Project Report: Task-oriented Conversational Chatbot for Shipment assistance

---

By: Aditya Prakash Khedekar

Acknowledgement

I would like to express by deep felt gratitude to my mentors Mr. Kartik Thakkar and Mr. Bhavish Mistry under whose guidance and mentorship this project could take form. Their unwavering support and knowledgeable advice helped me a lot during the challenging times of project development. Their assistance is invaluable to me and helped me grow a lot as a Software Developer.

I am also grateful for being provided this opportunity to intern at Black Box, which exposed me to a professional environment and provided me with a lot of practical experience in Artificial Intelligence, Natural Language Understanding and project management. This experience greatly helped me in my professional development and would help me reach greater heights in the future.

This internship is a huge milestone for my career and I would strive to make good use of this experience and knowledge in the best possible way.

Table of Contents

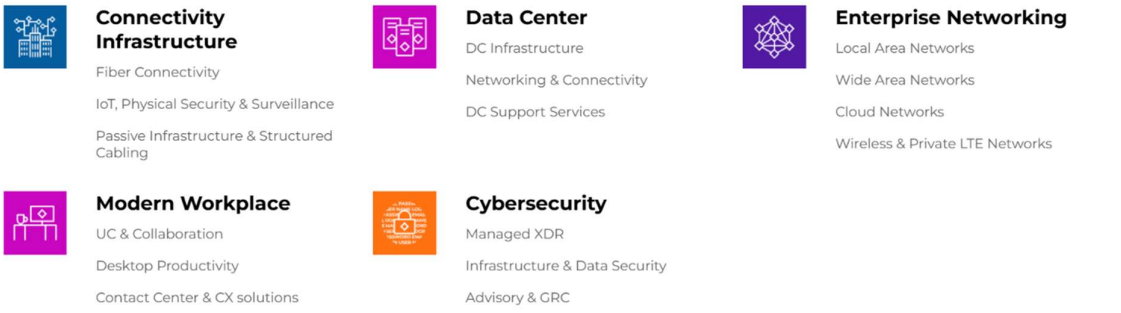
Company Overview

Black Box, a leading ICT Global Solutions Integrator and solutions specialist, is committed to delivering excellence in digital infrastructure solutions. With a strong focus on innovation, expertise, and customer-centricity through tailored Professional & Managed Services, Black Box provides comprehensive and cutting-edge products and solutions tailored to meet the unique needs of its clients.

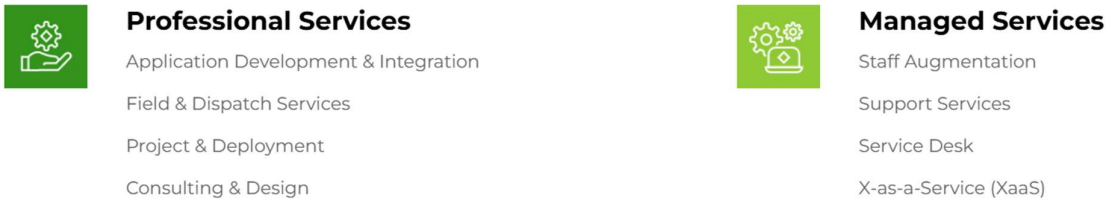
Black Box is a global digital infrastructure integrator delivering network and system integration services and solutions, support services, and technology products to businesses in the United States, Europe, India, Asia Pacific, the Middle East, and Latin America and has around 4,000 professionals globally.

Black Box has strong service offerings in network integration, digital connectivity infrastructure, data center build-out, modern workplace, and cybersecurity for businesses across various industries including financial services, technology, healthcare, retail, public services like airports; manufacturing, and other sectors.

Solutions Offered



Services Offered



Work Responsibilities

During my internship at Black Box Navi Mumbai (from June 9, 2025 to July 8, 2025), I was assigned the task of creating a Chatbot to assist in Shipment/Logistics with conversational abilities designed to guide the users through their problem or task at hand and provide satisfactory responses. My core responsibility in this project was to build such a bot using open-source tools which could serve the users in various actions like tracking their existing shipment, checking offers, changing address of the source and destination address and providing a rough estimate of shipment arrival. Additionally, it records customer feedback and generates a summary and sentiment of the user from the chat.

I conducted research on conversational AI frameworks and related open-source services. One such service which matched my requirements was Rasa Open Source. Rasa Open Source is an open-source conversational AI platform that allows us to understand and hold conversations, and connect to messaging channels and third-party systems through a set of APIs. It supplies the building blocks for creating virtual (digital) assistants or chatbots. Rasa offers a robust and modular NLU pipeline capable of handling complex intent classification and entity extraction tasks. Its support for both rule-based and machine learning-based components, combined with customization options, makes it well-suited for scalable and intelligent conversational systems.

To simplify development process, Rasa utilizes a modular structure. Rasa uses YAML format because it offers a clean, human-readable, and structured way to define conversational data like intents, rules, stories, and configurations. Backend logic is supported by python.

For this project, a client-side frontend was developed with support for buttons, card carousels and star ratings. The client-side frontend was connected to the Rasa server through the REST channel. The intents, entities, slot values, forms, actions and responses are defined in domain.yml, training examples for each intent and entity was listed in nlu.yml. The NLU pipeline and policies are defined in config.yml. The endpoints.yml file is used to configure external services that Rasa needs to connect to during runtime. Rules.yml and stories.yml contain rule-based conversation and example conversation flow respectively. To perform actions on data received from the user, an actions.py is used.

