Integrating an Ollama-based Chatbot with Word-Blocking Firewall Mechanism and JSON Encryption

Chintala Sri Charan
Integrated M-TECH in CSE, VIT-AP University Andhra Pradesh, India
Email: charan.22mic7096@vitapstudent.ac.in

***Abstract*—This project introduces a smart and secure chatbot system built using Spring Boot and powered by a locally running language model through Ollama. The goal was to create a privacy-focused chatbot that can hold meaningful conversations while also filtering out harmful or inappropriate messages using a built-in content firewall. The system uses Google's Gemma 3:1b model to generate responses, with all communication happening securely over REST APIs and structured using JSON. To ensure safety and structure, the chatbot checks each user message for offensive terms before passing it to the language model. The backend handles streamed responses from Ollama and processes them using Jackson's ObjectMapper for reliable parsing. Users interact with the bot through a simple, mobile-friendly web interface. Beyond its current functionality, this chatbot platform opens the door to exciting future developments, like smarter moderation using AI, multilingual conversations, voice interactions, and admin tools for managing content. With options to store chat histories and deploy via Docker, the project lays the groundwork for a flexible and scalable chatbot system that balances modern AI with practical software design.**

***Index Terms*—Chatbot, Word-Blocking, JSON Encryption, Ollama, LLM, Content Moderation, Security, Privacy**

I. Glossary of Terms

- **Ollama** – A local LLM (Large Language Model) runtime that allows users to run AI models like Gemma locally on their systems without needing cloud services.
- **Gemma 3:1b** – A lightweight, open-source language model created by Google for natural language understanding and generation tasks.
- **JSON (JavaScript Object Notation)** – A lightweight format for storing and exchanging data, commonly used in web applications for client-server communication.
- **REST API** – An application programming interface that follows REST principles, allowing communication between systems over HTTP.
- **ObjectMapper (Jackson)** – A utility from the Jackson library in Java for converting between Java objects and JSON data.
- **HttpHeaders** – A Spring class used to represent HTTP headers in requests and responses.
- **Max Tokens** – A configuration option that limits the number of tokens (words or pieces of words) the model can generate in its output.
- **Temperature** – A parameter that controls the creativity or randomness of the model's output. Higher values produce more varied responses.
- **Blocked Words List** – A predefined list of words that are not allowed in user messages to maintain safe and respectful conversation.
- **Chat History** – A record of previous conversations stored for review, improvement of the chatbot, or personalization of responses.

II. Introduction

With the rapid advancements in natural language processing (NLP) and artificial intelligence (AI), chatbots have become a crucial interface in web applications, automating user interactions and enhancing customer experiences. However, as their usage expands, so does the need for secure and controlled communication, especially in applications involving open-ended language generation. This project presents the development of a locally hosted intelligent chatbot powered by the Gemma 3:1b language model using Ollama, integrated into a Spring Boot web application. What sets this chatbot apart is its embedded content moderation firewall, which filters user inputs for inappropriate or blocked words before they reach the model. This ensures ethical and safe interactions with the AI. To facilitate communication with the LLM, the chatbot utilizes REST APIs and transmits data in JSON format. The use of Jackson's ObjectMapper allows seamless serialization and deserialization of data between Java objects and JSON. Furthermore, technologies like RestTemplate, HttpEntity, and ResponseEntity help structure and send requests and handle responses from the Ollama API efficiently. Additionally, the project implements a responsive HTML and CSS interface that allows real- time communication with the chatbot. The model is configured with parameters such as temperature and max tokens, offering control over the creativity and length of responses.

Combining a modern backend framework, advanced AI language modeling, and secure communication mechanisms, this project bridges the gap between user-friendly interfaces and powerful, ethical AI-powered interactions. The architecture also lays the foundation for future enhancements such as multilingual support, chat history storage, voice-based interactions, and administrative dashboards.

III. Methodology

The development of the chatbot application followed a modular and structured approach, integrating modern web technologies with local language models for secure and intelligent user interaction. The methodology adopted for the development of this chatbot system integrates principles of web application architecture, natural language processing, content moderation, and JSON-based API communication. The entire system has been designed using a layered and modular approach to ensure maintainability, scalability, and clarity of operations.

