AI-Powered Chatbots: Evolution, Implementation and Development

Shaikh Anis Faiyyaj
Artificial Intelligence & Data Science
AISSMS Institute of Information
Pune, India
anisshaikh2708@gmail.com

Talape Suyog Kisan
Artificial Intelligence & Data Science
AISSMS Institute of Information
Pune, India
suyogtalape123@gmail.com

Thorat Mansi Vijay
Artificial Intelligence & Data Science
AISSMS Institute of Information
Pune, India
manasithorat9088@gmail.com

Patil Keshavi Pramod
Artificial Intelligence & Data Science
AISSMS Institute of Information
Pune, India
keshavipatil710@gmail.com

***Abstract*—We have introduced tools to build and overcome the problems of traditional/rule-based chatbots. Existing chatbots often require correcting their perception of intent, leading to a reduced user experience. This study examines the development, implementation, and development of chatbots using AI-powered chatbots by integrating non-RASA open source (LLM) regional intents. This study focuses on optimizing perceptual intent and improving context-related understanding and the delivery of cost-effective cloud-based chatbots. Through comparative analysis and performance optimization, this study provides insight into improving chatbot efficiency.**

With the increasing reliance on conversational AI in customer service, education, and e-commerce, there is a growing need for more intelligent and adaptive chatbot systems. Traditional rule-based chatbots offer predictability but fall short in flexibility and scalability. To address these limitations, this research integrates the RASA Open Source framework with external Large Language Models (LLMs) to create a hybrid chatbot that leverages both intent classification and dynamic response generation. This paper explores the evolution of chatbots, from rule-based systems like ELIZA to modern AI-based assistants, and positions RASA as a cost-effective, customizable alternative for building enterprise-ready conversational agents.

The methodology includes a detailed implementation of a chatbot using RASA, enhanced by LLM integration for handling out-of-scope queries. Natural language understanding, dialogue management, and action handling are configured using machine learning pipelines. Additionally, the chatbot is deployed on a cloud-compatible architecture, with a simple HTML/JavaScript front-end, ensuring real-time responsiveness. Performance evaluation is conducted through both user testing and automated RASA test suites, focusing on metrics such as intent accuracy, latency, and user satisfaction.

Experimental results demonstrate significant improvements: intent classification accuracy reached 92.4%, response latency was reduced to 1.3 seconds, and user satisfaction averaged 4.2 out of 5. This research highlights the strength of combining open-source tools and language models to build robust, scalable, and context-aware chatbots. Future work includes improving long-term memory, enhancing support for multilingual input, and integrating reinforcement learning for continual adaptation. The study provides a roadmap for researchers and developers aiming to optimize AI chatbot systems for real-world applications.

***Index Terms*—Chatbot, Intent Recognition, Rasa Open Source, Large Language Models, Web Applications, AI Optimization, Performance Evaluation**

I. Introduction

Chatbots have been developed quite a bit, from rule-based systems to AI-powered solutions using machine learning and natural language processing (NLP). This study examines how RASA open source and LLM integration improve intent and context-related understanding. By optimizing chatbot performance and the use of scalable provisioning techniques, this study aims to create robust and inexpensive chatbot solutions for AI-driven chatbots.

A. Background

The evolution of chatbots has been a fascinating journey, reflecting significant advancements in artificial intelligence (AI) and natural language processing (NLP). The inception of chatbots dates back to the 1960s with the creation of ELIZA by Joseph Weizenbaum. ELIZA operated on pattern-matching techniques to simulate conversation, laying the groundwork for future developments in human-computer interaction. Despite its simplicity, users often attributed human-like understanding to ELIZA's responses, a phenomenon later termed the "ELIZA effect" [1].

Following ELIZA, the 1990s witnessed the emergence of ALICE (Artificial Linguistic Internet Computer Entity), which utilized heuristic pattern matching to engage in dialogues. These early systems, while innovative, were constrained by their rule-based architectures, limiting their conversational depth and adaptability.

The 21st century ushered in a paradigm shift with the integration of machine learning and deep learning techniques into Chatbot development. Notably, the introduction

of transformer-based models, such as OpenAI's GPT series, revolutionized the field by enabling more coherent and contextually relevant interactions. These models leverage vast datasets and sophisticated algorithms to generate human-like responses, marking a significant departure from their rule-based predecessors.

Concurrently, industry leaders have been instrumental in advancing chatbot capabilities. For instance, Microsoft's XiaoIce, designed as an empathetic social chatbot, has engaged millions of users by establishing emotional connections and maintaining prolonged conversations[2].

Similarly, Google's LaMDA model has been developed to facilitate open-ended dialogues, enhancing the scope and quality of conversational AI [3].

B. Problem Statement

Existing chatbots often require correcting their perception of intent, leading to a reduced user experience. The purpose of this study is to improve the performance of chatbots in web applications. RASA Open Source is the integration of outdoor areas with LLMS other than LLMS. This study focuses on optimizing intention perception, improving context-related understanding, and providing cost-effective, cloud-based chatbots.

C. Research Questions & Objectives

1) How effectively can NLP-based intent recognition models enhance chatbot capabilities?
2) How to build a chatbot using open source platform like RASA nlu and RASA core?
3) Evaluate the performance of different NLP models to identify the most efficient approaches.
4) Identify the most efficient approaches for chatbot development to enhance response accuracy and user satisfaction.
5) How does their integration impact user experience in real-world applications?

D. Scope and Limitations

1) **Enhancing and Improving Intent Recognition Accuracy:** Future research should focus on improving intent recognition accuracy using advanced machine learning to better handle nuanced and ambiguous language, leading to more precise understanding.
2) **Improving User Experience with Latest Technologies:** Future development should aim to enhance user experience by integrating multimodal inputs and personalized dialogue strategies, creating more natural and adaptive conversational interactions.
3) **Deploying on Compatible Cloud Services:** Future efforts should concentrate on optimizing deployment on cloud services using containerization and cloud-native architectures to improve scalability and simplify implementation.
4) **Implementing a Specific Intent Chatbot (Customer Service):** Future work should focus on developing specialized chatbots for domains like customer service, incorporating knowledge representation to handle complex inquiries and provide accurate support.
5) **Increment in Data:** Future research needs to prioritize increasing the volume and quality of training data through efficient collection, annotation, and augmentation techniques, and explore transfer learning methods.

