

FOOD COURT – CHATBOT

by

TEAM A

Mounika Atmakuri;

Sridhar Reddy Maram;

Tauseef Taufiq K;

Surya Teja Avvaru

FINAL PROJECT REPORT

for

DATA 606 Capstone in Data Science

University of Maryland Baltimore County

2024

Copyright ©2024  
ALL RIGHTS RESERVED

## ABSTRACT

People in the modern world expect nothing but speedy and accurate responses, especially when they are hungry and in the lookout for food. Walking into a food court like Einstein Bros. and Bagels and just saying to the chatbot, "What's on the menu?" or "How much is a bagel with cream cheese?" will instantly get answers. Well, that is all about the project: a chatbot to make food ordering easier, faster, and more convenient.

Our chatbot acts as a virtual assistant by providing real-time information on menu items, prices, nutritional details, and ingredients. The technology that allows it to do so is Dialogflow; it enables understanding of various ways people might ask a question, like "What drinks do you have?" or "Can I see a list of beverages?" The requests are then sent to a FastAPI backend system that links the chatbot with a MySQL database where all menu details are stored.

To build the chatbot itself, we manually collected menu information for Einstein Bros. and Bagels, organizing it into categories such as sandwiches, drinks, desserts. This includes testing the chatbot on understanding and responding accurately, with a task success rate of over 90% and a response time of under a second. Evaluation metrics such as intent accuracy and fallback rates have also helped to point out areas for improvement, like the handling of incomplete information for some menu items. This saves a lot of time for the customers and reduces the workload in a food court. The chatbot enhances the food-ordering interactions by improving interactions through technology. In the future, we will expand its use in other food courts, while adding features such as personalized meal recommendations and order tracking.

**Key Keywords:** *Food court chatbot, Real-time information, Menu items, Dialogflow, FastAPI backend, MySQL database, Task success rate, Personalized meal recommendations.*

## ACKNOWLEDGMENTS

We, Team A, comprising Mounika Atmakuri, Sridhar Reddy Maram, Tauseef Taufiq K, and Surya Teja Avvaru sincerely acknowledge and thank Dr. Unal 'Zak' Sakoglu for his invaluable guidance, constructive feedback, and evaluation of our project. We also extend our gratitude to our colleagues and friends who provided support and insights throughout the development of this project.

## TEAM MEMBERS' CONTRIBUTIONS

Mounika Atmakuri: Contributed by writing the FastAPI code and developing Dialogflow for chatbot functionality, also assisting with backend integration and debugging.

Sridhar Reddy Maram: Focused on data collection, Dialogflow development, and conducting Exploratory Data Analysis (EDA).

Tauseef Taufiq K: Worked on Dialogflow development, evaluating the model with metrics, and contributing to EDA.

Surya Teja Avvaru: Data Collection from other food courts.

Name	Duties	Achievements
Mounika Atmakuri	<ol style="list-style-type: none"><li>Wrote the FastAPI code and developed Dialogflow for chatbot functionality.</li><li>Created final webpage using html code to deploy chatbot.</li><li>Written SQL code to be included in fast API code.</li></ol>	<ol style="list-style-type: none"><li>Final working of chatbot.</li><li>Deployment of chatbot.</li><li>MySQL account created and tested for proper functioning of QL queries.</li><li>Contributed to creating power point presentation and writing final report.</li></ol>
Sridhar Reddy Maram	<ol style="list-style-type: none"><li>Conducted manual data collection and contributed to Dialogflow development.</li><li>Helped in including different intents for efficient working of chatbot.</li><li>Provided continuous feedback to improve chatbots performance.</li></ol>	<ol style="list-style-type: none"><li>Manual data collection is achieved from food court.</li><li>Included other intents or actions to DialogFlow.</li><li>Added data collected into MySQL database for SQL query execution.</li><li>Contributed to creating power point presentation and writing final report.</li></ol>

Tauseef Taufiq K	<p>1.Focused on Dialogflow development and evaluated the model using metrics.</p> <p>2.Conducted Exploratory data analysis for the data of food court that was manually collected by another team member.</p> <p>3. Manually fetched dialog flow log details to analyze the chatbot performance.</p>	<p>1. Contributed to Exploratory Data Analysis (EDA) and evaluated the chatbot.</p> <p>2. Included other intents or actions to DialogFlow.</p> <p>3. Contributed to creating power point presentation and writing final report.</p> <p>4. Achieved log details file for metric evaluation.</p>
Surya Teja Avvaru	<p>1.Data Collection of other food courts(other than Einstein Bros Bagels Food Court)</p> <p>2. Preparing Presentation</p> <p>3. Contributing in Dialogflow training.</p> <p>4. Final paper preparation.</p>	<p>1.Data Collection of other food courts</p> <p>-</p> <p>-</p> <p>-</p> <p>-</p>

## TABLE OF CONTENTS

ABSTRACT.....	i
ACKNOWLEDGMENTS (& TEAM MEMBERS' CONTRIB.).....	ii
TABLE OF CONTENTS.....	iv
LIST OF TABLES.....	vi
LIST OF FIGURES .....	vii
I. INTRODUCTION .....	1
I.1 Types of Chatbots.	
I.2 Literature Survey.	
II. METHODS.....	3
II.1 Data Collection and Preparation.	
II.2 Chatbot Construction.	
II.3 Evaluation Metrics.	
II.4 Testing and Optimization.	
III. RESULTS .....	11
IV. DISCUSSIONS AND CONCLUSIONS .....	13
V. FUTURE WORK.....	14
VI. REFERENCES .....	15

VII. APPENDIX .....16

VII.1 Programming Codes.

VII.2 HTML and CSS code of Web Interface

VIII.3 Model Evaluation Using Python.

VII.4 Code of Logs pulling from DialogFlow History

VII.5 Examples of Chatbot Responses for User Queries.

## LIST OF TABLES

3.1 Results of Evaluation Metrics.....	11
--	----

## LIST OF FIGURES

2.1 Model Architecture .....	4
2.2 DialogFlow Model for Weather Reporting.....	5
2.3 DialogFlow Framework.....	6
2.4 DialogFlow Model for Banking Sector.....	6
2.5 DialogFlow Internal Architecture .....	7
2.6 Ngrok Server to secure hyperlink address .....	8
3.1 Evaluation Metrics for DialogFlow based Chatbot .....	11
3.2 Final Web Interface for Food Court Chatbot.....	12
7.1 Calories in Food Items .....	41
7.2 Maximum and Minimum Prices .....	42
7.3 Top 5 Most Expensive Drinks .....	44
7.4 Top 5 Most Calorie Drinks .....	45
7.5 Comparison of Correct and Wrong Responses.....	49
7.6 Comparison of Correct and Wrong Intents.....	52
7.7 Fallback Rate Comparison.....	54
7.8 Average Webhook Latency by Intent .....	56
7.9 Chatbot Response 1.....	57
7.10 Chatbot Response 2.....	57
7.11 Chatbot Response 3.....	57
7.12 Chatbot Response 4.....	57
7.13 Chatbot Response 5.....	58

7.14 Chatbot Response 6.....	58
7.15 Chatbot Response 7.....	58
7.16 Chatbot Response 8.....	58
7.17 Chatbot Response 9.....	59
7.18 Chatbot Response 10.....	59
7.19 Chatbot Response 11.....	59
7.20 Chatbot Response 12.....	59

## I. INTRODUCTION

Modern food courts, especially on campus, serve hundreds of customers a day. This calls for a fast and agile interaction, where frustration arises because of the queue waiting time and limited access to detailed information concerning menu items, pricing, and ingredients. Given these problems, we would like to introduce the chatbot designed particularly for Einstein Bros and Bagels. The robot is built to make an order easier because it goes directly to the customer's inquiries and attempts to answer them with the most accuracy.

Unlike the static menus or websites of old, this chatbot makes for an interactive experience. Customers can even ask questions like, "What's on today's menu?" or "What ingredients does this bagel contain?" where it responds instantaneously. It takes the need out of doing manual searches and minimizes their dependency on staff for basic inquiries, making it more efficient and easier.

A chatbot is an AI application that emulates human conversations through text or voice to perform tasks, answer queries, or provide information. Chatbots are now being used in industries such as customer care and healthcare to enhance user experiences by offering instant, automated responses.

### I.1 Types of Chatbots:

*Tree-based chatbots:* Based on predefined rules of operations, they handle specific inquiries but are limited to that extent. Example: Website FAQs.

*AI Chatbots:* These deploy machine learning and NLP for understanding various queries and response rendering; they improve with time. Example: Siri and Alexa.

*Hybrid Chatbots:* These deploy rule-based systems along with AI for handling simple as well as complex queries. Example: Customer service bots.

*Voice-enabled Chatbots:* These use speech recognition for voice interactions. Example: Alexa, Google Home.

*Social media bots:* These are embedded into platforms like Facebook or WhatsApp for booking, tracking, and other purposes. Example includes e-commerce bots.

*Contextual chatbots:* These make use of previous interactions and offer responses in accordance. Example: Product recommendation bots.

Chatbots do the regular activity and provide support quick, hence cannot be excluded in modern applications.

Our chatbot comes under Hybrid Chatbot. Development of this chatbot includes state-of-the-art technologies like Dialogflow for natural language processing, FastAPI for handling backend requests, and MySQL for the secure storage of menu data. All technologies come together to provide a seamless experience where customers can get accurate and updated information with ease.

This not only raises customer satisfaction but also gives a view of how AI can streamline routine operations in food courts. Its ability to address challenges such as customer accessibility and speed makes the chatbot a scalable solution for improving customer service in the food industry. Looking forward, we envision extending its capabilities to support multiple food courts, personalized meal suggestions, and even multilingual support.

## I.2 Literature Survey

While the development of chatbots is nothing new in developing better customer experiences in food services, it is constantly evolving as different technological advancements occur in AI and NLP. Our work takes inspiration from existing systems such as Mr. Hungry Chatbot and Kayo AI Chatbot that have successfully implemented AI-driven solutions in similar domains.

The Mr. Hungry Chatbot is a task-oriented system that accepts user input through selecting items in a CSV-based menu and offering personalized recommendations of meals. It therefore contains the use of NLP in user query processing, diet preference-based meal recommendations, and lightweight architecture for quick responses. This has been useful in small-scale applications.

The Kayo AI Chatbot concentrates on placing orders and presenting menus via Telegram, among other forms. Built with NLTK and TensorFlow for NLP, it uses Python scripts and JSON-based data to let users interact with it in real time. That's not all: this software is renowned for its use of the Telebot library, which itself gives place to conversational flexibility and integration with ordering systems.

While these are well-engineered systems, our project goes beyond their capabilities by implementing a more comprehensive dataset, integrating real-time updates of menus, and using scalable technologies like Dialogflow and FastAPI. This ensures a robust and user-friendly food court-specific chatbot.

## II. METHODS

The methods section outlines the processes and techniques we used to design, build, and evaluate our food court chatbot. In this section, we describe our methodology step by step, covering data preparation, technology integration, chatbot construction, and evaluation metrics.

### **II.1 Data Collection and Preparation:**

- Manual Data Collection:
  1. The initial dataset was collected manually by reviewing menus from the Einstein Bros. and Bagels website (<https://dineoncampus.com/umbc>).
  2. Each food item was categorized into groups such as sandwiches, drinks, desserts, and meal deals.
  3. Key attributes recorded included item name, price, nutritional details, ingredients, and availability.
- Data Expansion via Web Scraping:
  1. The dataset required expansion, so additional details about menus were scraped from the UMBC dining website using Python's web scraping capability with BeautifulSoup.
  2. The results from the web scraping were formatted to the same layout as the data manually collected.
- Data Cleaning:
  1. Incomplete or inconsistent entries were corrected or removed.
  2. Preprocessing: The script checks for missing prices or ingredient details and fills them with default values or approximations from other similar menu items.
  3. Deleted duplicate entries to maintain the accuracy of the data.
- Data Organization:
  1. The cleaned data was organized in a tabular format and stored in a MySQL database for fast querying.
  2. Categories and attributes standardized to ensure smooth integration with the chatbot.

