

# FoodieFriend: An NLP-based food order and delivery chatbot

SHRUTI SUNDARAM,

Department of Computer Science, University of Nottingham, shruti.sundaram.28@gmail.com

This paper introduces FoodieFriend, a food order and delivery chatbot developed with a keen focus on Conversational User Interface Design principles. This chatbot can interact with users, provide menu options, and handle food orders. It makes use of NLP and user-centric design principles. FoodieFriend offers users seamless interactions for order placement, utilizing prompt design, personalization, discoverability, context tracking, confirmation, and error handling strategies. The design of the chatbot is in accordance with Responsible Research and Innovation (RRI) principles, emphasizing ethical considerations, privacy, and inclusivity. Through rigorous performance and usability testing, FoodieFriend's effectiveness is confirmed, establishing it as a conscientious and valuable tool for personalized and ethically sound conversational interactions.

CCS CONCEPTS • Human-centered computing → Conversational User Interface (CUI) design principles • Information systems → Chatbot architecture, Implementation strategies • Computing methodologies → Natural language processing, Intent matching • Social and professional topics → Responsible Research and Innovation (RRI) in AI applications • Computing methodologies → User testing, Usability testing, Performance testing

**Additional Keywords and Phrases:** Task-oriented chatbots, Conversational design principles, Identity management in chatbots, User-centric design, Responsible AI/ACM

## Reference Format:

First Author's Name, Initials, and Last Name, Second Author's Name, Initials, and Last Name, and Third Author's Name, Initials, and Last Name. 2018. The Title of the Paper: ACM Conference Proceedings Manuscript Submission Template: This is the subtitle of the paper, this document both explains and embodies the submission format for authors using Word.

## 1 INTRODUCTION

The FoodieFriend chatbot designed to digitalize the food ordering and delivery experience. Harnessing the power of Natural Language Processing (NLP), FoodieFriend serves as an intelligent intermediary between users and culinary services, offering a seamless and intuitive interface for choosing restaurants, and placing orders. Users can use FoodieFriend to initiate various actions, such as comparing menus, comparing prices, getting descriptions of various dishes. The chatbot is adept at managing small talk as well.

## 2 CHATBOT ARCHITECTURE

### 2.1 System Overview and Functionalities

The FoodieFriend chatbot incorporates various functionalities, each tailored to address specific intentions and transactions. *Figure 1* illustrates the flowchart of the chatbot architecture.

