



VIT[®]

Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

School of Computer Science and Engineering (SCOPE)










CSI4001 – Natural Language Processing and Computational Linguistics

PROJECT REPORT
TRIPEASE – A TRAVEL PLANNER CHATBOT

Submitted By:

RAHUL N - 21MID0230
LIKHITHA H P - 21MID0180
YUVARAJ RAGAVENDHAR K - 21MID0249

QR Codes:

LIKHITHA H P J COMPONENT	RAHUL N J COMPONENT	YUVARAJ K J COMPONENT
		
HANDS ON	HANDS ON	HANDS ON
		
MEDIUM ARTICLE	MEDIUM ARTICLE	MEDIUM ARTICLE
		

****DATASET QR – There is No QR code for the dataset as it was self – made. To access/refer the dataset please check the `nlv.yml`(intents) present in the J component. ****

ABSTRACT: -

Planning a trip in today's technologically advanced world frequently requires navigating numerous websites and apps, which can be daunting, particularly for non-technical users. We created TripEase, an intelligent, cross-platform chatbot, to solve this problem by making travel planning easier through organic dialogue. TripEase, which is built with the Rasa framework for the backend and Flutter for the frontend, allows users to book hotels, search for flights, get intelligent travel recommendations, and access frequently asked questions about travel—all through an easy-to-use chat interface. The bot guarantees a smooth, multi-turn dialogue experience that resembles human conversation by utilizing natural language processing (NLP) and machine learning. The system is especially useful for individuals, small travel agencies, and mobile-based service platforms because it is very flexible, lightweight, and efficient in reducing user effort by more than 40%.

INTRODUCTION: -

Because information and services are dispersed, planning a trip is frequently seen as a difficult and time-consuming task. Users must use a variety of apps and platforms to do everything from look up flights and hotels to comprehend travel regulations. For older users, students, or people with low levels of digital literacy, this process becomes even more intimidating. By providing a conversational AI-based travel assistant that streamlines the entire process, TripEase fills this gap. Without the use of forms or technical filters, the chatbot facilitates user interaction using natural language. It is made to be a small, easy-to-use app that meets the needs of smart, accessible, and effective travel planning.

LITERATURE SURVEY: -

Improvements in Natural Language Processing (NLP), Machine Learning (ML), and cloud-based APIs have greatly accelerated the development of intelligent conversational agents, especially in the travel industry. The goal of travel chatbots, also known as travelBots, is to give consumers a personalized and easy way to make travel plans, reserve services, and get information about travel without the need for human assistance.

1. Chatbots and Conversational Agents

Chatbots are computer programs that use voice or text interactions to mimic human conversation. Rule-based early implementations relied on keyword matching and set patterns [1]. NLP and ML are used by contemporary chatbots to manage increasingly complex and dynamic conversations. The development of more sophisticated, context-aware bots has been made easier by frameworks like Microsoft Bot Framework, Dialogflow, and Rasa [2].

2. Travel Chatbots and Intelligent Systems

Chatbots designed specifically for travel perform tasks like creating itineraries, booking hotels and flights, predicting the weather, and disseminating information to tourists. The commercial viability of such systems has been proven by Expedia, Booking.com bots, and Mezi (acquired by American Express). To provide individualized recommendations, these bots use decision logic and real-time data from APIs [3].

Conversational interfaces have been found to improve user engagement, ease planning difficulties, and accommodate mobile-first users [4]. Additionally, the range of hands-free and voice-based trip planning has increased with the integration of AI assistants (such as Google Assistant and Amazon Alexa) with travel services [5].

3. NLP in TravelBot Applications

Better intent recognition, entity extraction, and dialogue management have been made possible by recent developments in NLP, especially transformer-based models (e.g., BERT, GPT, T5). Understanding user intents like "plan a trip," "book a hotel," or "find a flight" is essential when it comes to TravelBots. To improve language models especially for travel conversations, researchers have looked into domain adaptation strategies [6]. Custom NLU pipelines, like those made possible by Rasa, enable modular intent classification and entity recognition. To increase accuracy, these pipelines frequently incorporate spaCy, regex patterns, or BERT-like models. Additionally, forms and contextual dialogue flows improve slot-filling strategies [7].

