho utilizzato **Google Dialogflow CX** per la creazione della logica e la gestione dei flussi di conversazione del chatbot; ho poi usato il linguaggio di programmazione **Python** per scrivere uno script principale che includesse alcune richieste HTTP, le quali sono state distribuite sul servizio **Google Firebase** (tale piattaforma, e in particolare il Realtime Database, come sarà descritto successivamente nella relazione, sono state utilizzate anche per la gestione della lista della spesa garantita dal bot); ho inoltre scritto degli script ausiliari in Python che permettessero di gestire le chiamate ad API esterne mantenendo una buona **modularità** del codice; infine, ho utilizzato il servizio **BotFather**, messo a disposizione da Telegram, per la creazione del bot sulla piattaforma di messaggistica e per la connessione alla logica gestita da Google Dialogflow CX.

Di seguito una descrizione più dettagliata delle tecnologie e dei servizi esterni utilizzati.

3.1 Google Dialogflow CX

Google Dialogflow CX è una piattaforma che offre un approccio basato sulle macchine a stati per la progettazione degli **agenti**. Questo consente un controllo chiaro ed esplicito delle **conversazioni**, migliorando l'esperienza dell'utente finale e ottimizzando il flusso di lavoro dello sviluppo.
Nel progetto ho utilizzato Dialogflow CX per la creazione e la gestione della **logica generale** e di tutti i flussi di conversazione tra l'utente e il chatbot. Di seguito una breve descrizione delle operazioni che ho svolto tramite Dialogflow CX:

- Creazione di un **agente virtuale** che gestisce le conversazioni seguendo degli scenari previsti.

- Creazione di più **flussi di conversazione**, utilizzati per definire gli argomenti e i percorsi conversazionali associati a una discussione tra l'utente e il chatbot. Ho creato tre flussi aggiuntivi al flusso di avvio predefinito.
- Creazione di una o più **pagine** per ogni flusso, che rappresentano i vari stati di un flusso di conversazione. Ho creato una pagina per ogni azione che il chatbot deve svolgere.
- Creazione di un **tipo di entità** per il controllo dei dati provenienti dall'input dell'utente. Ho utilizzato un'entità non di sistema, creata da me e definita *item*, per il riconoscimento di un prodotto o elemento da gestire nei vari flussi di conversazione del progetto. Quando l'utente chiede di aggiungere un nuovo elemento alla lista della spesa, Dialogflow riconosce quell'elemento come un parametro che ha un nome e il tipo di entità *item*.
- Creazione di **intent**, utili per gestire l'intenzione dell'utente attraverso delle frasi di addestramento.
- Creazione e utilizzo di **Webhook** e **Fulfillment**. I primi sono servizi che ospitano la logica o richiamano altri servizi (come, per esempio, le richieste HTTP); il fulfillment rappresenta invece la generazione di una risposta da parte di Dialogflow.
- Definizione di **route**, utili per gestire le transizioni da uno stato all'altro, sia a livello di pagina sia a livello di flusso.

3.2 Script in Python

Ho usato il linguaggio di programmazione Python per la stesura di alcuni script:

- **main script**, volto alla creazione e alla gestione delle richieste HTTP (funzioni cloud);
- **script ausiliari**, volti alla gestione delle API esterne usate nel progetto. Questi script sono stati realizzati per mantenere una buona modularità nel codice.

3.3 Firebase

Firebase è una piattaforma per la creazione di applicazioni per dispositivi mobili e web sviluppata da **Google**.
Nel progetto ho utilizzato Firebase per:

- effettuare il **deployment** (distribuzione) delle **funzioni cloud**, precedentemente create tramite script Python. Ho effettuato il deployment tramite Firebase CLI, uno strumento utilizzabile tramite linea di comando nel terminale; una volta effettuato il login e collegata la directory al progetto creato su Google Cloud ho eseguito il comando *firebase deploy –only functions* per la distribuzione delle funzioni.

- usufruire del **Realtime Database** per la gestione del primo servizio garantito dal chatbot (servizio di gestione della lista della spesa). All'interno del Realtime Database esiste, o viene creato qualora venisse eliminato, un **branch** denominato "*grocery_list*" all'interno del quale verranno creati i **nodi** contenenti gli elementi che vengono inseriti nella lista della spesa.

3.4 Telegram - BotFather

Telegram è un'applicazione di **messaggistica** mobile e desktop. Mette a disposizione un servizio di creazione di chatbot tramite un bot stesso (BotFather), la cui descrizione è sufficiente per capirne l'utilizzo: "*BotFather is the one bot to rule them all. Use it to create new bot accounts and manage your existing bots.*".

4 Scelte implementative

In questa sezione della relazione entro più nel dettaglio delle scelte implementative prese durante lo svolgimento del progetto.

4.1 Dialogflow CX

Ho creato tre flussi di conversazione, aggiuntivi al flusso di avvio predefinito. Sono, dunque, presenti i seguenti flussi:

- Default Start Flow
- Grocery List Flow
- Nutrition Analysis Flow
- Recipe Search Flow

4.1.1 Default Start Flow

Rappresenta il **flusso di avvio predefinito**. Come mostrato nella **Fig.3**, presenta una pagina di avvio attraverso la quale può ricondurre lo stato agli altri flussi di conversazione, in base all'evento catturato in input dall'utente.

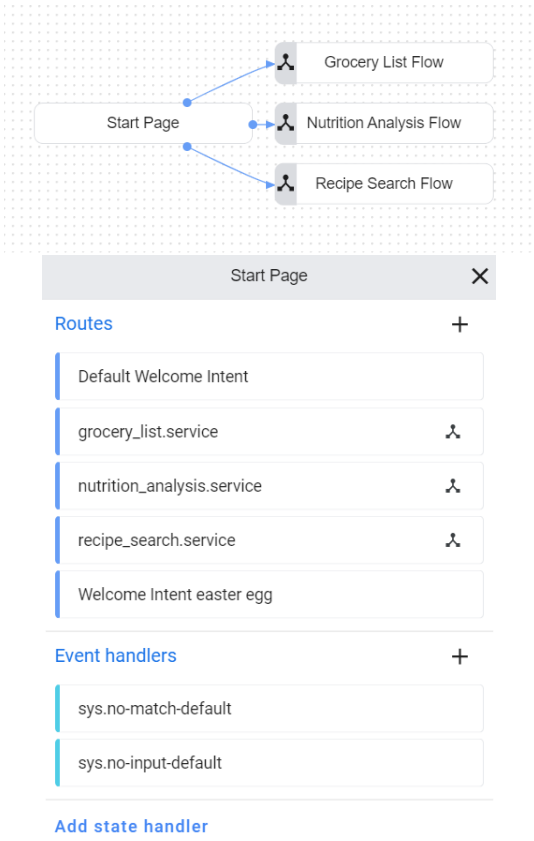

Figura 3: Default Start Flow e relativa Start Page

4.1.2 Grocery List Flow

Rappresenta il **flusso di conversazione più articolato** del progetto, in quanto all'interno di esso sono presenti tutte le possibili azioni e transizioni da una stato all'altro riguardanti il primo servizio garantito dal chatbot, ovvero la gestione di una lista della spesa.