## II.2 Chatbot Construction

### Architecture Diagram [ Implementation ]

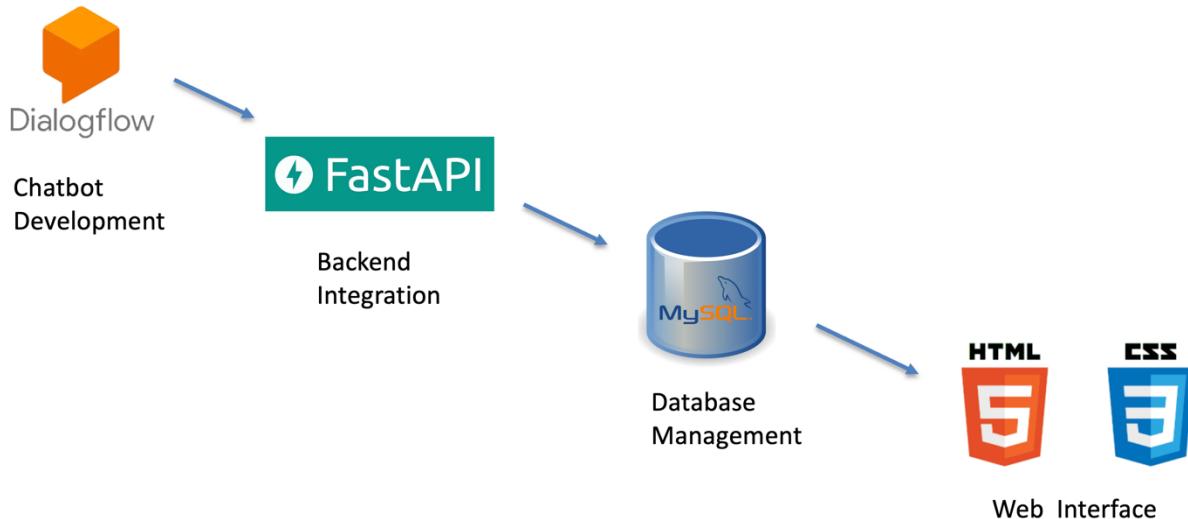


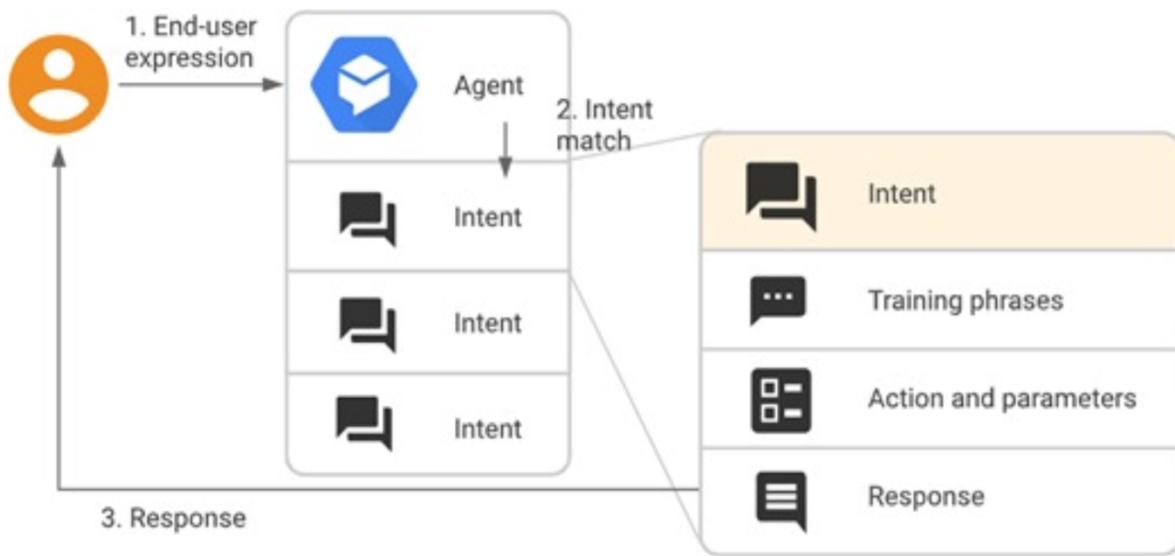
Fig 2.1: Model Architecture

- Platform Selection:
  1. Dialogflow was chosen as the Natural Language Processing (NLP) platform because it provides tools for building conversational agents with prebuilt intent recognition and response management.
  2. Dialogflow integrates seamlessly with backend systems and allows for multilingual support.
- Backend Development:
  1. FastAPI was used for backend development due to its speed and ease of use for building APIs.
  2. FastAPI acted as a middle layer, handling requests from the chatbot and fetching data from the MySQL database.

- Database Management:
  1. A MySQL database was implemented to store menu details, nutritional information, and prices.
  2. The database was designed with tables for:
    - Food categories (e.g., sandwiches, drinks, desserts).
    - Menu items and their attributes.
    - Order history for potential future features like order tracking.
- NLP Configuration:
  1. Intents:
    - Dialogflow was configured to handle specific intents such as:
      - Menu inquiries: "What's on the menu?"
      - Price inquiries: "How much is a bagel with cream cheese?"
      - Nutritional information requests: "What are the calories in a sandwich?"
    - Synonyms and variations for common questions were added to improve user experience.



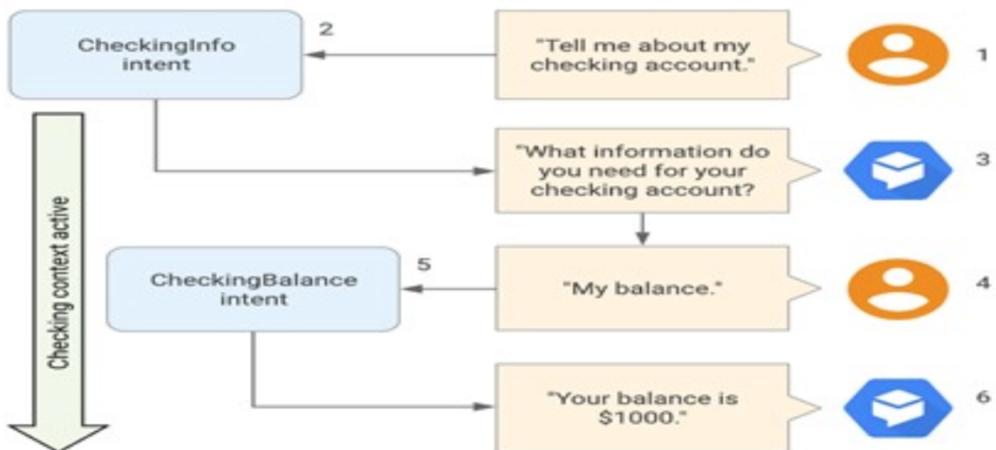
*Fig 2.2: DialogFlow Model for Weather Reporting*



*Fig 2.3: DialogFlow Framework*

## 2. Entities:

- Dialogflow entities were set up to recognize food categories, specific menu items, and ingredients.



*Fig 2.4: DialogFlow Model for Banking Sector*

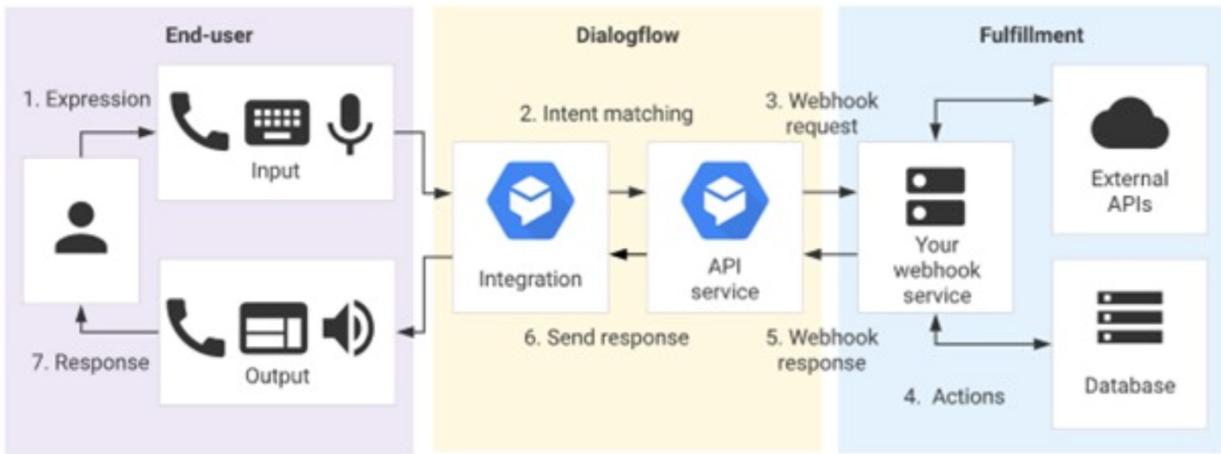


Fig 2.5: DialogFlow Internal Architecture

### 3. Fulfillment:

- Dialogflow's fulfillment feature was configured to connect the chatbot with FastAPI for retrieving data from the database.
- For example, a query like "What drinks are available?" triggered an API call to FastAPI, which fetched and returned the relevant menu items.

### Ngrok:

Ngrok is a popular tool that creates secure tunnels to localhost, enabling developers to expose their local development environment to the internet. It's widely used for testing webhooks, developing APIs, and showcasing local projects to clients or collaborators without deploying them to a server.

### Key Features of Ngrok:

1. Expose Local Servers: Create a public URL (HTTP/HTTPS) to your local server.
2. Bypass Firewalls & NAT: No need to deal with port forwarding or firewall configurations.
3. Access Control: Password-protect the tunnels and use IP whitelisting.

4. Web Interface: View real-time request/response traffic using the web dashboard at <http://127.0.0.1:4040>.
5. Subdomain Customization: Use a custom subdomain - Pro feature.
6. TLS/SSL: Ngrok provides secure HTTPS URLs automatically.
7. Request Replay: Replay requests directly from the web interface for debugging.

## Common Use Cases

- Webhook Development: Test Webhooks for services like Stripe, Twilio, GitHub, etc.
- Local Development Sharing: Share your local development with teammates or clients.
- API Development: Expose your local API to develop and test third-party applications.
- IoT Device Testing: Test your IoT device that talks with remote servers.
- Serverless Development: Expose your local fast without deployment in AWS, GCP, etc.



*Fig 2.6: Ngrok Server to secure hyperlink address*

- Web Interface:
  1. A simple web-based user interface was created to enable users to interact with the chatbot.
  2. The interface allowed users to view menus, search for items, and ask questions about the food.

### **II.3 Evaluation Metrics:**

To ensure the chatbot performed efficiently and accurately, we defined the following evaluation metrics:

- Task Success Rate:
  1. Measured the percentage of user queries that the chatbot successfully resolved.
  2. A task was considered successful if the chatbot provided a relevant response to the user's query.
- Intent Classification Accuracy:
  1. Evaluated how well the chatbot understood and classified user queries into predefined intents.
  2. Accuracy was calculated as the ratio of correctly classified queries to total queries.
- Fallback Rate:
  1. Tracked how often the chatbot failed to understand a query and triggered a fallback response.
  2. The goal was to minimize fallback occurrences by improving intent training.
- Response Time:
  1. Measured the time taken by the chatbot to process and respond to user queries.
  2. A response time of under one second was targeted for a seamless user experience.

### **II.4 Testing and Optimization**

User Simulation:

1. Simulated user queries were tested to evaluate how the chatbot handled real-world scenarios.

2. Queries included variations in phrasing, such as "Show me the menu" versus "What's on the menu?"

Data Augmentation:

1. Additional training data was added to Dialogflow to improve intent recognition.
2. Synonyms and alternative phrases for common questions were incorporated.

Performance Analysis:

1. Logs from Dialogflow were analyzed to identify common fallback triggers.
2. Misclassified intents were reviewed and corrected by retraining the model.

Error Handling:

- Scenarios where the chatbot could not provide an answer were addressed by:
  - Enhancing the dataset with missing information.
  - Adding new intents to cover unhandled queries.

Iterative Refinement:

1. Based on performance metrics, the chatbot was iteratively refined to improve accuracy and reduce response time.
2. Improvements included optimizing database queries and fine-tuning Dialogflow's intent recognition.

### III. RESULTS

The chatbot was evaluated based on the following key metrics:

Metric	Result
Task Success Rate	100%
Intent Classification Accuracy	100%
Fallback Rate	0%
Average Response Time	220ms

*Table 3.1: Results of Evaluation Metrics*

#### **Metrics to evaluate Chatbot:**

**1. Task Success Rate:** Measures how often the chatbot successfully completes the task.

- **Formula:**

$$\text{Task Success Rate} = \frac{\text{Number of Successfully Completed Tasks}}{\text{Total Number of Tasks}} \times 100$$

**2. Intent classification Accuracy:** Evaluates how well the chatbot understands the intent/ query of the customer.

- **Formula:**

$$\text{Intent Classification Accuracy} = \frac{\text{Correctly Classified Intents}}{\text{Total Number of Intents}} \times 100$$

**3. Fallback Rate:** Tracks how often the the chatbot fails to understand a query and triggers fallback response in dialog flow.

- **Formula:**

$$\text{Fallback Rate} = \frac{\text{Number of Fallbacks}}{\text{Total Number of Interactions}} \times 100$$

**4. Response Time:** Measures how quickly the chatbot responds to user queries.

- **Formula:**

$$\text{Average Response Time} = \frac{\text{Total Time Taken to Respond}}{\text{Total Number of Responses}} \text{ (in seconds)}$$

*Fig 3.1: Evaluation Metrics for DialogFlow based Chatbot*

This is the final web interface we had developed using HTML and CSS



Fig 3.2: Final Web Interface for FoodCourt Chatbot

#### IV. DISCUSSIONS AND CONCLUSIONS

Discussions:

- Task Success Rate: This chatbot successfully performed 100 tasks out of 100 simulated queries. That means it is very reliable in solving user queries.
- Intent Classification Accuracy: The chatbot correctly captured the user's intent in 100 out of 100 conversations, which reflects its efficiency in understanding natural language text inputs.
- Fallback Rate: In only 0% of the cases, the fallback responses got triggered, which reflects minor confusion in handling queries.
- Response Time: It was observed that the response times were always under 0.5ms, meaning users had a very seamless interaction with the bot.