The user action flow was controlled with the use of buttons, to prevent the users from asking out of the scope, ambiguous queries that could lead to unpredictable and suboptimal responses. Validation of shipment tracking-id and source-destination pincode was implemented to prevent users from entering wrong format. To gauge customer satisfaction, a rating system was implemented. A chat summary and sentiment was generated based on user logs and rating with use of LlaMA3.2:latest hosted on local device using Ollama.

About the topic: Conversational Chatbots

A conversational Chatbot is an AI agent which can talk with its users. It uses large volumes of data, machine learning and natural language processing to mimic human interactions, recognize speech and text inputs and translate meaning from one language to another. Organizations use them for customer support and it can respond to the customers in a personalized manner.

Benefits of Conversational AI

1. Improved customer experience: Conversational AI chatbots can provide 24/7 support and immediate customer response, a service modern customers prefer and expect from all online systems. Instant response increases both customer satisfaction and the frequency of engagement with the brand.

2. Improved operational efficiency: Organisations can use conversational AI solutions to streamline your customer service workflows. They can answer frequently asked questions or other repetitive input, freeing up human workforce to focus on more complex tasks.

3. Wider accessibility: Conversational AI can be used to improve accessibility for customers with disabilities. It can also help customers with limited technical knowledge, different language backgrounds, or nontraditional use cases.

Use Cases of Conversational AI

Conversational AI has several use cases in business processes and customer interactions mainly grouped into four broad categories.

1. Informational: In an informational context, conversational AI primarily answers customer inquiries or offers guidance on specific topics.

2. Data capture: Organisations can use conversational AI tools to collect essential user details or feedback.

3. Transactional: In transactional scenarios, conversational AI facilitates tasks that involve any transaction.

4. Proactive: When conversational AI is used proactively, the system initiates conversations or actions based on specific triggers or predictive analytics.

Problem Statement

In my internship at Black Box Navi Mumbai, I was assigned the task of creating a chatbot with conversational abilities which could understand user input and provide appropriate output. It was also required to track the user ratings and generate a summary and identify the sentiment of the conversations from the chat log using a local LlaMA3.2:latest model.

Black Box develops support chatbots for their clients. The need for this project was to create a low-cost chatbot for shipment task for their cost-conscious clients. Rasa was selected for being open source which keeps costs at a minimum and it had a vast array of features which makes it a best fit tool for the task.

Key Objectives

1. Develop a cost-effective Conversational Chatbot: Design and develop a Conversational Chatbot which can address user's issues related to shipment and logistics, using open-source rasa framework to minimize development cost.

2. Enable Core Shipment Functionalities: Support essential shipment related tasks such as tracking shipment status, estimating delivery date, changing source and destination pincodes and showing offers through a guided conversational flow.

3. Ensure input validation: Implement necessary input validation to reduce user errors and ensure accurate processing.

4. Robust Conversational flow: Implement button-based responses to guide users through interactions efficiently.

5. Integrate Feedback mechanisms: Implement a user feedback system at the end of each conversation to gauge user satisfaction with chatbot's responses and services.

6. Generate Conversation Summary and Sentiment: Use a locally hosted LlaMA3.2:latest model through the Ollama API to process the user's chat log and rating, generating a brief summary of the interaction along with sentiment classification (Positive, Neutral, or Negative).

7. Design an intuitive, user-friendly interface: Build a simple, user-friendly client-side chatbot interface which has all of the necessary feature support like buttons, carousal, star rating widget and shortcuts to the main-menu and offers page.

Project Structure and Architecture

This project is developed using rasa open-source. It follows a modular structure and separates program files by its purpose. The files are organised and separated under a broad classification of user interactions, conversational flows, configuration settings and custom actions. A dedicated section further will describe MVC and client-server architecture in detail, this section will focus on how the files and folders are structured and how they interact and enable the chatbot's functionalities.

Key files and Folders:

Backend

- **nlu.yml:** Contains training examples for different intents such as track_shipment, change_address, estimate_delivery, etc., along with entity annotations for extracting information like tracking ID, source and destination pincodes.

- **domain.yml:** Defines the project's intents, entities, slots, forms, responses, and actions. It acts as a central configuration that governs how the chatbot behaves during conversations.

- **rules.yml**: Contains rule-based conversation paths such as how to trigger a form, how to handle user denial, or how to collect feedback using buttons.

- **stories.yml:** Defines sample user-bot interactions to train the dialogue management system using machine learning-based policies.

- **config.yml:** Specifies the NLU pipeline and policies used by Rasa. The NLU pipeline includes tokenization, featurization, and intent/entity classifiers, while the policies decide how the bot should respond.

- **actions.py:** Contains custom Python logic including:
    - Validation of tracking ID and pincodes
    - Fetching shipment status from SHIPMENT_DB
    - Estimating delivery time
    - Changing addresses
    - Recording user ratings
    - Generating conversation summaries and sentiment using LlaMA3.2 model via Ollama

- **endpoints.yml:** Configures external service endpoints such as the custom action server and REST.

- **data/ folder:** Stores training examples (NLU, rules and stories).

- **models/ folder:** Contains the trained Rasa models (.tar.gz) generated after training the chatbot.