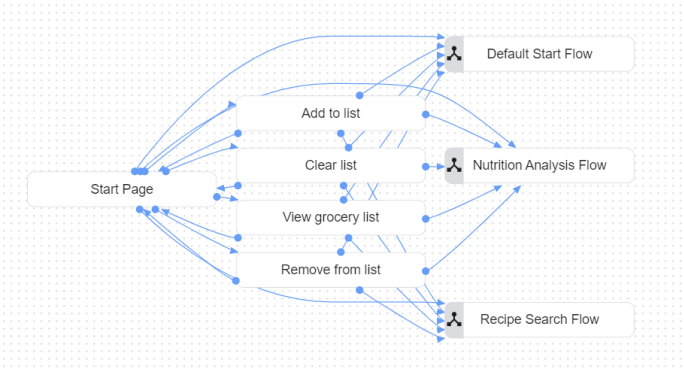

Figura 4: Grocery List Flow

Ogni pagina presente all'interno del flusso presenta determinate **route** per la gestione delle transizioni di stato e la gestione delle azioni da svolgere. Nella **Fig.5** mostro la pagina "Add to list" e il suo Fulfillment. Con la transizione a questa pagina viene chiamata la richiesta HTTP relativa al webhook "*add_items*". Nelle specifiche del webhook, ho inserito l'URL relativo alla funzione cloud che gestisce l'inserimento nel Realtime Database di Firebase degli elementi passati come parametri dall'utente.

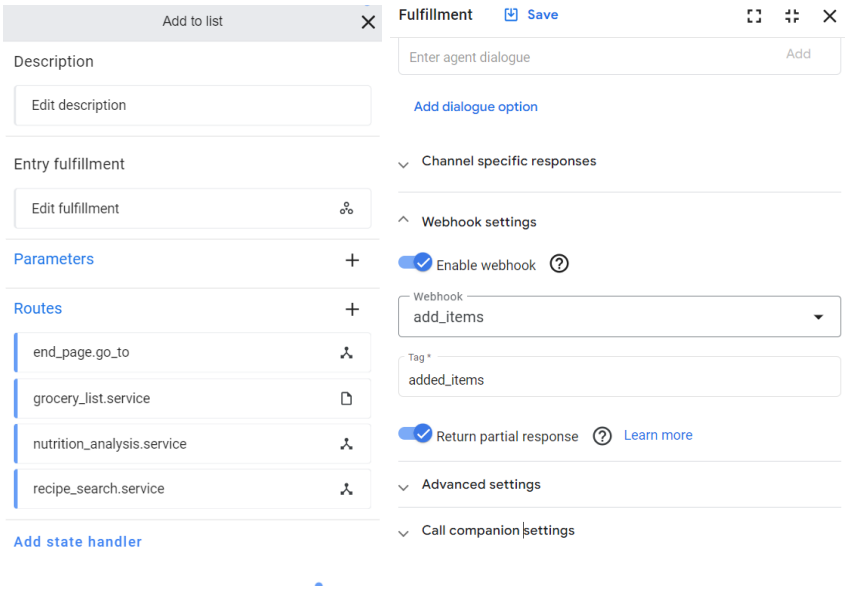

Figura 5: Add to list page e Fulfillment associato

4.1.3 Nutrition Analysis Flow

Rappresenta il **flusso di conversazione per la gestione del servizio di analisi nutrizionale** garantito e offerto dal chatbot Foodmate. Presenta la pagina "Get nutritional analysis" che, in modo analogo a quanto spiegato prima per la pagina "Add to list", gestisce la richiesta dell'utente per quanto riguarda l'analisi nutrizionale da fare tramite l'API fornita dal sito **Edamam.com**.
Il webhook associato al Fulfillment della pagina, anche in questo caso, presenta l'URL della richiesta HTTP specifica per la gestione dell'azione. Come spiegherò nella sezione riguardante il codice in Python, questa funzione cloud (come anche altre) fa una richiesta a sua volta alle API fornite da Edamam.com in modo da usufruire dei servizi offerti dalla piattaforma.

Figura 6: Nutrition Analysis Flow

4.1.4 Recipe Search Flow

Rappresenta il **flusso di conversazione per la gestione del servizio di ricerca di ricette** garantito e offerto dal chatbot Foodmate. Presenta la pagina "Recipe Search" che, in modo analogo a quanto spiegato prima per la pagina "Get nutritional analysis", gestisce la richiesta dell'utente per la ricerca di una ricetta da fare tramite l'API fornita dal sito **Edamam.com**.
Il webhook associato al Fulfillment della pagina, anche in questo caso, presenta l'URL della richiesta HTTP specifica per la gestione dell'azione. La funzione cloud, come quanto descritto per il servizio precedente, fa una richiesta a sua volta alle API fornite da Edamam.com in modo da usufruire dei servizi offerti dalla piattaforma.

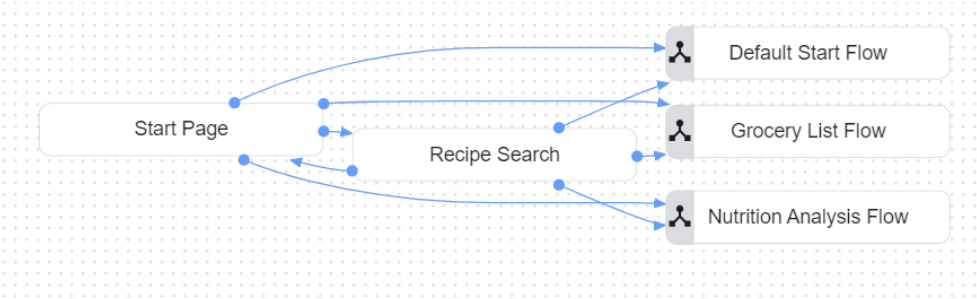

Figura 7: Recipe Search Flow

4.2 Firebase

Ho utilizzato Firebase per usufruire del **Realtime Database** e del servizio di **deployment** delle **funzioni cloud**.

4.2.1 Realtime Database

Ho usato il Realtime Database fornito da Firebase per la gestione del primo servizio. Infatti, quando un utente effettua un'azione inerente alla **gestione della lista della spesa**, nel database vengono creati o eliminati i nodi relativi agli elementi **inseriti** o **rimossi** dalla lista. Nella **Fig.8** mostro alcune schermate istantanee del database dopo l'inserimento e durante la rimozione di un elemento.

