Distributed Agent-Based System for Logical Reasoning from Natural Language using JADE and LLMs

Gabriele Francesco Berruti
gabriele.berruti00@gmail.com

September 14, 2025

Details of the proposal

Proposal Summary

This project develops a distributed Multi-Agent System (MAS) with JADE that enables symbolic reasoning from natural-language inputs. Users express rules, facts, and questions in English; a parser agent uses LLM APIs to translate them into Prolog clauses; a logic agent maintains a knowledge base and executes reasoning; and a query agent manages user questions and responses. The system targets Internet-connected, containerized deployment and scales by running multiple reasoning and query agent instances.

Kind and Difficulty

Kind: Creative
Degree of difficulty: Hard. The integration of an Internet-scale deployment and testing process, Prolog-based symbolic reasoning, LLM-based natural language parsing, and a distributed JADE-based MAS during the course period is reflected in this rating.

System Specification

- **Agents:** UserAgent (user I/O), ParserAgent (LLM-based NL→Prolog translation), LogicAgent (Prolog KB + inference), QueryAgent (query routing and responses)

- **Parsing:** use Java wrappers to invoke OpenAI/OpenRouter-compatible APIs to convert natural language into Prologue clauses and queries, with sanitization and validation.
- **Reasoning:** preserve established knowledge; respond to queries and provide answers; store facts and rules in a Prologue engine that is accessible from Java.
- **Deployment:** package agents as Docker containers; run one or more instances of LogicAgent and QueryAgent across machines/containers; connect agents over the Internet via JADE's main container.
- **Frontend:** A Node.js web user interface is used to submitting facts, rules, and questions and displaying answers.
- **Testing scope:** use ten to twenty representative rules/facts and multiple questions to confirm the accuracy of the translation and reasoning from beginning to end.

Evaluation and Stress Testing

To find bottlenecks, I run several LogicAgent and QueryAgent instances across computers and containers (at least one instance of each type; more if resources permit). In addition to reporting the maximum sustainable configuration and known limits on my hardware, I will measure failure behaviour.

Example Use Case

Academic rule checking:

```
User: "A student can graduate if they have paid the fees and passed all exams."
Parser -> Prolog:
  can_graduate(X) :- paid_fees(X), passed_all_exams(X).

User: "Luca has passed all exams."
User: "Luca has paid the fees."
User: "Can Luca graduate?"

Interpreted as:
  passed_all_exams(luca).
  paid_fees(luca).
  ?- can_graduate(luca).
```

```
LogicAgent -> true
UserAgent -> "Yes, Luca can graduate."
```

Out-of-Scope TODO works and Extensions

Some planned extensions are excluded due to time constraints:

- Mobile/Android deployment: not a primary goal; if time allows, it might be investigated later.
- Port cleanup: deprecate/remove servers bound to ports **4001** and **40005**, if possible, to prevent conflicts when multiple Parser or Logic agents share the same host.
- Federated multi-main topology: manage per-type agent pools with busy/free status across multiple JADE main containers; each main keeps a local registry of agents that are connected to it and communicates with other mains to maintain a consistent knowledge base, creating a distributed network.

Introduction

Project Overview

This project proposal proposes the development of a distributed multi-agent system using the JADE framework that enables symbolic reasoning based on natural language input.
The system will leverage Large Language Models (LLMs) to translate user-provided natural language rules and facts into logical formulas in Prolog (Horn Clauses), and then evaluate them using a reasoning engine distributed across multiple agents connected through the Internet. So the main idea is to first create a system of agents which is distributed possibly all over the internet or in a general a local network in such a way to allow the different agents to communicate .
The extend version of the project contemplate also to implement a GUI which is similar to the one implement by Paul Tarau inside the cited paper which is the following one :
https://deepllm.streamlit.app

Technologies Used

- **Docker**: used for creating an isolated environment for the system, ensuring that all dependencies are managed and the system can be easily deployed on any machine. It is required to use a version greater than 3.9. During the build process, one image will be created per agent (named after the agent) and an additional image named “sdai”.
- **JADE (Java Agent Development Environment)**: for building the distributed multi-agent system. It is used to simplify agent communication via the ACL (Agent Communication Language) .
- **Java 8**: to maintain compatibility with JADE.
- **Prolog engine** : to store and reason over logical formulas, resolve queries and retrive result from fact and relations .
- **LLMs** : to parse natural language into logical clauses. The model is not run locally; access is performed via an API key to connect to an "OpenRouter.ai" service.

