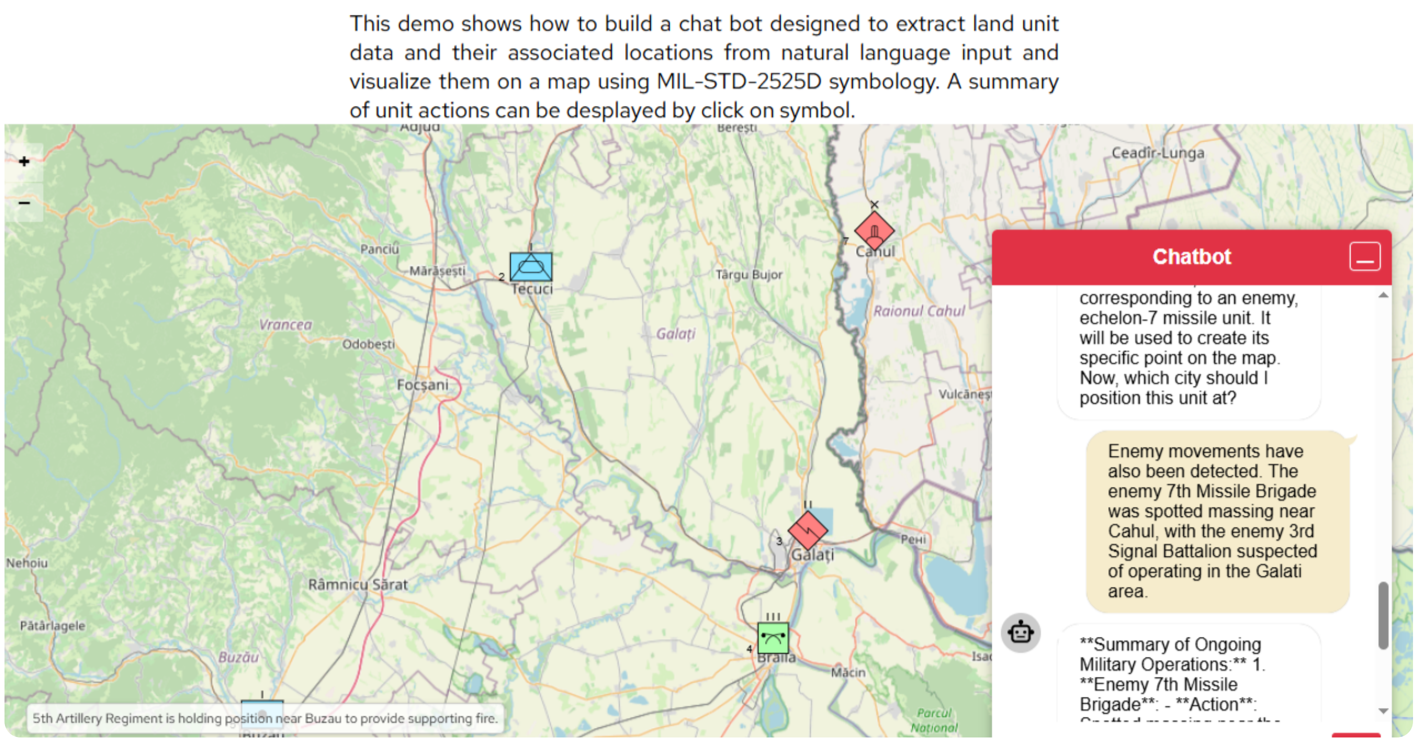
Articles People Learning Jobs Games Get the app


This demo shows how to build a chat bot designed to extract land unit data and their associated locations from natural language input and visualize them on a map using MIL-STD-2525D symbology. A summary of unit actions can be desplayed by click on symbol.

Chatbot

corresponding to an enemy, echelon-7 missile unit. It will be used to create its specific point on the map. Now, which city should I position this unit at?

Enemy movements have also been detected. The enemy 7th Missile Brigade was spotted massing near Cahul, with the enemy 3rd Signal Battalion suspected of operating in the Galati area.

Summary of Ongoing Military Operations: 1. **Enemy 7th Missile Brigade**: - **Action**:

5th Artillery Regiment is holding position near Buzau to provide supporting fire.

Exploring Generative AI for Military C2 Systems: A Practical Approach

Traian Nicula
IT professional
Published May 12, 2025

+ Follow

In recent years, generative AI and large language models (LLMs) have rapidly transformed the way we interact with technology. Their influence is spreading across countless domains, and their full impact is only beginning to unfold. The integration of LLMs into real-world applications—especially through AI agents, tool use, and Retrieval-Augmented Generation (RAG)—has become a major focus in modern software development.