Frontend

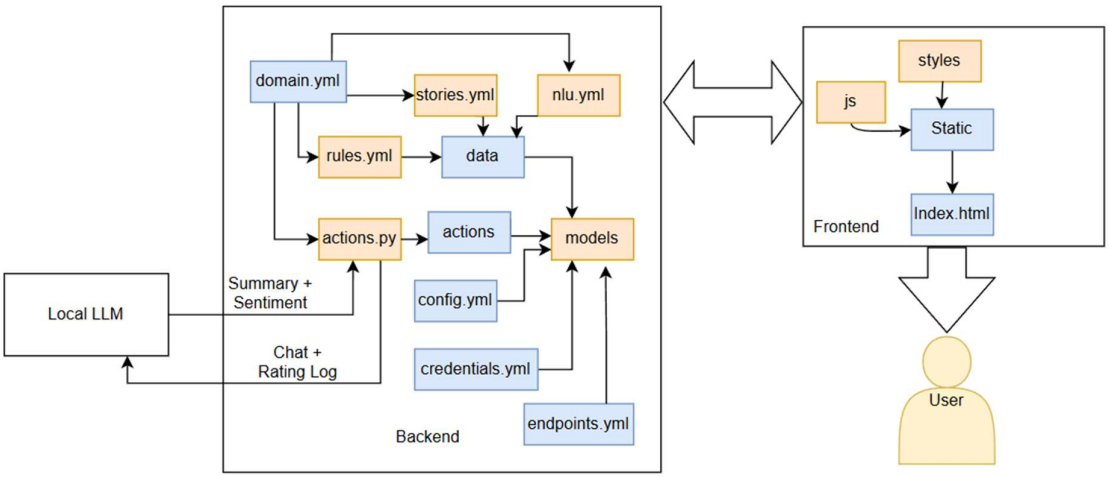
- **index.html:** Contains all styles and script imports, basic chatbot structures.
- **static/ folder:** contains css folder which contains all widget styles, img folder with all image files and js folder with all script files.

The frontend is a simple web-based interface that:

- Connects to the Rasa server via REST.
- Sends messages and receives responses.
- Displays custom buttons and carousels.
- Collects user ratings at the end of the chat.

Local LLM Integration

To generate feedback summaries and detect sentiment, the project integrates a locally hosted LlaMA3.2 model using the Ollama API. This component reads the latest conversation and rating from a CSV log and returns a formatted summary to the admin or user.



MVC Architecture Implementation

The Model-View-Controller (MVC) framework is an architectural/design pattern that separates an application into three main logical components Model, View, and Controller. Each architectural component is built to handle specific development aspects of an application. It isolates the business logic and presentation layer from each other. The Shipment Chatbot project follows the Model-View-Controller (MVC) architecture to ensure a clean separation of concerns, making the system modular, scalable, and easier to maintain.

Model: The Model represents the data layer of the application. In this project, it includes:

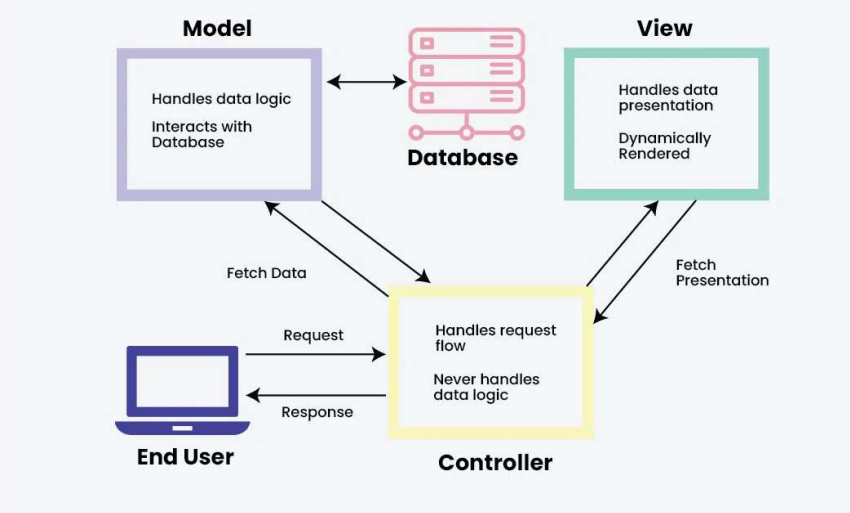
- SHIPMENT_DB dictionary which stores shipment details like tracking ID, source and destination addresses, and status.
- The rating_log.csv file that logs user ratings and feedback for analysis.
- Validation logic and data-handling functions in the actions.py file also fall under this layer, especially when reading/writing data.

View: The View is the frontend chat interface that users interact with. It renders the bot responses visually and enhances the overall user experience. It includes:

- A custom-built HTML/CSS/JavaScript interface which includes buttons, carousels, etc.

Controller: The Controller acts as the bridge between the View and Model. In this chatbot:

- Rasa Core acts as the controller by processing user inputs, determining the appropriate action based on the dialogue history (rules/stories), and triggering the correct logic.
- Custom actions written in actions.py handle business logic and data operations.
- The NLU pipeline helps route the intent of the user to the appropriate function/controller logic.