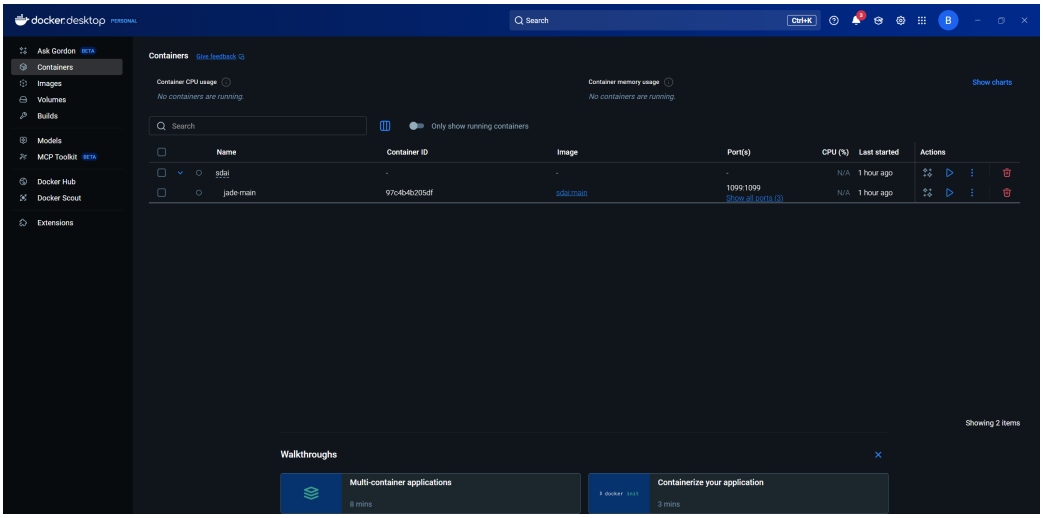

Figure 1: Docker is used for containerized deployment and reproducible environments.

Architecture of the System

The user interface receives the results back, where they are humanized into short, natural language replies.
The main container for JADE, called `jade-main`, is situated in the primary center of the platform.
It hosts the AMS, DF, and message transit protocols and acts as the platform's representative for agent names. While local and remote users connect to this container using the standard RMI registry on port `1099`, the HTTP Message Transport Protocol (HTTP-MTP) is incorporated into the monitor and available on port `7778` for remote containers. Following the local execution of the main container in our deployment, agents can join from the same host or from remote hosts and register using `jade-main`.
A lightweight monitoring UI available at `http://localhost:4100/monitor` periodically probes the main container and displays platform details together with the list of currently connected agents.

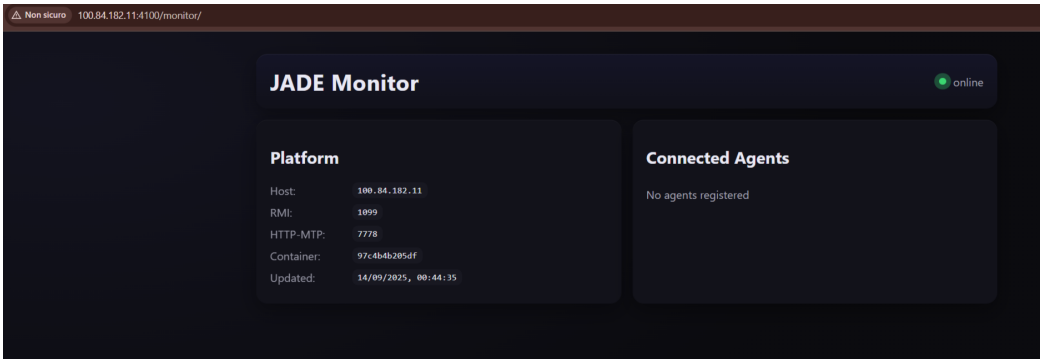

Figure 2: JADE Monitor connected to the local `jade-main` platform (RMI 1099, HTTP-MTP 7778) showing platform status and connected agents.