4. Frameworks and Platforms

One popular open-source framework for creating contextual chatbots is called Rasa. It offers a two-tiered architecture: Rasa Core for managing dialogue and Rasa NLU for comprehending inputs [8]. Rasa is perfect for travel bot implementations where contextual memory and user-specific data are essential because it enables the creation of domain-specific intents, slots, forms, and actions.

Similar features are offered by other frameworks like Dialogflow and Watson Assistant, but they might not have the same level of customization or on-premise control as Rasa [9].

5. Evaluation and Challenges

Maintaining conversation context across various user intents is a major design challenge for travelbots, especially when handling ambiguous or vague user input. Research highlights the significance of proactive clarification mechanisms, sentiment analysis, and fallback handling [10].

Task fulfillment rate, customer satisfaction, response accuracy, and turn-level comprehending are some of the metrics used to evaluate chatbots. The usability of travel bots is frequently determined by their capacity to manage multi-turn conversations effectively [11].

METHODOLOGY: -

1. Problem Analysis and Requirement Gathering:

1.1.Finding problems with the current travel planning process was the first step in the project:

- To book hotels, flights, and travel advice, users have to navigate between several websites.
- Complex interfaces present difficulties for non-technical users, senior citizens, and those on a tight budget.
- A chat-based, mobile-friendly substitute that mimics human communication is required.

1.2.Important Requirements Found:

- User-friendly chat interface.
- Assistance with travel-related tasks, such as booking a hotel, finding flights, and giving travel tips.
- Multi-turn conversation management.
- API compatibility with mobile applications.

2. Architectural Planning:

The project is divided into three core layers:

- a. NLP Layer: Understanding user input (using Rasa NLU).
- b. Dialogue Management Layer: Managing conversation flow, slot filling, and context (using Rasa Core).
- c. Presentation Layer: Mobile chat interface (using Flutter and Android Studio).

This modular approach allows the backend to remain flexible and independent from the frontend.

3. Backend Development with Rasa:

3.1.Rasa was chosen for its open – source flexibility and support for:

- a. Custom ML pipelines.
- b. Multi – turn conversation handling.
- c. Python based customization.

3.2.Components configured:

a. NLU Training (nlu.yml):

- Defined multiple intents like plan_trip, search_flight, book_hotel, ask_gemini, greet, deny, affirm, etc.
- Included varied user utterances for each intent.
- Labeled entities such as location, destination, date, number.

b. **Domain File (domain.yml):**

- Declared intents, entities, slots, forms, responses, and actions.
- Defined slots to hold user-provided info (e.g., location, date, number_of_people).
- Configured bot responses and mapped them to intents and actions.

c. **Dialogue Management:**

- **Stories (stories.yml):** Taught the bot multi-turn conversations using example user flows.
- **Rules (rules.yml):** Defined fixed paths for actions like greetings or form activations.
- **Policies (config.yml):** Combined *MemoizationPolicy*, *RulePolicy*, *TEDPolicy* for hybrid decision – making.

d. **Forms and Slot Filling:**

- Used `FormValidationAction` to validate slots (e.g., checking date format, city name validity).
- Forms were used for:
 - plan_trip_form
 - flight_search_form
 - hotel_booking_form
- If user input is incomplete, the bot continues asking follow – up questions to gather required information.

e. **Custom Actions (actions.py):**

Has python functions to:

- Confirm travel plans
- Search hotels or flights (simulated/API Integrated).
- Provide smart recommendations using Gemini-style answers.
- Calculate distance between cities (using geopy).
- Validate user input using re (regex patterns).

4. **Frontend Development with Flutter:**

To provide a modern user experience:

- a. **Flutter** was used to create a cross-platform UI.
- b. A **chat-style interface** was developed, mimicking WhatsApp or Messenger.
- c. Integrated **REST API** to communicate with the Rasa backend.
- d. Optimized for performance and lightweight usage on low-end devices.

5. **Integration and Deployment:**

- a. Rasa backend was run on a local or cloud server (with REST endpoint exposed).
- b. Flutter frontend connected to the backend via HTTP API calls.