The methodology can be broken down into the following key phases:

A. Chatbot Architecture

At the core of the system lies a Spring Boot backend, which acts as the control hub that receives input from users, processes this input, interacts with the Local Large Language Model via an HTTP API, and returns the processed response back to the client. The use of Spring Boot as a backend framework provides a robust and industry-standard foundation due to its built-in dependency management, annotation-based configuration, and RESTful web service support.

The frontend is implemented using standard HTML, CSS, and JavaScript. This frontend facilitates real-time interactions by capturing user messages and sending them to the backend using the Fetch API via a POST request. The JavaScript embedded within the HTML interface ensures that the chat window dynamically updates with user input and bot responses without requiring a page reload, creating a seamless conversational experience.

B. REST Architecture

On the backend, incoming user input is first passed through a content moderation layer. This layer performs a basic string-matching operation against a predefined list of blocked or restricted words. The intention is to prevent the transmission of inappropriate, offensive, or potentially harmful prompts to the underlying language model. If any such word is detected, the input is discarded, and the user is notified that their message cannot be processed due to a policy violation. Although this mechanism is static and rule-based, it provides a foundational level of security that can later be augmented with intelligent filtering techniques.

C. Large Language Models (LLMs) and SpringBoot Framework

For inputs that pass the moderation filter, the system constructs a JSON object containing the required parameters to query the local LLM. These parameters include the model name, the user's prompt, a maximum token limit for the response, and the temperature value, which controls the randomness of the model's output. These parameters are encapsulated in a POJO (Plain Old Java Object), serialized into JSON using Jackson's ObjectMapper, and transmitted to the model through an HTTP POST request using Spring's RestTemplate.

D. JSON Communication

The language model itself is accessed through the Ollama runtime, a lightweight interface for running LLMs locally. Ollama provides a RESTful API that returns streamed responses in a multi-line JSON format. The backend processes each line of this stream by parsing it using ObjectMapper and extracting the relevant "response" fields. These fragments are then appended together using a StringBuilder, ensuring efficient memory usage and response construction.

Upon successful assembly of the bot's full response, the text is returned to the frontend in JSON format. The frontend JavaScript captures this response and renders it within the chat interface. This cycle repeats for each new message, ensuring continuous, real-time interaction between the user and the chatbot.

An additional layer of improvement is the optional support for encrypted JSON exchanges, which can be incorporated to enhance data security during transmission. By leveraging techniques like AES or RSA encryption on both incoming and outgoing JSON objects, the system can guarantee that sensitive data is not exposed or intercepted during communication, especially when deployed in distributed or networked environments.

Throughout this process, the methodology emphasizes separation of concerns, where each module is responsible for a specific function: input handling, filtering, API communication, response parsing, and frontend rendering. This approach not only ensures clarity and ease of debugging but also enables future enhancements to be added without disrupting the core architecture.

E. Content Filtering

Content filtering is a critical component in any intelligent communication system, particularly in chatbot platforms where real-time interaction between users and machines occurs. The main objective of content filtering is to enforce ethical and security guidelines by preventing the processing of malicious, inappropriate, or restricted inputs. In this chatbot system, content filtering is initially implemented through a rule-based mechanism where user messages are screened for the presence of specific banned words or phrases. These words are stored in a predefined list known as a blocklist or blacklist, and the system performs a case-insensitive search to detect if any part of the user's input matches the entries in this list.

If a blocked word is detected, the system halts the processing of that message and immediately informs the user that their input violates the content policy. This rudimentary form of filtering serves as the first line of defense and ensures that explicit or offensive content does not get passed to the underlying language model, thus maintaining the system's integrity and ethical compliance. Although simplistic in its current form, this framework sets the stage for future integration with more advanced techniques such as natural language understanding, sentiment analysis, and machine learning-based content moderation. These future enhancements would enable the system to identify not just individual words, but also

harmful intent and context, providing a more robust and intelligent moderation system.