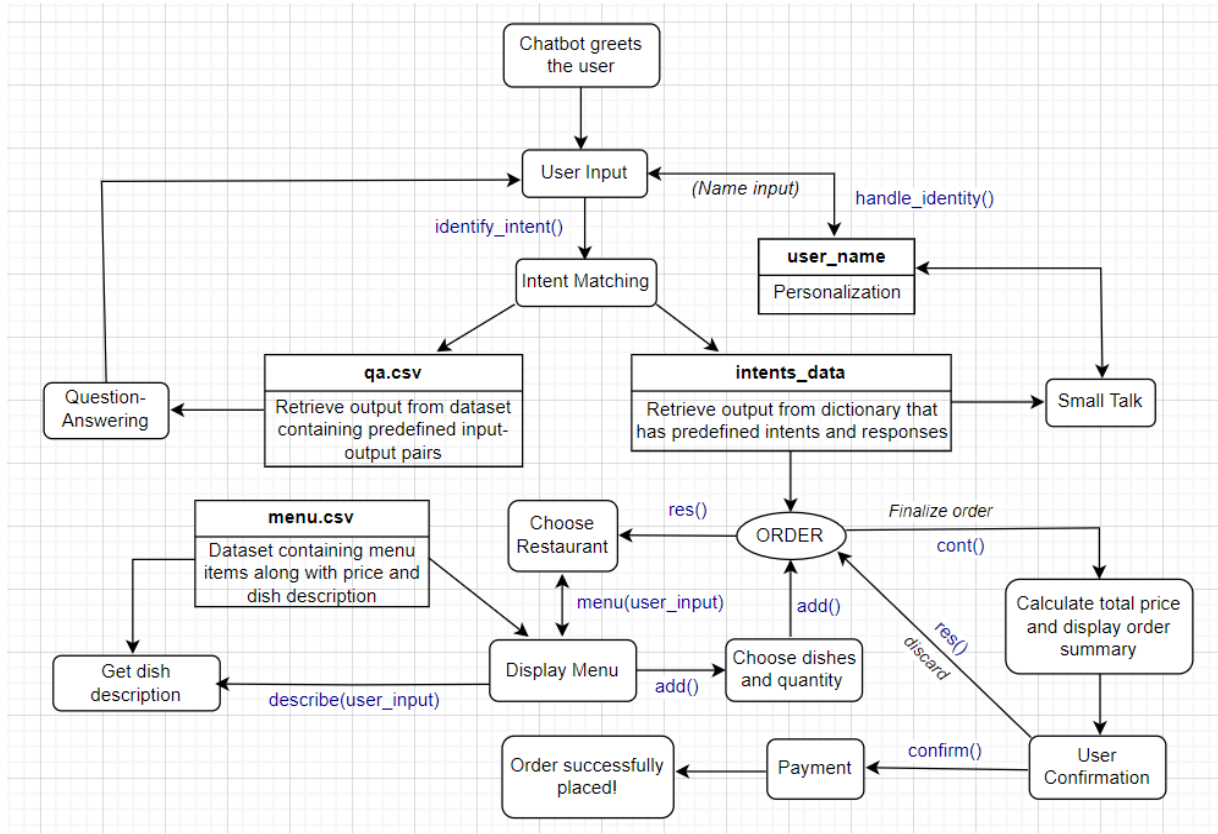


Figure 1: Flowchart for chatbot architecture.

#### Steps involved:

- The chatbot initiates the interaction by extending a greeting to the user and soliciting information regarding their name.
- The user's name is extracted through the execution of a function, and the obtained name is stored in a variable for subsequent retrievals as required.
- An array of predefined intents and corresponding responses is instantiated. A dataset comprising question-answer pairs is leveraged to generate responses to user inputs, employing intent identification and Jaccard similarity scores.
- User engagement may extend to casual conversation, with the option to place food delivery orders.
- Upon identification of the user's intent to place an order, an array of available restaurants is presented, allowing the user a selection opportunity.
- Diverse functions are invoked contingent upon the recognized intent, determined by the user's input.

- vii. Following the user's selection of a restaurant, pertinent details such as menu items, dish descriptions, and prices are presented, sourced from the associated menu dataset.
- viii. The user may access detailed descriptions of individual dishes by complying with the chatbot's prompts, facilitating a comprehensive exploration of menu offerings.
- ix. The user can proceed to the ordering stage by selecting the dishes along with their respective quantities.
- x. After finalizing the order, the chatbot systematically presents a comprehensive order summary inclusive of individual items selected and the cumulative cost.
- xi. Upon user confirmation, the chatbot transitions to the payment phase.
- xii. Upon successful completion of the payment process, entailing the input of pertinent information such as a valid card number and delivery address, the order is successfully placed, concluding the user's transactional interaction with the chatbot.

## 2.2 Implementation

The implementation of the FoodieFriend chatbot encompasses various tasks and functionalities.