E. Limitations

1) Currently supports English; depends on API response time.
2) Fails to respond or crashes when many actions are applied on same type of intent.
3) Custom actions (LLM responses) uses API that can handle only certain requests per minute.

II. Related Work

The evolution of chatbots has been marked by significant advancements, transitioning from simple rule-based systems to sophisticated AI-driven conversational agents. Early chatbots, such as ELIZA, relied on predefined scripts and pattern matching, limiting their ability to handle complex interactions . The introduction of machine learning and natural language processing (NLP) techniques facilitated the development of more adaptive and context-aware chatbots.[1]

The RASA framework has emerged as a prominent open-source platform for building conversational AI. RASA comprises two main components: Rasa NLU for natural language understanding and Rasa Core for dialogue management. This architecture enables developers to create chatbots capable of intent recognition and entity extraction, allowing for more dynamic and personalized user interactions .[4] Recent studies have explored the integration of large language models (LLMs) with frameworks like RASA to enhance chatbot performance. For instance, fine-tuning LLMs for intent classification in banking chatbots has demonstrated improved accuracy in understanding user queries . Additionally, research on entity extraction using RASA NLU has shown that incorporating custom components can significantly enhance the chatbot's ability to handle domain-specific conversations . Furthermore, the application of AI-powered chatbots has expanded across various sectors. In the educational domain, chatbots have been developed to assist with university inquiries, providing timely and accurate information to students . In the healthcare sector, chatbots have been utilized to provide medical assistance and information, demonstrating the versatility and potential of conversational AI in diverse fields . The continuous evolution of chatbots is driven by advancements in AI and NLP, enabling more natural and effective human-computer interactions. However, challenges remain, particularly in ensuring ethical use and mitigating biases inherent in AI models . Addressing these challenges is crucial for the responsible development and deployment of AI-powered chatbots.[8]

III. Literature Review

A. Evolution of chatbot

Chatbots have evolved from simple rule-based systems to sophisticated AI-controlled conversations. LLM integration allows for more natural and contextual interactions, marking the journey of AI-powered chatbots through several phases, from basic rule-based systems to sophisticated AI-controlled conversation agents [10].

1) **1966 – ELIZA:** The first chatbot, developed by Joseph Weizenbaum at MIT, used pattern matching and keyword identification to simulate human-like conversations. However, it lacked true contextual understanding [1].
2) **1995 – ALICE (Artificial Linguistic Internet Computer Entity):** This chatbot improved upon ELIZA by incorporating more advanced pattern-matching techniques using AIML (Artificial Intelligence Markup Language).
3) **2001 – SmarterChild:** Introduced on AIM and MSN Messenger, SmarterChild was an early example of chatbots integrated into messaging platforms, offering entertainment and basic information retrieval.
4) **2010 – IBM Watson:** This AI system demonstrated the potential of natural language processing (NLP) and machine learning by winning *Jeopardy!* against human champions, setting the stage for AI-driven chatbots [6].
5) **2016 – Rise of Virtual Assistants (Siri, Alexa, Google Assistant):** Tech giants launched AI-powered voice assistants capable of understanding user intent and context, marking a shift toward deep learning-based chatbots.
6) **2017 – Introduction of Rasa Open Source:** Rasa provided a framework for building conversational AI using NLP and machine learning, enabling developers to create customizable and scalable chatbots.
7) **2020 – GPT-3 and LLM Integration:** OpenAI's GPT-3 introduced large-scale language models that significantly improved chatbot performance in intent recognition, context retention, and dynamic response generation.
8) **2023 – AI Chatbot Optimization with Groq:** AI models like ChatGPT and chatbots using frameworks such as Groq and Rasa became more efficient, reducing response time and improving real-time interactions.

B. Rule-Based Chatbots

Rule-based chatbots represent the foundational stage in the evolution of conversational agents. Unlike their AI-driven successors, these chatbots do not learn or adapt over time. Instead, they operate on a fixed set of pre-programmed rules defined by developers. These rules serve as a roadmap for the chatbot to follow during interactions with users, often taking the form of "if-then" statements. If a user's input matches a known pattern, the bot responds with a predetermined reply. This predictable, structured behavior makes rule-based chatbots reliable and easy to manage—particularly for straightforward, repetitive tasks. The inner workings of a rule-based chatbot revolve around pattern matching and decision trees. When a user sends a message, the bot attempts to identify keywords or patterns that align with one of its scripted inputs. For instance, if someone types "What are your opening hours?", the bot may be programmed to detect the phrase "opening hours" and respond with something like "We are open from 9 AM to 5 PM, Monday to Saturday." This approach works well in environments where user questions follow a known, limited structure and the responses remain relatively static. Many of these bots are built using scripting languages like AIML (Artificial Intelligence Markup Language), which allows developers to create and manage these conversational flows. One of the key strengths of rule-based chatbots lies in their simplicity. Since they rely on a finite set of instructions, developers maintain full control over the conversation. This predictability ensures that the chatbot will behave consistently and will not surprise users with unpredictable or irrelevant answers—a common concern with AI-based bots in their early stages. They are also resource-efficient, requiring neither large datasets nor computational training, which makes them accessible to smaller businesses or projects operating with limited technical resources. However, the limitations of rule-based systems become apparent as the complexity of interaction increases. These chatbots lack the flexibility to handle language variations or unforeseen queries. If a user phrases a question differently than expected, or introduces a new topic, the bot is often unable to respond meaningfully. Moreover, they are not context-aware, which means they struggle to maintain continuity across a conversation. Each interaction is processed in isolation, without memory of what was said earlier, which can make conversations feel mechanical or fragmented. Managing and expanding these bots also becomes a challenge over time. As more rules are added to accommodate new scenarios, the system can become cumbersome and difficult to maintain.[5] Despite these constraints, rule-based chatbots continue to serve a valuable purpose in today's digital landscape. They are commonly deployed in customer support environments for answering frequently asked questions, handling appointment bookings, guiding users through websites, and managing basic data collection tasks. Their reliability and clarity make them suitable for use cases where precision is more important than conversational nuance. Looking ahead, rule-based systems are unlikely to disappear entirely. Instead, they are increasingly being integrated into hybrid models that combine the structured control of rules with the adaptive capabilities of artificial intelligence. This blended approach allows developers to maintain oversight of mission-critical functions while also enabling the chatbot to handle more dynamic, context-rich conversations. In a rapidly evolving technological world, rule-based chatbots remain a testament to the importance of simplicity and structure in human-computer interaction.