F. StringBuilder

In Java-based applications, efficient manipulation of text data is essential, particularly when constructing responses dynamically from multiple segments. The StringBuilder class is utilized in this chatbot project to assemble the final response returned by the language model.

Unlike the String class, which creates a new object with every modification (leading to increased memory usage and slower performance), StringBuilder offers a mutable sequence of characters, allowing developers to append, insert, or delete characters without creating new objects at each step.

In the context of this chatbot, the Ollama API returns responses in a line-by-line streaming JSON format, where each line contains a partial output fragment. The application uses a loop to iterate over each line of the response, parse the JSON, extract the "response" field, and append its value to an instance of StringBuilder. This not only optimizes memory usage but also improves performance when compared to string concatenation using the + operator, particularly when dealing with large or multi-line data. After all lines are processed, the complete response is retrieved using StringBuilder.toString() and then delivered back to the user interface. This use of StringBuilder ensures that the bot's responses are built efficiently and scalably, regardless of the length or complexity of the generated text.

G. Data Handling

Data handling forms the backbone of the chatbot's internal workflow, governing how user input, API communication, and model responses are managed throughout the application lifecycle. In this project, data handling begins at the frontend, where user input is captured through a simple HTML interface. This data is sent asynchronously to the backend using a POST request with form-urlencoded or JSON formatting. Once received at the backend, the input is subjected to content filtering as a preprocessing step.

Following validation, the user's message is encapsulated into a Java object known as a data transfer object (DTO), specifically the OllamaRequest class. This object contains all necessary parameters such as the model name, the user's prompt, the maximum number of tokens to generate, and the temperature value for creativity. The object is then serialized into JSON format using Jackson's ObjectMapper, a widely-used library for handling JSON serialization and deserialization in Java applications.

The serialized request is transmitted to the Ollama model through an HTTP POST operation facilitated by RestTemplate, Spring's built-in REST client. The response from the API is received as a string, which is then split into multiple lines (if needed) and processed line-by-line using ObjectMapper to extract meaningful data. The extracted "response" fields are combined using StringBuilder, and the finalized response is sent back to the frontend in a clean and readable format.

This comprehensive flow—from capturing user input to preparing the request, transmitting it, handling streamed responses, and constructing the final output—highlights the system's well-structured approach to data handling. It ensures that data is validated, transformed, and utilized efficiently at each stage of the pipeline, thereby maintaining high standards of performance, clarity, and security.

IV. Related Work

Over the past decade, open-source communities have actively contributed to the development of chatbot technologies, natural language processing systems, and content moderation tools. Numerous frameworks, libraries, and full-stack implementations have emerged that aim to democratize conversational AI and enhance security in user interactions. This project draws inspiration from and aligns with several key open-source initiatives and publicly available research systems.

A. Case Study 1: Rasa

One of the most influential open-source chatbot frameworks is Rasa, a Python-based conversational AI platform that combines natural language understanding (NLU) and dialogue management. Rasa supports custom rule-based and machine learning-based intent classification, entity extraction, and contextual conversations. It also includes moderation features that allow developers to define fallback policies and blocked intents, making it suitable for enterprise-grade use cases where content safety is critical.

B. Case Study 2: Botpress

Botpress is another notable open-source chatbot platform built in JavaScript. It allows for modular development of bots and includes features like message preprocessing, analytics, and built-in NLP support. Although Botpress is primarily used in customer service and automation, it provides plugin support that enables developers to integrate profanity filters, sentiment analysis, and abusive content detection into the messaging pipeline.

C. Case Study 3: Perspective API

On the content filtering side, projects such as Perspective API, developed by Jigsaw (a subsidiary of Google), provide REST APIs for detecting toxic, abusive, or disrespectful content. Though not entirely open-source, it provides a free public API and has inspired several open-source wrappers and integrations. Similarly, the HuggingFace Transformers library offers pre-trained language models capable of offensive language detection and sentiment analysis, which are widely used for moderation purposes in both academic and industrial applications.