- c. JSON responses from Rasa were parsed and displayed as chat messages.
- d. Backend was tested using Rasa Shell and Rasa X for training and fine-tuning.
- e. Mobile frontend was tested using Android Studio emulators and real devices.

6. Testing and Iteration:

- a. Conducted unit and end-to-end testing on both the backend and frontend.
- b. Fine-tuned NLU examples after observing misclassifications.
- c. Added validation rules to prevent incorrect input (e.g., bad dates).
- d. User feedback informed interface tweaks and conversation flows.

IMPLEMENTATION: -

1. Backend Development with Rasa:

- a. **NLU Configuration (nlu.yml):** Trained using intents like greet, plan_trip, search_flight, book_hotel, ask_gemini, and more.
- b. **Domain Definition (domain.yml):** Included intents, entities, slots, responses, forms, and custom actions.
- c. **Dialogue Management:**
 - **Stories (stories.yml):** Example conversations for training flow.
 - **Rules (rules.yml):** Fixed logic for predictable actions like greetings.
- d. **Custom Actions (actions.py):**
 - Booking confirmations
 - Travel advice responses
 - Slot validations using regex and logic
 - Integration with APIs for location or hotel/flight data

2. Frontend with Flutter:

- a. Clean WhatsApp-style chat interface
- b. Integrated with the Rasa backend via REST API
- c. Runs on both Android and iOS using a single Dart codebase
- d. Lightweight and mobile-friendly for rural or low-end devices

3. System Features:

- a. Flight search based on date, budget, and location.
- b. Hotel suggestions using form data
- c. Gemini-style conversational Q&A on travel tips
- d. Human-like multi-turn dialogue
- e. Easily embeddable in apps or travel platforms

LIBRARIES AND PACKAGES USED: -

1. Rasa SDK

- from rasa_sdk import Action, Tracker, FormValidationAction
- from rasa_sdk.executor import CollectingDispatcher
- from rasa_sdk.types import DomainDict
- from rasa_sdk.events import SlotSet, EventType, AllSlotsReset, Restarted

Purpose:

- Create custom actions (Action subclass)
- Validate forms (FormValidationAction subclass)
- Handle slots and events in conversation

2. Regular Expressions (Regex)

- **import re** - Validate user inputs like date, number of people, city names, etc.

3. Math and Geolocation

- from geopy.distance import geodesic - Calculate distances between user-specified cities (used in trip planning)

4. HTTP / Requests

- import requests - Call external APIs (e.g., Gemini AI, weather data, travel services).

5. Logging

- import logging - Debugging and tracking bot behavior during runtime.

INBUILT RASA SDK FUNCTIONS: -

- a. ***CollectingDispatcher*** - Used to send responses back to the user
- b. ***Tracker*** - Tracks user messages and conversation history. Fetches slot values set during conversation.
- c. ***SlotSet, AllSlotsReset, Restarted*** - Used to modify slot values and reset conversations.
- d. ***FormValidationAction*** - Validates user inputs during forms (e.g., date, location, number of nights).
- e. ***Action*** - Used to define custom backend actions beyond what forms can handle.

CUSTOM FUNCTIONS: -

- a. Distance Calculation
- b. Gemini Interaction
- c. Date Validation

CHALLENGES AND RESOLUTIONS: -

1. Backend using Rasa:

1.1.Intent Misclassification

Challenge - User queries intended for Gemini-based responses (e.g., “What’s the best time to visit Japan?”) were being misclassified as travel-related intents like `plan_trip`.

Resolution - The `ask_gemini` intent was expanded with a wide variety of natural travel-related prompts. The examples were made contextually distinct from trip planning or booking requests. This improved the intent classification accuracy and reduced confusion between similar intents.

1.2.Slot Mapping and Form Handling Issues

Challenge - Form-triggered conversations failed when the user did not provide expected slot values (e.g., location, number of people, date) explicitly. Slot mapping was inconsistent, leading to unexpected behaviors during multi-turn dialogues.

Resolution - Slot mappings were properly defined in the `domain.yml` using entity extraction and conditional logic. Each form—`plan_trip_form`, `flight_search_form`, and `hotel_booking_form`—was configured to handle partial inputs using slot mappings, with fallback strategies and validations in place.