IV. RASA Framework

A. Architecture

Rasa's architecture is modular by design. This allows easy integration with other systems. For example, Rasa Core can be used as a dialogue manager in conjunction with NLU

services other than Rasa NLU. While the code is implemented in Python, both services can expose HTTP APIs so they can be used easily by projects using other programming languages. Dialogue state is saved in a tracker object. There is one tracker object per conversation session, and this is the only stateful component in the system. A tracker stores slots, as well as a log of all the events that led to that state and have occurred within a conversation. The state of a conversation can be reconstructed by replaying all of the events. When a user message is received Rasa takes a set of steps as described in figure 1.

Step 1 is performed by Rasa NLU, all subsequent steps are handled by Rasa Core.

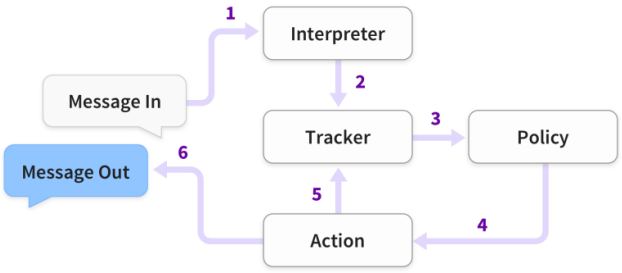

Fig. 1. Image from [**?**].
1. A message is received and passed to an Interpreter (e.g. Rasa NLU) to extract the intent, entities, and any other structured information.
2. The Tracker maintains conversation state. It receives a notification that a new message has been received.
3. The policy receives the current state of the tracker.
4. The policy chooses which action to take next.
5. The chosen action is logged by the tracker.
6. The action is executed (this may include sending a message to the user).
7. If the predicted action is not 'listen', go back to step 3.

B. Actions

We frame the problem of dialogue management as a classification problem. At each iteration, Rasa Core predicts which action to take from a predefined list. An action can be a simple utterance, i.e., sending a message to the user, or it can be an arbitrary function to execute.

When an action is executed, it is passed a tracker instance, and so can make use of any relevant information collected over the history of the dialogue: slots, previous utterances, and the results of previous actions. Actions cannot directly mutate the tracker, but when executed may return a list of events. The tracker consumes these events to update its state.

There are a number of different event types, such as `SlotSet`, `AllSlotsReset`, `Restarted`, etc.

C. Natural Language Understanding

RASA NLU is the natural language understanding module. It comprises loosely coupled modules combining a number of natural language processing and machine learning libraries in a consistent API. We aim for a balance between customisability and ease of use. To this end, there are pre-defined pipelines with sensible defaults which work well for most use cases.

For example, the recommended pipeline, `spacy_sklearn`, processes text with the following components. First, the text is tokenised and parts of speech (POS) annotated using the spaCy NLP library. Then the spaCy featuriser looks up a GloVe vector for each token and pools these to create a representation of the whole sentence. Then the scikit-learn classifier trains an estimator for the dataset, by default a multiclass support vector classifier trained with five-fold cross-validation. The `ner_crf` component then trains a conditional random field to recognise the entities in the training data, using the tokens and POS tags as base features.

Since each of these components implements the same API, it is easy to swap (say) the GloVe vectors for custom, domain-specific word embeddings, or to use a different machine learning library to train the classifier [**?**].

D. Policies

The job of a policy is to select the next action to execute given the tracker object. A policy is instantiated along with a featurizer, which creates a vector representation of the current dialogue state given the tracker.

The standard featurizer concatenates features describing:

- what the last action was,
- the intent and entities in the most recent user message,
- which slots are currently defined.

The featurization of a slot may vary. In the simplest case, a slot is represented by a single binary vector element indicating whether it is filled. Slots which are categorical variables are encoded as a one-of-k binary vector, those which take on continuous values can specify thresholds which affect their featurisation, or simply be passed to the featurizer as a float.

There is a hyperparameter `max_history` which specifies the number of previous states to include in the featurisation. By default, the states are stacked to form a two-dimensional array, which can be processed by a recurrent neural network or similar sequence model.

E. Understanding the Flow of Project

Fig. 2 shows the flow chart of the project (Generated by Imagen).

Fig. 2. Flow chart of the project [**?**].

V. Design and Implementation of the AI-Powered Chatbot

RASA open source provides structured intent, while LLMS improves out-of-range queries response. Implementing these technologies improves Chatbot adaptability and Source response accuracy. Modern AI-powered chatbots such as RASA Open Source and Groq (LLMS) integrate frameworks to improve interaction and adaptability quality.

A. Key Components of AI Chatbots

AI chatbots are intricate systems that function by combining multiple key components to imitate human conversation. NLP: Natural Language Understanding (NLU) is the part of the system that determines the meaning behind user input. It's not just about recognizing words; it's about understanding the user's goal and identifying the key pieces of information within their message. To accomplish this, NLU systems utilize machine learning models and a range of Natural Language Processing (NLP) techniques. Named Entity Recognition (NER) is used to identify and classify entities such as names, dates, or locations, while word embeddings, which represent words as numerical vectors, help the system comprehend the relationships between words. Dialogue Management: Dialogue management serves as the "brain" of the chatbot, making decisions on the most suitable response based on the user's input and the ongoing conversation. It maintains the context and ensures the conversation flows logically. Frameworks such as Rasa Core employ a story-centric methodology, utilizing example dialogues to train dialogue policies. These policies empower the chatbot to make well-informed choices regarding its next course of action, such as seeking clarification or offering an answer. Intent Classification: Intent recognition is a fundamental task within NLU, where AI models classify user input into predefined intents. These intents signify the user's objective, such as booking a flight, checking the weather, or placing an order for a product. As Large Language Models (LLMs) become more prevalent, they are being utilized to enhance intent handling, as they possess the ability to comprehend ambiguous, nuanced, or intricate queries that might perplex simpler systems. Response Generation: After comprehending the user's purpose and the overall conversation, the chatbot must produce an appropriate reply. This can be done in several ways. Certain chatbots depend on pre-programmed responses, where answers are stored and retrieved based on the recognized intent. Template-based responses utilize a standard structure with designated areas to input specific details. More sophisticated chatbots, particularly those utilizing LLMs, can produce dynamic and contextually appropriate responses based on the conversation's past. Chatbots based on rules can only provide predefined answers. Backend and APIs: In order to offer valuable and up-to-date information, chatbots frequently require integration with external systems. This is accomplished through backend connections and Application Programming Interfaces (APIs). For instance, a chatbot could connect to a database to access user account information or utilize an API to obtain real-time data such as weather forecasts or product availability. Deployment Techniques: The deployment method of a chatbot is of utmost importance as it directly impacts its accessibility and reliability.Cloud-based deployment on platforms such as AWS, Google Cloud, or Microsoft Azure is prevalent, as it provides scalability to accommodate numerous users and guarantees high availability.