Client-Server Architecture Implementation

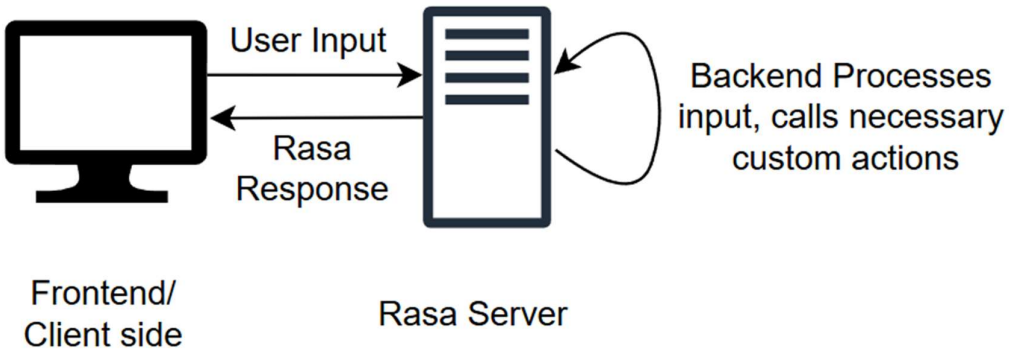
The Shipment Chatbot follows a Client-Server Architecture, which is a fundamental design model in distributed applications. In this architecture, the client and server are separated into independent components that communicate with each other over a network.

Client (Frontend): The Client is the web-based chat interface where the user interacts with the bot. It includes:

- A custom HTML/CSS/JavaScript UI, which renders the chat messages, buttons, quick replies, and star ratings.
- It collects user inputs and sends them to the Rasa server using REST API calls.
- It also displays bot responses, including text, buttons and carousels, in a user-friendly format.

Server (Backend): The Server is powered by Rasa Open Source, which processes incoming messages and generates intelligent responses. It includes:

- The Rasa NLU module for intent classification and entity extraction.
- The Rasa Core dialogue manager that decides the next action based on conversation history, rules, and policies.
- A custom action server (actions.py) that executes business logic such as:
- Validating tracking IDs and pincodes
- Accessing or updating the shipment database
- Generating summaries and sentiments using a locally hosted LlaMA model



Expected outcome and choice of service

Expected Outcome

The expected outcome of the project was to develop a fully functional, intelligent chatbot that can assist users with common shipment-related tasks in a natural, conversational manner.

The chatbot should be able to:

1. Engage in human-like conversation using a structured dialogue flow.

2. Allow users to ask shipment related queries like:

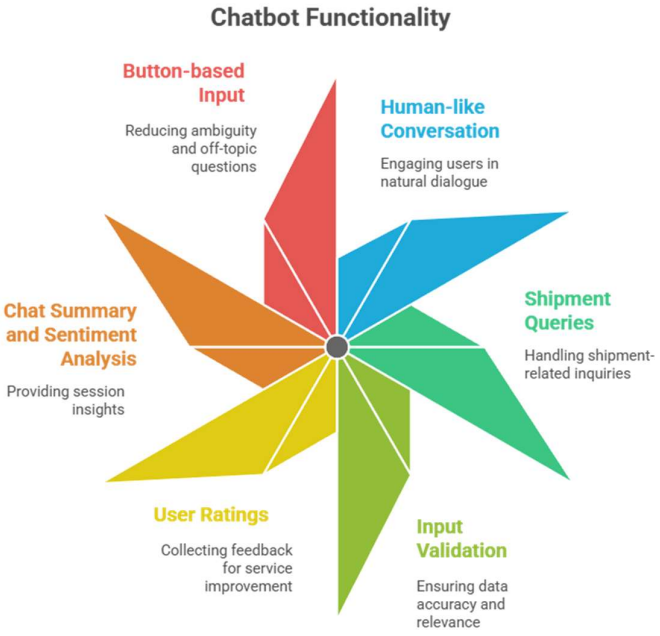
    - Track existing shipments using a tracking ID.
    - Get estimated delivery times for a shipment.
    - Change shipment address by updating source and destination pincodes.
    - View Offers.

3. Handle input validation

4. Collect user ratings after each session and log them into a CSV file.

5. Generate a chat summary and sentiment analysis using a local LLM model (llama3.2:latest) hosted via Ollama.

6. Support button-based input to reduce ambiguity and prevent off-topic questions.



Choice of Service

To build this chatbot, the following services and tools were carefully chosen:

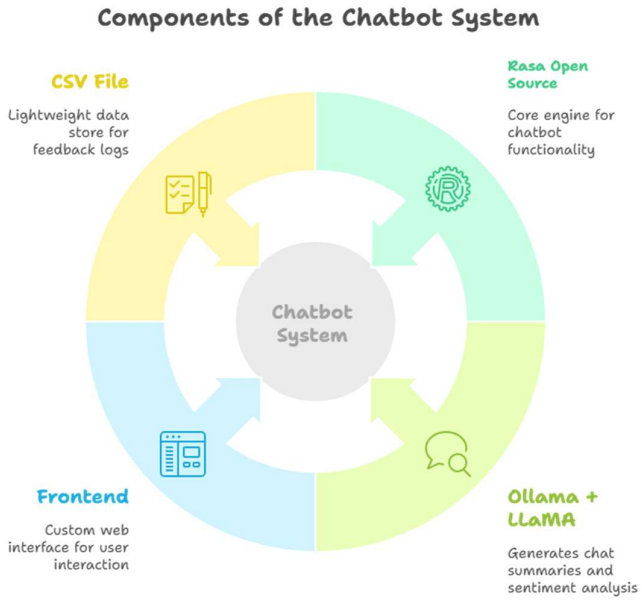
1. **Rasa Open Source**: Selected as the core engine for the chatbot due to its:
   - Open-source nature (no licensing costs)
   - Robust NLU pipeline for intent classification and entity extraction
   - Rule and ML-based dialogue handling
   - Support for custom actions and validation
   - Easy integration with REST, messaging platforms, and custom frontends