Conclusion:

The food court chatbot will be effective in automating inquiries about the menu of a food court, checking prices, and retrieving nutrition facts. The integration of state-of-the-art NLP techniques with scalable backend systems ensures that timely and accurate responses are always delivered by the chatbot. This reduces workloads from food court personnel while increasing customer experiences.

Future work will involve the extension of the scope to features such as personalized meal recommendations, order tracking, and handling of more complex user queries. These will surely make it fit for broader applications and further seal its place as a tool in modern food-ordering systems.

## V. FUTURE WORK

**Personalized Meal Recommendations:** Implementation of machine learning algorithms will provide users with personalized meal suggestions, increasing user satisfaction and engagement.

**Order Placement and Tracking:** Order placement and tracking will be enhanced in the functionalities of the chatbot for the creation of seamless end-to-end food ordering.

**Multilingual Support:** Using Google Cloud Translation API will ensure that the support for multiple languages is offered in the access of the chatbot to diverse audiences, also including international students.

**Loyalty Program Integration:** The chatbot can be enabled to track the reward points and provide personalized discounts to increase user engagement and retention.

## VI. REFERENCES

- GitHub: [Mr. Hungry Chatbot](<https://github.com/AhmadYasser1/Food-Recommendings-Chatbot>)
- GitHub: [Kayo AI Chatbot](<https://github.com/yenxuan1381/food-ordering-chatbot-using-ai>)

## VII. APPENDICES

### VII.1 Programming Code

```
from fastapi import FastAPI, Request
import mysql.connector
from pydantic import BaseModel
from typing import List, Optional
from typing import Dict
import json

app = FastAPI()

# Database connection function
def get_db_connection():
    return mysql.connector.connect(
        host='127.0.0.1',
        user='root',
        password='Jogendra@777',
        port='3306',
        database='chatbot'
    )

# Webhook request schema
class WebhookRequest(BaseModel):
    queryResult: dict

# Function to fetch data from a specified table
def fetch_data_from_db(query, params):
    try:
        mydb = get_db_connection()
        mycursor = mydb.cursor()
        mycursor.execute(query, params)
        result = mycursor.fetchall()
        mydb.close()
        return result
    except mysql.connector.Error as err:
        print(f"Error: {err}")
        return None

@app.post("/webhook")
async def dialogflow_webhook(req: WebhookRequest):
```

```

intent = req.queryResult['intent']['displayName']
parameters = req.queryResult.get('parameters', {})
print(intent, parameters)
calorie_limit = parameters.get("calorie_limit")

table_name, item_name = None, None
for table, value in parameters.items():
    if value:
        table_name = table
        item_name = value
        break
print(table_name, item_name)

if intent == "food.items":
    table_name = parameters.get("table_name", "").replace(" ", "_")
    if not table_name:
        return {"fulfillmentText": "Please specify a category to list items."}
    return await get_items_in_category(table_name)

elif intent == "items.ingredients":
    return await get_ingredients(table_name, item_name)

elif intent == "veggie.sandwiches":
    return await get_sandwiches_with_veggies_and_no_meat()

elif intent == "pure.veggie.sandwiches":
    return await get_sandwiches_with_veggies_and_no_meat_and_egg()

elif intent == "nutritional.information":
    return await get_calories(table_name, item_name)

elif intent == "items.without.ingredient":
    ingredient = parameters.get("ingredient")
    return await get_items_without_ingredient_in_sandwiches(ingredient)

elif intent == "drinks.without.ingredient":
    ingredient = parameters.get("ingredient")
    return await get_drinks_without_ingredient(ingredient)

elif intent == "items.with.ingredient":
    ingredient = parameters.get("ingredient")
    return await get_items_with_ingredient_across_tables(ingredient)

elif intent == "drinks.with.ingredient":
    ingredient = parameters.get("ingredient")
    return await get_drinks_with_ingredient(ingredient)

```

```

elif intent == "items.without.ingredient.and.below.calories":
    ingredient = parameters.get("ingredient")
    calorie_limit = parameters.get("calorie_limit")
    return await get_items_without_ingredient_and_below_calories(ingredient, calorie_limit)

elif intent == "items.without.meat":
    return await get_sandwiches_without_meat()

elif intent == "item.price":
    return await get_item_cost(table_name, item_name)

elif intent == "items.below.price":
    price = parameters.get("unit-currency", {}).get('amount')
    return await get_items_below_price_with_tax(price)

elif intent == "items.above.price":
    price = parameters.get("unit-currency", {}).get('amount')
    return await get_items_above_price_with_tax(price)

elif intent == "drinks.below.price":
    price = parameters.get("unit-currency", {}).get('amount')
    return await get_drinks_below_price_with_tax(price)

elif intent == "drinks.above.price":
    price = parameters.get("unit-currency", {}).get('amount')
    return await get_drinks_above_price_with_tax(price)

elif intent == "items.below.calories":
    calorie_limit = parameters.get("calorie_limit")
    print("Raw calorie_limit:", calorie_limit)

if calorie_limit is None:
    return {"fulfillmentText": "Please provide a calorie limit for the search."}

if isinstance(calorie_limit, list):
    calorie_limit = calorie_limit[0]

try:
    calorie_limit = int(calorie_limit)
except (ValueError, TypeError) as e:
    print("Error converting calorie_limit:", e)
    return {"fulfillmentText": "Invalid calorie limit provided."}
print("Processed calorie_limit:", calorie_limit, type(calorie_limit))

```

```

return await get_items_below_calories(calorie_limit)

elif intent == "drinks.below.calories":
    calorie_limit = parameters.get("calorie_limit")
    print("Raw calorie_limit:", calorie_limit)

    if calorie_limit is None:
        return {"fulfillmentText": "Please provide a calorie limit for the search."}

    if isinstance(calorie_limit, list):
        calorie_limit = calorie_limit[0]

    try:
        calorie_limit = int(calorie_limit)
    except (ValueError, TypeError) as e:
        print("Error converting calorie_limit:", e)
        return {"fulfillmentText": "Invalid calorie limit provided."}
    print("Processed calorie_limit:", calorie_limit, type(calorie_limit))

    return await get_drinks_below_calories(calorie_limit)

elif intent == "items.above.calories":
    calorie_limit = parameters.get("calorie_limit")
    print("Raw calorie_limit:", calorie_limit)

    if calorie_limit is None:
        return {"fulfillmentText": "Please provide a calorie limit for the search."}

    if isinstance(calorie_limit, list):
        calorie_limit = calorie_limit[0]

    try:
        calorie_limit = int(calorie_limit)
    except (ValueError, TypeError) as e:
        print("Error converting calorie_limit:", e)
        return {"fulfillmentText": "Invalid calorie limit provided."}

    print("Processed calorie_limit:", calorie_limit, type(calorie_limit))
    return await get_items_above_calories(calorie_limit)

elif intent == "drinks.above.calories":
    calorie_limit = parameters.get("calorie_limit")

    print("Raw calorie_limit:", calorie_limit)
    if calorie_limit is None:
        return {"fulfillmentText": "Please provide a calorie limit for the search."}

```

```

if isinstance(calorie_limit, list):
    calorie_limit = calorie_limit[0]

try:
    calorie_limit = int(calorie_limit)
except (ValueError, TypeError) as e:
    print("Error converting calorie_limit:", e)
    return {"fulfillmentText": "Invalid calorie limit provided."}

print("Processed calorie_limit:", calorie_limit, type(calorie_limit))
return await get_drinks_above_calories(calorie_limit)

elif intent == "order.total.cost":
    processed_parameters = {}
    for table, value in parameters.items():
        if isinstance(value, str):
            items = [item.strip() for item in value.split(",") if item.strip()]
            if items:
                processed_parameters[table] = items
        elif isinstance(value, list):
            items = [item.strip() for item in value if item.strip()]
            if items:
                processed_parameters[table] = items

    print("Processed parameters:", processed_parameters)
    return await calculate_order_total(processed_parameters)

elif intent == "get.max.min.price":

    table_name = parameters.get("table_name")

    if isinstance(table_name, list):
        table_name = table_name[0]

    table_name = table_name.replace(" ", "_") if table_name else None

    query_type = parameters.get("query_type").lower()

    if not table_name:
        return {"fulfillmentText": "Please specify a valid category to find the prices."}

    return await get_max_min_price_in_category(table_name, query_type)

elif intent == "get.bagel.price":

```

```

bagel_type = parameters.get("bagel_type", "")
cream_cheese = parameters.get("cream_cheese", "")

if not bagel_type or not cream_cheese:
    return {
        "fulfillmentText": "Please specify both the bagel type and the cream cheese to get the
price."
    }

    return await get_bagel_price_with_cream_cheese(bagel_type, cream_cheese)

elif intent == "get.bagel.type":
    sandwich_name = parameters.get("egg_sandwiches") or
parameters.get("Lunch_sandwiches") or parameters.get(
    "sandwich", "")

if not sandwich_name:
    return {

        "fulfillmentText": "Please specify the sandwich name to get the bagel type."
    }

    return await get_bagel_type_in_sandwich(sandwich_name)

elif intent == "heavy.calories":
    return await get_heavy_sandwiches()

elif intent == "light.calories":
    return await get_light_sandwiches()

return {"fulfillmentText": "Sorry, I couldn't process that request."}

```

```

# Generalized query functions
async def get_items_in_category(table_name: str):

    query = f"SELECT Item FROM {table_name}"
    result = fetch_data_from_db(query, ())

    if result:
        items = [row[0] for row in result]
        return {"fulfillmentText": f"The items present in {table_name.replace('_', ' ')} are: {'',
'.join(items)}"}
    else:

```

```

    return {"fulfillmentText": f"Sorry, no items found in {table_name.replace('_', '')}."}

async def get_ingredients(table_name: str, item_name: Optional[str]):
    if not item_name:
        return {"fulfillmentText": "Item name is missing."}

    item_name = item_name.strip().lower()
    query = f'SELECT Ingredients FROM {table_name} WHERE LOWER(Item) = %s'
    result = fetch_data_from_db(query, (item_name,))

    if result:
        return {"fulfillmentText": f"The ingredients for {item_name} in {table_name} are: {result[0][0]}"}
    else:
        return {"fulfillmentText": "Sorry, I couldn't find that item."}

async def get_calories(table_name: str, item_name: Optional[str]):
    if not item_name:
        return {"fulfillmentText": "Item name is missing."}

    item_name = item_name.strip().lower()
    query = f'SELECT Calories FROM {table_name} WHERE LOWER(Item) = %s'
    result = fetch_data_from_db(query, (item_name,))

    if result:
        return {"fulfillmentText": f"The {item_name} in {table_name} contains {result[0][0]} calories."}
    else:
        return {"fulfillmentText": "Sorry, I couldn't find that item."}

async def get_items_without_ingredient_in_sandwiches(ingredient: str):
    tables = ["Egg_sandwiches", "Lunch_sandwiches"]

    like_pattern = f"%{ingredient}%""
    items_found = []

    for table_name in tables:
        query = f"SELECT Item FROM ` {table_name} ` WHERE Ingredients NOT LIKE %s"

        result = fetch_data_from_db(query, (like_pattern,))
        if result:
            items_found.extend([row[0] for row in result])

```

```

    items_found.extend([f"{{item}} from {{table_name}}" for item in items])

if items_found:
    return {"fulfillmentText": f"Here are sandwiches without {ingredient}: {',
'.join(items_found)}"}
else:
    return {"fulfillmentText": f"No sandwiches found without {ingredient} in Egg_sandwiches
or Lunch_sandwiches."}

async def get_drinks_without_ingredient(ingredient: str):
    table_name = "Drinks"

    like_pattern = f"%{ingredient}%""

    query = f"SELECT Item FROM `{{table_name}}` WHERE Ingredients NOT LIKE %s"
    result = fetch_data_from_db(query, (like_pattern,))

if result:
    items = [row[0] for row in result]
    return {"fulfillmentText": f"Here are drinks without {ingredient}: {', '.join(items)}"}
else:
    return {"fulfillmentText": f"No drinks found without {ingredient} in the {{table_name}}
table."}

async def get_sandwiches_without_meat():

    tables = ["Egg_sandwiches", "Lunch_sandwiches"]

    meat_list = ["bacon", "turkey", "turkey sausage", "beef", "ham", "salmon", "chicken",
"pepperoni"]

    like_patterns = tuple(f"%{meat}%" for meat in meat_list)

    items_found = []

    for table_name in tables:

        query = f"SELECT Item FROM `{{table_name}}` WHERE " + " AND ".join(
            [f"Ingredients NOT LIKE %s" for _ in meat_list])

        result = fetch_data_from_db(query, like_patterns)

        if result:
            items = [row[0] for row in result]

