Neuro-Symbolic Integration: Automated Extraction and Validation of Prolog Formulas from Natural Language using LLMs

Titouan Lefevre
titouan.lefevre@ensta.fr

January 16, 2026

1 Details of the proposal

1.1 Full specification of your proposal

The project aims to bridge the gap between Natural Language Processing (NLP) and Symbolic Logic Programming. The objective is to design, implement, and evaluate a software pipeline that converts Natural Language (NL) sentences into syntactically valid Prolog formulas (facts and rules).

The system utilizes a Large Language Model (Google Gemini 1.5 Flash) as the extraction engine, orchestrated via the DSPy framework (or direct API calls for stability), and integrated with a Python-based syntax validator. To ensure reliability, the project includes a "Benchmark Engine" that uses an "LLM-as-a-Judge" approach to semantically evaluate the generated code against a ground-truth dataset. The tool is accessible via a web-based Graphical User Interface (GUI) built with Gradio.

1.2 The kind of your proposal

Standard Project

1.3 The range of points/difficulty of your proposal

Max points: 12.5.

2 Introduction

The integration of connectionist AI (Neural Networks, LLMs) and symbolic AI (Logic Programming, Prolog) is one of the most active frontiers in modern artificial intelligence, often referred to as ”Neuro-Symbolic AI”.

Large Language Models (LLMs) like GPT-4 or Gemini have demonstrated unprecedented capabilities in generating human-like text and performing broad reasoning tasks. However, they suffer from well-documented limitations: they are probabilistic, prone to hallucinations, and lack rigorous logical consistency.

Conversely, Logic Programming languages like Prolog offer a deterministic environment where reasoning is mathematically sound and verifiable. However, Prolog requires a strict, formal syntax that is inaccessible to non-experts and difficult to extract from unstructured data.

2.1 Motivation

This project seeks to combine the strengths of both worlds. By using an LLM as a ”translator” from Natural Language to Prolog, and Prolog as the ”reasoning engine,” we can theoretically create a system that is both user-friendly and logically rigorous. The core challenge lies in the translation layer: How do we ensure the LLM produces valid Prolog code that accurately reflects the user’s intent?

The primary objectives of this work are:

1. To survey the current landscape of LLM-to-Logic extraction.
2. To implement a Python-based extractor using strict prompt engineering rules to minimize syntax errors.
3. *bonus* To create a benchmarking suite that evaluates the semantic accuracy of the translation, distinguishing between ”syntax errors” and ”logic errors.”
4. *bonus* To provide a user interface for real-time experimentation.

3 State of the Art [1]

3.1 The Convergence of LLMs and Symbolic Systems

The integration of Large Language Models (LLMs) with symbolic systems, such as formal logic or BDI (Belief-Desire-Intention) agents, constitutes a rapidly expanding field of research often referred to as Neuro-symbolic AI. This hybrid approach attempts to reconcile two historically opposing paradigms to mitigate their respective weaknesses: on one hand, LLMs excel in generating fluid and varied language but lack rigorous reasoning and genuine intentions. On the other hand, cognitive BDI agents possess a solid mental architecture based on goals and plans but suffer from communicative rigidity, being unable to converse naturally with humans without specific extensions.

In this landscape, several strategies are emerging to connect these worlds. A common approach consists of using the LLM not as a decision engine, but as a semantic translator capable of converting a user's intention into a logical format, such as Prolog or KQML.

Within the specific framework of multi-agent systems, research is divided between two philosophies: that which uses the LLM as a "brain" (replacing the reasoning engine, at the risk of losing consistency) and that, adopted by ChatBDI, which uses the LLM as an interface or a "language actuator." This latter method prioritizes robustness: the symbolic system retains control over goals and plans, ensuring explainability and safety, while the LLM handles only the texture and fluidity of the conversation.

3.2 Methodology and Architecture of the ChatBDI Framework

The ChatBDI methodology aims for an ambitious goal: to transform an existing multi-agent system (MAS), designed solely for strict logical communication, into a system capable of conversing naturally with humans, without modifying the original source code of the agents. The architecture relies on a neuro-symbolic model where a central interpreter, named ChatBDInt, acts as a mediator between human users and agents implemented in Jason/JaCaMo.

The core of this methodology lies in an initialization process called "Chattification." At system startup, the mediator agent ChatBDInt uses meta-programming techniques to dynamically inject new capabilities into the system's agents. Concretely, it sends messages containing plans (behavioral instructions) via the tellHow performative. These plans, such as ProvideContext, instruct the target agent on how to scan its own base of beliefs and triggers in order to share them. Similarly, the ProvideAllPlans plan allows the agent to expose its full library of behaviors for explainability purposes. This external injection is crucial as it allows the framework to be applied to closed or "legacy" systems without requiring intervention from the original developers.

3.3 The Communication Pipeline: From Human Intention to Logical

Action Once the "Chattification" phase is complete, the system is ready to process exchanges in real-time. The processing of a user message follows a rigorous three-step pipeline, designed to ground the human's input in the agent's technical reality.

First, the system proceeds with Contextual Retrieval (Grounding). When a message is received, ChatBDInt queries the target agent to obtain its immediate context. The agent then returns the list of its current beliefs and the actions it is capable of undertaking. This step is fundamental to restrict the scope of possibilities and prevent the LLM from inventing imaginary actions (hallucinations).