### 1. Intent matching

Predefined intents and their associated phrases are stored in a dictionary. Jaccard similarity is computed between the user's cleaned input and the phrases associated with each intent. The *identify\_intent* function (refer *Figure 2*) iterates through the predefined intents, calculates Jaccard similarity for each, and selects the intent with the highest similarity. The identified intent is then used to determine the appropriate response or action.

```
# Function to identify user intent
def identify_intent(user_input, intents_data):
    max_similarity = 0
    identified_intent = "default"

    for intent, phrases in intents_data.items():
        similarity = calculate_jaccard_similarity(user_input, phrases)
        #print(user_input, ":", phrases, ":", similarity)
        if similarity > max_similarity:
            max_similarity = similarity
            identified_intent = intent
    #print(max_similarity)
    if max_similarity < 0.125:
        identified_intent = "qa"

    return identified_intent
```

Figure 2: Code snippet of Jaccard similarity computation for intent identification

### 2. Identity management

The *user\_identities* dictionary is initialized to manage user identities. This dictionary is intended to store information about user identities for future interactions. The *handle\_identity* function takes the user's input (*user\_input*) and performs POS tagging using NLTK. The user's input is tokenized into individual words, and POS tagging is applied to those tokens, assigning a grammatical label (tag) to each word. The function then iterates through the POS-tagged tokens and identifies potential names by looking for proper nouns (NNP), common nouns (NN), and adjectives (JJ) that are not explicitly excluded (e.g., 'name', 'T', 'I'). If a suitable name is found, it is assigned to the variable *user\_name*. *Figure 3* and *4* illustrates the code block used for handling identity.

```

# Function to handle user identity using POS tagging
def handle_identity(user_input):
    tokens = word_tokenize(user_input)
    pos_tags = pos_tag(tokens)
    user_name = None
    for token, tag in pos_tags:
        #Check for proper and common nouns
        if tag in ['NNP', 'NN', 'JJ'] and token not in ['name', 'Name', 'I', 'i']:
            user_name = token
            break
    return user_name

```

Figure 3: Code snippet of tokenization and POS tagging to extract user name.

```

elif intent == "identity" or is_greet == True:
    user_name = handle_identity(user_input)
    if user_name:
        user_identities["user"] = user_name
        print(f"Chatbot: Nice to meet you, {user_name}! I'm your food order and delivery assistant.")
    else:
        print("Chatbot: I didn't catch your name. Can you please provide it?")
    is_greet = False

elif intent == "retrieval":
    print("Chatbot: " + user_name)

```

Figure 4: Code snippet of acquiring and storing the user's name for subsequent retrieval

### 3. Information retrieval and question answering

A dataset containing question-answer pairs is pre-processed and stored in a dictionary. If the identified intent is "qa" based on Jaccard Similarity measure, then the code performs question answering by comparing the user's input with predefined input-output pairs stored in the dictionary, and printing the output accordingly (refer Figure 5 for the backend code).

```

#Question-answering
elif intent == "qa":
    max_similarity=0
    answer=''
    for inp, output in inp_out.items():
        cleaned_user_input = clean_text(user_input)
        cleaned_input = clean_text(inp)
        similarity = calculate_jaccard_similarity(cleaned_user_input, cleaned_input.split())
        if similarity > max_similarity:
            max_similarity = similarity
            answer = "Chatbot: "+output
    if max_similarity == 0:
        print("Chatbot: Sorry I can't understand.")
    print(answer)
else:
    print("Chatbot: I'm not sure how to respond to that. Can you please provide more information?")

```

Figure 5: Code snippet of question-answering based on similarity score

### 4. Small talk

A list of pre-defined intents and responses is created (refer Figure 6). A function involving keyword matching is used to identify intent (refer Figure 7). Once the intent is identified, the appropriate response associated with that intent is retrieved.

```

# List of predefined intents and responses
intents_data = {
    "greeting": ["hello", "hi", "hey", "greetings"],
    "farewell": ["bye", "goodbye", "see you later", "take care"],
    "identity": ["name", "am", "my name is", "I am", "what's", "your", "name"],
    "thanks": ["thank you", "thanks", "appreciate it", "appreciate", "thank"],

```

Figure 6: Code snippet of intent dictionary

```

# Process user input based on identified intent
if intent == "greeting":
    print("Chatbot: Hello! What is your name?")
    is_greet = True

elif intent == "farewell":
    print("Chatbot: Goodbye! Have a great day.")

elif intent == "thanks":
    print("Chatbot: You're welcome! If you have more questions, feel free to ask.")