Containerization and Runtime

The platform is containerized so that both the main node and the agents run from the same Docker image while adopting different roles at runtime. Building either `jade-main` or any agent service uses `docker/Dockerfile`, which installs Ubuntu 22.04, OpenJDK 8, SWI–Prolog with JPL, and Node.js via NVM.
The build process copies the entire `jade/` tree to `/app`, normalizes and makes executable the scripts under `/app/docker`, and exposes ports for the web user interface (`4000`, `5000`, `5001`, `5002`), the monitor (`4100`), and JADE (`1099` for RMI/IMTP and `7778` for HTTP – MTP). By reading the `ROLE` environment variable and delegating to `start_main.sh` or `start_agent.sh`, the container is started via `docker/entrypoint.sh`.
The `docker-compose.yml` file orchestrates roles and wiring. It defines a service for the main platform that publishes `1099`, `7778`, and `4100` and several agent presets that reuse the same image with `ROLE=agent`. Compose profiles let you start only what you need:
`docker compose -profile main up jade-main` launches AMS/DF and the monitor, while `docker compose -profile agent up user-ui` starts an agent container connected to the main by service name and, when a `UserAgent` is present, also brings up the web UI. The distinction between services is driven by environment variables such as `PUBLIC_HOST`, `MAIN_HOST`, and the `AGENTS` string passed to `jade.Boot`, not by different images.

At runtime `start_main.sh` cleans and compiles Java sources under `/app/web-ui/{agents,utils}` into `/app/out`, optionally starts a lightweight monitor on port `4100`, and then boots JADE in main mode with AMS/DF only.

Conversely, `start_agent.sh` performs the same compilation step, sets the knowledge base path, and starts a JADE container connected to the main; if a `UserAgent` is included it also launches the UI server in the background. The folder `Project/JADE-bin-4.6.0/jade/docker/` contains the Dockerfile, the compose files (including a Tailscale variant to connect to remote agents), the entrypoint, and the two role-specific startup scripts that implement this behavior end-to-end.

There is no external database attached to the main container by default; AMS and DF live in memory inside the main JVM. The only on-disk knowledge base shipped with the project is the Prolog file `web-ui/kb/knowledge.pl`, which agents consult and update. To persist it across rebuilds, i would have mount a host volume to that path or to `/app/out` for compiled classes.

Regarding the web UI, `web-ui/server.js` is an Express app that serves the static frontend from `web-ui/public`, exposes local endpoints, sends inputs to `UserAgent` and `QueryAgent` over TCP, and listens on port `5002` for answers from `LogicAgent`. The file `web-ui/server-main.js` runs a separate Express/WebSocket server on port `4100` to monitor the main node's heartbeat and agent events.
The client script `web-ui/public/monitor.js` drives the browser dashboard by opening a WebSocket, showing online/offline status, and rendering the list of connected agents.

Agent Roles

The **UserAgent** acts as the entry point for facts and rules coming from the UI. It exposes a TCP listener on `127.0.0.1:5000`, reads one line per request, looks up a running ParserAgent in the Directory Facilitator (service type "parser"), and forwards the raw message as an ACL `INFORM`. If the parser is not yet available it retries for a few seconds and, on repeated failures, notifies the frontend through a TCP message to `FRONT_HOST:FRONT_PORT` (defaults to `127.0.0.1:5002`). It also responds to application-level liveness probes by answering an ACL `REQUEST` with content "ping" with an `INFORM` "pong".