With a background in the military, I have been particularly interested in how generative AI might enhance Command and Control (C2) systems. In this article, I share my experience building a proof-of-concept C2 chatbot designed to extract land units and their associated locations from natural language input. Once a unit is identified, the system derives its 30-character Symbol Identification Code (SIDC), extracts the unit's unique designator, and generates a brief summary of its activities. If a location is mentioned, it is geocoded to obtain its latitude and longitude. All extracted data is visualized on a map using MIL-STD-2525D symbology and returned to the user in structured JSON format.

This project demonstrates how LLMs can serve as a bridge between free-form battlefield reporting and structured, machine-interpretable military data, opening up new possibilities for the next generation of C2 applications.

Background and Motivation

Throughout my career, I have worked extensively in the field of military computer information systems (CIS), as well as in modeling and simulation (M&S) for training and exercises. One of my most significant contributions has been the design, development, and maintenance of a Command and Control (C2) application called ***ForTer PC***, used by the Romanian Land Forces. This system served as an operational tool for nearly six years, supporting land forces personnel in various contexts.

In military exercises and operations, data typically flows in two distinct forms:

- On one side, we deal with highly structured, standardized, and normalized data formats—such as those defined by the APP-11 NATO Message Catalogue or the JC3IEDM data model—commonly used in C2 systems.

- On the other side, we encounter unstructured, informal communications exchanged through channels like chat or email.

Traditionally, bridging the gap between these two types of data has required manual intervention from human operators. This is where generative AI, with its semantic understanding and natural language processing capabilities, offers a transformative potential: automating the conversion of informal, free-text messages into structured, machine-readable formats.

This concept lies at the heart of both this article and the associated GitHub project, *c2_chatbot*. The goal is to demonstrate how modern language models can streamline and enrich data flows within C2 environments, making the systems both more responsive and more intelligent.

As an associate lecturer at the Military Technical Academy of Bucharest—where I teach courses on general-purpose modeling & simulation and programming paradigms in Java—I am also driven to inspire students. I want them to see the real-world power of Java as a language and appreciate the robust, modern ecosystem it offers for building complex, impactful applications.

System Architecture

The application is built using the **Quarkus Framework**, leveraging its powerful extensions to support backend development. Integration with large language models (LLMs) is achieved through the **Quarkus LangChain4j** extension, which offers robust support for AI agents, tools, and Retrieval-Augmented Generation (RAG)—all while remaining developer-friendly and easy to use.

This project relies entirely on an **Azure OpenAI**-hosted language model, which has proven to be both performant and reliable for the use case. Although I have not tested alternative providers, the architecture is built on **Quarkus LangChain4j**, which supports multiple LLM backends. In principle, the system should work with any other LLM supported by LangChain4j—such as OpenAI (non-Azure), Cohere, Hugging Face models, or local LLMs—provided the prompts and tool interfaces remain compatible.

Backend Overview

A simplified UML diagram of the backend architecture is shown below:

The backend UML diagram

To avoid getting too deep into implementation details, I'll provide a high-level overview of the system components and their responsibilities.

The **Chatbot AI Service** acts as the main entry point for user inputs, received via WebSocket from the frontend. Using a carefully crafted prompt, this service analyzes the incoming text to detect references to military land units and their associated locations. If at least one unit is identified, the LLM extracts several key elements:

- the unit type (e.g., Infantry, Armored),
- the unit's unique designator (e.g., 114),
- a short summary of the unit's actions,

and calls an AI tool—**SidcGenAgent**—to generate the **Symbol Identification Code (SIDC)**.

The **SidcGenAgent** orchestrates several internal tools (as illustrated in the diagram) to assemble the full 30-character SIDC. One of the most complex parts of this process is obtaining the **Entity Type Code** (a 6-digit component of the SIDC). This is handled by the **EntityTypeRepository** tool, which queries a local database to retrieve the code associated with the extracted unit type. For example, for "Infantry", the code returned is 121100. This mapping is based on data from the open-source Esri Military Symbology project.

Once the unit is processed, the next step is to geolocate the associated place mentioned in the text. This is handled by the **GeocodingService**, a REST client that queries the GeoNames Search Webservice. If the user has zoomed or panned the map, the bounding box of the visible area is passed as a parameter to narrow the search. The GeoNames API also supports fuzzy matching, which helps resolve misspelled or ambiguous place names.

Finally, with all data gathered and structured, a **JSON message** is created and passed to the **MapUpdater** tool. This component sends the message back to the frontend for rendering the unit and location on the map using **MIL-STD-2525D military symbology**.

Frontend Overview

The frontend for this project was not developed from scratch. Instead, it builds upon the sample frontend code available in the quarkus-langchain4j GitHub project. The interface is built using **Web Components**, developed with the **Lit Framework**, which provides a lightweight and efficient foundation for building modern web applications.