```

Figure 7: Keyword matching

## 5. Confirmation

The `cont()` function is called (refer Figure 8) when the user wants to continue with the order, and the total cost of the order based on the selected dishes and quantities is calculated. It displays an order summary, including the items selected, their prices, and the total bill. The function prompts the user to type 'confirm' to proceed to payment or 'discard' to cancel the order.

```

#Function to continue and calculate price
cost=0
price_lst=[]
def cont():
    for i in dish:
        filtered_df = df[df['dish'] == i.lower()]
        price = list(filtered_df['price'])
        #print(price)
        currency_str= str(price[0])
        currency=int(currency_str[1:])
        price_lst.append(int(currency))
    print("Chatbot: Here is your Order Summary ~\n")
    for element1, element2, element3 in zip(dish, price_lst, quantity):
        print("\t\tf{element1} ({element2}) : {element3}")
    result1 = [a * b for a, b in zip(price_lst, quantity)]
    result2 = sum(result1)
    print("\n\tTotal Bill is $" +str(result2))
    print("\nPlease type 'confirm' to proceed to payment OR type 'discard' to discard this order.")

```

Figure 8: Function to confirm order and calculate bill.

## 6. Transaction

The `confirm()` function is called (refer Figure 9) when the user confirms the order and proceeds to payment. It prompts the user to enter their card details and address for delivery. The card details and address are appended to the card and address lists, respectively. If the entries are valid, it prints a confirmation message that the order has been successfully placed, and if not, it prompts the user to enter the details again.

```

# Confirm payment
card=[]
address=[]
def confirm():
    global error_count
    card_input= input("Chatbot: Please enter your card details[enter 16 digit code]: ")
    address_input=input("Chatbot: Please enter address for delivery[block,street,area]: ")

    if len(str(card_input).replace(" ", "").strip()) <16 and len(list(address_input.strip().split(' '))) <=3:
        print("The entries are not valid.\n")
        error_count+=1
        print("Please type 'confirm' to try again.")
        print("OR type 'back' to discard the order.")
    else:
        print("\n\tYour order has been successfully placed. Thank You!")

```

Figure 9: Function to confirm payment and validate user inputs.

## 2.3 Justification

This chatbot system aims to create a user-friendly and conversational experience for users interacting with a food order and delivery assistant. Some justifications for the design choices and implementation:

- **Natural Language Processing (NLP) and Intent Matching:**

The chatbot incorporates intent matching using Jaccard similarity, a basic but effective NLP technique. This allows the chatbot to understand user intents and respond accordingly. The code includes predefined intents and responses, covering a range of user interactions from greetings to order confirmation.

- **Identity Management and Personalization:**

The chatbot captures user identity during the conversation, enhancing the user experience by addressing the user by name and personalizing responses. This feature contributes to a more engaging and human-like interaction, making the user feel recognized and valued.

- **Use of Jaccard Similarity:**

Jaccard similarity is chosen for its flexibility, simplicity, and suitability for short, informal text inputs in a conversational setting. It allows the chatbot to effectively identify the user's intent even when the phrasing is not an exact match to predefined phrases.

- **Order Placement and Confirmation:**

The chatbot supports users in placing food orders, providing a structured process for selecting dishes, confirming the order, and entering payment details. The confirmation process ensures that users have the opportunity to review their order before proceeding, enhancing user satisfaction.

- **Conversational Design and Question-Answering Capability:**

The code follows conversational design principles by incorporating small talk elements, acknowledging user inputs, and responding in a conversational tone. Conversational markers and personalized prompts contribute to a more engaging and natural user experience. The question-answering feature allows users to ask general questions. This adds versatility to the chatbot, making it more capable of handling a variety of user inputs.

- **Modularity and Readability:**

The code is modular, with functions handling specific tasks such as menu display, order placement, and confirmation. This promotes code readability and maintainability.