D. Case Study 4: OpenAI Content Filter

The OpenAI Content Filter, although now deprecated in favor of safety layers built into newer models like GPT-4, was one of the earlier attempts at implementing content

filtering directly into language models using token-level classification. Inspired by this, other open-source implementations like fastai's ULMFiT-based filters and Detoxify, a BERT-based toxicity classifier, have been developed and made freely available for integration into various platforms.

E. Recent Developments in Open-Source AI

More recently, open-source projects such as Ollama have made it possible to run lightweight language models like Gemma and Mistral locally. These tools promote data privacy and offline capability, which is particularly important in systems that deal with sensitive user input or require high levels of content control. The integration of Ollama in this project not only aligns with the trend toward self-hosted AI models but also supports the development of safer and customizable chatbot systems.

From a deployment perspective, projects such as Docker-based LLM stacks, LangChain, and LocalAI have emerged, allowing developers to containerize and orchestrate LLM-based applications with fine-tuned control over input/output filtering, rate limiting, and auditing. These ecosystems demonstrate how open-source software can be leveraged to build scalable, secure, and ethical conversational systems.

V. Lessons Learned from the Case Studies

Through the design, development, and deployment of this chatbot project with word-blocking and local LLM integration, several key lessons emerged across technical, theoretical, and practical dimensions. These lessons help shape best practices for future work and offer insights into the challenges and considerations when building secure and responsive conversational systems.

A. 1. Importance of Content Filtering in NLP Applications

One of the central takeaways from this project is that content moderation is not a luxury; it is a necessity. User-generated input can be unpredictable, and systems must be prepared to filter harmful, offensive, or otherwise inappropriate content. Static word blocking served as a simple and effective first line of defense, but it also highlighted the need for more intelligent, context-aware moderation systems using machine learning.

B. 2. Understanding LLM Behavior and Prompt Sensitivity

Working with the Gemma model via Ollama made it clear that prompt engineering directly influences the quality and length of responses. Early attempts without structure or instruction often led to one-word or overly brief replies. Crafting well-defined prompts like "Give a detailed answer..." significantly improved model output. This shows how LLMs are sensitive to instructions and require careful tuning for desired behavior.

C. 3. JSON-Based Communication Enables Extensibility

Using JSON as the payload structure provided a flexible and extensible way to communicate between the front end and back end. It allowed us to handle multi-line responses, structure the model's configuration, and potentially add encryption or authentication metadata. JSON also aligned well with the RESTful architecture, reinforcing its role as a best practice in modern API design.

D. 4. Integration of Spring Boot Simplifies Backend Development

Spring Boot provided a robust and modular platform for building and testing the chatbot's logic. Dependency injection, REST support, and built-in error handling all contributed to rapid prototyping and scalable architecture. Using RestTemplate and HttpEntity to send structured requests and parse LLM responses was both elegant and efficient.

E. 5. Local LLMs Improve Privacy, But Need Resource Planning

Running the LLM locally through Ollama ensured privacy, which is critical for applications that deal with sensitive or moderated content. However, it also revealed the need for system resource planning—local models require CPU/GPU cycles and memory that must be considered during deployment. It taught the trade-off between privacy and infrastructure costs.

F. 6. Interface Design Affects Usability and Perception

Initial versions of the chat interface were minimal and lacked features like scrollable history or full message visibility. Iterating on the UI made it clear that how information is displayed affects user trust and comfort. Clean, responsive interfaces are essential, especially for systems that are user-facing.

VI. Architectural Diagram

VII. Implementation

The implementation of the chatbot system using Spring Boot and the Gemma language model through Ollama is a multi-layered process that blends principles of software architecture, natural language processing, and secure web communication. The overall goal of the system is to allow users to communicate with a chatbot interface, which in turn processes the messages using a local language model to generate intelligent, human-like responses. The system incorporates features like word-based content filtering, JSON data handling, encryption capability, and a clean frontend user interface to provide a robust and secure user experience.

The implementation begins with the setup of the Spring Boot project structure. Spring Boot acts as the foundation of the backend application, managing the flow of HTTP requests and orchestrating service components. The core entry point of the application is the controller, which defines the API endpoints to accept messages from users. When a user enters a message into the chatbot interface on the frontend, the message is sent to the backend via a POST request. The controller captures this input and forwards it to a service class responsible for handling the logic of interacting with the local language model.