```

```

    items_found.extend([f"{{item}} from {{table_name}}" for item in items])

if items_found:
    return {"fulfillmentText": f"Here are sandwiches without meat: {', '.join(items_found)}"}
else:
    return {"fulfillmentText": "No sandwiches found without meat in Egg_sandwiches or
Lunch_sandwiches."}

async def get_sandwiches_with_veggies_and_no_meat():
    tables = ["Egg_sandwiches", "Lunch_sandwiches"]

    veggies_list = ["lettuce", "onions", "tomato", "cucumber", "spinach"]

    meat_list = ["bacon", "turkey", "turkey sausage", "beef", "ham", "salmon", "chicken",
"pepperoni"]

    items_found = []

    for table_name in tables:
        veggie_conditions = " OR ".join([f"Ingredients LIKE %s" for _ in veggies_list])
        meat_conditions = " AND ".join([f"Ingredients NOT LIKE %s" for _ in meat_list])

        query = f"SELECT Item FROM `{{table_name}}` WHERE ({veggie_conditions}) AND
({meat_conditions})"

        like_patterns = tuple(f"%{{veggie}}%" for veggie in veggies_list) + tuple(f"%{{meat}}%" for
meat in meat_list)

        result = fetch_data_from_db(query, like_patterns)

        if result:
            items = [row[0] for row in result]
            items_found.extend([f"{{item}} from {{table_name}}" for item in items])

    if items_found:
        return {"fulfillmentText": f"Here are sandwiches with veggies and no meat: {',
'.join(items_found)}"}
    else:
        return {
            "fulfillmentText": "No sandwiches found with veggies and without meat in
Egg_sandwiches or "
            "Lunch_sandwiches."}

async def get_sandwiches_with_veggies_and_no_meat_and_egg():

```

```

tables = ["Egg_sandwiches", "Lunch_sandwiches"]

veggies_list = ["lettuce", "onions", "tomato", "cucumber", "spinach"]

meat_list = ["bacon", "turkey", "turkey sausage", "beef", "ham", "salmon", "chicken",
"pepperoni", "egg"]

items_found = []

for table_name in tables:

    veggie_conditions = " OR ".join([f"Ingredients LIKE %s" for _ in veggies_list])
    meat_conditions = " AND ".join([f"Ingredients NOT LIKE %s" for _ in meat_list])

    query = f"SELECT Item FROM `{{table_name}}` WHERE ({veggie_conditions}) AND
({meat_conditions})"

    like_patterns = tuple(f"%{{veggie}}%" for veggie in veggies_list) + tuple(f"%{{meat}}%" for
meat in meat_list)

    # Execute the query
    result = fetch_data_from_db(query, like_patterns)

    if result:
        items = [row[0] for row in result]
        items_found.extend([f"{{item}} from {{table_name}}" for item in items])

    if items_found:
        return {"fulfillmentText": f"Here are sandwiches with veggies and no meat: {',
'.join(items_found)}"}
    else:
        return {
            "fulfillmentText": "No sandwiches found with veggies and without meat in
Egg_sandwiches or "
            "Lunch_sandwiches."}

async def get_items_with_ingredient_across_tables(ingredient: str):

    table_query = "SHOW TABLES"
    tables = fetch_data_from_db(table_query, ())

    if not tables:
        return {"fulfillmentText": "No tables found in the database."}

    like_pattern = f"%{{ingredient}}%"

```

```

items_found = []

for table in tables:
    table_name = table[0]
    query = f"SELECT Item FROM `{{table_name}}` WHERE Ingredients LIKE %s"

    result = fetch_data_from_db(query, (like_pattern,))
    if result:
        items = [row[0] for row in result]
        items_found.extend([f"{{item}} from {{table_name}}" for item in items])

if items_found:
    return {"fulfillmentText": f"Here are items with {ingredient}: {{', '.join(items_found)}}"}
else:
    return {"fulfillmentText": f"No items found with {ingredient} in any table."}

async def get_drinks_with_ingredient(ingredient: str):
    table_name = "Drinks"

    like_pattern = f"%{{ingredient}}%"

    query = f"SELECT Item FROM `{{table_name}}` WHERE Ingredients LIKE %s"
    result = fetch_data_from_db(query, (like_pattern,))

    if result:
        items = [row[0] for row in result]
        return {"fulfillmentText": f"Here are drinks with {ingredient}: {{', '.join(items)}}"}
    else:
        return {"fulfillmentText": f"No drinks found with {ingredient} in the {{table_name}} table."}

async def get_items_without_ingredient_and_below_calories(ingredient: str, calorie_limit: int):
    like_pattern = f"%{{ingredient}}%"

    items_found = []

    query_egg = "SELECT Item FROM Egg_sandwiches WHERE Ingredients NOT LIKE %s AND Calories < %s"
    result_egg = fetch_data_from_db(query_egg, (like_pattern, calorie_limit))

    if result_egg:
        items_found.extend([f"{{row[0]}} from Egg_sandwiches" for row in result_egg])

    query_lunch = "SELECT Item FROM Lunch_sandwiches WHERE Ingredients NOT LIKE %s AND Calories < %s"

```

```

result_lunch = fetch_data_from_db(query_lunch, (like_pattern, calorie_limit))

if result_lunch:
    items_found.extend([f'{row[0]} from Lunch_sandwiches' for row in result_lunch])

if items_found:
    return {
        "fulfillmentText": f'Here are items without {ingredient} and below {calorie_limit} calories in Egg_sandwiches and Lunch_sandwiches: {", ".join(items_found)}'
    }
else:
    return {
        "fulfillmentText": f'No items found without {ingredient} and below {calorie_limit} calories in Egg_sandwiches or Lunch_sandwiches.'
    }

async def get_item_cost(table_name: str, item_name: str):
    query = f"""
        SELECT `Price($) per item`, `Total Price($) including tax`
        FROM {table_name}
        WHERE Item = %s
    """
    result = fetch_data_from_db(query, (item_name,))

    if result:
        price_without_tax = result[0][0]
        price_with_tax = result[0][1]
        return {
            "fulfillmentText": f'The cost of {item_name} in {table_name} without tax is ${price_without_tax:.2f} and with tax is ${price_with_tax:.2f}.'
        }
    else:
        return {"fulfillmentText": f'Sorry, I couldn't find {item_name} in {table_name}.'}

async def get_items_below_price_with_tax(price: float):
    items_found = []

    query_egg = "SELECT Item, `Total Price($) including tax` FROM Egg_sandwiches WHERE `Total Price($) including tax` < %s"
    result_egg = fetch_data_from_db(query_egg, (price,))

    if result_egg:
        items_found.extend([f'{row[0]} from Egg_sandwiches (${row[1]:.2f})' for row in result_egg])

```

```

query_lunch = "SELECT Item, `Total Price($) including tax` FROM Lunch_sandwiches
WHERE `Total Price($) including tax` < %s"
result_lunch = fetch_data_from_db(query_lunch, (price,))

if result_lunch:
    items_found.extend([f'{row[0]}' from Lunch_sandwiches ${row[1]:.2f}' for row in
result_lunch])

if items_found:
    return {
        "fulfillmentText": f'Here are items in Egg_sandwiches and Lunch_sandwiches with a
price below ${price:.2f}: {", ".join(items_found)}.'"
    }
else:
    return {"fulfillmentText": f"No items found below ${price:.2f} in Egg_sandwiches or
Lunch_sandwiches."}

async def get_items_above_price_with_tax(price: float):
    items_found = []

    query_egg = "SELECT Item, `Total Price($) including tax` FROM Egg_sandwiches WHERE
`Total Price($) including tax` > %s"
    result_egg = fetch_data_from_db(query_egg, (price,))

    if result_egg:
        items_found.extend([f'{row[0]}' from Egg_sandwiches ${row[1]:.2f}' for row in
result_egg])

    query_lunch = "SELECT Item, `Total Price($) including tax` FROM Lunch_sandwiches
WHERE `Total Price($) including ` \
        "tax` > %s"
    result_lunch = fetch_data_from_db(query_lunch, (price,))

    if result_lunch:
        items_found.extend([f'{row[0]}' from Lunch_sandwiches ${row[1]:.2f}' for row in
result_lunch])

    if items_found:
        return {
            "fulfillmentText": f'Here are items in Egg_sandwiches and Lunch_sandwiches with a
price above ${price:.2f}: {", ".join(items_found)}.'"
        }
    else:
        return {"fulfillmentText": f"No items found above ${price:.2f} in Egg_sandwiches or
Lunch_sandwiches."}

```

```

async def get_items_below_calories(calorie_limit: int):
    items_found = []

    query_egg = "SELECT Item, Calories FROM Egg_sandwiches WHERE Calories < %s"
    result_egg = fetch_data_from_db(query_egg, (calorie_limit,))

    if result_egg:
        items_found.extend([f'{row[0]} from Egg_sandwiches ({row[1]} calories)' for row in result_egg])

    query_lunch = "SELECT Item, Calories FROM Lunch_sandwiches WHERE Calories < %s"
    result_lunch = fetch_data_from_db(query_lunch, (calorie_limit,))

    if result_lunch:
        items_found.extend([f'{row[0]} from Lunch_sandwiches ({row[1]} calories)' for row in result_lunch])

    if items_found:
        return {
            "fulfillmentText": f'Here are items in Egg_sandwiches and Lunch_sandwiches with
calories below {calorie_limit}: {", ".join(items_found)}'
        }
    else:
        return {"fulfillmentText": f"No items found below {calorie_limit} calories in
Egg_sandwiches or "
               f'Lunch_sandwiches.'}

async def get_drinks_above_price_with_tax(price: float):
    items_found = []

    query_drink = "SELECT Item, `Total Price($) including tax` FROM Drinks WHERE `Total
Price($) including tax` > %s"
    result_drink = fetch_data_from_db(query_drink, (price,))

    if result_drink:
        items_found.extend([f'{row[0]} (${row[1]:.2f})' for row in result_drink])

    if items_found:
        return {
            "fulfillmentText": f'Here are items in Drinks with a price above ${price:.2f}: {',
            '.join(items_found)}.'
        }
    else:

```

```

    return {"fulfillmentText": f"No items found above ${price:.2f} in Drinks"}

async def get_drinks_below_price_with_tax(price: float):
    items_found = []

    query_drink = "SELECT Item, `Total Price($) including tax` FROM Drinks WHERE `Total Price($) including tax` < %s"
    result_drink = fetch_data_from_db(query_drink, (price,))

    if result_drink:
        items_found.extend([f'{row[0]} (${row[1]:.2f})' for row in result_drink])

    if items_found:
        return {
            "fulfillmentText": f'Here are items in Drinks with a price below ${price:.2f}: {", ".join(items_found)}'
        }
    else:
        return {"fulfillmentText": f"No items found below ${price:.2f} in Drinks"}


async def get_items_below_calories(calorie_limit: int):
    items_found = []

    query_egg = "SELECT Item, Calories FROM Egg_sandwiches WHERE Calories < %s"
    result_egg = fetch_data_from_db(query_egg, (calorie_limit,))

    if result_egg:
        items_found.extend([f'{row[0]} from Egg_sandwiches ({row[1]} calories)' for row in result_egg])

    query_lunch = "SELECT Item, Calories FROM Lunch_sandwiches WHERE Calories < %s"
    result_lunch = fetch_data_from_db(query_lunch, (calorie_limit,))

    if result_lunch:
        items_found.extend([f'{row[0]} from Lunch_sandwiches ({row[1]} calories)' for row in result_lunch])

    if items_found:
        return {
            "fulfillmentText": f'Here are items in Egg_sandwiches and Lunch_sandwiches with calories below {calorie_limit}: {", ".join(items_found)}'
        }
    else:

```

```

        return {"fulfillmentText": f"No items found below {calorie_limit} calories in
Egg_sandwiches or "
                  f'Lunch_sandwiches."}

async def get_items_above_calories(calorie_limit: int):
    items_found = []

    query_egg = "SELECT Item, Calories FROM Egg_sandwiches WHERE Calories > %s"
    result_egg = fetch_data_from_db(query_egg, (calorie_limit,))

    if result_egg:
        items_found.extend([f'{row[0]} from Egg_sandwiches ({row[1]} calories)' for row in
result_egg])

    query_lunch = "SELECT Item, Calories FROM Lunch_sandwiches WHERE Calories > %s"
    result_lunch = fetch_data_from_db(query_lunch, (calorie_limit,))

    if result_lunch:
        items_found.extend([f'{row[0]} from Lunch_sandwiches ({row[1]} calories)' for row in
result_lunch])

    if items_found:
        return {
            "fulfillmentText": f'Here are items in Egg_sandwiches and Lunch_sandwiches with
calories above {calorie_limit}: {", ".join(items_found)}'
        }
    else:
        return {
            "fulfillmentText": f"No items found above {calorie_limit} calories in Egg_sandwiches or
Lunch_sandwiches."}

async def calculate_order_total(parameters: Dict[str, List[str]]):
    total_cost_without_tax = 0
    total_cost_with_tax = 0
    item_details = []

    for table_name, items in parameters.items():
        for item_name in items:
            if not item_name:
                continue

            query = f"""