- **Context Tracking and Discoverability:**

The chatbot maintains context throughout the conversation, adapting its responses based on user input and past interactions. Conversational markers and personalized prompts contribute to a more engaging and natural user experience. The code includes clear prompts, contextual help, and guidance messages to enhance discoverability and help users understand the available functionality.

- **User Exit Feature:**

The chatbot includes an exit command ("exit") for users to end the conversation at any point, providing a clear and intuitive way to terminate the interaction.

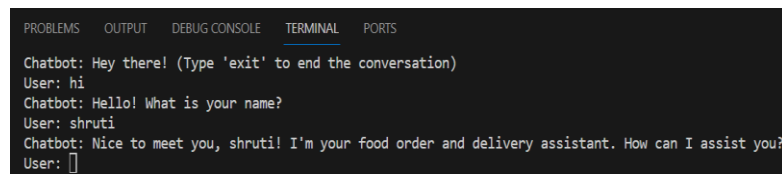
### 3 CONVERSATIONAL DESIGN

#### 3.1 Prompt Design:

In the formulation of all prompts employed within the system, several crucial considerations were meticulously addressed:

- **Greetings and introduction**

The chatbot integrates starts with a friendly greeting. It introduces itself and prompts the user to type 'exit' to end the conversation. It allows personalization by storing and using the user's name in subsequent interactions. *Figure 10* illustrates the prompt.



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
Chatbot: Hey there! (Type 'exit' to end the conversation)
User: hi
Chatbot: Hello! What is your name?
User: shruti
Chatbot: Nice to meet you, shruti! I'm your food order and delivery assistant. How can I assist you?
User: 
```

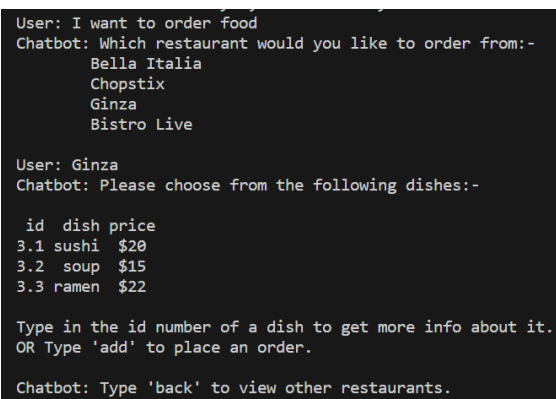
*Figure 10: Greeting prompt*

- **User input and processing**

The main loop continually prompts the user for input. The user's input is processed, and the chatbot responds accordingly based on the identified intent.

- **Information prompts**

Prompts are used to display information to the user, such as the menu, order summary, and payment details. Examples include displaying the menu items and their prices, showing the order summary, and requesting card details and delivery address (refer *Figure 11*).



```
User: I want to order food
Chatbot: Which restaurant would you like to order from:-
        Bella Italia
        Chopstix
        Ginza
        Bistro Live

User: Ginza
Chatbot: Please choose from the following dishes:-

  id  dish  price
3.1  sushi  $20
3.2  soup   $15
3.3  ramen  $22

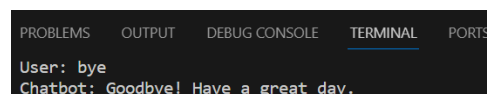
Type in the id number of a dish to get more info about it.
OR Type 'add' to place an order.