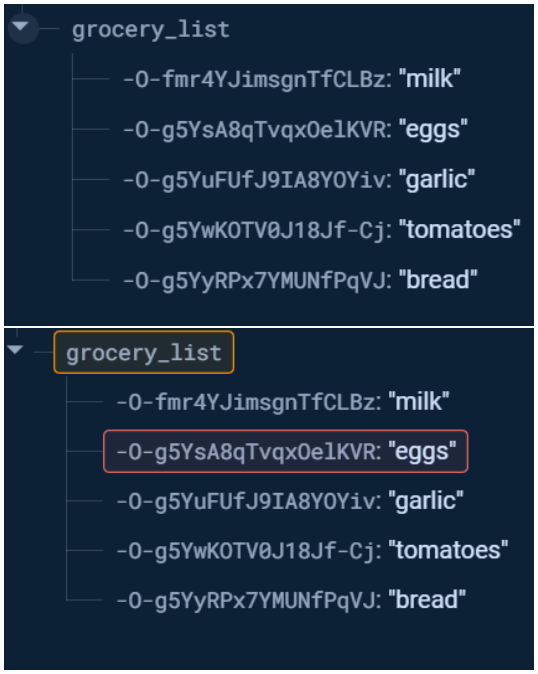

Figura 8: Istantanee del Realtime Database

4.2.2 Funzioni Cloud

Ho usato la sezione **Functions** offerta da Firebase per il deployment (distribuzione) delle richieste HTTP sotto forma di funzioni cloud. Gli **URL** relativi ad ogni funzione sono stati inseriti nei webhook su Dialoflow CX associati alle azioni corrispondenti. Nella **Fig.9** mostro la schermata della sezione Functions di Firebase, comprendente tutte le funzioni cloud distribuite.

Figura 9: Funzioni cloud distribuite

4.3 Script in Python

Come già precedentemente spiegato, ho utilizzato il linguaggio Python per la stesura di alcuni script necessari al funzionamento generale del chatbot, in particolare per collegare la logica presente su Dialogflow con l'applicazione di messaggistica Telegram.
Alcuni sono degli script **ausiliari**, creati per migliorare la **modularità** del codice, utilizzati nel **main** script.

4.3.1 Gemini API script

Script ausiliario che utilizza le API di **Google Gemini** per gestire la **categorizzazione automatica** durante la visualizzazione della lista della spesa.
Nello script, dopo aver effettuato l'autenticazione, utilizzo la libreria **google.generativeai** per configurare un modello. A questo modello invio un prompt contenente le istruzioni sulla categorizzazione in reparti del supermercato e la lista della spesa come parametro di input da categorizzare.
Di seguito un estratto del codice, più in particolare la funzione *categorize_grocery_list()* e un suo possibile esempio di utilizzo.

```
def categorize_grocery_list(grocery_list):
    try:
        api_key = load_api_key("gemini-key.json")
        genai.configure(api_key=api_key)
        model = genai.GenerativeModel()
```

```
        prompt = (
            f"You have to categorize items in a grocery list "
            f"to help a customer finding the right "
            f"supermarket section for every product in the list. "
            f"I will give you in input the list "
            f"and you have to return more list divided by "
            f"category (supermarket section). The grocery "
            f"list includes: {grocery_list}. "
            f"Please categorize the items by supermarket "
            f"section. Be careful to be "
            f"precise in your categorization")

        response = model.generate_content(prompt)
        return response.text
    except Exception as e: # Catch any unexpected errors
        print(f"Error categorizing grocery list: {e}")
        return [] # Return an empty list on error
```

Di seguito, un esempio di utilizzo della funzione *categorize_grocery_list()*.

```
# Example usage
grocery_list = ["Milk", "Bread", "Apples", "Eggs", "beer", "almonds", "juice"]
categorized_list = categorize_grocery_list(grocery_list)
print(categorized_list)
```

Eseguendo il codice, l'output ottenuto è il seguente.

```
**Dairy:**
- Milk
- Eggs

**Bakery:**
- Bread

**Produce:**
- Apples

**Beverages:**
- Juice
- Beer

**Nuts:**
- Almonds
```

Il prompt viene, dunque, passato come parametro alla funzione di generazione del modello, dalla quale si ottiene il testo della risposta che verrà gestita da Dialogflow CX per l'invio all'utente.

4.3.2 Edamam Nutrition Analysis API script

Script ausiliario che utilizza le API di **Edamam.com** per gestire la **raccolta automatica di dati nutrizionali** degli ingredienti. Nello script, dopo aver effettuato l'autenticazione, utilizzo la libreria **requests** per inviare una richiesta GET alle API di Edamam con un ingrediente come parametro di input. Di seguito un estratto del codice, più in particolare una porzione della funzione *get_nutrition_data()*.

```
def get_nutrition_data(ingredient):
    try:
        app_id, app_key = load_api_key("edamam_nutritionAPI_key.json")
        url = (f"https://api.edamam.com/api/nutrition-data?"
        f"app_id={app_id}&app_key={app_key}&nutrition-type=logging&ingr={ingredient}")
        headers = {
            'accept': 'application/json'
        }

        response = requests.get(url, headers=headers)

        if response.status_code == 200:
            data = response.json()
            filtered_data = {
                # Gestione e filtraggio dei dati ricevuti...
            }

            formatted_text = f"Food Name Matched: {filtered_data['food_name']}\n"

            if filtered_data["calories"]:
                # Aggiunta a formatted_text delle informazioni sulle calorie
            if filtered_data["FAT"]:
                # Aggiunta a formatted_text delle informazioni sui grassi
            # Altri if per gestione di formatted_text...

            nutrient_mapping = {
                "ENERC_KCAL": "Total Kcal",
                "PROCNT_KCAL": "Kcal from protein",
                "FAT_KCAL": "Kcal from fat",
                "CHOCDF_KCAL": "Kcal from carbohydrates"
            }

            formatted_text += "Total Nutrients KCal.\n"
            for nutrient, value in filtered_data["totalNutrientsKCal"].items():
                nutrient_name = nutrient_mapping.get(nutrient, nutrient)
                formatted_text += (f" {nutrient_name}: "
                f"{value['quantity']} {value['unit']}\n")
```

```
            if filtered_data["food_name"] == "Unknown":
                out = (f"Sorry, the database of Edamam.com "
                        f"did not find what you were looking for.")
                return out
            else:
                return formatted_text

        else:
            return {"success": False, "error": f"Error: {response.status_code}"}

    except Exception as e:  # Catch any unexpected errors
        print(f"Error using Edamam Nutrition API: {e}")
        return e
```

Ho omesso, nella relazione, la gestione di tutte le informazioni riguardanti la risposta di Edamam.com alla chiamata GET effettuata.
Ho preferito, d'altra parte, mostrare in modo più generale la logica della funzione, che svolge il compito di filtraggio e la formattazione dei dati ricevuti. Il testo formattato viene, quindi, inviato a Dialogflow, da cui a sua volta verrà inviato all'utente.