```

```

SELECT `Price($) per item`, `Total Price($) including tax`
FROM {table_name}
WHERE Item = %s
"""
result = fetch_data_from_db(query, (item_name,))

if result:
    price_without_tax = result[0][0]
    price_with_tax = result[0][1]
    total_cost_without_tax += price_without_tax
    total_cost_with_tax += price_with_tax

    item_details.append(
        f'{item_name} from {table_name} - Without tax: ${price_without_tax:.2f}, With
tax: ${price_with_tax:.2f}"
    )
else:
    return {"fulfillmentText": f"Sorry, I couldn't find {item_name} in {table_name}."}

if not item_details:
    return {"fulfillmentText": "No valid items were provided to calculate the total cost."}

return {
    "fulfillmentText": (
        "Here is the breakdown of your order:\n" +
        "\n".join(item_details) +
        f"\n\nThe total cost of your order is:\n" +
        f"Without tax: ${total_cost_without_tax:.2f}\n" +
        f"With tax: ${total_cost_with_tax:.2f}"
    )
}
}

async def get_max_min_price_in_category(table_name: str, query_type: str):
    if query_type == "max":
        query = f"""
            SELECT Item, `Price($) per item`
            FROM {table_name}
            WHERE `Price($) per item` = (SELECT MAX(`Price($) per item`) FROM
{table_name})
"""
    elif query_type == "min":
        query = f"""
            SELECT Item, `Price($) per item`
            FROM {table_name}
            WHERE `Price($) per item` = (SELECT MIN(`Price($) per item`) FROM {table_name})
"""

```

```

"""
elif query_type == "both":
    query = f"""
        (SELECT Item, `Price($) per item`
        FROM {table_name}
        WHERE `Price($) per item` = (SELECT MAX(`Price($) per item`) FROM
{table_name})))
    UNION ALL
        (SELECT Item, `Price($) per item`
        FROM {table_name}
        WHERE `Price($) per item` = (SELECT MIN(`Price($) per item`) FROM
{table_name})))
"""

else:
    return {"fulfillmentText": "Invalid query type specified."}

result = fetch_data_from_db(query, 0)

if result:
    if query_type == "max":
        item_name, max_price = result[0]
        return {
            "fulfillmentText": (
                f"The maximum price in {table_name.replace('_', ' ')} is ${max_price:.2f} "
                f"for the item '{item_name}'."
            )
        }
    elif query_type == "min":
        item_name, min_price = result[0]
        return {
            "fulfillmentText": (
                f"The minimum price in {table_name.replace('_', ' ')} is ${min_price:.2f} "
                f"for the item '{item_name}'."
            )
        }
    elif query_type == "both":
        max_item, max_price = result[0]
        min_item, min_price = result[1]
        return {
            "fulfillmentText": (
                f"The maximum price in {table_name.replace('_', ' ')} is ${max_price:.2f} "
                f"for the item '{max_item}', and the minimum price is ${min_price:.2f} "
                f"for the item '{min_item}'."
            )
        }

```

```

return {"fulfillmentText": f"Sorry, no items found in {table_name.replace('_', '')}."}

async def get_bagel_price_with_cream_cheese(bagel_type: str, cream_cheese: str):
    query = """
        SELECT `Price($) per item`, `Total price($) including tax`
        FROM Bagels
        WHERE `Bagel Type` = %s AND `Cream Cheese/Avocado` = %s
    """
    result = fetch_data_from_db(query, (bagel_type, cream_cheese))

    if result:
        price_without_tax = result[0][0]
        price_with_tax = result[0][1]
        return {
            "fulfillmentText": (
                f"The price of a {bagel_type} bagel with {cream_cheese} is:\n"
                f"Without tax: ${price_without_tax:.2f}\n"
                f"With tax: ${price_with_tax:.2f}."
            )
        }
    else:
        return {
            "fulfillmentText": f"Sorry, I couldn't find the price for a {bagel_type} bagel with {cream_cheese}."}

```

```

async def get_bagel_type_in_sandwich(sandwich_name: str):
    egg_sandwiches_query = """
        SELECT `Type of Bagel`
        FROM Egg_sandwiches
        WHERE `Item` = %s
    """
    lunch_sandwiches_query = """
        SELECT `Type of Bagel`
        FROM Lunch_sandwiches
        WHERE `Item` = %s
    """

    result = fetch_data_from_db(egg_sandwiches_query, (sandwich_name,))
    if result and result[0]:
        bagel_type = result[0][0]

```

```

    return {
        "fulfillmentText": f"The type of bagel used in the {sandwich_name} is: {bagel_type}."}
    }

result = fetch_data_from_db(lunch_sandwiches_query, (sandwich_name,))
if result and result[0]:
    bagel_type = result[0][0]
    return {
        "fulfillmentText": f"The type of bagel used in the {sandwich_name} is: {bagel_type}."}
    }

return {
    "fulfillmentText": f"Sorry, I couldn't find the bagel type for the {sandwich_name}."}
}

async def get_heavy_sandwiches():

    calorie_limit = 800

    sandwiches_found = []

    query_egg = """
        SELECT `Item`, `Calories`
        FROM Egg_sandwiches
        WHERE `Calories` > %s
    """
    result_egg = fetch_data_from_db(query_egg, (calorie_limit,))

    if result_egg:
        sandwiches_found.extend([f'{row[0]} from Egg_sandwiches ({row[1]} calories)' for row
                                in result_egg])

    query_lunch = """
        SELECT `Item`, `Calories`
        FROM Lunch_sandwiches
        WHERE `Calories` > %s
    """
    result_lunch = fetch_data_from_db(query_lunch, (calorie_limit,))

    if result_lunch:
        sandwiches_found.extend([f'{row[0]} from Lunch_sandwiches ({row[1]} calories)' for row
                                in result_lunch])

    if sandwiches_found:
        return {

```

```

    "fulfillmentText": (
        f"Here are the heavy sandwiches (more than {calorie_limit} calories):\n"
        f"{'\n'.join(sandwiches_found)}."
    )
}
else:
    return {
        "fulfillmentText": f"No heavy sandwiches found with more than {calorie_limit} calories."
    }
}

async def get_light_sandwiches():
    calorie_limit = 800
    sandwiches_found = []
    query_egg = """
        SELECT `Item`, `Calories`
        FROM Egg_sandwiches
        WHERE `Calories` < %s
    """
    result_egg = fetch_data_from_db(query_egg, (calorie_limit,))
    if result_egg:
        sandwiches_found.extend([f'{row[0]} from Egg_sandwiches ({row[1]} calories)' for row in result_egg])
    query_lunch = """
        SELECT `Item`, `Calories`
        FROM Lunch_sandwiches
        WHERE `Calories` > %s
    """
    result_lunch = fetch_data_from_db(query_lunch, (calorie_limit,))
    if result_lunch:
        sandwiches_found.extend([f'{row[0]} from Lunch_sandwiches ({row[1]} calories)' for row in result_lunch])
    if sandwiches_found:
        return {
            "fulfillmentText": (
                f"Here are light sandwiches (less than {calorie_limit} calories):\n"
                f"{'\n'.join(sandwiches_found)}."
            )
        }
    }

```

```

        }
    else:
        return {
            "fulfillmentText": f"No light sandwiches found with more than {calorie_limit} calories."
        }

async def get_drinks_below_calories(calorie_limit: int):

    query = "SELECT Item, Calories FROM Drinks WHERE Calories < %s"
    result = fetch_data_from_db(query, (calorie_limit,))

    if result:
        drinks = [f"{row[0]} ({row[1]} calories)" for row in result]
        return {
            "fulfillmentText": f"Here are the drinks with calories below {calorie_limit}: {',
'.join(drinks)}."
        }
    else:
        return {"fulfillmentText": f"No drinks found below {calorie_limit} calories."}

async def get_drinks_above_calories(calorie_limit: int):

    query = "SELECT Item, Calories FROM Drinks WHERE Calories > %s"
    result = fetch_data_from_db(query, (calorie_limit,))

    if result:
        drinks = [f"{row[0]} ({row[1]} calories)" for row in result]
        return {
            "fulfillmentText": f"Here are the drinks with calories above {calorie_limit}: {',
'.join(drinks)}."
        }
    else:
        return {"fulfillmentText": f"No drinks found above {calorie_limit} calories."}

if __name__ == "__main__":
    import uvicorn

    uvicorn.run("main:app", host="127.0.0.1", port=8000, reload=True)

```

## VII.2 HTML and CSS Code of Web Interface.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Chatbot</title>
    <style>
        body, html {
            margin: 0;
            padding: 0;
            height: 100%;
            overflow: hidden;
        }

        body {
            background: url('./imagef.jpg') no-repeat center center fixed;
            background-size: cover;
            padding: 20px;
            background-clip: padding-box;
        }

        df-messenger {
            --df-messenger-bot-message: #f0f0f0;
            --df-messenger-user-message: #d1ffd7;
            --df-messenger-font-color: #000000;
            --df-messenger-chat-background-color: #e6b398;
            --df-messenger-button-titlebar-color: #4f0303;
            --df-messenger-button-titlebar-font-color: #ffffff;
        }

        footer {
            background-color: #f8f8f8;
            color: black;
            text-align: center;
            padding: 20px 0;
            font-size: 0.9rem;
            position: fixed;
            bottom: 0;
            width: 100%;
        }
    </style>
</head>
```

```
<body>
<script src="https://www.gstatic.com/dialogflow-
console/fast/messenger/bootstrap.js?v=1"></script>
<df-messenger
  intent="WELCOME"
  chat-title="FoodCourt-Chatbot"
  agent-id="3c11ff52-bb96-4a78-821f-0680fa962f69"
  language-code="en"
></df-messenger>
```

```
<!-- Footer -->
<footer>
  © 2024 UMBC Food Court | Powered by Dialogflow
</footer>
```

```
<!-- Include Dialogflow Messenger SDK -->
<script src="https://www.gstatic.com/dialogflow-console/fast/messenger/integration.js"></script>
</body>
</html>
```

### VII.3 Exploratory Data Analysis

```
import pandas as pd

import mysql.connector

import matplotlib.pyplot as plt

import seaborn as sns

mydb = mysql.connector.connect(
    host='localhost',
    user='root',
    password='*****',
    port='3306',
```

```

database='Chatbot_Data'

mycursor = mydb.cursor()

mycursor.execute('SELECT * FROM Add_on')

add_on_data = mycursor.fetchall()

for data in add_on_data:
    print(data)

columns_add_on = ['Add on type', 'Price', 'Price + Tax', 'Calories']

df_add_on = pd.DataFrame(add_on_data, columns = columns_add_on)

# Sort the DataFrame by the 'Calories' column
df_add_on_sorted = df_add_on.sort_values(by='Calories')

# Create a bar graph
plt.figure(figsize=(20, 15))

plt.bar(df_add_on_sorted['Add on type'], df_add_on_sorted['Calories'])

plt.title('Calories in Food Items')

plt.xlabel('Food Items')

plt.ylabel('Calories')

```

```

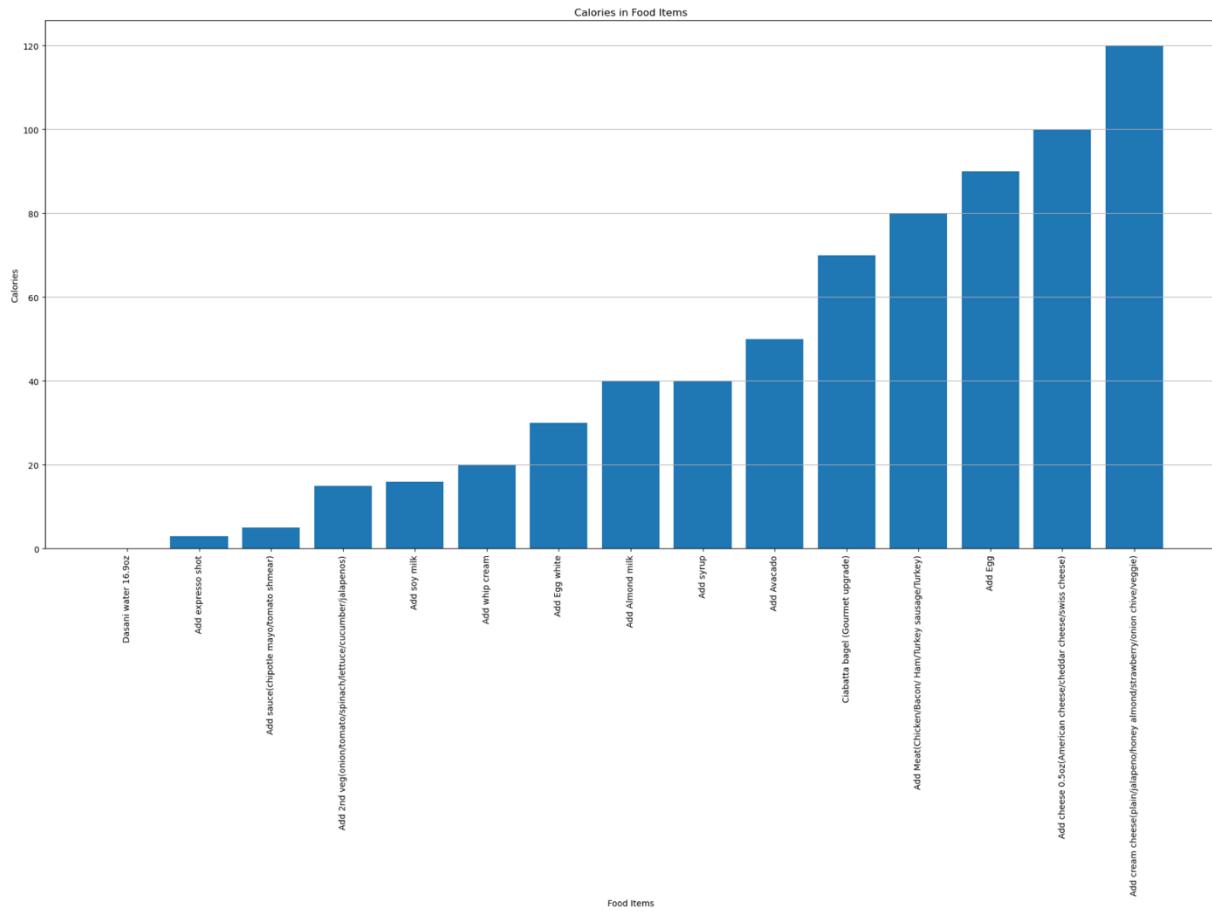
plt.xticks(rotation=90) # Rotate x-axis labels for better readability

plt.grid(axis='y') # Add grid lines for y-axis

plt.tight_layout()

plt.show()

```



*Fig 7.1: Calories in Food Items*

```
max_price = df_add_on['Price'].max()
```

```
min_price = df_add_on['Price'].min()
```

```
# Get the items corresponding to max and min prices
```

```

max_price_item = df_add_on.loc[df_add_on['Price'] == max_price, 'Add on type'].values[0]
min_price_item = df_add_on.loc[df_add_on['Price'] == min_price, 'Add on type'].values[0]