Chatbot: Type 'back' to view other restaurants.
```

*Figure 11: Information prompt for viewing menu items.*

- **Farewell**

The chatbot provides prompts for farewell. The farewell message is displayed when the user types 'exit' (refer *Figure 12*).



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
User: bye
Chatbot: Goodbye! Have a great day.
```

*Figure 12: Farewell prompt*

### 3.2 Discoverability:

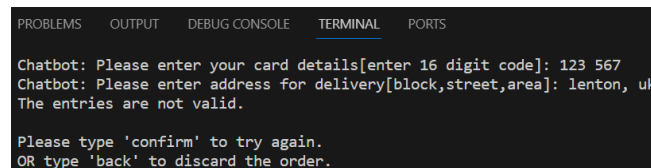
The chatbot suggests specific commands for the user to input in certain situations. For example - "Type 'add' to add more dishes, or type 'continue' to proceed to your payment" (Suggesting commands for adding more dishes or continuing to payment). By incorporating command suggestions and error handling guidance, the code aims to enhance discoverability by providing clear instructions, guidance, and feedback to the user throughout the interaction. This ensures that users can easily understand the available features and how to interact with the chatbot.

### 3.3 Error Handling:

The chatbot provides guidance in case of errors, helping the user understand what went wrong and how to proceed. It is needed to manage certain situations where unexpected input or issues may arise. Some of the error handling aspects in the code include:

- **Invalid entries during confirmation**

In the `confirm()` function, the code checks if the user inputs for the card and address is valid. If not, the code prints an error message (refer Figure 13). It then recursively calls the `confirm()` function, prompting the user to re-enter their card details and address.



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Chatbot: Please enter your card details[enter 16 digit code]: 123 567
Chatbot: Please enter address for delivery[block,street,area]: lenton, uk
The entries are not valid.

Please type 'confirm' to try again.
OR type 'back' to discard the order.
```

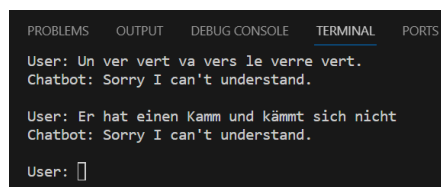
Figure 13: Error handling during transaction

- **Invalid Input Handling in QA (Question-Answering)**

In the QA part of the code, if the calculated Jaccard similarity is below a certain threshold ( $\text{max\_similarity} < 0.125$ ), the identified intent is set to "qa". This is done to handle cases where the user's input doesn't closely match any predefined intents. The code then provides a response indicating that it couldn't understand the input in case there is no match.

- **General Response for Unrecognized Input**

In the main loop, if none of the predefined intents match the user's input, the chatbot provides a generic response (refer Figure 14). This serves as a catch-all response when the chatbot cannot identify the user's intent based on the predefined categories.



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
User: Un ver vert va vers le verre vert.
Chatbot: Sorry I can't understand.

User: Er hat einen Kamm und kämmt sich nicht
Chatbot: Sorry I can't understand.

User: []
```

Figure 14: Default response for unrecognized input

### 3.4 Personalization:

The incorporation of personalization involves greeting users with their names and acknowledging them by name.

### 3.5 Confirmation:

The confirmation process involves presenting the user with a summary of their order, asking for confirmation, and collecting necessary information for order fulfilment. The chatbot ensures that the user's entries are valid before confirming the order. This process provides a structured and user-friendly way for the user to confirm their actions and proceed with the transaction.



- **Confirmation Prompt**  
After the user has added dishes to their order, the *cont()* function is called, which provides a summary of the order and asks the user to confirm.
- **Order Summary and Confirmation**  
The *cont()* function calculates the total cost of the order and prints a summary, asking the user to confirm by typing 'confirm' or discard by typing 'discard'.
- **Payment and Address Collection**  
The *confirm()* function prompts the user to enter their card details and delivery address. The code checks the card and address details to ensure they are valid.
- **Order Confirmation Message**  
If the card details and address are valid, the chatbot confirms the successful placement of the order.

### 3.6 Context tracking:

The chatbot maintains context throughout the conversation, adapting its responses and actions based on information gathered during earlier interactions.