2. **Ollama + LlaMA 3.2:latest:** Used to generate chat summaries and sentiment analysis based on user logs and ratings. It was chosen for being local, cost-free, and easy to host without relying on cloud-based LLM APIs.

3. **Frontend (HTML + CSS + JS):** Developed a simple custom web interface with support for:

   - Buttons, carousels, and quick replies
   - Star-rating system
   - Markdown rendering for formatted messages
   - Connected to Rasa backend using REST API channel.

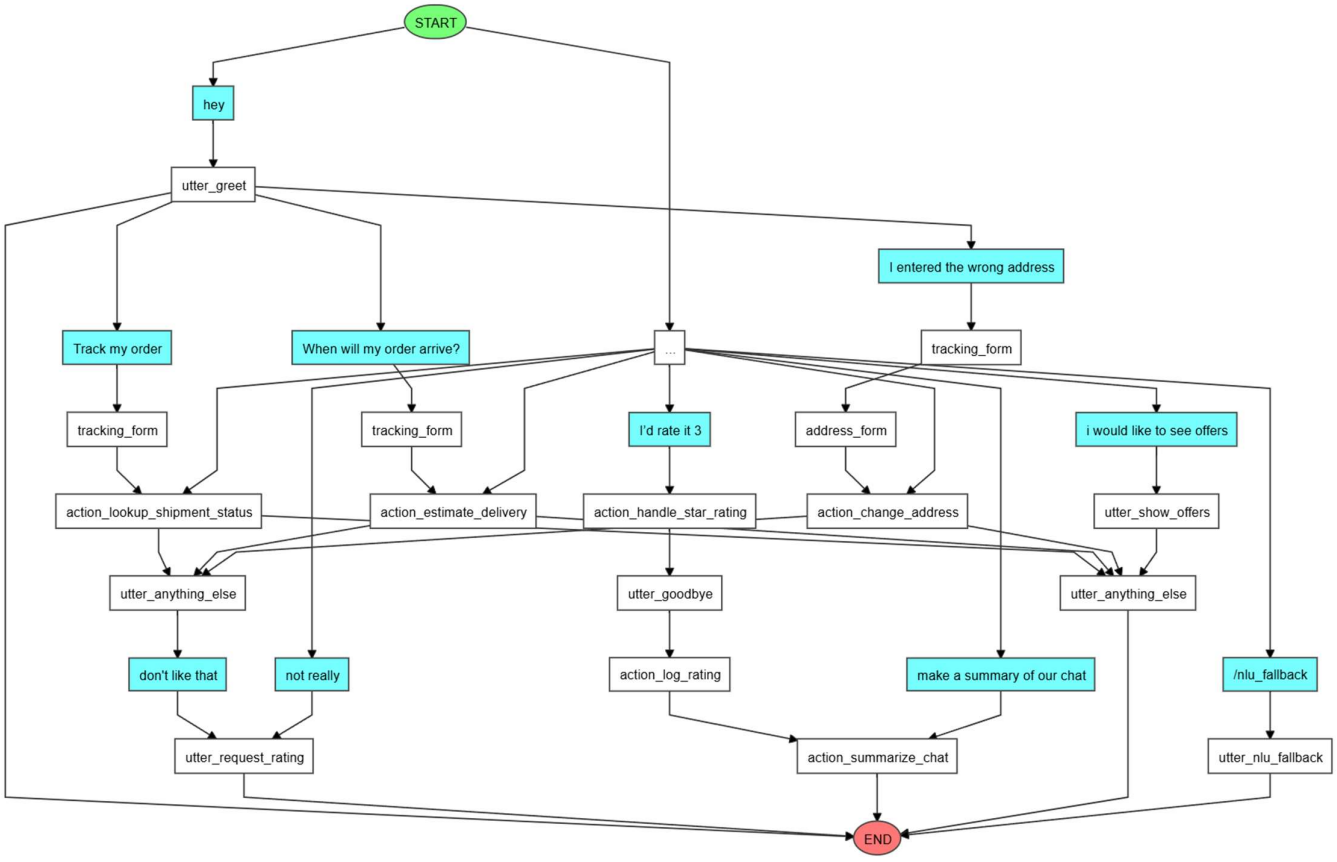
4. **CSV File (Rating Logs)**: Used as a lightweight data store to store feedback logs, sentiment and summary. It was chosen for simplicity, portability, and ease of use in a prototype setup.