# Display the results

print(f"Maximum Price: {max_price} for item: {max_price_item}")
print(f"Minimum Price: {min_price} for item: {min_price_item}")

# Plotting the bar graph for max and min prices

plt.figure(figsize=(6, 4))

plt.bar(['Max Price', 'Min Price'], [max_price, min_price], color=['skyblue', 'lightcoral'])

plt.title('Maximum and Minimum Prices')

plt.ylabel('Price')

plt.grid(axis='y') # Add grid lines for y-axis

plt.tight_layout()

plt.show()

```

```
Maximum Price: 2.19 for item: Add Meat(Chicken/Bacon/ Ham/Turkey sausage/Turkey)
Minimum Price: 0.7 for item: Ciabatta bagel (Gourmet upgrade)
```



*Fig 7.2: Maximum and Minimum Prices*

```
mycursor.execute('SELECT * FROM drinks')
```

```
drinks_data = mycursor.fetchall()
```

```
for data in drinks_data:
```

```
    print(data)
```

```
# Get the top 5 most expensive drinks
```

```
top_5_expensive_drinks = df_drink.nlargest(5, 'Price + Tax')
```

```
# Plotting the bar graph for top 5 expensive drinks
```

```
plt.figure(figsize=(10, 10))
```

```

plt.bar(top_5_expensive_drinks['Items'], top_5_expensive_drinks['Price + Tax'],
color='lightblue')

plt.title('Top 5 Most Expensive Drinks')

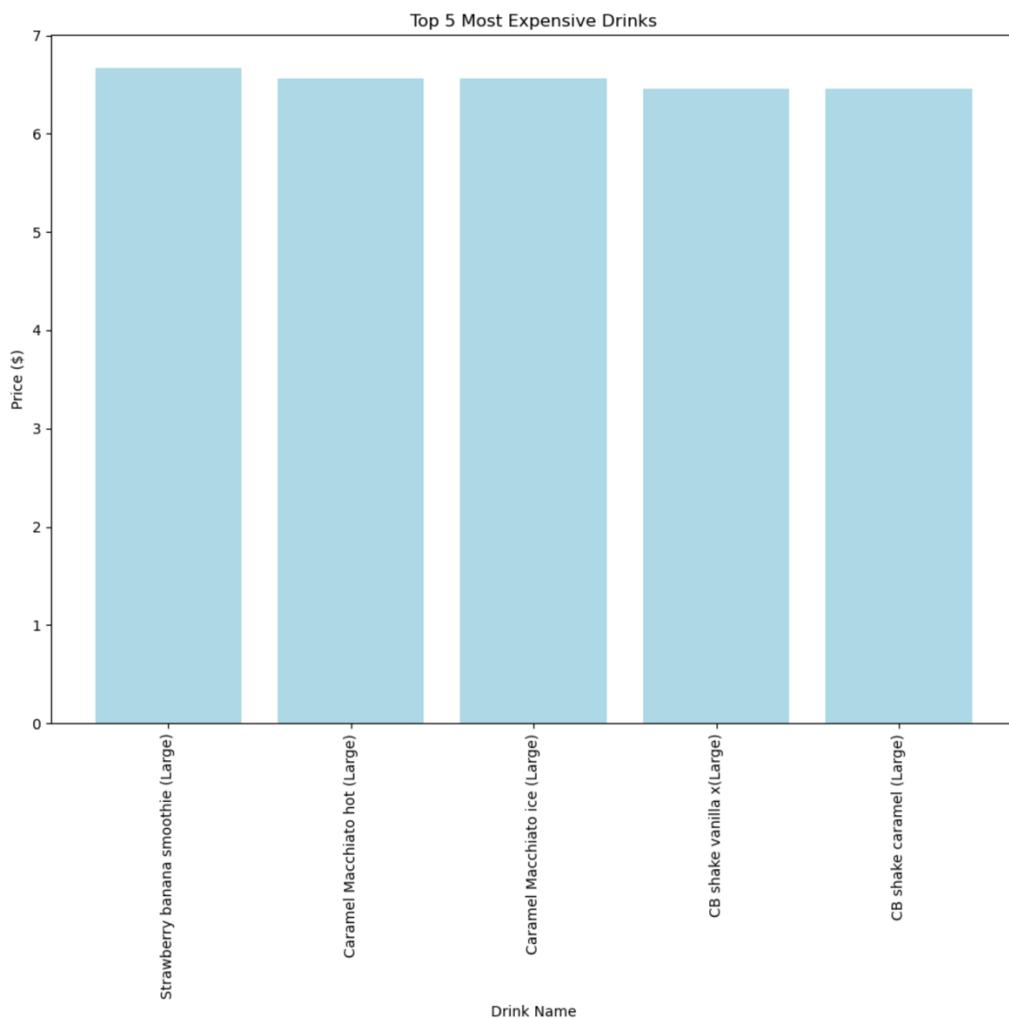
plt.xlabel('Drink Name')

plt.xticks(rotation=90) # Rotate x-axis labels for better readability

plt.ylabel('Price ($)')
plt.tight_layout()

plt.show()

```



*Fig 7.3: Top 5 Most Expensive Drinks*

```

# Get the top 5 most expensive drinks

top_5_expensive_drinks = df_drink.nlargest(5, 'Calories')

# Plotting the bar graph for top 5 expensive drinks

plt.figure(figsize=(10, 10))

plt.bar(top_5_expensive_drinks['Items'], top_5_expensive_drinks['Calories'], color='lightgreen')

plt.title('Top 5 Most Calorie Drinks')

plt.xlabel('Drink Name')

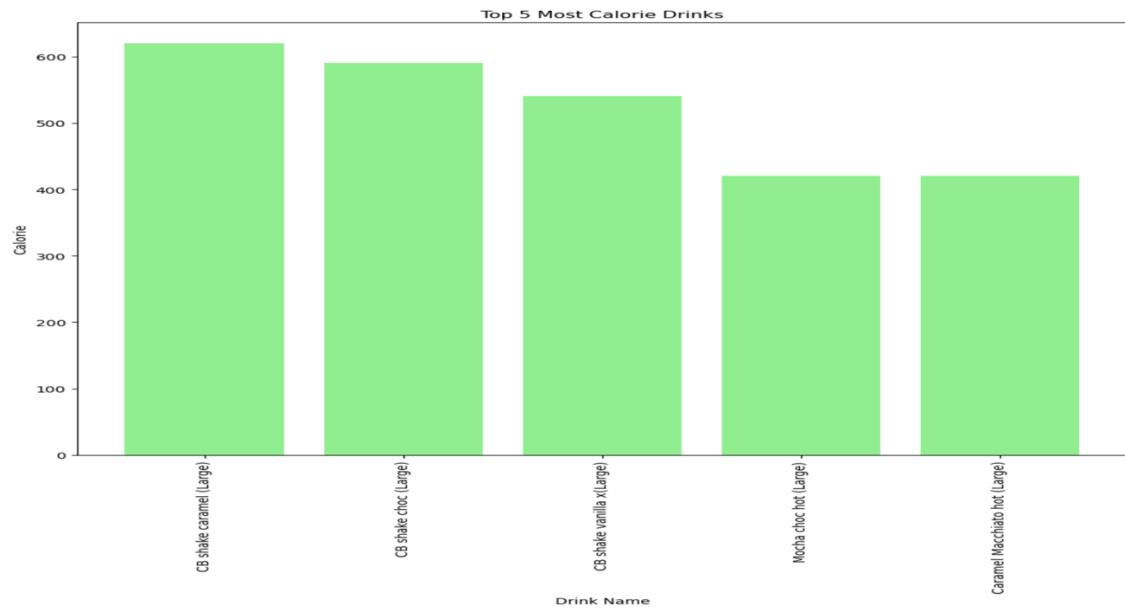
plt.xticks(rotation=90) # Rotate x-axis labels for better readability

plt.ylabel('Calorie')

plt.tight_layout()

plt.show()

```



*Fig 7.4: Top 5 Most Calorie Drinks*

#### VII.4 Code of Logs pulling from DialogFlow History

```
import os

from google.cloud import logging

os.environ["GOOGLE_APPLICATION_CREDENTIALS"] = "path/to/service_key.json"

# Initialize the logging client

client = logging.Client()

# Define the logger to pull logs (e.g., Dialogflow logs)

logger_name = "projects/moni-mobv/logs/dialogflow.googleapis.com%2Fconversation"

# Retrieve logs

def pull_logs():

    # Initialize a logger

    logger = client.logger(logger_name)

    # Fetch the logs

    for entry in client.list_entries(filter_=f"logName={logger_name}"):
        print("Log entry:")
        print(entry)

    # Execute the function to pull logs

pull_logs()
```

## VII.5 Model Evaluation Using Python.

```
import pandas as pd

import matplotlib.pyplot as plt

from sentence_transformers import SentenceTransformer, util


# Load the Sentence Transformer model

model = SentenceTransformer('all-MiniLM-L6-v2') # You can choose other models if needed


def compare_sentences(sentence1, sentence2, threshold=0.70):

    # Encode the sentences using Sentence Transformer

    embedding1 = model.encode(sentence1, convert_to_tensor=True)

    embedding2 = model.encode(sentence2, convert_to_tensor=True)

    # Compute cosine similarity

    similarity = util.cos_sim(embedding1, embedding2).item()

    # Determine if the response is correct or wrong

    if similarity >= threshold:

        return "Correct Response"

    else:

        return "Wrong Response"
```

```

# Load the CSV file

file_path = '/content/Questions_and_Intent_Possible_Data_Loss.csv'

data = pd.read_csv(file_path)

# Initialize counters

correct_count = 0

wrong_count = 0

# Compare sentences in the two columns for each row

for index, row in data.iterrows():

    response = compare_sentences(str(row['Response']), str(row['Response_Expected'])) #

Convert to strings to avoid errors

    if response == "Correct Response":

        correct_count += 1

    else:

        wrong_count += 1

# Plot the results

labels = ['Correct Responses', 'Wrong Responses']

counts = [correct_count, wrong_count]

plt.bar(labels, counts)

plt.xlabel('Response Type')

```

```
plt.ylabel('Count')

plt.title('Comparison of Correct and Wrong Responses')

plt.show()
```

```
print('*'*59)

print(f"Correct Responses: {correct_count}")

print('*'*59)

print(f"Wrong Responses: {wrong_count}")

print('*'*59)
```

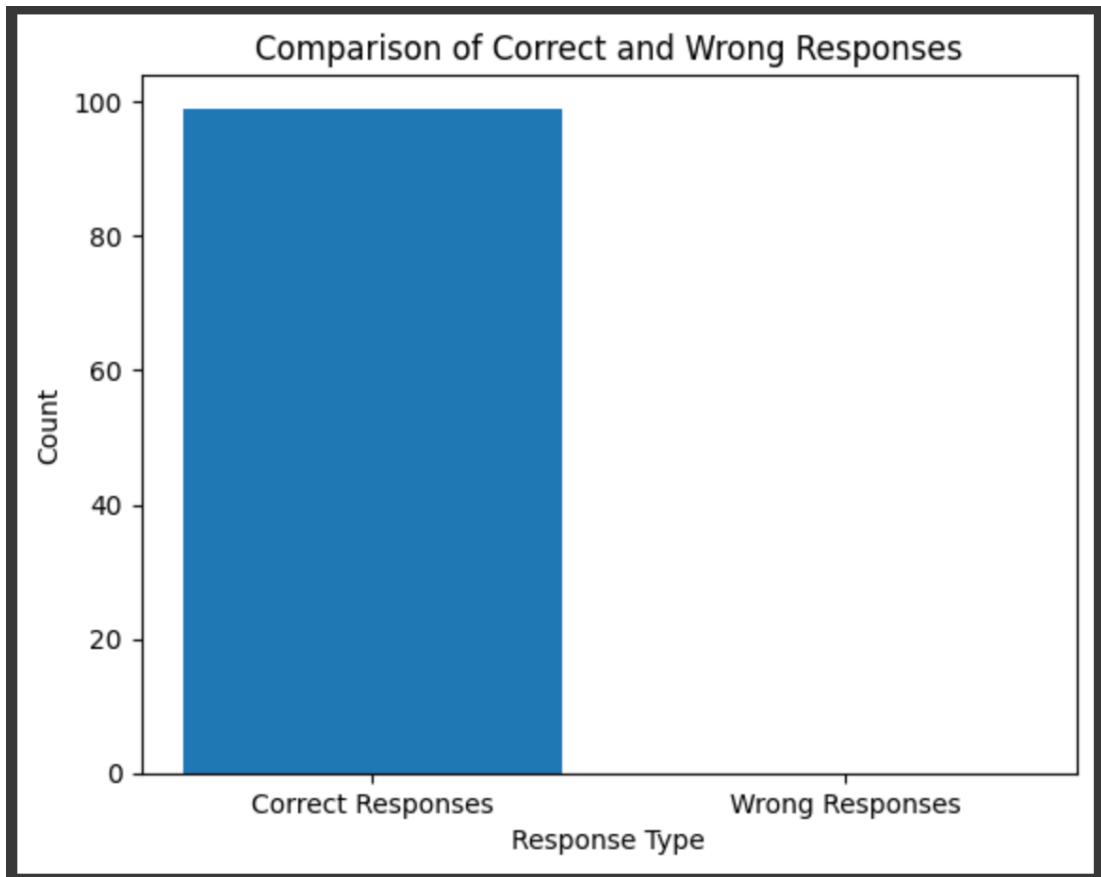


Fig 7.5: Comparison of Correct and Wrong Responses