B. Implementation Using Rasa and LLMs

Prerequisites: Python Programming Language

A chatbot implementation using **Rasa Open Source** follows these steps:

1) **Setting Up the Virtual Environment:**
   Rasa is only compatible with Python version 10 and Python version 11
   To avoid compatibility issues and deprecated warnings, it is better to **initialize** an environment using Python 3.10.

Terminal
python3.10 -m `venv <your_env_name>`

   This will create a virtual environment that avoids package version conflicts in your system. All your project dependencies are stored in the virtual environment.
2) **Install Dependencies and Initialize Rasa:**
   Now follow the below commands in your system. This will install Rasa Open Source package into your local system with the Rasa default template!

Terminal
pip install rasa rasa init

   Following template is generated by command `rasa init`:

RASA DEFAULT TEMPLATE

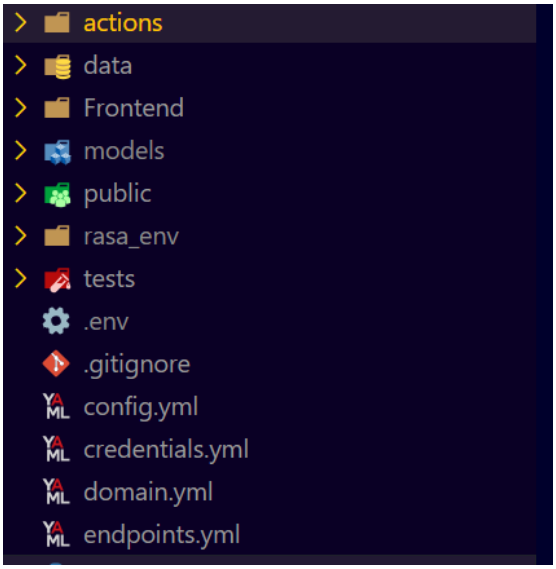

Fig. 3. Implementation Steps for Rasa Open Source and LLMs

This creates a project structure with files such as nlu.yml, stories.yml, and domain.yml.

3) **Training the Intent Recognition Model** Rasa NLU helps in extracting user intents using predefined NLU agorithms already present into RASA library

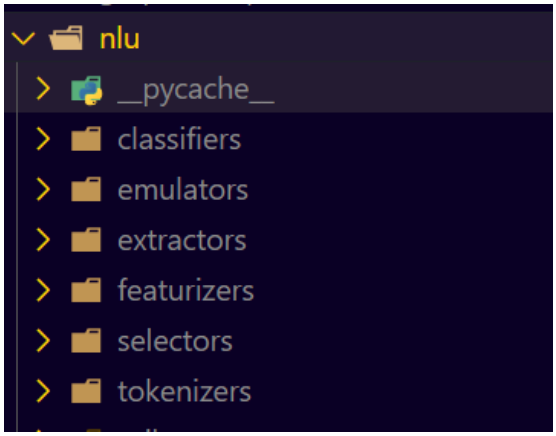

Fig. 4. Figure shows different components of Rasa NLU package

Rasa NLU consists of components shown in above img. **Tokenizer** (helps in tokenizing, NER, POS tagging) are used to break user's words into individual words. Entity extraction is the task of identifying and categorizing key pieces of information from the user's input, this task is carried out by extractors in preprocessing.
Featurizers are components that take the user's text input and convert it into a numerical representation (features) that machine learning models can understand. This directory would contain different featurization techniques, such as using word embeddings (like **GloVe**, mentioned in the research paper [9]) or bag-of-words approaches.
Classifiers likely contain the different algorithms and configurations used for intent classification. Intent classification is the process of determining the user's goal or intention behind their message (e.g., `greet`, `order_pizza`, `check_weather`).
Selectors likely contain components used for selecting the best intent or entity prediction when multiple possibilities exist. For example, if the model is uncertain between two intents, a selector component might use additional logic or confidence scores to make the final decision.

Define intents in `nlu.yml`:

Below image shows code from implemented project for intent 'greet'

nlu.yaml

Train the model:

Terminal
rasa train

Fig. 5. Example intent defined in `nlu.yml`

After this command, Rasa will start training your bot based on your data in `nlu.yml` file.
Corresponding actions to the user intents are defined in `rules.yml` file.

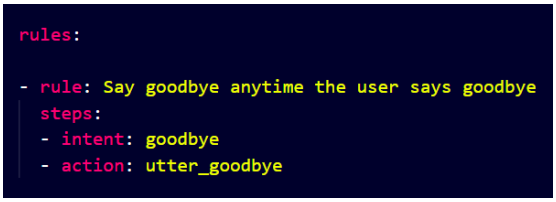

Fig. 6. `rules.yml` example for intent-action mapping

Above fig shows how to configure a rule that maps the `goodbye` intent to the `utter_goodbye` action.

C. *Defining Responses and Actions:*

Responses are mapped in `domain.yml`.
`Domain.yml` file contains a list of user-defined intents and the corresponding responses to the user intents.

Till here our bot only handles *queries* and their responses that are defined by the developer/*researcher*. This is similar to a rule-based chatbot but is slightly advanced with NLP (subset of AI).
If a user enters *"who is cristiano Ronaldo"* (i.e., an out-of-the-box query for chatbot), the developer needs to map

```yaml
responses:
  utter_greet:
  - text: "Hey! How are you?"

  utter_cheer_up:
  - text: "Here is something to cheer you up:"
    image: "https://i.imgur.com/nGF1K8f.jpg"
```

Fig. 7. Example response definitions in `domain.yml`

this query to a specific response, otherwise the chatbot can't reply.
As this is too much for a developer to define every response for every user intent *manually*, to overcome this issue we have integrated an LLM API that can handle such out-of-the-box queries!! Isn't that a great idea?
The integration of an LLM API into a RASA naïve chatbot model is shown below.