4.3.3 Edamam Recipe Search API script

Script ausiliario che utilizza le API di **Edamam** per gestire la **raccolta automatica di ricette** basate su un ingrediente.
Nello script, dopo aver effettuato l'autenticazione, utilizzo la libreria **requests** per inviare una richiesta GET alle API di Edamam con un ingrediente come parametro di input.
Di seguito un estratto del codice, più in particolare la funzione *get_recipe_data()* e un suo possibile esempio di utilizzo.

```
def get_recipe_data(ingredient):
    try:
        app_id, app_key = load_api_key("edamam_recipeAPI_key.json")
            url = (f"https://api.edamam.com/api/recipes/"
            f"v2?type=public&q={ingredient}&app_id={app_id}&app_key={app_key")

        headers = {
            'accept': 'application/json'
        }

        response = requests.get(url, headers=headers)

        if response.status_code == 200:
            data = response.json()

            recipes_info = []
```

```
            for i, recipe in enumerate(data.get("hits", [])[:4], 1):
                recipe_data = recipe["recipe"]

                # Controllo se il valore delle calorie è un dizionario
                if isinstance(recipe_data.get("calories"), dict):
                    calories_quantity = recipe_data["calories"].get("quantity", "N/A")
                    calories_unit = recipe_data["calories"].get("unit", "")
                else:
                    calories_quantity = recipe_data.get("calories", "N/A")
                    calories_unit = ""

                # Seleziono solo i campi desiderati con quantitativo e unità di misura
                recipe_info = {
                    "name": recipe_data.get("label", "Unknown"),
                    "image_url": recipe_data.get("image"),
                    # Gestione delle informazioni desiderate
                    # ...
                }

                recipes_info.append(recipe_info)

            formatted_text = ""
            for i, recipe_info in enumerate(recipes_info, 1):
                formatted_text += f"Recipe {i}:\n"
                formatted_text += f"Name: {recipe_info['name']}\n"
                # Formattazione del testo per l'output
                # ...
            return formatted_text
        else:
            return {"success": False, "error": f"Error: {response.status_code}"}

    except Exception as e:  # Catch any unexpected errors
        print(f"Error using Edamam Recipe API: {e}")
        return e
```

Ho omesso, nella relazione, la gestione di tutte le informazioni riguardanti la risposta di Edamam.com alla chiamata GET effettuata e della relativa formattazione del testo.

Di seguito un esempio di utilizzo della funzione *get_recipe_data()*.

```
# Example usage
print(get_recipe_data("chicken breast"))
```

Eseguendo il codice, l'output ottenuto è il seguente.

```
Recipe 1:
Name: Sous Vide Chicken Breast Recipe
Calories: 312.80057999999997
Ingredients:
- 2 bone-in, skin-on chicken breast halves
- Kosher salt and freshly ground black pepper
- 4 sprigs thyme or rosemary (optional)
Recipe URL: https://www.seriouseats.com/sous-vide-chicken-breast-recipe

Recipe 2:
Name: Roasted Chicken Breast
Calories: 150.29511
Ingredients:
- 1 bone-in, skin-on chicken breast half, about 10 ounces
- Coarse salt and freshly ground black pepper
Recipe URL: https://www.marthastewart.com/1090479/roasted-chicken-breasts
...
...
...
```

4.3.4 Main script

Rappresenta lo script principale. Svolge i seguenti compiti:

- inizializzazione dell'**applicazione Firebase**;
- caricamento delle **credenziali** necessarie per comunicare con Dialogflow CX e con il chatbot su Telegram;
- gestione dello **scambio di messaggi** tra Dialogflow e Telegram;
- definizione delle **funzioni cloud** di cui effettuare il **deployment**.

Di seguito, una spiegazione dello script e delle sue funzioni.

Inizializzazione dell'applicazione Firebase: avviene attraverso il caricamento delle informazioni contenute nel file "*chiave.json*" in combinazione con l'URL del Realtime Database.

```
# Initialize Firebase app
cred = credentials.Certificate("chiave.json")
firebase_admin.initialize_app(cred,
    {'databaseURL':
'https://nlp-chatbot-project-420413-default-rtdb.europe-west1.firebasedatabase.app/'})
```

Caricamento credenziali per comunicazione con Dialogflow e Telegram: avviene tramite l'utilizzo delle funzioni ausiliarie *load_telegram_key()* e *load_dialogflow()*. Queste caricano le credenziali a partire dalle informazioni presenti nei file "*telegram_bot_father_key.json*" e "*dialogflow_infos.json*".

```
def load_telegram_key(file_path):
    with open(file_path, "r") as f:
        credentials = json.load(f)
    api_key = credentials.get("TELEGRAM_BOT_KEY")
    if not api_key:
        raise ValueError(f"Missing 'TELEGRAM_BOT_KEY' key in {file_path}.")
    return api_key

def load_dialogflow(file_path):
    with open(file_path, "r") as f:
        credentials = json.load(f)
    project_ID = credentials.get("PROJECT_ID")
    agent_ID = credentials.get("AGENT_ID")
    if not project_ID or not agent_ID:
        raise ValueError(f"Missing 'PROJECT_ID' or 'AGENT_ID' in {file_path}.")
    return project_ID, agent_ID

# Token del bot Telegram
TELEGRAM_BOT_TOKEN = load_telegram_key("telegram_bot_father_key.json")

# Informazioni riguardanti il progetto dialogflow
PROJECT_ID, AGENT_ID = load_dialogflow("dialogflow_infos.json")
REGION = "europe-west2"
LANGUAGE_CODE = 'en'

# Caricamento delle credenziali di servizio dal file JSON
DIALOGFLOW_CREDENTIALS = service_account.Credentials.from_service_account_file(
    'chiave.json',
    scopes=['https://www.googleapis.com/auth/cloud-platform']
)
```

Gestione dello scambio di messaggi tra Dialgoflow e Telegram: ogni qual volta che si verifica una condizione all'interno di una funzione cloud che comporta la generazione di una risposta all'utente, allora viene chiamata la funzione "*create_dialogflow_response()*" con *message_text* come parametro in input.

```
# Helper function to create Dialogflow response
def create_dialogflow_response(message_text):
    response = {
        "fulfillment_response": {
            "messages": [
                {
                    "text": {
                        "text": [
                            message_text
```

```
                    ]
                }
            }
        ]
    }
}
return json.dumps(response)
```