```
def compare_sentences(sentence1, sentence2, threshold = 1.00):

    # Encode the sentences using Sentence Transformer
    embedding1 = model.encode(sentence1, convert_to_tensor=True)
    embedding2 = model.encode(sentence2, convert_to_tensor=True)

    # Compute cosine similarity
    similarity = util.cos_sim(embedding1, embedding2).item()

    # Determine if the intent is correct or wrong
    if similarity >= threshold:
        return "Correct Intent"
    else:
        return "Wrong Intent"

# Load the CSV file
file_path = '/content/Questions_and_Intent_Possible_Data_Loss.csv'
data = pd.read_csv(file_path)

# Initialize counters
correct_intent_count = 0
wrong_intent_count = 0
```

```

# Compare sentences in the two specified columns for each row

for index, row in data.iterrows():

    response = compare_sentences(str(row['Known_Intent']), str(row['Intent'])) # Replace
    'Response' and 'Response_Expected' with your actual column names

    if response == "Correct Intent":

        correct_intent_count += 1

    else:

        wrong_intent_count += 1


# Plot the results

labels = ['Correct Intent', 'Wrong Intent']

counts = [correct_intent_count, wrong_intent_count]

plt.bar(labels, counts)

plt.xlabel('Intent Type')

plt.ylabel('Count')

plt.title('Comparison of Correct and Wrong Intents')

plt.show()

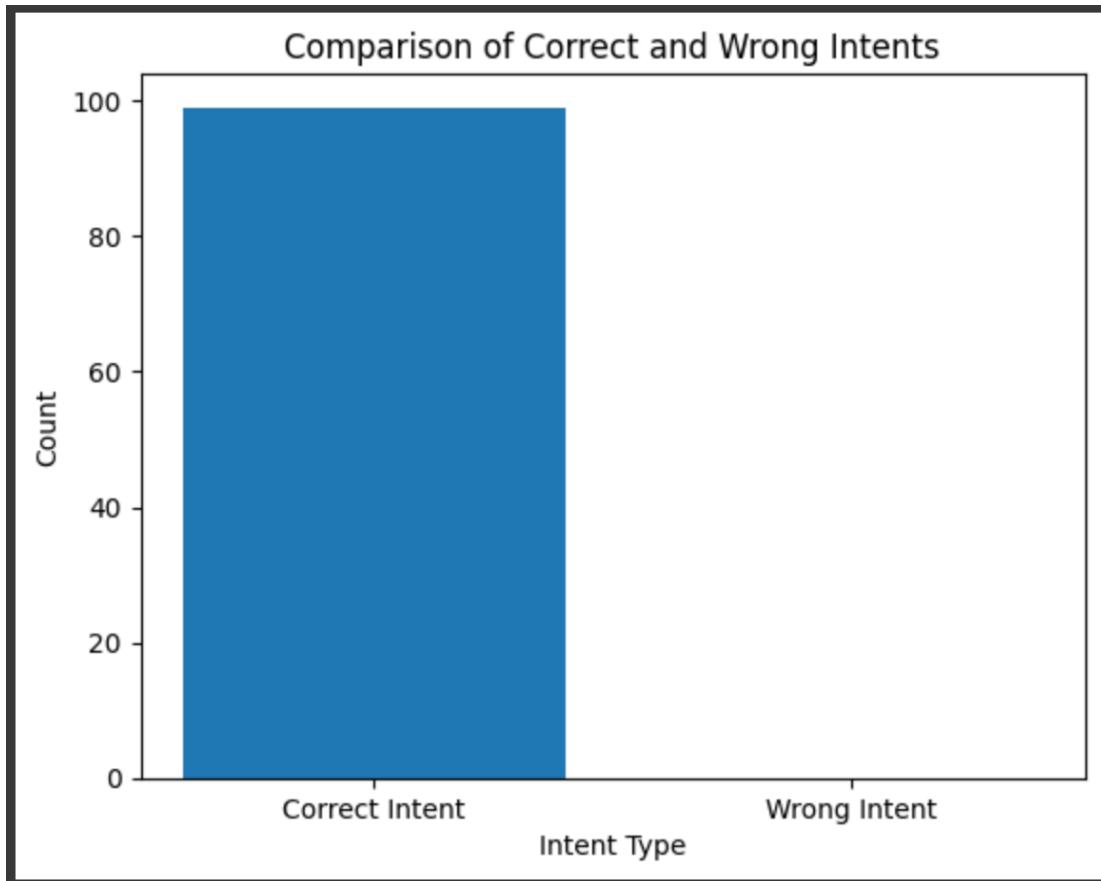
print('*'*59)

print(f"Correct Intent: {correct_intent_count}")

print('*'*59)

```

```
print(f"Wrong Intent: {wrong_intent_count}")  
print('*'*59)
```



*Fig 7.6: Comparison of Correct and Wrong Intents*

```
import pandas as pd  
  
import matplotlib.pyplot as plt  
  
  
def check_fallback_rate(fallback_value):  
    # If the fallback rate is False, the model is working fine  
    if not fallback_value:  
        return "Model Working Fine"
```

```

else:
    return "Wrong Response"

# Load the CSV file

file_path = '/content/Questions_and_Intent_Possible_Data_Loss.csv'
data = pd.read_csv(file_path)

# Initialize counters

model_working_fine_count = 0
wrong_response_count = 0

# Check fallback rates for each row

for index, row in data.iterrows():

    # Check fallback rate

    fallback_response = checkFallbackRate(row['Fallback_rate']) # Replace 'Fallback_Rate' with
    your actual column name for fallback

    if fallback_response == "Model Working Fine":
        model_working_fine_count += 1
    else:
        wrong_response_count += 1

# Plot the fallback rate results

labels_fallback = ['Model Working Fine', 'Wrong Fall Back']

```

```
counts_fallback = [model_working_fine_count, wrong_response_count]
```

```
plt.bar(labels_fallback, counts_fallback)
```

```
plt.xlabel('Fallback Rate')
```

```
plt.ylabel('Count')
```

```
plt.title('Fallback Rate Comparison')
```

```
plt.show()
```

```
print('*'*59)
```

```
print(f"Model Working Fine: {model_working_fine_count}")
```

```
print('*'*59)
```

```
print(f"Wrong Fall Back: {wrong_response_count}")
```

```
print('*'*59)
```

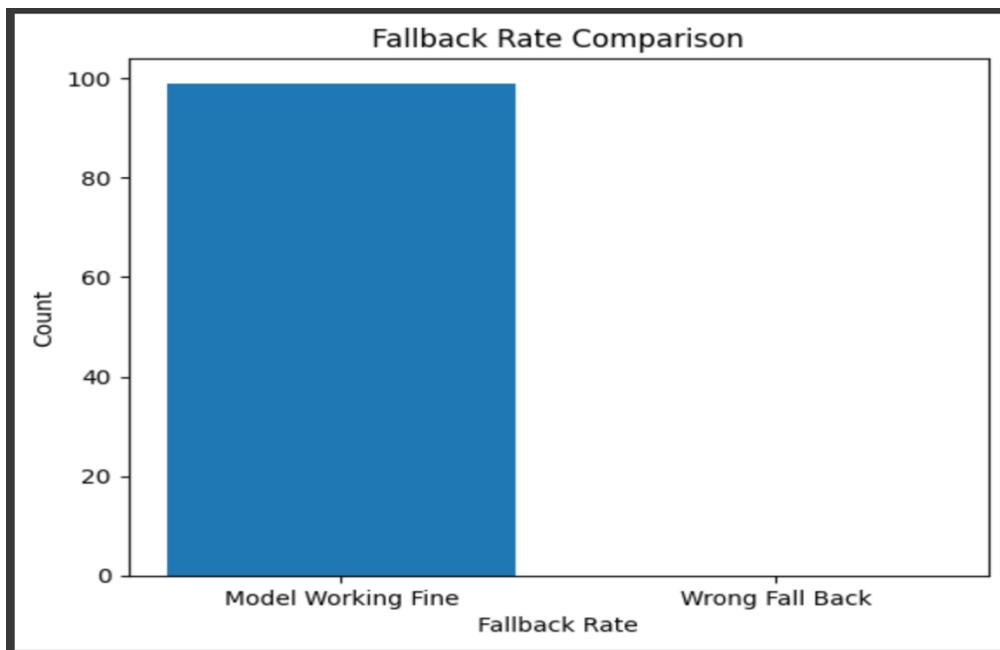


Fig 7.7 Fallback Rate Comparison

```

# Calculate the average latency

average_latency = data['webhook_latency'].mean() # Replace 'webhook_latency' with the actual
column name if different

print('*59)

print(f"Average Latency: {average_latency} ms")

print('*59)

# Group by intent and calculate the average webhook_latency

average_latency_by_intent = data.groupby('Intent')['webhook_latency'].mean() # Replace
'Intent' with the correct column name if different

# Plot the average latency for each intent

average_latency_by_intent.plot(kind='bar')

plt.xlabel('Intent')

plt.ylabel('Average Webhook Latency (ms)')

plt.title('Average Webhook Latency by Intent')

plt.xticks(rotation=45)

plt.tight_layout()