The **QueryAgent** plays the symmetric role for questions. It listens on the local version at the port `5001` and forwards each query to the ParserAgent. Before sending it performs a liveness probe using an ACL `REQUEST` "ping" and an *AchieveREInitiator*; if the parser replies with `INFORM` "pong", the query is sent as an `INFORM`, otherwise the agent backs off and retries. A short-term suppression cache avoids hammering unreachable agents.
Persistent failures are reported to the UI on the same `5002` socket. Like the other agents, QueryAgent also answers to "ping".

The **ParserAgent** is responsible for turning natural language into Prolog.
It registers with type "parser" in the DF and receives `INFORM` messages from UserAgent and QueryAgent.
The sender is used to decide whether the input should be treated as a fact/rule ("user...") or a query ("query..."). The agent calls : `LLMService.translateToLogic(input, type)`, which contacts a remote model via OpenRouter with an API key and returns a single Prolog clause. For queries, the agent ensures the "?-" prefix is present. Before delivering the clause, ParserAgent pings a LogicAgent (type "logic") and, on success, forwards the message as an `INFORM` with a prefix "##TYPE:*fact/query*## " followed by the clause. A backoff map suppresses retries toward failing logic agents for around 60 seconds; repeated failures are surfaced to the UI.

The **LogicAgent** is the execution engine.
On startup it consults the Prolog knowledge base at `web-ui/kb/knowledge.pl` through JPL.
Incoming `INFORM` messages are sanitized (code fences and comments removed, whitespace normalized, final period ensured) and then classified.
If the input is a fact or a rule it is asserted into SWI-Prolog using `assertz/1` (rules via `assertz((Head :- Body))`) and appended to `knowledge.pl` so that the knowledge persists.

If the input is a query, the agent evaluates it in Prolog and collects all solutions; when the query is ground and true it reports success.
The raw logical result is then transformed into a short English sentence by `LLMService.humanizeAnswer(logicResult, query)`, and the final answer is sent back to the frontend through a TCP connection to `FRONT_HOST:FRONT_PORT` (default `127.0.0.1:5002`).
LogicAgent also answers application-level "ping" messages with "pong". In the running system the web server under `web-ui/server.js` exposes HTTP endpoints for the GUI, bridges `/send-fact` to the UserAgent socket on port `5000` and `/send-query` to the QueryAgent socket on port `5001`, and listens on port `5002` to collect the answers produced by LogicAgent.
Agent discovery and liveness are managed through the JADE Directory Facilitator and explicit ping/pong handshakes implemented by all agents.

Usage Instructions

Startup with Docker Compose

The docker folder includes Docker Compose profiles to launch the JADE platform (Main) and the agents (User GUI, Parser, Logic, Query).

0) Open Docker Desktop

Ensure Docker Desktop is running on your machine. It can be Windows, macOS, or Linux.
The version of Docker Desktop in Windows must be at least 3.9 to support Compose profiles.

1) Build base image

Open a shell and move to the compose directory:

```
cd Project/JADE-bin-4.6.0/jade/docker
```

Rebuild the base image used by services:

```
docker compose build jade-main
```

Tip: add –no-cache when you change Java/Node or scripts.
Note: agent services inherit their build from the shared jade-agent service; the first

```
docker compose --profile agent up user-ui
```

will automatically build the `sdai:main` image if not present locally.

2) Start Main (DF/AMS + Monitor UI)

```
docker compose --profile main up jade-main
```

- JADE Main listens on `1099` (IMTP/RMI) and `7778` (HTTP-MTP).
- Monitor dashboard: `http://localhost:4100/monitor`

3) Start agents

GUI + UserAgent (same container, GUI on port 4000):

```
docker compose --profile agent up user-ui
# GUI: http://localhost:4000
```

Start Parser, Logic and Query as dedicated services (they connect to Main):

```
docker compose --profile agent up parser
docker compose --profile agent up logic
docker compose --profile agent up query
```