D'altra parte, quando è necessario inviare un messaggio al chatbot su Telegram e volerne ricevere la risposta (per poterla comunicare a Dialogflow o per effettuare debugging tramite log) viene chiamata la funzione "*send_message_to_telegram()*" con *chat_id* e *text* come parametri in input.

```
def send_message_to_telegram(chat_id, text):
    url = f"https://api.telegram.org/bot{TELEGRAM_BOT_TOKEN}/sendMessage"
    payload = {
        'chat_id': chat_id,
        'text': text
    }
    headers = {
        'Content-Type': 'application/json'
    }

    response = requests.post(url, headers=headers, json=payload)
    logging.debug(f"Ricevuta risposta da Telegram: {response.status_code} {response.text}")
    return response.json()
```

Funzioni cloud: di seguito mostro una spiegazione delle funzioni cloud distribuite. Inoltre, mostro anche come ottengo la reference al Realtime Database di Firebase. La reference sarà necessaria per poter effettuare le operazioni sul database, tramite alcune funzioni cloud.

```
# Reference to grocery list in database
grocery_list_ref = db.reference("grocery_list")
```

Funzione telegram_webhook: necessaria per l'invio dei messaggi da Dialogflow CX a Telegram. Usufruisce della funzione ausiliaria "*detect_intent_texts(session_id, text, user_id, username, chat_id, update_id, message_id, date)*" e della funzione "*send_message_to_telegram(chat_id,text)*" (vista in precedenza).

```
@https_fn.on_request(cors=options.CorsOptions(cors_origins="*", cors_methods=["post"]))
def telegram_webhook(request):
    try:
        request_data = request.get_json()
        if not request_data:
            return {"success": False, "error": "Request data is missing"}, 400
```

```
        message = None
        update_id = request_data.get('update_id')

        if 'message' in request_data:
            message = request_data['message']
        elif 'edited_message' in request_data:
            # ...
        elif 'my_chat_member' in request_data:
            return {"success": True, "message": "Chat member update received."}, 200
        else:
            return {"success": False, "error": "Invalid message format"}, 400

        if not message:
            return {"success": False, "error": "No message found in request data"}, 400

        chat_id = message.get('chat', {}).get('id')
        text = message.get('text')
        user_id = message.get('from', {}).get('id')
        username = message.get('from', {}).get('username', '')
        message_id = message.get('message_id')
        date = message.get('date')

        if not chat_id or not text:
            return {"success": False, "error": "Invalid message format"}, 400

        session_id = str(chat_id)

        # Chiamata a Dialogflow CX con "it" come LANGUAGE_CODE
        dialogflow_response = detect_intent_texts(session_id, text, user_id,
        username, chat_id, update_id, message_id, date)

        # Estrazione di tutti i messaggi di testo da responseMessages
        response_messages = dialogflow_response.get('queryResult',
        {}).get('responseMessages', [])
        response_texts = []
        for message in response_messages:
            if 'text' in message and 'text' in message['text']:
            response_texts.extend(message['text']['text'])

        response_text = ' '.join(response_texts) if response_texts
        else "I didn't get that. May you try again please?"

        # Invio risposta a Telegram
        telegram_response = send_message_to_telegram(chat_id, response_text)
        return {"success": True, "response": telegram_response}
    except Exception as e:
```

```
        return {"success": False, "error": str(e)}, 500
```

Di seguito la **funzione ausiliaria** "*detect_intent_texts()*".

```
def detect_intent_texts(session_id, text, user_id,
username, chat_id, update_id, message_id, date):

    url = (f"https://{REGION}-dialogflow.googleapis.com/v3/projects/{PROJECT_ID}/"
    f"locations/{REGION}/agents/{AGENT_ID}/sessions/{session_id}:detectIntent")

    # Aggiorna il token se necessario
    auth_req = Request()
    DIALOGFLOW_CREDENTIALS.refresh(auth_req)
    token = DIALOGFLOW_CREDENTIALS.token

    headers = {
        'Authorization': f'Bearer {token}',
        'Content-Type': 'application/json'
    }

    data = {
        "query_input": {
            "language_code": LANGUAGE_CODE,
            "text": {
                "text": text,
            }
        },
        "query_params": {
            "payload": {
                "data": {
                    "update_id": update_id,
                    "message": {
                        ...
                        ...
                        ...
                    }
                },
                "source": "telegram"
            }
        }
    }

    response = requests.post(url, headers=headers, json=data)

    return response.json()
```

Funzione add_to_grocery_list: necessaria per l'aggiunta di elementi alla lista della spesa. Nella funzione controllo innanzitutto se vi sono elementi già presenti all'interno della lista; successivamente controllo che i prodotti passati come input nella richiesta non siano all'interno della lista sul Database. Quindi, aggiungo alla lista solo gli elementi nuovi.

```
@https_fn.on_request(cors=options.CorsOptions(cors_origins="*", cors_methods=["post"]))
def add_to_grocery_list(request):
    try:
        request_data = request.get_json()
        if request_data is None:
            return {"success": False, "error": "Request data is missing"}, 400

        parameters = request_data.get("intentInfo", {}).get("parameters", {})
        items_to_add = parameters.get("item", {}).get("resolvedValue", [])

        current_items = grocery_list_ref.get()
        if current_items is None:
            current_items = []
        else:
            current_items = list(current_items.values())

        items_added = []
        for item in items_to_add:
            if item not in current_items:
                grocery_list_ref.push(item)
                items_added.append(item)

        if not items_added:
            response_no_items_added = create_dialogflow_response(
            "No element was added to grocery list."
            "They were all already in.")
            return response_no_items_added
        else:
            response_items_added = create_dialogflow_response(
            (f"New items added to grocery list successfully: "
            f"{items_added} are now in the list."))
            return response_items_added

    except Exception as e:
        print("Error adding new items:", e)
        response_error = create_dialogflow_response(f"Error adding new items: {e}")
        return response_error, 500
```

Funzione remove_from_grocery_list: necessaria per la rimozione di elementi dalla lista della spesa. Nella funzione controllo innanzitutto se nella lista sono presenti dei prodotti, e in caso positivo li salvo in un array per poterli confrontare in seguito con gli elementi da rimuovere; successivamente itero sulla lista dei prodotti da rimuovere ricevuti come input nella richiesta e li elimino dal Database solo se effettivamente presenti. Infine, gestisco le risposte inviate all'utente in base a ciò che è stato effettuato e agli elementi rimossi.

```
@https_fn.on_request(cors=options.CorsOptions(cors_origins="*", cors_methods=["delete"]))
def remove_from_grocery_list(request):
    try:
        request_data = request.get_json()
        if request_data is None:
            return {"success": False, "error": "Request data is missing"}, 400
        parameters = request_data.get("intentInfo", {}).get("parameters", {})
        items_to_remove = parameters.get("item", {}).get("resolvedValue", [])

        current_items = grocery_list_ref.get()
        if current_items is None:
            response_no_items_ = create_dialogflow_response(
            "The grocery list is already empty!")
            return response_no_items_
        else:
            current_items = list(current_items.values())
            items_removed = []

            for item in items_to_remove:
                if item in current_items:
                    query_result = grocery_list_ref.order_by_value().equal_to(item).get()
                    for key, value in query_result.items():
                        if value == item:
                            grocery_list_ref.child(key).delete()
                            items_removed.append(item)

            if not items_removed:
                response_no_items_removed = create_dialogflow_response(
                "No element was removed from grocery list. They were not in.")
                return response_no_items_removed
            else:
                response_items_removed = create_dialogflow_response(
                (f"Items removed from grocery list successfully:"
                f"{items_removed} are no longer in the list."))
                return response_items_removed

    except Exception as e:
        print("Error removing items:", e)
```

```
        response_error = create_dialogflow_response(f"Error removing items: {e}")
        return response_error, 500
```