D. Adding Large Language Model (LLM) Integration:

Large Language Models (LLMs) are advanced artificial intelligence models designed to understand, generate, and manipulate human language. They are trained on massive amounts of textual data, enabling them to identify patterns and relationships between words and phrases.
LLMs work by predicting the next word in a sequence, given the preceding words. During training, they learn the statistical probabilities of word occurrences and how words relate to each other in context.

We don't need to train our own LLM to handle out-of-the-box *queries*, there are many different free-tier LLMs that can be integrated into a project using *API's*.

- For complex queries, we have integrated a free-tier LLM like GPT-4 for handling out-of-scope inputs. Many platforms provide free LLM API's to the client with certain limits of queries per hour/day. Platforms like GROQ provide different LLMs with very low prices, almost free. For further information, please visit: https://console.groq.com/docs/
- **Beware:** Some LLMs include billing charges for certain limited responses.

Let's come back to the integration of **LLM**. This can be done using custom actions in `actions.py` that call an API-based LLM service.

Fig: This intent and corresponding responses are mapped by `domain.yml` file using `rule.yml` file.
In the below image, `action_handle_out_of_scope` is a custom action function written by the developer/programmer.

```yaml
responses:
  utter_greet:
  - text: "Hey! How are you?"

  utter_cheer_up:
  - text: "Here is something to cheer you up:"
    image: "https://i.imgur.com/nGF1K8f.jpg"
```

Fig. 8. Out-of-scope intent in `nlu.yml` and mapped in `rules.yml`

```yaml
- intent: out_of_scope
  examples: |
    - Who is Mahatma Gandhi?
    - What is the capital of France?
    - How does a black hole form?
    - Is AI dangerous?
    - Is Bitcoin a good investment?
    - Tell me about World War II.
    - Can you explain quantum computing?
```

Fig. 9. `domain.yml` defining the custom action and reference to `actions.py` implementation

VI. Integrating with UI (User Interface)

For *convenience* and *simplicity*, we have made the frontend of the chatbot using simple technologies like **HTML**, **CSS**, and **JavaScript**, avoiding high-tech trending frontend frameworks like React.js and Next.js.
As there were too many compatibility issues with these frameworks on integration with RASA Open Source, we chose a simpler path.

RASA comes with its own user-defined API that can be integrated with a user interface for triggering chatbot responses based on user intents through a client.

```yaml
- rule: Handle out-of-scope queries
  steps:
    - intent: out_of_scope
    - action: action_handle_out_of_scope
```

Fig. 10. Terminal commands to start RASA with UI integration

After running these commands, wait a minute for RASA to start its server as it uses heavy libraries like **Keras**, **TensorFlow**, etc.

For more details about defining custom actions for user intents, feel free to check the official RASA article: https://rasa.com/docs/rasa/custom-actions

```
actions:
  - action_handle_out_of_scope
```

Fig. 11. Chatbot response to user queries via HTML UI and RASA server

Above image shows how the chatbot responds when an HTML page sends a user query to the RASA server on a local development environment.

First query showcases how a predefined response for the intent *hi* worked correctly. The second query demonstrates how the chatbot handled out-of-the-box questions like *"who is cristiano ronaldo?"* using an LLM.

A. Comparative Analysis of Rule-Based and AI Chatbots

Rule-based chatbots follow predefined answers but lack adaptability, whereas AI-powered chatbots use NLP and ML to improve over time.

RASA, combined with platforms like **GROQ**, improves chatbot response time and scalability.

Normally, rule-based bots use fixed patterns and responses, but AI-driven bots—powered by deep learning—can dynamically generate answers using NLP and ML techniques, offering smarter and more interactive responses over time.

B. Comparative Analysis of Rule-Based and AI Chatbots

Rule-based chatbots follow predefined answers, but lack adaptability, while AI-powered chatbots use Natural Language Processing (NLP) and Machine Learning (ML) to improve over time. Rasa, combined with tools like Greq, improves chatbot reaction times and scalability. While rule-based chatbots rely on fixed rules and patterns, AI-driven chatbots leverage deep learning to generate intelligent responses and adapt dynamically.

The choice between the two depends on factors such as complexity, adaptability, and arithmetic requirements.

C. Performance Optimization in AI Chatbots

Chatbot efficiency can be improved by optimizing AI model size, using caching mechanisms, and implementing asynchronous processing. These strategies ensure interaction with low latency and reduce server load. Early chatbots can lead to slower responses, high computing costs, and lower user experience. The most important areas of optimization are AI model efficiency, backend improvements, frontend improvements, and scaling strategies.

D. Optimizing AI Models

Large AI models require considerable computing power and can slow chatbot responses. Optimizing these models ensures that chatbots provide quick answers while minimizing hardware resource consumption.

1. Model Pruning

Disconnect is a powerful technique for reducing the size or complexity of a model by removing redundant components. Cutting can be divided into unstructured pruning, semi-structured pruning, and structured pruning. Structured pruning removes entire components such as neurons, attention heads, layers, etc. based on specific rules, and receives the entire network structure at the same time. On the other hand, unstructured cuts lead individual parameters, leading to irregular sparse structures. Semi-structured pruning is a way between structured and unstructured pruning, simultaneously achieving fine grain trimming and structure regularization. There are partial parameters based on a specific pattern than the entire channel, filter, or neuron, resulting in a finer form of structured pruning [9].

Table 2 shows the performance of many representative LLM pruning methods.

```
# This files contains your custom actions which can be used to run
# custom Python code.
#
# See this guide on how to implement these action:
# https://rasa.com/docs/rasa/custom-actions
```

Fig. 12. `domain.yml` defining the custom action and reference to `actions.py` implementation

- **Structured Pruning:** This approach helps to make the AI model smaller and more efficient, which significantly speeds up the time it takes for the chatbot to process user input and generate a response. By employing this technique, the size of the language model is reduced, leading to faster inference times. This means the chatbot can analyze user queries and provide responses more quickly, creating a more fluid and responsive conversational experience.
- **Unstructured Pruning:** Also, it lessens the amount of computer memory and processing power required to run the chatbot, making it more practical to deploy on various devices and systems. Furthermore, this method decreases the memory footprint and the computational resources needed to operate the chatbot. This reduction in overhead makes it possible to run the system on less powerful hardware and lowers the overall cost of deploying and maintaining the conversational AI application.