plt.show()

```

```
print(average_latency_by_intent)
```

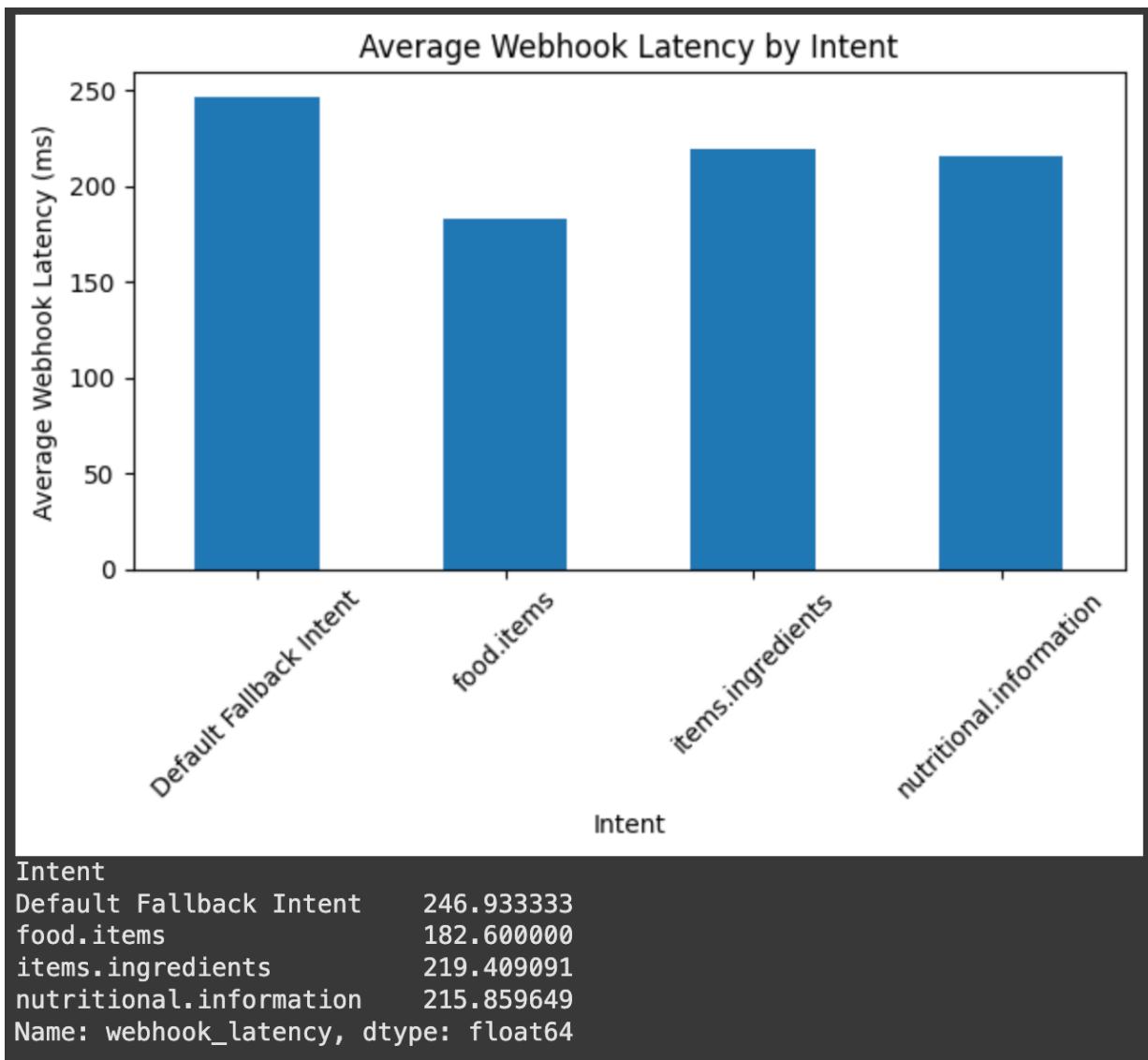
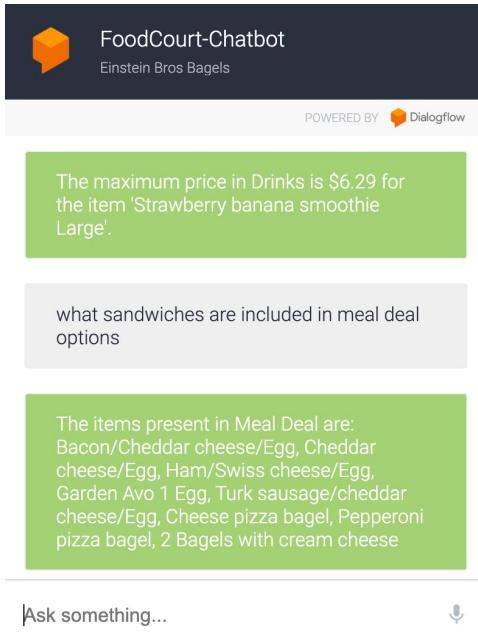
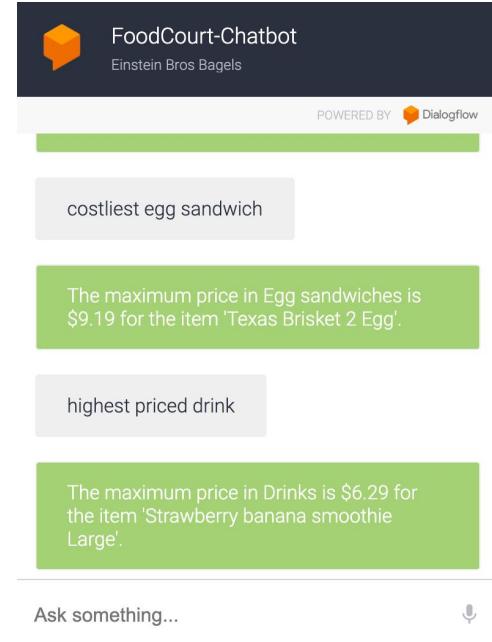


Fig 7.8 Average Webhook Latency by Intent

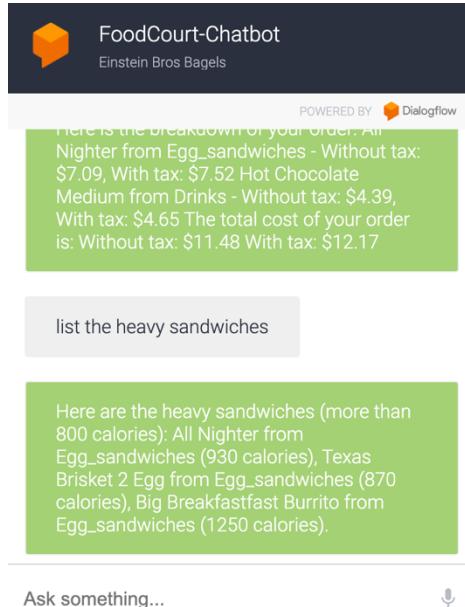
## VII.5 Examples of Chatbot Responses for User Queries.



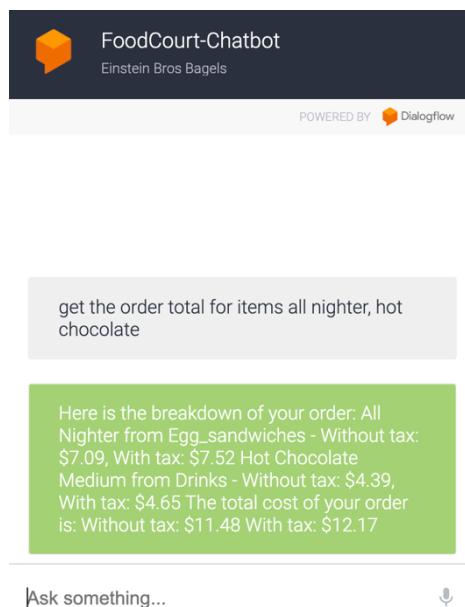
*Fig 7.9: Chatbot Response 1*



*Fig 7.10: Chatbot Response 2*



*Fig 7.11: Chatbot Response 3*



*Fig 7.12: Chatbot Response 4*

**FoodCourt-Chatbot**  
Einstein Bros Bagels  
POWERED BY Dialogflow

list the veggie sandwiches

Here are sandwiches with veggies and no meat: Cheesy veg melt from Lunch\_sandwiches, Cheese pizza bagel from Lunch\_sandwiches

list the sandwiches that have only veggies

Here are sandwiches with veggies and no meat: Cheesy veg melt from Lunch\_sandwiches, Cheese pizza bagel from Lunch\_sandwiches

|Ask something...|  
|Microphone|

**FoodCourt-Chatbot**  
Einstein Bros Bagels  
POWERED BY Dialogflow

drinks without milk

Here are drinks without milk: Cold brew Medium, Cold brew Large, Cold brew vanilla Medium, Cold brew vanilla Large, Cold brew caramel Medium, Cold brew caramel Large, Cold brew chocolate Medium, Cold brew chocolate Large, Cold brew hazlenut Medium, Cold brew hazlenut Large, Strawberry banana smoothie Medium, Strawberry banana smoothie Large, Tea hot Medium, Tea hot Large, Ice coffee Medium, Ice coffee Large, Coffee Regular Medium, Coffee Large

|Ask something...|  
|Microphone|

Fig 7.13: Chatbot Response 5

Fig 7.14: Chatbot Response 6

**FoodCourt-Chatbot**  
Einstein Bros Bagels  
POWERED BY Dialogflow

sandwiches below 8\$

Here are items in Egg\_sandwiches and Lunch\_sandwiches with a price below \$8.00: B/T/A Egg white from Egg\_sandwiches (\$6.88), Bacon/Cheddar cheese/Egg from Egg\_sandwiches (\$6.46), Cheddar cheese/Egg from Egg\_sandwiches (\$6.14), All Nighter from Egg\_sandwiches (\$7.52), Sante Fe Egg from Egg\_sandwiches (\$7.41), Sante Fe Egg wht from Egg\_sandwiches (\$7.41), Ham/Swiss cheese/Egg from Egg\_sandwiches (\$6.35), Garden Avocado from Egg\_sandwiches (\$7.41), Farmhouse Egg from Egg\_sandwiches (\$7.41), Turk sausage/cheddar cheese/Egg from Egg\_sandwiches (\$7.41)

|Ask something...|  
|Microphone|

**FoodCourt-Chatbot**  
Einstein Bros Bagels  
POWERED BY Dialogflow

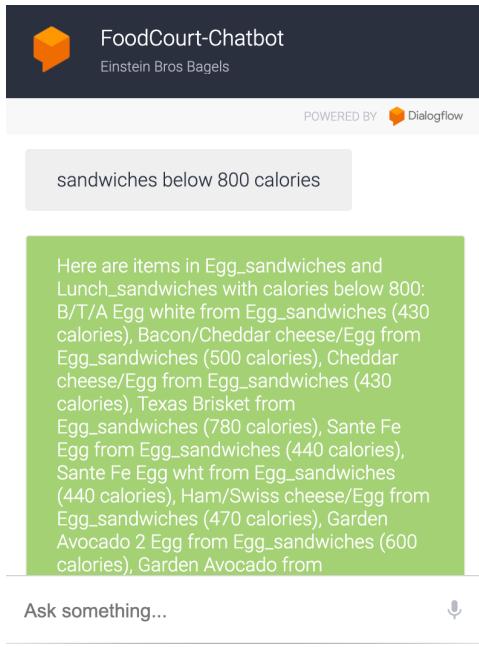
sandwiches without bacon and below 800 calories

Here are items without bacon and below 800 calories in Egg\_sandwiches and Lunch\_sandwiches: Cheddar cheese/Egg from Egg\_sandwiches, Texas Brisket from Egg\_sandwiches, Sante Fe Egg from Egg\_sandwiches, Sante Fe Egg wht from Egg\_sandwiches, Ham/Swiss cheese/Egg from Egg\_sandwiches, Garden Avocado 2 Egg from Egg\_sandwiches, Garden Avocado from Egg\_sandwiches, Turk sausage/cheddar cheese/Egg from Egg\_sandwiches, Avocado Toast 2 Egg from Egg\_sandwiches, Cheesy veg melt from Egg\_sandwiches

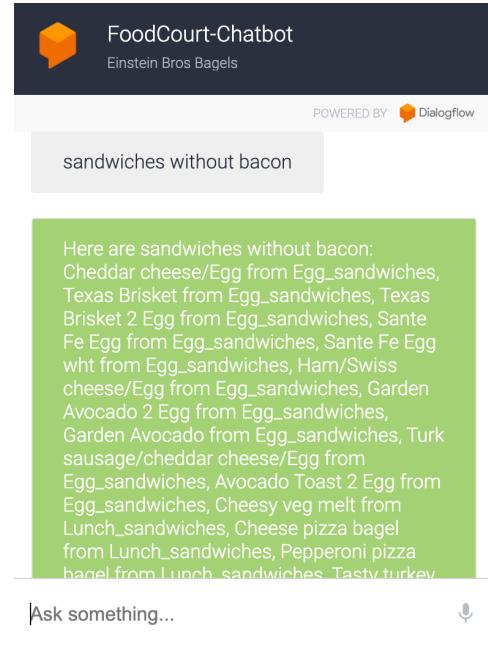
|Ask something...|  
|Microphone|

Fig 7.15: Chatbot Response 7

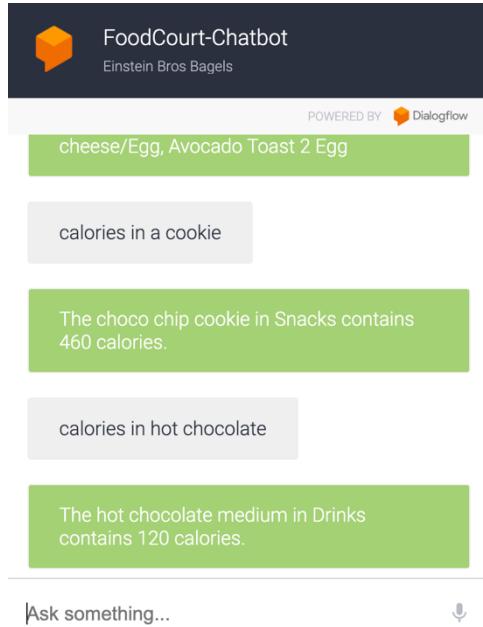
Fig 7.16: Chatbot Response 8



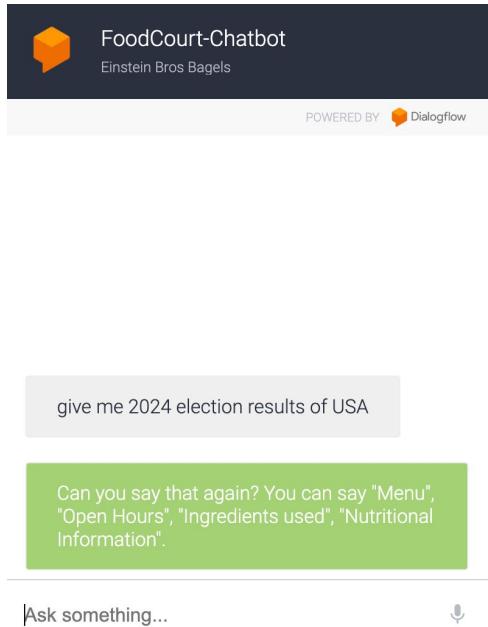
*Fig 7.17: Chatbot Response 9*



*Fig 7.18: Chatbot Response 10*



*Fig 7.19: Chatbot Response 11*



*Fig 7.20: Chatbot Response 12*