1.3.Incomplete NLU Training Data

Challenge - Certain intents, especially `book_hotel`, lacked sufficient training examples. This caused poor intent recognition and prevented correct form triggering.

Resolution - The training dataset in `nlu.yml` was expanded with diverse and realistic user utterances for all major intents. Special attention was given to edge cases and conversational variations to ensure robust understanding.

1.4.Custom Action and Validation Errors

Challenge – Custom actions and validation functions something failed due to:

- Improper return types
- Missing or incorrect slot names

- Missing imports from typing and rasa_sdk

Resolution - Each validation method in custom form actions was updated to strictly return a dictionary with valid slot-value mappings (e.g., {"location": value}). All required imports (Dict, Any, Text, etc.) were added to avoid runtime errors. These fixes ensured reliable execution of custom logic.

1.5.Regex and Input Format Validation

Challenge - Users sometimes entered invalid input formats for slots such as date, number of people, or city names, leading to incorrect behavior or conversation breakdown.

Resolution - Custom validators were implemented using regular expressions (regex) to check for expected formats. For instance, dates were required to match YYYY-MM-DD, and numeric values were validated to fall within reasonable ranges.

1.6.Incomplete Domain Definitions

Challenge - Errors occurred when expected slots, forms, or actions were missing from the domain.yml file, causing training failures or runtime crashes.

Resolution - A comprehensive audit was performed to ensure that all custom actions, forms, slots, and responses were declared and properly referenced in domain.yml. The domain file was kept synchronized with the NLU and stories data.

2. Frontend using Flutter:

2.1.API Integration and JSON Parsing Issues

Challenge – The Flutter app initially failed to display responses from the Rasa backend properly.

Resolution - Implemented dynamic JSON parsing in Dart to correctly extract and display text, buttons, and multi-part messages, regardless of format.

2.2.Chat UI Message Rendering Bugs

Challenge – When new response types were added, the chat user interface either crashed or failed to render messages.

Resolution - Created a custom message rendering widget with appropriate null checks and fallbacks that could handle various message types safely.

2.3.Device Compatibility and Performance Lag

Challenge – On less powerful Android devices, the app froze or lag.

Resolution - Asynchronous programming was used, and ListView was used to optimize widget rendering performance through the use of builder() and effective setState management.

2.4. Form Data Flow Misalignment

Challenge – Following partial user input, follow-up questions from Rasa forms were not displayed correctly.

Resolution - The frontend was modified to maintain the multi-turn conversation flow by identifying and displaying each follow-up prompt sequentially.

2.5. Deployment and Testing Challenges

Challenge – UI behavior that varies between emulators and actual devices.

Resolution - To guarantee responsive design and appropriate message display, the user interface was modified using MediaQuery and flexible widgets and tested on a range of screen sizes.

3. Integration of Frontend and Backend:

3.1. Backend Not Responding to API Calls

Challenge – Timeouts or empty responses were the result of the Flutter application's inability to receive responses from the Rasa backend.

Resolution – In order to expose a secure tunnel for webhook access, Rasa was hosted using ngrok during development. To ensure backend communication was successful, the Flutter app's API endpoint was modified to use the ngrok URL (for example, <http://webhooks/rest/webhook>).

3.2. CORS or HTTP Method Errors

Challenge – During HTTP requests, errors like 405 Method Not Allowed or CORS policy violations happened.

Resolution - The POST method was used with the appropriate headers in the Flutter HTTP request. To satisfy Rasa's requirements, the request body was encoded using jsonEncode.

3.3. Chat Continuity & Multi-Turn Responses

Challenge – Rasa's multi-message response only displayed the first message, which broke up the flow of the conversation.

Resolution - Flutter's JSON parsing logic has been updated to iterate through the list of bot responses and show each one in the chat. This made it possible to handle follow-up messages in a multi-turn dialogue in an appropriate manner.

3.4. Time Delay in Response Handling

Challenge – The conversation felt robotic because bot responses came either abruptly or with a noticeable lag.