Results

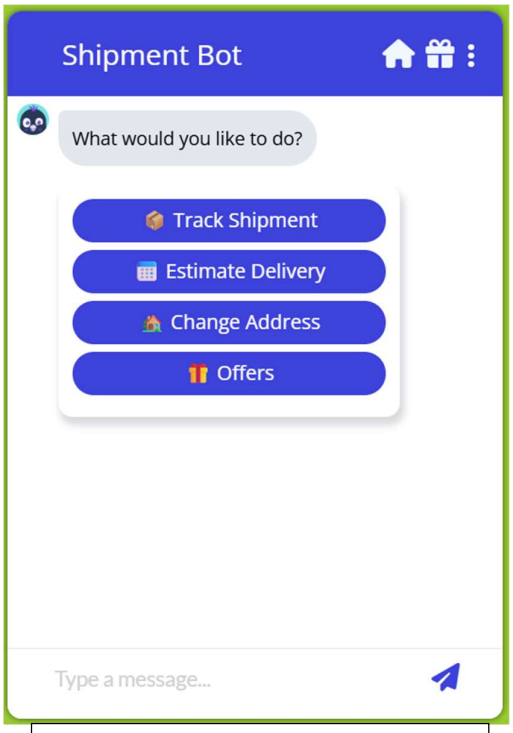
The Shipment Chatbot project successfully achieved its objectives and demonstrated a functional end-to-end conversational system for handling logistics-related tasks.

1. **Functional Conversational Chatbot:** A working chatbot was developed using Rasa Open Source, capable of engaging users through structured conversations and guiding them through shipment-related tasks with appropriate prompts, buttons, and responses.

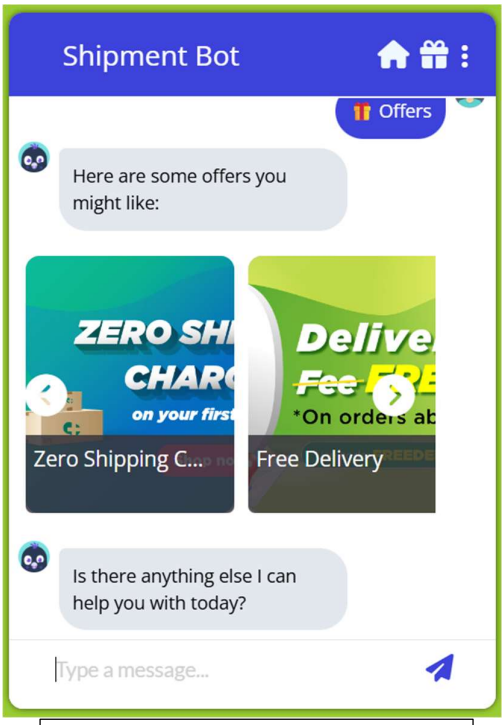


Conversational flow

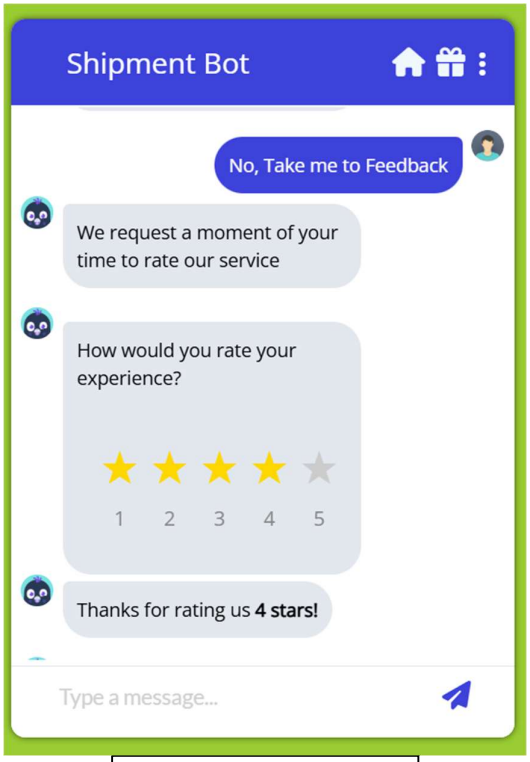
2. **User Interface Integration**: A frontend interface was implemented allows message inputs and replies and button, carousal, star rating support.