Funzione view_grocery_list: necessaria per la visualizzazione della lista della spesa. Usufruisce dello script ausiliario "*gemini_api_script*", attraverso la funzione "*categorize_grocery_list(items)*" per la categorizzazione automatica dei prodotti all'interno della lista della spesa. Nella funzione cloud gestisco il caso in cui non vi sia alcun elemento nel Database.

```
@https_fn.on_request(cors=options.CorsOptions(cors_origins="*", cors_methods=["get"]))
def view_grocery_list(request):
    try:
        grocery_list = grocery_list_ref.get()
        if grocery_list is None or not grocery_list.values():
            response_no_items_in_the_list = create_dialogflow_response(
            "The grocery list is empty.")
            return response_no_items_in_the_list

        items_in = list(grocery_list.values())
        response_categorized_items = create_dialogflow_response(
            categorize_grocery_list(items_in))
        return response_categorized_items

    except Exception as e:
        print("Error reading items from grocery list:", e)
        response_error = create_dialogflow_response(
            f"Error reading items from grocery list: {e}")
        return response_error, 500
```

Funzione clear_grocery_list: necessaria per l'eliminazione di tutti gli elementi all'interno della lista della spesa senza dover specificare i nomi dei prodotti. Nella funzione gestisco il caso in cui la lista sia già vuota, restituendo un output opportuno.

```
@https_fn.on_request(cors=options.CorsOptions(cors_origins="*", cors_methods=["delete"]))
def clear_grocery_list(request):
    try:
        current_items = grocery_list_ref.get()
        if current_items is None:
            response_no_items_ = create_dialogflow_response(
                "The grocery list is already empty!")
            return response_no_items_
        else:
            grocery_list_ref.delete()
            response_success_delete = create_dialogflow_response(
                "All items removed from grocery list successfully.")
```

```
            return response_success_delete
    except Exception as e:
        print("Error removing all items from grocery list:", e)
        response_error = create_dialogflow_response(
            f"Error removing all items from grocery list: {e}")
        return response_error, 500
```

Funzione get_nutrition_analysis_single_ingredient: necessaria per la gestione della richiesta di analisi nutrizionale. Usufruisce dello script ausiliario "*edamam_nutrition_api_script*" e, in particolare, della funzione "*get_nutrition_data(ingredient)*".

```
@https_fn.on_request(cors=options.CorsOptions(cors_origins="*", cors_methods=["get"]))
def get_nutrition_analysis_single_ingredient(request):
    try:
        request_data = request.get_json()
        if request_data is None:
            return {"success": False,
            "error": "Request data is missing"}, 400

        parameters = request_data.get("intentInfo",
            {}).get("parameters", {})
        item_to_analyze = parameters.get("item",
            {}).get("resolvedValue", [])
        nutrition_data = get_nutrition_data(item_to_analyze)

        response_nutrition_data = create_dialogflow_response(f"{nutrition_data}")

        return response_nutrition_data
    except Exception as e:
        response_error = create_dialogflow_response(
            f"Error analyzing nutrition data: {e}")
        return response_error, 500
```

Funzione get_recipes_search: necessaria per la gestione della richiesta di ricerca ricette. Usufruisce dello script ausiliario "*edamam_recipe_api_script*" e, in particolare, della funzione "*get_recipe_data(ingredient)*".

```
@https_fn.on_request(cors=options.CorsOptions(cors_origins="*", cors_methods=["get"]))
def get_recipes_search(request):
    try:
        request_data = request.get_json()
        if request_data is None:
            return {"success": False,
                "error": "Request data is missing"}, 400

        parameters = request_data.get("intentInfo",
```

```
            {}).get("parameters", {})
        item_to_search_recipe = parameters.get("item", {}).get("resolvedValue", [])

        recipe_data = get_recipe_data(item_to_search_recipe)
        response_recipe_data = create_dialogflow_response(f"{recipe_data}")

        return response_recipe_data
    except Exception as e:
        response_error = create_dialogflow_response(
            f"Error searching recipes data: {e}")
        return response_error, 500
```

4.4 Telegram - BotFather

Durante la fase di sviluppo del progetto ho testato l'assistente virtuale tramite l'applicazione di messaggistica integrata su Dialogflow CX. Successivamente, quando ho raggiunto una versione stabile del chatbot ho deciso di procedere con la distribuzione su Telegram.
Ho, dunque, utilizzato il servizio **BotFather** per la creazione di un nuovo bot. Ho dovuto, dapprima, scegliere un nome e un username; ho poi ricevuto il token per l'accesso all'uso delle API; in seguito, ho inserito una immagine profilo (logo, visibile nella **Fig.1**) e modificato la descrizione e le informazioni di contesto del bot.
Infine, dopo la distribuzione su Firebase della funzione cloud necessaria per il collegamento e lo scambio di messaggi tra Dialogflow e il chatbot su Telegram, ho dovuto collegare la funzione cloud tramite webhook anche al bot su Telegram. Il collegamento è stato effettuato da **linea di comando**, nel seguento modo:

```
curl -F "url=<my_webhook_url>" https://api.telegram.org/bot<my_telegram_token>/setWebhook
```

Da cui, ho ottenuto il seguente **output**:

```
{"ok":true,"result":true,"description":"Webhook is already set"}
```

In breve, il processo di scambio messaggi, partendo da Telegram, può essere descritto nel seguente modo:

1. **Ricezione del messaggio**: un utente invia un messaggio al bot su Telegram.
2. **Invio al webhook**: Telegram invia una richiesta HTTP POST all'URL con i dati del messaggio.
3. **Elaborazione del messaggio**: Il server all'URL specificato processa il messaggio e può rispondere di conseguenza.

Quindi, ogni volta che si invia un messaggio sull'applicazione di messaggistica all'assistente Foodmate, verrà effettuata una richiesta POST per poter inviare il messaggio a Firebase, su cui il corrispondente webhook di Dialogflow è in **ascolto**, in attesa di nuovi messaggi.