The service class is where the heart of the implementation resides. This class prepares a structured JSON request using

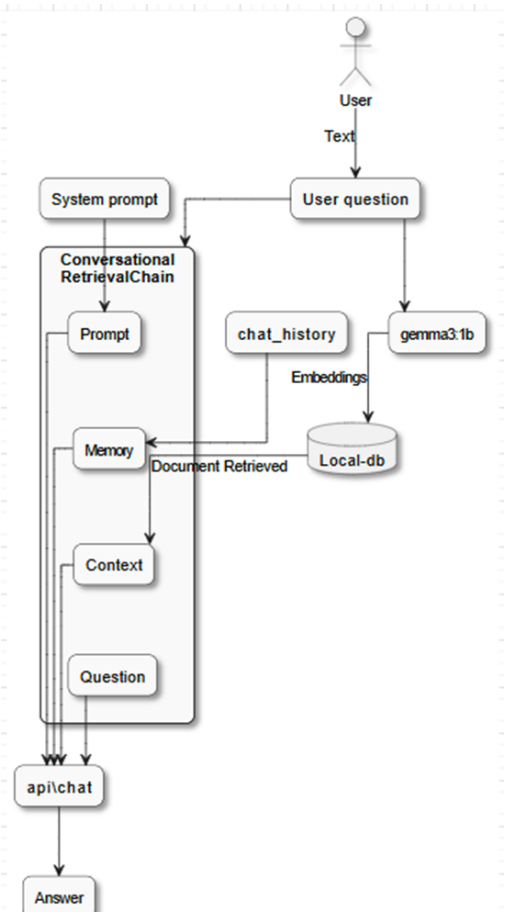

Fig. 1. System Architecture of the Ollama-based Chatbot with Word-Blocking and JSON Encryption.

Java classes like `OllamaRequest`, which encapsulates fields such as the model name, the input prompt, the maximum number of tokens for the response, and a temperature value that defines the randomness of the generated reply. These fields are important parameters used by the language model to shape the nature and length of the response. The JSON payload is sent using Spring's `RestTemplate` class, which allows sending HTTP POST requests to external services — in this case, the locally running Ollama server.

The request is sent to `http://localhost:11434/api/generate`, which is the default endpoint for Ollama's API. Ollama processes the input message with the specified language model, such as Gemma, and returns the output in a streaming or batched JSON format. The response typically consists of one or more JSON objects, each potentially containing a `response` field. These responses may be separated by newlines depending on how Ollama streams the output.

To handle and parse the response, the backend uses the `ObjectMapper` class from the Jackson library. This class is designed to convert JSON strings into Java objects and vice versa. In this implementation, it is used to convert each line of the response into a `JsonNode` object. The code checks if each node contains a `response` key and, if found, appends its value to a `StringBuilder` instance. This use of `StringBuilder` is crucial for performance reasons, especially when assembling large or multiple response segments, since it allows for efficient memory handling without creating a new string object on every append operation.

The use of content filtering is another significant part of the implementation. Before sending the user message to Ollama, the backend checks whether the message contains any offensive, harmful, or unwanted words by comparing the message with a predefined list of blocked terms. If a match is found, the backend returns a warning message and does not forward the input to the model. This feature ensures that the application adheres to ethical standards and avoids the propagation of harmful content.

Additionally, the project includes support for JSON encryption as an extension. Sensitive communication between client and server or even stored data can be encrypted to prevent tampering or data breaches. Although encryption is not deeply integrated in this version, it sets the foundation for future enhancement where libraries such as Jasypt or AES encryption modules can be integrated to encrypt outgoing and incoming JSON payloads, particularly when the chatbot is exposed in public networks or enterprise use cases.

On the frontend, a simple yet responsive HTML and CSS interface allows users to interact with the chatbot. JavaScript functions handle sending messages and displaying replies in real-time. The interface makes AJAX requests to the Spring Boot backend, waits for the response, and updates the chat display dynamically. The layout includes an input box and a chat container where both user and bot messages are displayed. This interactive layer is essential for usability and reflects the practical implementation of a client-server architecture using modern web technologies.