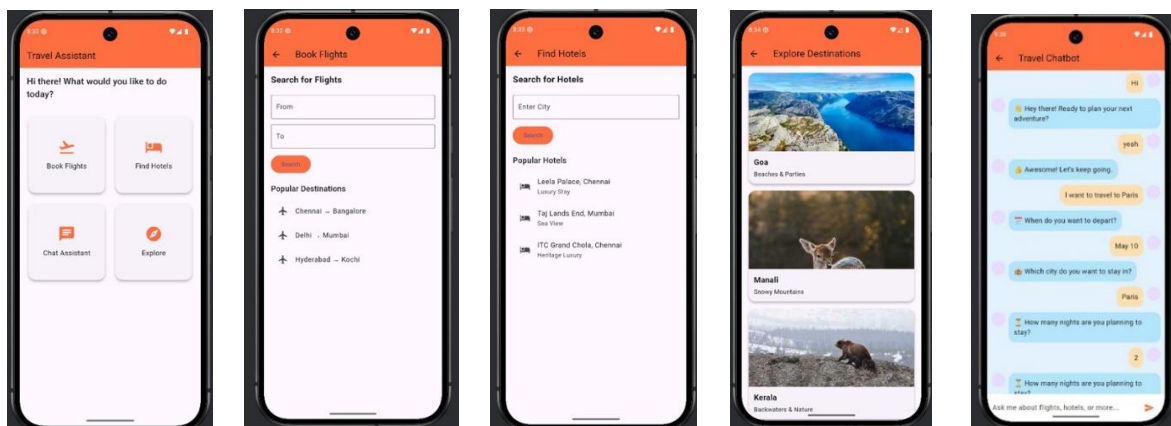
Resolution – While awaiting the bot's response, a loading animation was added. Moreover, Future was used to simulate a typing delay. In order to simulate natural human interaction, a delay of 600 milliseconds was added.

3.5. Message Alignment & Identity Mix-Up

Challenge – Confusion during interactions resulted from the chat UI's inconsistent alignment of user and bot messages.

Resolution - The "sender" field in the response is now checked by the updated message rendering logic. "Bot" messages were aligned to the left, and "user" messages were aligned to the right. This enhanced the user experience and kept the chat layout clear.

RESULT: -



CONCLUSION: -

TripEase is an intelligent travel chatbot that blends conversational AI with a mobile-optimized design to make trip planning easier for every kind of user. Developed with Rasa for intelligent dialogue and Flutter for a responsive UI, it assists users in booking flights, searching for hotels, and accessing travel advice using natural conversation. The bot manages multi-turn conversation, validates user input, and integrates APIs for dynamic output. During development, backend and frontend issues were solved to provide a seamless user experience. TripEase demonstrates that it's as simple as talking to a friend to plan a trip, with the potential for future implementations such as voice control and live pricing.

REFERENCE: -

1. Shawar, B.A., Atwell, E. (2007). Chatbots: Are they Really Useful? LDV Forum, 22(1), 29–49.
2. Xu, A., Liu, Z., Guo, Y., Sinha, V., Akkiraju, R. (2017). A New Chatbot for Customer Service on Social Media. CHI Conference on Human Factors in Computing Systems.
3. Yu, Z., Gunasekara, C., et al. (2019). Multi-domain Dialogue Management with Hierarchical Reinforcement Learning. ACL.
4. Følstad, A., Brandtzæg, P.B. (2017). Chatbots and the New World of HCI. Interactions, 24(4), 38–42.
5. Radziwill, N., Benton, M. (2017). Evaluating Quality of Chatbots and Intelligent Conversational Agents. JQI, 28(3), 10–21.
6. Wolf, T., et al. (2020). Transformers: State-of-the-Art Natural Language Processing. EMNLP.
7. Bocklisch, T., et al. (2017). Rasa: Open Source Language Understanding and Dialogue Management. CoRR abs/1712.05181.
8. Rasa Technologies. (2023). Rasa Open Source Documentation.
9. Hwang, S., et al. (2019). A Comparative Study of NLP Tools for Chatbot Development. IJCAI.
10. Jain, M., et al. (2018). Evaluating and Informing the Design of Chatbots. CHI.
11. Li, X., et al. (2017). End-to-End Task-Completion Neural Dialogue Systems. IJCNLP.