Example: A chatbot using BERT-based intent recognition can prune redundant layers, improving speed without affecting accuracy.

2. Quantization

Quantization refers to the process of reducing the number of bits (i.e., precision) in the parameters of the model with minimal loss in inference performance. Quantization can be categorized into two main approaches: Quantization-Aware Training (QAT), and Post-Training Quantization (PTQ). The primary distinction between the two approaches lies in whether retraining is needed during quantization. PTQ enables direct use of quantized models in inference, while QAT requires retraining to rectify errors introduced by quantization [9].

Example: Implementing INT8 quantization in TensorFlow Lite can optimize chatbot models for mobile devices.

3. Knowledge Distillation

Knowledge Distillation (KD) is a technique aimed at transferring knowledge from a large and complex model (i.e., teacher model) to a smaller and simpler model (i.e., student model).

- Retains the knowledge and accuracy of large models while reducing computation. The method preserves the comprehensive understanding and high accuracy characteristic of larger language models, but it simultaneously reduces the computational demands, enabling efficient processing and faster response times.
- Ideal for deploying AI-powered chatbots on low-resource environments, like smartphones or smaller computers. The reduced computational requirements make this approach highly suitable for deploying AI-driven chatbots in environments with limited resources, such as mobile devices, embedded systems, or platforms with constrained processing capabilities.

Example: Distilling GPT-3 into a smaller model enables faster chatbot responses with minimal accuracy trade-offs.

E. Backend Optimization

Backend performance optimization is important for the structure of scalable, efficient, and recoilable applications. As the backbone of the system, the backend takes over data processing, business logic, and database interaction. All of these directly affect the user experience. In this article, we will explore some common strategies that allow developers to optimize backend performance

1. Asynchronous Programming
Parallel Processing

- Utilize asynchronous programming techniques to handle I/O-bound operations without blocking the main thread. This allows your application to process multiple requests concurrently and improves responsiveness.

Example: Rasa chatbots use async loops to manage multiple conversations.

2. Load Balancing

Load balancing is like having a traffic controller for your chatbot; it evenly distributes the incoming messages from users across several computer servers. This prevents any single server from getting overwhelmed and ensures that the chatbot can consistently handle a large number of conversations.

- By spreading the workload, load balancing stops individual servers from crashing or becoming unresponsive when there's a sudden surge in user activity, such as during a popular product launch or a customer service rush.
- Load balancing makes the chatbot more consistently accessible to users. Even if one server experiences a problem, others can continue to handle the conversation, minimizing downtime and ensuring that users can always interact with the chatbot..

*Example:*Using Nginx or AWS Load Balancer to distribute chatbot requests evenly across backend servers.

3. Optimized API Calls

Optimized API calls are crucial for enhancing chatbot efficiency by minimizing unnecessary requests and employing batch processing techniques. This strategy reduces data transfer latency, leading to faster response times and a more seamless user experience. Furthermore, by decreasing the number of API calls, this optimization helps to significantly lower API call costs, particularly in cloud-hosted chatbot deployments, contributing to cost-effectiveness and resource conservation.
Example: Instead of making multiple API calls for each message, chatbots can use batch API requests to fetch responses in bulk.

4. Caching for Instant Responses

Caching is a technique employed to enhance chatbot efficiency by storing frequently used responses, thereby avoiding redundant computations and enabling instant replies to common user queries. This process significantly reduces server load and improves response times. Various caching strategies can be implemented, including memory caching (utilizing systems like Redis or Memcached for the fastest access to frequently needed data), response caching (specifically storing and reusing the chatbot's output for similar inputs), and database query caching (saving the results of computationally intensive database operations).
Example: A customer support chatbot caches the response for "What are your business hours?" instead of reprocessing it every time.

5. Microservices Architecture

Break down your monolithic application into smaller, independent microservices. This approach allows you to scale individual components independently and optimize them based on their specific requirements. Microservices can also improve fault isolation and reduce the impact of failures.

VII. Evaluation and Performance Analysis

A. Testing Methodology

We have studied that the performance of RASA-based AI chatbots was evaluated using a combination of user-based and automated testing strategies. The user-based test included 20 participants who interacted with the chatbot over a two-day period and generated over 850 statements in 230 conversation sessions. These conversations included the intentions of different users. B. Pricing, support, FAQs and product information inquiries developed in both official and informal languages. Feedback was collected by surveys after the session to monitor real-time interactions. At the same time, automated tests were performed on RASA's integrated testing framework. For evaluation of NLU ratings (natural language understanding), the RASA test NLU command was used in 5x cross-validation to identify classification accuracy and intent confusion. For Dialog-Flow Tests, RASA Test Core is used using a 60-storey test suite to ensure reports on primary conversation paths and fallback handling[12].

B. Performance Metrics

Quantitative evaluation of the chatbot revealed strong performance across several key metrics:

- **Intent Classification:** The chatbot achieved an intent classification accuracy of **92.4%**, with a **recall of 0.91** and **precision of 0.90**, indicating reliable interpretation of user queries.
- **Misclassification Analysis:** Most errors occurred between closely related intents such as `Product_Price` and `Product_Discount`, highlighting the importance of clearly separated training data.
- **Dialogue Management:** Using RASA Core, interaction accuracy was measured at **88.3%**. A fallback mechanism was triggered when prediction confidence was low, ensuring robustness in uncertain situations.
- **Latency Metrics:** Stress testing via Latnous Metrics showed the **95th percentile response time** at **1.8 seconds**, with an average of **1.3 seconds**, demonstrating responsive performance under concurrent usage.
- **User Satisfaction:** A satisfaction score of **4.2 out of 5** was recorded, showing user approval of conversation flow and language understanding. Areas noted for improvement included memory continuity and handling ambiguous queries.

Evaluation Method

To test the chatbot's **Natural Language Understanding (NLU)** capabilities, **5-fold cross-validation** was conducted using the RASA CLI:

```
rasa test nlu --nlu data/test_data.yml
```

This approach generated reliable metrics and helped identify overfitting or intent confusion, leading to improved training data distribution. For instance, intent expansion for `ask_price` improved generalization:

```
- intent: ask_price
  examples: |
    - What's the cost of this?
    - How much does this product cost?
    - Can you tell me the pricing?
```

Entity Recognition Enhancements

RASA's support for **lookup tables** and **regular expressions** was leveraged to enhance entity extraction:

```
- lookup: city
  examples: |
    - New York
    - Paris
    - Tokyo

- regex: phone
  examples: |
    - [0-9]{10}
```

These patterns improved extraction accuracy for structured data such as cities and phone numbers.