- **User Identity Tracking**  
The code tracks user identity throughout the conversation. When the chatbot identifies the intent as "identity" or when the user provides their name, the user's name is stored in the *user\_identities* dictionary. The stored user identity is later used in responses to personalize the interaction. For example, when confirming the user's name, the chatbot responds with a personalized message.
- **Menu Interaction Context**  
When the user interacts with the menu, the context of the current restaurant selection is maintained. For example, when the user selects a restaurant, the *menu()* function is called with the restaurant name as an argument, and the function displays the menu for that specific restaurant.
- **Order Context**  
The code keeps track of the dishes and quantities selected by the user for placing an order. The *add()* function is called to add a dish and quantity to the order. The *cont()* function calculates the total cost based on the selected dishes and quantities, providing a context for the user during the payment process.

## 4 EVALUATION

User Testing was conducted among 15 people. The users are required to test the chatbot with the following tasks:

**Task 1:** Performance Testing: Use the chatbot for at least 5 transactions.

Transactions to try:

1. Place an order
2. Discard an order
3. Proceed to payment
4. Discard payment
5. Do a small talk, e.g. What can you do?

**Task 2:** Usability testing: Record your Feedback here. There are 6 questions. All questions are mandatory.

#### 4.1 Performance Testing

Performance testing encompasses the real-time recording of both user errors and successful task completions, subsequently calculating the resulting error rate and exporting the data to a CSV file

$$\text{Error\_rate} = (\text{User\_Errors\_Count} / \text{Task\_Completion\_Count}) * 100$$

Figure 15 illustrates the results of test sessions with 15 users. The feedback was used to improve the chatbot functionalities.

Error count	Success count	Error rate
0	0	0
6	3	200
5	2	250
4	2	200
0	3	0
1	4	25
1	4	25
1	4	25
0	3	0
0	3	0
0	3	0
0	3	0

Figure 15: Performance Testing results

After each round of test execution, the chatbot was optimized. The above results clearly depict the error rate (0-100) has significantly come down as the round progresses.

#### 4.2 Usability Testing

Evaluation encompasses user testing through a [Google Form](#) questionnaire consisting of 6 questions from the Chatbot Usability Questionnaire (CUQ). 3 rounds of testing with 15 users each were conducted. Figure 16 illustrates the results for the usability testing.

CUQ Questions	Answer - Yes	Answer - Maybe	Answer - No
The Chatbot is friendly and engaging	100%	0%	0%
The Chatbot explained its scope and purpose effectively	82%	12%	0%
The Chatbot was easy to navigate	100%	0%	0%
The Chatbot could understand me fairly easily	94%	6%	0%
I could understand the Chatbot with ease	100%	0%	0%
The Chatbot handled my errors well	82.4%	11.8%	6%

Figure 16: Results for Usability Testing

Conclusion from the usability test results:

- Users are content with the chatbot's friendly prompts and conversational ability.
- Users are highly satisfied with the transactional processes and easily comprehensible instructions.
- The chatbot's proficiency in error handling received mostly positive feedback.

After each round of test execution, the chatbot was optimized. The above results clearly depict the error rate (0-100) has significantly come down as the round progresses.

## 5 CONCLUSION

### 5.1 Ethical thinking

The chatbot prompts and the dataset don't contain any words that may convey racism, sexism, or bias. Gender-neutral pronouns are employed throughout the process.

### 5.2 Privacy

User input is only stored during runtime and reverts to a null state upon the restart of the chatbot. Performance metrics data is exclusively retained during user testing, with explicit disclosure provided in the description accompanying the testing phase.

### 5.3 Reflection on possible issues

- The chatbot is accessible exclusively to individuals with computer access.
- Lack of multilingual support to enable users to interact with the chatbot in their preferred language.
- Voice Recognition not incorporated to accommodate users who may prefer or require voice-based interactions.

#### **Future work may include:**

- Allowing users to set preferences and store order history.
- Integrate with external services for additional functionalities, such as wait time, distance between restaurant and delivery address etc.
- Implement user segmentation based on preferences, demographics, or behavior. This allows for targeted interactions and personalized recommendations.
- Ensure the chatbot is accessible to users with disabilities.