Main Menu showcasing buttons



Offers showcasing Carousels



Star Review System

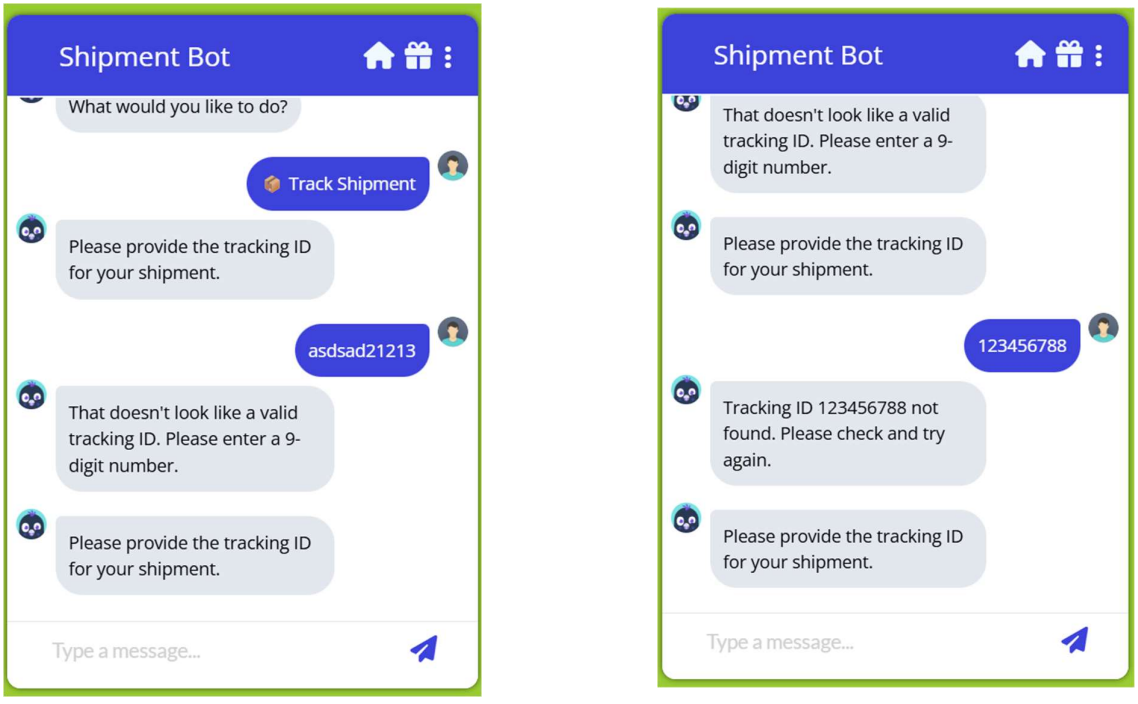
3. **User Feedback Analysis:** Ratings provided by users were logged into a CSV file and used to generate chat summaries and sentiment analysis with the help of a locally hosted LlaMA 3.2 model via Ollama.

ratings_log.csv

```
1   timestamp,user_id,rating,summary,sentiment
2   2025-07-08 16:24:08,544e3892-b013-4138-8db3-8c4e57c6b11d,4," The
    conversation starts with the bot asking what the user wants to do,
    then offers some options, asks if there's anything else it can help
    with, and finally requests a rating after being denied. The user
    gives a 4-star rating and thanks the bot before leaving. ", Neutral
3
```

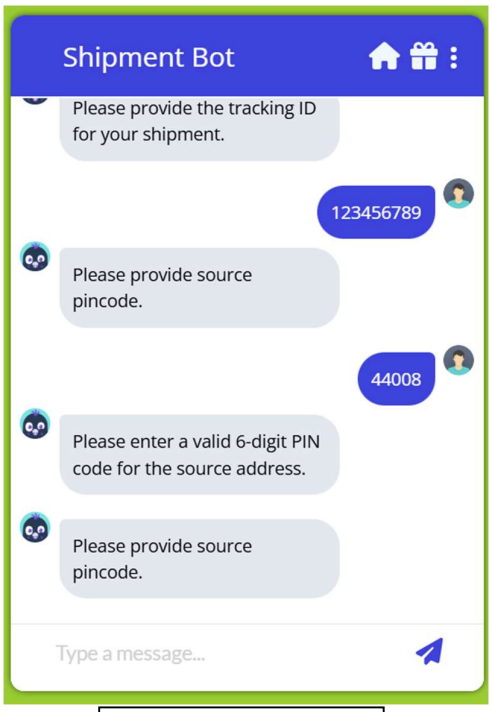
Rating log and summary

4. **Input Validation and Error Handling:** The bot was able to reject invalid entries and prompt users to re-enter correct information.
    - Tracking IDs (must be 9 digits and present in the shipment database)



Validating tracking ID

- Pincodes (must be 6-digit numeric values)



Pincode Validation

Limitations

While the Shipment Chatbot fulfilled its core objectives, there were certain limitations due to time constraints, scope, and available resources:

1. **Static Shipment Data**:
   The shipment tracking information was based on a locally defined dictionary (mock database). In real-world applications, integration with a live shipment tracking API or enterprise-grade database (like PostgreSQL, MySQL) would be required for dynamic updates.

2. **Limited Natural Language Flexibility**:
   Although Rasa provides a powerful NLU engine, the chatbot was primarily button-driven to avoid ambiguity. Free-text queries outside of predefined intents might not always result in accurate responses without additional training data.

3. **No Multilingual Support**:
   The chatbot only supported English. Support for regional or multiple languages would require additional intent training, entity recognition, and multilingual NLU pipelines.

4. **Simplified Address Handling**:
   Source and destination addresses were validated only through 6-digit pincode format. There was no geolocation, address normalization, or external validation through postal APIs.

5. **No Live Backend Integration**:
   All business logic (shipment status, address update, ETA, etc.) was handled locally. In a real deployment, APIs would be required to communicate with backend services for booking, updates, or tracking.

6. **Basic Sentiment Analysis**:
   Sentiment was generated using a local Ollama LLaMA3.2 model, which was not extensively fine-tuned for logistic-specific feedback. More robust feedback analysis would require fine-tuning or using external APIs.

7. **No Authentication or Security**:
   There was no user authentication or data privacy mechanisms implemented. In a production system, secure user identification and encrypted communication would be necessary.

Reflection on Learning Experience

This internship was a valuable opportunity for me to apply theoretical knowledge to a practical, real-world project. Building a shipment chatbot from scratch deepened my understanding of not just chatbot development, but also of AI/ML concepts, backend logic, conversational design, and software integration. I was introduced to Rasa Open Source, an industry-grade framework for building conversational AI, and learned how to effectively structure intents, slots, entities, forms, and custom actions.