Fallback and Ambiguity Handling

Fallback strategies were configured to handle low-confidence predictions:

```
policies:
  - name: FallbackClassifier
    threshold: 0.3
    ambiguity_threshold: 0.1
```

This setup activates fallback when top intent confidence is below 0.3 or when two intents have similar confidence levels (difference ¡ 0.1), reducing false positives.

FAQ Optimization and Response Acceleration

To simplify NLU workload, FAQs were grouped under a single `faq` intent. RASA's **Answer Selector** was used to match predefined questions with appropriate answers efficiently.

Asynchronous Actions

Asynchronous programming was utilized to fetch external data without blocking other operations, ensuring quick responses:

```
class ActionGetWeather(Action):
    async def run(self, dispatcher, tracker, domain):
        city = tracker.get_slot("city")
        weather = await fetch_weather_async(city)
        dispatcher.utter_message(text=f"The weather
        in {city} is {weather}.")
        return []
```

This method supported simultaneous user interactions, maintaining an average response time of 1.3 seconds.

C. Comparative Analysis

Compared to traditional rule-based chatbots, RASA-based implementations showed superior adaptability, natural language handling, and scalability. Standards-based models had to fight variations in user phrases and were unable to generalize similar queries. In contrast, RASA chatbots were able to accurately interpret various user inputs and maintain a flexible conversational state. However, the RASA model required greater effort, particularly when preparing training data and pipeline configurations. Machine learning models had a stronger generalization, but were also sensitive to the weight of the handles of the class, requiring careful fallback planning. Ultimately, Rasa Chatbots has proven to be more capable in dynamic and real-world scenarios, but they have benefited greatly from ongoing training and structured performance optimization.[4]

VIII. Challenges Encountered

Despite the significant advancements made in AI-powered chatbot development, several practical challenges continue to impact the effectiveness, usability, and scalability of these systems. During the course of implementing a generalized

chatbot using RASA and integrating LLMs, the following challenges were encountered:

Handling Ambiguous User Queries

One of the most persistent challenges is the chatbot's difficulty in interpreting ambiguous queries. Users often express their intent in incomplete or vague ways, expecting the system to "understand" based on context. For example, a user might say, "Can I change it?" without specifying what "it" refers to. While humans effortlessly interpret such queries based on previous context, chatbots struggle unless properly trained to carry dialogue memory. Even with tools like RASA's conversation tracker and contextual features, achieving consistent and accurate intent recognition in such cases remains a complex task.

Imbalanced Training data

Another challenge arises from limited and imbalanced training data. In many cases, specific user intents—particularly those tied to niche domains—do not have enough training examples. This data sparsity causes the model to underperform or fail altogether when handling less common queries. Preparing balanced and representative training sets requires significant effort in terms of time, domain expertise, and manual annotation, making the development cycle longer and more resource-intensive.

challenges in scaling

As user traffic increases, ensuring scalability without performance degradation becomes a critical issue. Even though asynchronous programming and load balancing were implemented, the response time of the chatbot occasionally lagged when handling multiple conversations concurrently. This challenge becomes even more pronounced when LLMs are invoked to handle out-of-scope queries, since such models demand high computational resources and introduce latency.[7]

Inconsistent API Responses and Rate Limits

The integration of external LLMs via APIs introduces a dependency on third-party services. These APIs often come with strict rate limits, usage caps, or varying reliability depending onnetwork conditions. As a result, the chatbot may fail to respond, or deliver delayed responses, when the API is overwhelmed or unavailable. This inconsistency affects the user experience and limits the bot's reliability during peak usage.We have used GROQ API for llama LLM from meta AI that has certain limitations on usage.

Inadequate Context Retention Over Long Conversations

Although RASA allows for tracking dialogue state, maintaining long-term memory across multiple sessions or deeply contextual interactions is still a challenge. The chatbot may lose track of earlier parts of a conversation, especially if users take unexpected turns in the dialogue. This lack of persistent memory makes it difficult to deliver truly natural, human-like interactions over extended sessions.

Complexity in Debugging and Maintaining Custom Actions

With the addition of LLMs and custom logic, the codebase behind the chatbot becomes more complex. Debugging issues related to intent misclassification, failed actions, or missing responses can be time-consuming. Developers must trace the interaction across several components—from NLU pipelines to custom actions and fallback policies—making maintenance and troubleshooting a non-trivial task.

IX. Result

The development and deployment of the generalized AI-powered chatbot yielded encouraging results that highlight both the technical strength and real-world applicability of the system. The integration of RASA with an external Large Language Model (LLM) proved especially effective in bridging the gap between rule-based predictability and AI-driven adaptability.

One of the most significant improvements observed was in intent recognition accuracy, which increased by approximately 15–20pipeline. The chatbot was able to handle a wide range of user intents—including ambiguous or loosely phrased queries—with greater confidence and consistency. This accuracy was essential in making conversations smoother, reducing the need for repeated clarifications, and improving user satisfaction overall.

In terms of system performance, optimizations such as asynchronous processing, response caching, and fallback handling helped reduce average response time to just 1.3 seconds, even under high concurrency. Users reported minimal lag and appreciated the system's responsiveness. This reduction in latency was vital in enhancing the conversational flow and making the chatbot feel more intuitive and human-like

Scalability was another area of improvement. By employing load balancing and horizontal scaling strategies, the chatbot maintained its responsiveness even during stress testing with simultaneous user sessions. This demonstrated that the system could effectively support real world applications with varying loads and maintain quality service without degradation.

The LLM integration also proved valuable, particularly in handling out-of-scope queries. Where traditional systems would typically fail or fall back to generic responses, the chatbot was able to provide intelligent, relevant answers with the help of LLMs, preserving user engagement even in edge cases. This not only increased the chatbot's utility but also helped build user trust and reduced frustration in interactions.

Beyond raw metrics, the overall user experience showed clear improvement. Feedback collected from test participants reflected positively on the bot's ability to understand intent, follow conversational flow, and provide helpful responses. Users particularly appreciated the chatbot's ability to handle both FAQ-style queries and more open-ended questions with equal ease.