5 Conclusione

La soluzione da me proposta come progetto d'esame per l'insegnamento **Natural Language Processing** per l'anno accademico 2023/2024 consiste, dunque, in un chatbot, denominato **Foodmate**.

Foodmate rappresenta un assistente virtuale completo per la cucina, in grado di semplificare la gestione della lista della spesa, suggerire ricette basate sugli ingredienti disponibili e fornire analisi nutrizionali dettagliate. Le tecnologie utilizzate e le scelte implementative si sono dimostrate efficaci, ma ci sono margini di miglioramento.

5.1 Valutazione della soluzione proposta

Lo scopo del progetto, come precedentemente descritto, è fornire servizi culinari agli utenti che desiderano usufruirne, garantendo praticità e completezza.
L'uso di Dialogflow per la logica mi ha permesso di gestire al meglio i flussi di conversazione. La gestione della lista della spesa tramite Firebase è efficace per garantire un aggiornamento in tempo reale. L'integrazione con le API di Edamam fornisce risultati accurati e dettagliati. La distribuzione tramite Telegram permette la fruizione del chatbot su più ampia scala.

Di seguito mostro una dimostrazione d'uso dell'assistente.

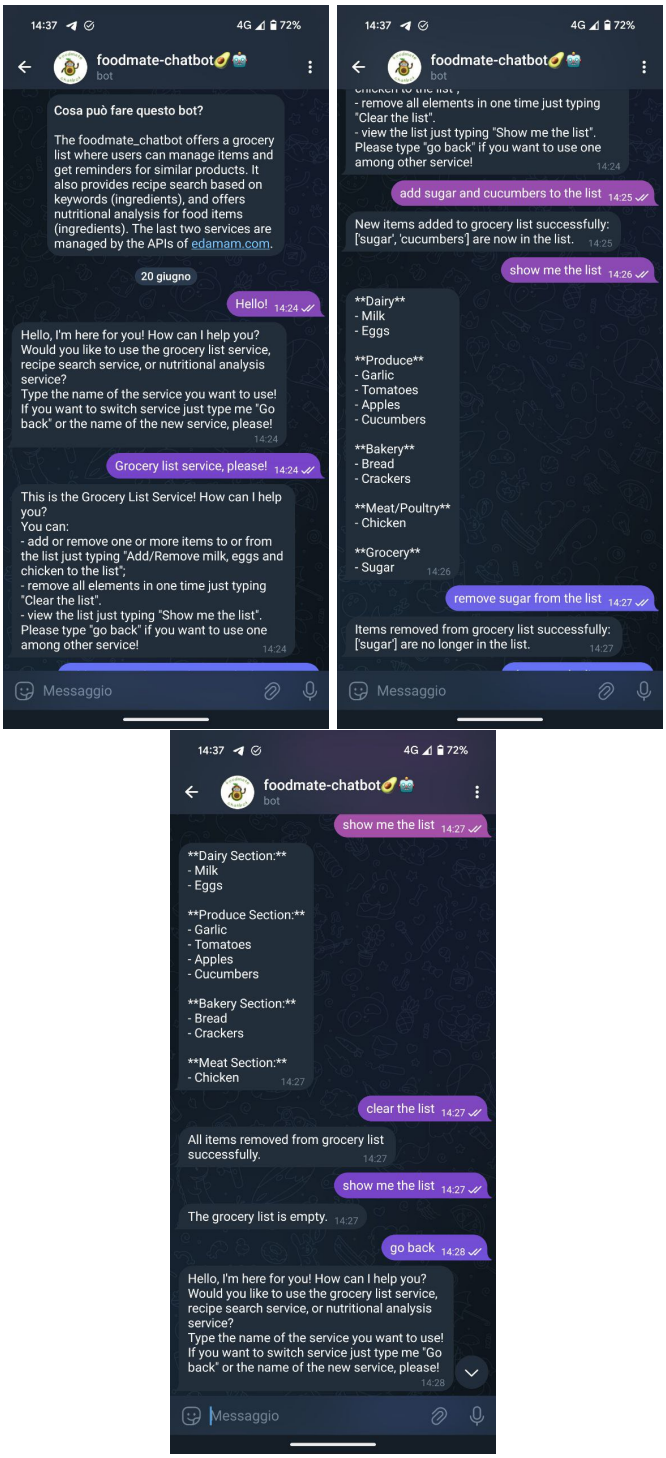

Figura 10: Saluti iniziali e servizio di gestione della lista della spesa

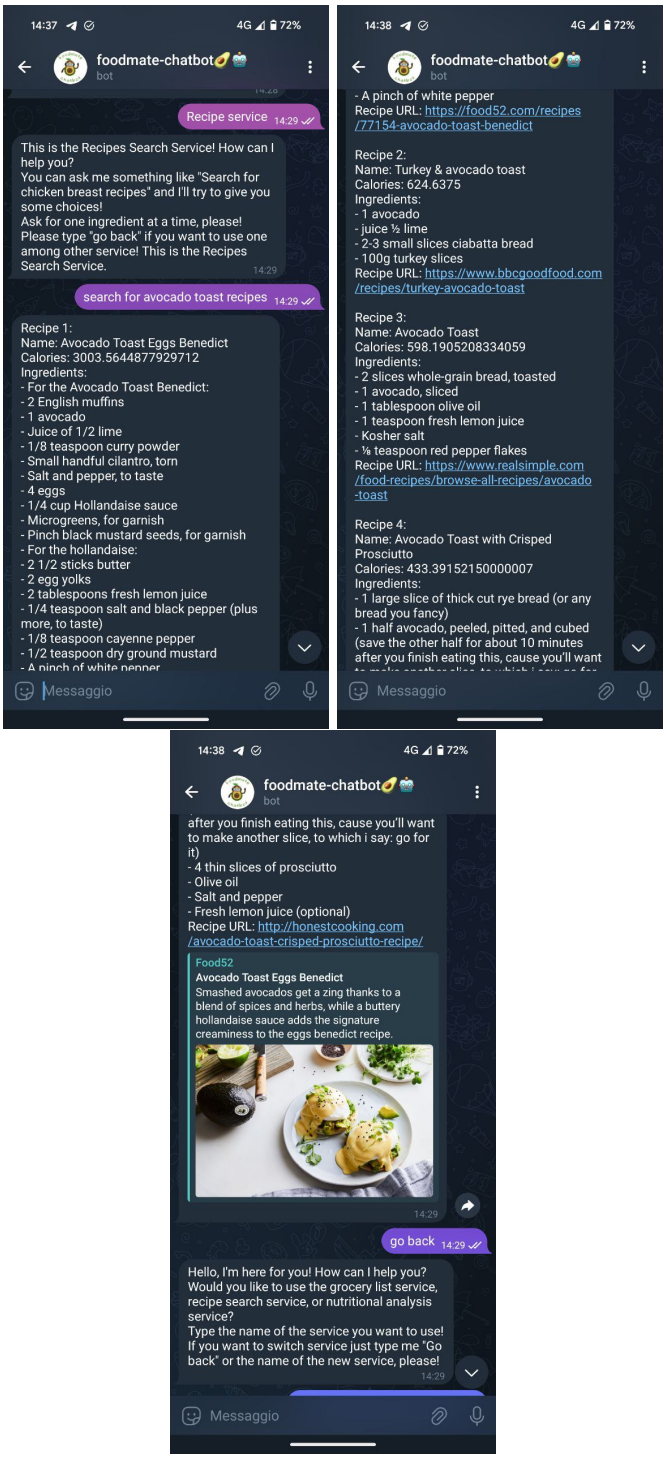

Figura 11: Servizio di ricerca ricette

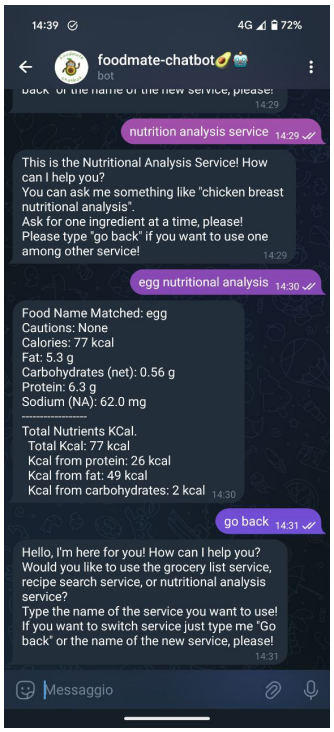

Figura 12: Servizio di analisi nutrizionale

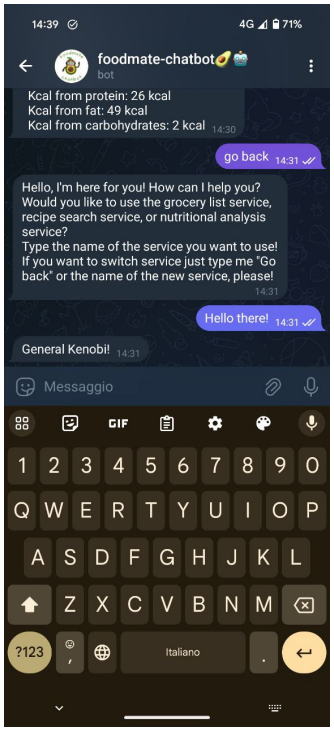

Figura 13: Saluto personalizzato - easter egg: omaggio simpatico per i fan di Star Wars

5.2 Difficoltà riscontrate

Ho riscontrato alcune difficoltà durante la realizzazione del progetto.
In particolare, durante la fase iniziale, ho avuto problemi nell'utilizzo della piattaforma **Dialogflow CX**, fornita da Google. Infatti, le documentazioni trovate sono esaustive per quanto riguarda le nozioni teoriche, ma povere di esempi pratici. Ho inoltre cercato online svariati tutorial, trovando spesso soltanto riferimenti alla piattaforma Dialogflow ES, che permette di realizzare progetti con più libertà e strumenti a disposizione.
Successivamente, ho trovato difficoltà nella fase di distribuzione tramite Telegram del chatbot Foodmate. Infatti, ho avuto problemi nel riuscire a comunicare correttamente con Dialogflow e, successivamente anche con il bot su Telegram. Sono riuscito a risolvere questi problemi dopo un'attenta revisione dei **payload** inviati e ricevuti dalle suddette piattaforme.
Infine, ho avuto difficoltà con le API di Edamam.com e più in particolare per quanto riguarda il **servizio di analisi nutrizionale**. Infatti, dapprima non ho gestito correttamente l'invio del payolad con la richiesta: una volta trovata la struttura corretta il problema che si verificava non si è più presentato.