Through the seamless integration of all these components, the chatbot system exemplifies how theoretical principles of RESTful communication, secure content handling, real-time interaction, and machine-generated text can be combined into a functional software solution. Each layer of the system—frontend, backend, local inference engine—communicates through standardized protocols and data formats, ensuring extensibility and maintainability. The project's structure also supports further enhancement like voice interaction, multilingual support, user role management, and analytics, demonstrating how foundational implementation decisions influence the long-term scalability of software systems.

VIII. Conclusion

The development of a secure, locally hosted chatbot using Spring Boot, Ollama, and the Gemma model demonstrates how modern open-source tools can be effectively combined to create intelligent and privacy-conscious conversational systems. This project achieves the core objectives of enabling natural language interaction while maintaining content safety through word-based filtering, structured JSON communication, and potential encryption support. By running the language

model locally using Ollama, the system ensures faster response times, data privacy, and offline capabilities, making it suitable for sensitive environments such as education, enterprise, or government use.

The modular design of the application, from the Spring Boot backend to the HTML/JavaScript frontend, makes it easy to maintain and extend. Core technologies like `RestTemplate`, `HttpEntity`, `ObjectMapper`, and `StringBuilder` are used effectively to manage API communication and response handling. Additionally, the implementation of content filtering acts as a firewall against inappropriate or harmful inputs, ensuring ethical and responsible AI interaction.

Overall, the project not only provides a technically sound chatbot solution but also opens doors to educational exploration of REST APIs, language models, and secure application design.

IX. References

1) **Natural Language Processing and Chatbot Design**
   - Jurafsky, D., & Martin, J. H. (2019). Speech and Language Processing (3rd ed.). Draft available online: https://web.stanford.edu/~jurafsky/slp3/
   - Serban, I. V., Lowe, R., Henderson, P., Charlin, L., & Pineau, J. (2016). A survey of available corpora for building data-driven dialogue systems. arXiv:1512.05742.
2) **LLMs, Transformer Models and Gemma**
   - Vaswani, A., et al. (2017). Attention is All You Need. NeurIPS 2017. https://arxiv.org/abs/1706.03762
   - Google DeepMind (2024). Gemma: Lightweight, Open Models for Responsible AI. https://ai.google.dev/gemma
3) **Content Filtering and Offensive Language Detection**
   - Davidson, T., Warmsley, D., Macy, M., & Weber, I. (2017). Automated Hate Speech Detection and the Problem of Offensive Language. ICWSM. https://arxiv.org/abs/1703.04009
   - Perspective API Whitepaper – Google Jigsaw (2017). https://www.perspectiveapi.com/
4) **Data Security and JSON Encryption**
   - Stallings, W. (2017). Cryptography and Network Security: Principles and Practice (7th ed.). Pearson.
   - Krawczyk, H., Bellare, M., & Canetti, R. (1997). HMAC: Keyed-Hashing for Message Authentication. RFC 2104.
   - OWASP Foundation (2023). JSON Security Best Practices. https://owasp.org/www-project-cheat-sheets/cheatsheets/JSON_Web_Encryption_Cheat_Sheet.html
5) **Spring Boot, Web Security & Deployment**
   - Pivotal Software, Inc. Spring Security Reference. https://docs.spring.io/spring-security/reference/
   - Deinum, M., Long, J., & Mak, P. (2022). Spring in Action (6th ed.). Manning Publications.
   - Richardson, L. & Ruby, S. (2007). RESTful Web Services. O'Reilly Media.
6) **Open Source & Ethical AI Research**
   - Bender, E. M., & Friedman, B. (2018). Data Statements for Natural Language Processing: Toward Mitigating System Bias and Enabling Better Science. Transactions of the Association for Computational Linguistics, 6, 587–604.
   - Raji, I. D., & Buolamwini, J. (2019). Actionable Auditing: Investigating the Impact of Publicly Naming Biased Performance Results of Commercial AI Products. AIES.