A simplified component UML diagram is shown below:

The frontend component diagram

The core structure includes three main JavaScript modules:

- demo-title.js (lightly modified from the original sample)
- demo-chat.js, responsible for managing user interaction with the chatbot interface
- demo-map.js, the workhorse of the application

The demo-map.js module handles:

- WebSocket communication with the backend
- Dynamic rendering of military symbols on the map
- Real-time map updates based on incoming data

For mapping functionality, the project uses the excellent **OpenLayers** JavaScript library (openlayers.org), which I've successfully used in previous projects. The map is rendered using **OpenStreetMap (OSM)** as the base tile layer, offering a clean and open-source-friendly solution.

To render **military symbology**, the application uses the outstanding MilSymbol library developed by **SpatialIllusions**. This library fully supports **MIL-STD-2525** and provides flexible and accurate symbol rendering directly in the browser, making it a perfect fit for C2-style visualizations.

Lessons Learned

The cornerstone of this project was the **generation of SIDC** codes, which are essential for rendering military symbols. In the beginning, with only a limited understanding of how LLMs work—especially embeddings and Retrieval-Augmented

Generation (RAG)—I assumed RAG would be the right solution for extracting SIDCs from user input.

Naively, I believed that ingesting the MIL-STD-2525D document into a vector database would allow the LLM to reliably retrieve and generate the correct SIDC based on free-text queries. What I got instead were **hallucinations**—lots of them.

With the help of ChatGPT, I experimented with various strategies:

- Adjusting chunk sizes and overlaps
- Using different embedding models
- Adding metadata to vector chunks

Unfortunately, these efforts didn't help. The issue turned out to be more fundamental: the MIL-STD-2525D content is **tabular, decontextualized**, and tightly bound to a **closed semantic domain**. This structure doesn't work well with RAG, which thrives on natural language context and open-ended semantic associations.

Eventually, I realized that RAG was **not** a good fit for my use case. I shifted toward using **tools and structured agent orchestration**, which delivered **significantly better results**.

Another major insight came from **prompt engineering**. Some call it an art, not a science—and my experience confirms this. A poorly designed prompt will yield poor results, no matter how powerful the underlying model. I relied on ChatGPT to refine my prompts, and while it helped, I discovered a subtle trap: if you iteratively improve your prompt without providing fresh context or simplifying structure, you risk creating **overly complex and cluttered prompts** that **perform worse**.

Finally, I learned that **orchestrating many tools and agents**—especially in **LangChain4j**—can quickly become challenging. As the number of tools grows, **response time increases**, and debugging becomes harder. I again used **ChatGPT** to help me **optimize and consolidate tools**, leading to better latency and cleaner logic.

These lessons shaped both the architecture and my personal understanding of working with LLMs in production-grade applications.

Conclusion

This project started as a curiosity: could generative AI—particularly LLMs—be meaningfully applied in the specialized domain of military Command and Control (C2) systems? The result is a working proof-of-concept that shows what's possible when modern AI meets the rigor and specificity of military information systems.

Throughout this journey, I learned some hard but valuable lessons: not every problem is suited to Retrieval-Augmented Generation (RAG), prompt engineering is more art than science, and orchestrating multiple tools and agents can quickly become a performance and complexity challenge. Yet, with the right abstractions, good tooling, and a strong understanding of the domain, it's possible to build powerful AI-driven capabilities.

Being a subject of growing international interest, I am well aware that many researchers and developers are working in this area globally. I sincerely hope that within the **Romanian military** there are also dedicated professionals exploring and advancing this field. This project is my small contribution, and I hope it serves as both a technical reference and an inspiration.

The project is available on GitHub and open for collaboration, experimentation, and improvement. I look forward to seeing where others take it from here.

Acknowledgements

I would like to sincerely acknowledge the contribution of my daughter, Maria Nicula, to this project. She was the one who introduced me to the world of large language models (LLMs) and integration frameworks like LangChain. Our discussions—often deep, technical, and forward-looking—have been both insightful and motivating, helping me shape the direction and architecture of this work.

I also want to recognize the invaluable assistance provided by **ChatGPT** and **GitHub Copilot** throughout the project. From code generation and debugging to prompt engineering and writing support, these tools have played a key role in accelerating development and enhancing the quality of both the application and this article.

#CommandAndControl #C2Systems #LLMToC2Systems

#LangChain4J #Quarkus #OpenLayers

Like Comment Share 10

To view or add a comment, sign in

More articles by Traian Nicula

Jul 31, 2024

Swarm of Drones Simulation Testbed – Part 1

The concept of autonomous drone swarms is gaining significant attention. This project aims to develop an experimental...

11 · 1 Comment

Explore topics

- Sales
- Marketing
- IT Services
- Business Administration
- HR Management
- Engineering
- Soft Skills
- See All