Inoltre, non ho gestito fin dal primo momento i casi in cui la richiesta **GET** andasse a buon fine, ma con un esito inaspettato. Capita che il servizio offerto dal sito non trovi uno degli elementi di cui viene richiesta l'analisi nutrizionale. Nonostante ciò, la richiesta andava a buon fine, restituendo come output un risposta priva di informazioni.
Ho, dunque, gestito questo caso nel seguente modo: nel caso in cui la richiesta restituisca un output privo di informazioni, suppongo che il servizio fornito da Edamam.com non abbia informazioni relative a quell'ingrediente nel proprio database e modifico l'output inviato all'utente di conseguenza.

5.3 Sviluppi futuri

Nonostante sia molto contento del lavoro svolto, sono consapevole dei limiti che Foodmate presenta e ho già pensato ad alcuni possibili miglioramenti che potrebbero essere implementati come sviluppi futuri del progetto.
Possibili miglioramenti:

- aumentare la quantità di **frasi di addestramento** con le quali allenare ogni intent creato su Dialogflow CX;
- integrare in modo diretto l'utilizzo dell'assistente **Gemini**;
- garantire una maggiore fruizione del chatbot tramite la possibilità di **interagire in più lingue** (a partire dalla lingua italiana);
- utilizzare **dataset più affidabili** per il servizio di analisi nutrizionale (e di conseguenza, gestire tutto lato codice anziché tramite richiesta al sito Edamam.com);

- diminuire il tempo di attesa per le risposte generate dal bot. Forse un meccanismo di gestione di invio-ricezione dell'input utente diverso potrebbe comportare un miglior trade-off.

5.4 Considerazioni finali

La realizzazione di questo progetto mi ha permesso di conoscere una nuova tecnologia per la creazione di un chatbot: Dialogflow CX.
Per sviluppare la logica di un assistente virtuale, infatti, è una piattaforma molto utile e anche abbastanza semplice da utilizzare. Ha i suoi limiti ed è necessario leggere molto attentamente le documentazioni, per evitare di ottenere risultati inaspettati o errori, come è successo a me. Se però si volesse sviluppare un progetto più specifico e con più libertà, l'utilizzo della versione Dialogflow ES potrebbe comportare dei vantaggi.

La realizzazione di questo progetto, inoltre, mi ha permesso di creare un'assistente utilizzabile quotidianamente. Infatti, fin dal primo momento in cui Foodmate è stato distribuito tramite Telegram, l'ho utilizzato personalmente e l'ho fatto utilizzare ad amici e parenti, anche per raccogliere feedback e riscontri basati su test reali.
Sono molto contento di aver creato Foodmate e sono sicuro che continuerò ad utilizzarlo nella mia quotidianità, sperando comunque di riuscire a poterlo migliorare ulteriormente in futuro.

6 Appendice

Foodmate è usabile tramite il seguente link: https://t.me/foodmate_chatBot.
Il codice pubblico è disponibile alla seguente repository GitHub: https://github.com/gabM27/foodmate-chatbot.