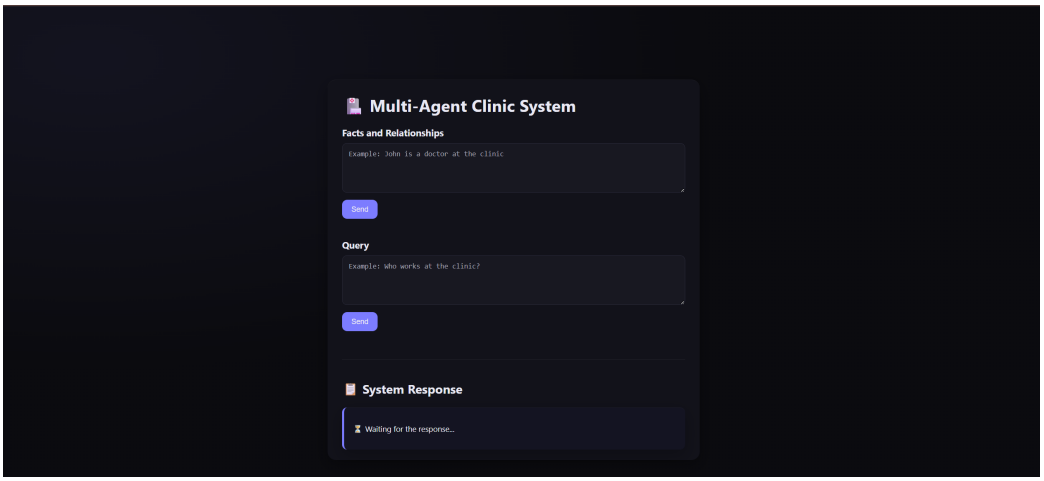

Figure 3: User interface served at `http://localhost:4000`.

Agents on remote machines

To attach an agent container running on another machine (e.g., via Tailscale), export the reachable IP via `PUBLIC_HOST` and point `MAIN_HOST` to the Main's IP. Ensure ports `1099` and `7778` are exposed.

```
PUBLIC_HOST=100.xx.yy.zz MAIN_HOST=100.84.182.11 \
  docker compose --profile agent up user-ui
```

Notes:

- Agent names are automatically made unique (random suffix) to avoid already-registered conflicts.
- The GUI (user-ui) sends queries to the query service; Logic returns answers to user-ui (configurable via env in compose).

4) Handling unavailable agents

If an agent is stopped during processing (e.g., container stopped):

- **QueryAgent:** retries contacting ParserAgent up to 5 times (2s interval). On failure, it notifies the frontend via socket to `FRONT_HOST:FRONT_PORT` (default `127.0.0.1:5002`).
- **ParserAgent:** retries contacting LogicAgent up to 5 times (2s interval). On failure, it notifies the frontend on the same channel (`FRONT_HOST:FRONT_PORT`).
- The Monitor dashboard continues to show up/down state and removes unreachable agents after a TTL.

Useful environment variables:

- `FRONT_HOST`: host of the GUI server receiving LogicAgent answers (default `127.0.0.1`).
- `FRONT_PORT`: port of the GUI server receiving LogicAgent answers (default `5002`).
- `QUERY_TIMEOUT_MS`: GUI-side timeout for LogicAgent responses (default `10000` ms).

5) Stop / Cleanup

Stop services for the used profiles:

```
docker compose --profile agent down
docker compose --profile main  down
```

or stop a single service, e.g. Logic:

```
docker compose stop logic
```

Workflow Example Domain and Assumptions

The end-to-end workflow connects the web UI with the four agent roles (User, Parser, Logic, Query) and the JADE Main. Users submit facts, rules, and queries from the UI; agents translate, store, reason, and return results.

Figure 4: Workflow example

To ground the demos, examples also use a simple hospital scenario (e.g., "John is in the room 204").
The architecture and agents are domain-agnostic and can be applied in

any setting (healthcare, manufacturing, logistics, finance, etc.). For a fully working system, all agent roles must be running concurrently – one process per role – started from the command line (or via Docker Compose): `UserAgent`, `ParserAgent`, `LogicAgent`, and `QueryAgent` must all be registered and reachable.