Working through the project helped me improve my skills in Python programming, form validations, and modular development using YAML configuration. I learned the importance of designing user-centric interfaces by incorporating feedback buttons, guided flows, and robust input validation to avoid ambiguity.

Integrating Ollama's LLaMA3.2 for sentiment analysis taught me how modern LLMs can be used locally, enabling me to explore prompt engineering and lightweight inference without relying on cloud APIs. I also learned to debug logs, handle errors and interpret stack traces to improve stability.

Additionally, collaborating with my mentors and adapting to feedback taught me how to work independently, manage my time, and deliver a working prototype within a limited timeline. Overall, this experience enhanced my confidence in AI development and strengthened my problem-solving and communication skills.

Conclusion

The development of the Shipment Chatbot during my internship at Black Box, Navi Mumbai, provided me with practical exposure to building intelligent conversational systems using open-source tools. The project successfully demonstrated how AI-powered chatbots can enhance user experience by simplifying logistics tasks such as shipment tracking, address updates, and booking processes.

By leveraging the Rasa Open Source framework and integrating with a local LLM (LLaMA3.2) model for sentiment analysis, I was able to implement a complete solution that is both cost-effective and extensible. The chatbot was designed with modularity in mind, using structured YAML files, Python-based backend actions, and a REST-enabled frontend interface.

This experience strengthened my understanding of conversational AI architectures, form validation, client-server models, and user interaction design. It also highlighted the importance of user-centric flow, error handling, and natural language understanding in real-world applications.

The knowledge and skills gained during this project have laid a solid foundation for my future endeavors in AI, machine learning, and full-stack development.

References

1. Rasa Documentation

https://legacy-docs-oss.rasa.com/docs/rasa/

https://rasa.com/docs/

2. Frontend

https://github.com/JiteshGaikwad/Chatbot-Widget

3. Ollama

https://ollama.com/

4. Python Documentation

https://www.w3schools.com/python/python_reference.asp

Appendices

Appendix A: List of Intents and Entities used in the chatbot and how they are used in nlu.

```
intents:
  - track_shipment
  - estimate_delivery
  - change_address
  - provide_tracking_id
  - give_rating
  - offers
  - summarize_chat
  - provide_source_address
  - provide_dest_address

entities:
  - tracking_id
  - source_address
  - dest_address
  - rating
```

```
  - intent: provide_tracking_id
    examples: |
      - my tracking id is [123456789](tracking_id)
      - the id is [987654321](tracking_id)
      - it's [112233445](tracking_id)
      - here you go [554433221](tracking_id)
      - [778855446](tracking_id)
      - ok, the number is [187547875](tracking_id)
```

Appendix B: A few select rules and stories used.

```
- rule: show offers
  steps:
    - intent: offers
    - action: utter_show_offers
    - action: utter_anything_else
```

```
- story: Change address full flow
  steps:
    - intent: greet
    - action: utter_greet
    - intent: change_address
    - action: tracking_form
    - active_loop: tracking_form
    - slot_was_set:
        - tracking_id: "123456789"
    - active_loop: null
    - action: address_form
    - active_loop: address_form
    - slot_was_set:
        - source_address: "500001"
        - dest_address: "500009"
    - active_loop: null
    - action: action_change_address
    - action: utter_anything_else
    - intent: deny
    - action: utter_request_rating
```

Appendix C: Custom Action snippets (Rating Log)

```
class ActionLogRating(Action):
    def name(self) -> str:
        return "action_log_rating"
    def run(self, dispatcher: CollectingDispatcher,
            tracker: Tracker,
            domain: dict) -> list:
        # Get the rating and user ID
        rating = tracker.get_slot("rating") or tracker.latest_message['entities'][0]['value']
        sender_id = tracker.sender_id
        timestamp = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        # CSV file path
        file_path = "ratings_log.csv"
        # Check if file exists, if not write header
        file_exists = os.path.isfile(file_path)
        with open(file_path, mode="a", newline="") as csv_file:
            fieldnames = ["timestamp", "user_id", "rating","summary","sentiment"]
            writer = csv.DictWriter(csv_file, fieldnames=fieldnames)
            if not file_exists:
                writer.writeheader()
            # Write the rating entry
            writer.writerow({
                "timestamp": timestamp,
                "user_id": sender_id,
                "rating": rating
            })
        return [SlotSet("rating", rating)]
```

Appendix D: Config.yml

```
language: en
pipeline:
- name: WhitespaceTokenizer
- name: RegexFeaturizer
- name: RegexEntityExtractor
- name: LexicalSyntacticFeaturizer
- name: CountVectorsFeaturizer
- name: CountVectorsFeaturizer
  analyzer: char_wb
  min_ngram: 1
  max_ngram: 4
- name: DIETClassifier
  epochs: 100
- name: EntitySynonymMapper
- name: ResponseSelector
  epochs: 100
- name: FallbackClassifier
  threshold: 0.3
  ambiguity_threshold: 0.1
policies:
- name: RulePolicy
  core_fallback_threshold: 0.3
  core_fallback_action_name: "action_default_fallback"
  enable_fallback_prediction: True
- max_history: 6
  name: AugmentedMemoizationPolicy
- name: TEDPolicy
  max_history: 10
  epochs: 20
  batch_size:
  - 32
  - 64
```