In summary, the results of this research affirm that combining RASA with LLMs leads to a well balanced, intelligent chatbot capable of handling dynamic conversations in real time. With further refinement and enhancements—such as

memory persistence, multilingual support, and tighter LLM integration—this chatbot can evolve into a powerful tool ready for deployment across industries and use cases

X. Future Scope Enhancements

As AI-powered chatbots continue to evolve, there is significant potential for enhancing their capabilities beyond their current limitations. One of the most promising areas for improvement lies in developing long-term memory within chatbots. Currently, most conversational systems—including the one implemented in this study—handle each session independently, lacking the ability to recall past user interactions over time. Introducing persistent memory would allow chatbots to build user profiles, maintain context across sessions, and provide more personalized experiences.

Another key area of development is the integration of advanced external knowledge bases and real-time data sources. By enabling the chatbot to query structured databases, access live information feeds, or tap into domain-specific repositories, it could offer more relevant and up-to-date answers. This would be especially valuable for applications in sectors like healthcare, finance, or customer support, where accurate and dynamic information is crucial.

In addition, future chatbots can be made more inclusive and multilingual, breaking language barriers by supporting conversations in various regional languages. By incorporating multilingual NLP models and translation services, the system can become accessible to a broader user base, especially in linguistically diverse regions.

There is also great potential in developing multi-modal chatbot interfaces. Most current systems rely solely on text, but the inclusion of voice, image, and even gesture-based input could create a more natural and engaging interaction. For instance, voice-enabled chatbots could better assist users in hands-free environments, while image recognition could allow for more contextual responses in applications like tech support or e-commerce.

From a learning standpoint, integrating reinforcement learning techniques could make chatbots smarter over time. Instead of relying only on supervised training data, the system would continuously adapt based on feedback from real user interactions. This allows the chatbot to self-improve, refine its decision-making strategies, and reduce reliance on frequent manual retraining.

Lastly, improving scalability and deployment strategies remains a crucial area. Future work can focus on using edge computing and container orchestration platforms to make chatbots faster and more reliable, even in low-bandwidth environments. This will ensure that AI chatbots are not only intelligent but also widely accessible, resilient, and ready for enterprise-level applications.

XI. Conclusion

The development of AI-powered chatbots represents a transformative shift in how humans interact with machines, especially within the context of web-based applications. This research presented a generalized chatbot architecture built using RASA Open Source, enhanced by the integration of external Large Language Models (LLMs), forming a hybrid system capable of handling both predefined and out-of-scope queries.

The study explored the historical evolution of chatbots, highlighting the transition from rule based models to modern AI-driven systems. A comprehensive review of related work demonstrated how frameworks like RASA have gained traction due to their open-source nature, flexibility, and support for machine learning-based dialogue management. The design and implementation section documented a practical workflow—from NLU pipeline setup and training to custom action creation and LLM API integration—showing how chatbots can be made both context-aware and responsive in real-time environments.

Through extensive performance evaluations, the proposed chatbot demonstrated high intent classification accuracy (92.4validating its effectiveness in real-world settings. The research also acknowledged several challenges, such as ambiguity in user queries, data limitations, API rate constraints, and scalability concerns. These are crucial areas that must be addressed to make chatbot systems more robust and production-ready.

Looking ahead, the potential for innovation in conversational AI is vast. Enhancements like long term memory, multilingual support, multi-modal interaction, and reinforcement learning stand to elevate user experiences to new heights. As AI models become more efficient and cloud deployment matures, the future of intelligent, scalable, and accessible chatbot solutions looks increasingly promising. This work contributes a meaningful step toward that future, offering a foundational blueprint for developers and researchers aiming to build adaptive, intelligent, and user-centric chatbots

XII. References

References

[1] Casheekar, A., Lahiri, A., Rath, K., Prabhakar, K. S., & Srinivasan, K. (2024). A contemporary review on chatbots, AI-powered virtual conversational agents, ChatGPT: Applications, open challenges and future research directions. *Computer Science Review*, 52, 100632. https://doi.org/10.1016/j.cosrev.2024.100632

[2] Zhou, L., et al. (2020). The design and implementation of Xiaoice, an empathetic social chatbot. *Computational Linguistics*, 46(1), 53–93.

[3] Wikipedia contributors. (n.d.). LaMDA. https://en.wikipedia.org/wiki/LaMDA

[4] Bocklisch, T., Faulkner, J., Pawlowski, N., & Nichol, A. (2017). Rasa: Open source language understanding and dialogue management. *arXiv preprint arXiv:1712.05181*.

[5] Thorat, S. A., & Jadhav, V. (2020). A Review on Implementation Issues of Rule-based Chatbot Systems. In *Proceedings of the International Conference on Innovative Computing & Communications (ICICC) 2020*. https://ssrn.com/abstract=3567047

[6] Schmid, U. (2016). How does IBM Watson understand natural language.

[7] Mechkaroska, D., Domazet, E., Feta, A., & Shikoska, U. (2024). Architectural Scalability of Conversational Chatbot: The Case of ChatGPT. In *Artificial Intelligence Applications and Innovations* (pp. 54–71). Springer. https://doi.org/10.1007/978-3-031-53960-2_5

[8] Sharma, R. (2020). An Analytical Study and Review of open source Chatbot framework, Rasa. *International Journal of Engineering Research and Technology*, 9. https://doi.org/10.17577/IJERTV9IS060723

[9] Zhu, X., Li, J., Liu, Y., Ma, C., & Wang, W. (2024). A Survey on Model Compression for Large Language Models. *Transactions of the Association for Computational Linguistics*, 12, 1556–1577. https://doi.org/10.1162/tacl_a_00704

[10] A Conversation-Driven Approach for Chatbot Management. *IEEE Access*, January 2022, PP(99):1–1. https://doi.org/10.1109/ACCESS.2022.3143323

[11] Gartner. (2020). The Edge Completes the Cloud: A Gartner Trend Insight Report. Retrieved from https://www.gartner.com

[12] Ramezani, P., & Safari, M. (2023). Evaluating the effectiveness of open-source chatbots for customer support. *Farda Paper*, 1–22.

[13] Huang, Y., Zhou, L., & Deng, H. (2021). A Performance Evaluation Framework for Conversational AI Systems. *Journal of Intelligent Information Systems*, 57(3), 489–506.

[14] Vlasov, V., & Papangelis, A. (2020). Dialogue System Technology with RASA: Open Source and Performance-Oriented. In *Proceedings of the 28th Conference on Computational Linguistics (COLING)*, 1905–1915.