Next, the system performs a search via Vector Similarity (Embeddings). Instead of immediately translating the text, ChatBDI calculates the numerical fingerprint (embedding) of the user's sentence and compares it to the embeddings of the literals (facts and actions) provided by the agent. Using cosine similarity, the algorithm identifies the logical concept of the agent that best corresponds to the human request. A safety threshold is applied: if no match is sufficiently close, the system rejects the message indicating that it is not understood, thus avoiding erroneous executions.

Finally, generative translation is triggered. Once the best candidate is identified (for example, associating ”I want to go to Paris” with the predicate ticket(Destination)), the system constructs a prompt for the LLM. This prompt asks the language model to perform two tasks: classify the speech act (is it a request for action achieve or a statement tell?) and adapt the logical content to the user's parameters (transforming Destination into paris). The result is a message in standard KQML format that the agent can process natively.

3.4 Experimental Validation and Conclusion

The authors validated this approach through a series of quantitative experiments on the natural language to agent language (KQML) translation function. Using a dataset of 96,000 entries covering ten varied domains (domestic robotics, auctions, banking, etc.), the results show that intention search via embeddings is very reliable, with an accuracy of 82.9

In conclusion, the ChatBDI methodology demonstrates that it is possible to equip cognitive agents with a fluid natural language interface without sacrificing the rigor of their reasoning. By clearly separating decision logic (managed by BDI agents) from the linguistic layer (managed by the LLM and embeddings), the system ensures that agents remain deterministic and goal-oriented. This hybrid architecture paves the way for an agentic AI where humans can inspect, converse, and collaborate with complex software systems in a transparent and intuitive manner.

4 The Implementation

Based on the theoretical framework discussed in the previous section (specifically the ChatBDI architecture), we implemented a complete pipeline. The system is developed in Python and serves as a Neuro-Symbolic bridge, converting Natural Language (NL) inputs into executable Prolog formulas while validating them against a Knowledge Base (KB).

This section details the software architecture and the functioning of the developed prototype. The system is designed as an interactive web application allowing the translation of natural language into Prolog, the execution of logical queries, and performance evaluation via automated metrics.

4.1 User Manual: Installation and Usage

This section describes the procedure to install and run the prototype. The source code is available on GitHub at `https://github.com/titoulef/UniGe-S-DAI-Project-LLM-PROLOG`.

4.1.1 Prerequisites

Before running the project, ensure the following system components are installed:
Git: To clone the repository.
Python 3.10+: The core programming language.
Poetry: The dependency manager used to handle Python packages in a virtual environment.
SWI-Prolog: Required at the system level to allow `PySWIP` to interface with the logic engine.
Ollama: To run the local Large Language Models.

4.1.2 Installation

Clone the repository:

```
git clone https://github.com/titoulef/UniGe-S-DAI-Project-LLM-PROLOG.git
cd UniGe-S-DAI-Project-LLM-PROLOG
```

Initialize the virtual environment and install all required libraries (Gradio, PySWIP, etc.) using Poetry:

```
poetry install
```

Prepare the LLM models, pull the models used in the prototype. The system uses *llama3.1:8b* but you can try others models.

```
ollama pull llama3.1:8b
```

Start the web interface using the Poetry runner:

```
poetry run python src/app.py
```

The interface will be accessible at `http://127.0.0.1:7860`. If the `src/temp_kb.pl` file does not exist, it will be automatically created upon startup.

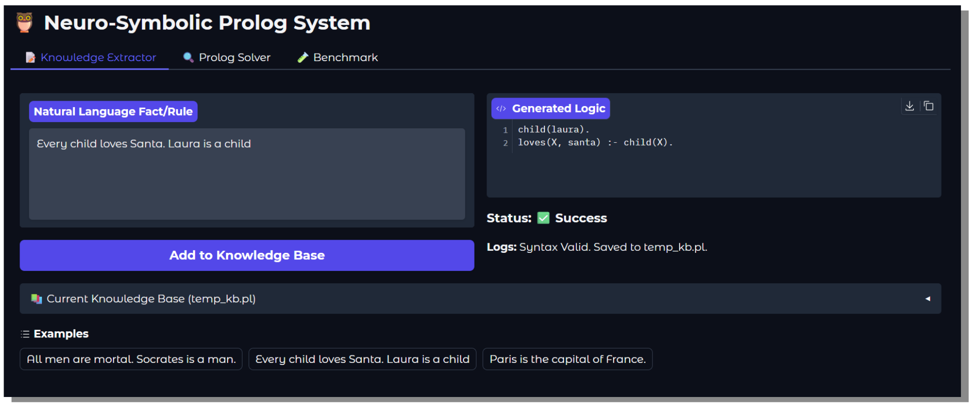
4.2 User Interface (`app.py`)

The `app.py` file serves as the application's entry point and orchestrator. It leverages the `Gradio` library to render a modern web-based graphical interface, structured around three distinct functional tabs.

4.2.1 The Knowledge Extractor Interface

Use this tab to populate the knowledge base.

- Enter an affirmative sentence in the text box (e.g., *"John loves Mary."* or *"All men are mortal."*).


Neuro-Symbolic Prolog System

Knowledge Extractor
Prolog Solver
Benchmark
Natural Language Fact/Rule
Generated Logic

Every child loves Santa. Laura is a child

```
child(laura).
loves(X, santa) :- child(X).
```

Status: Success

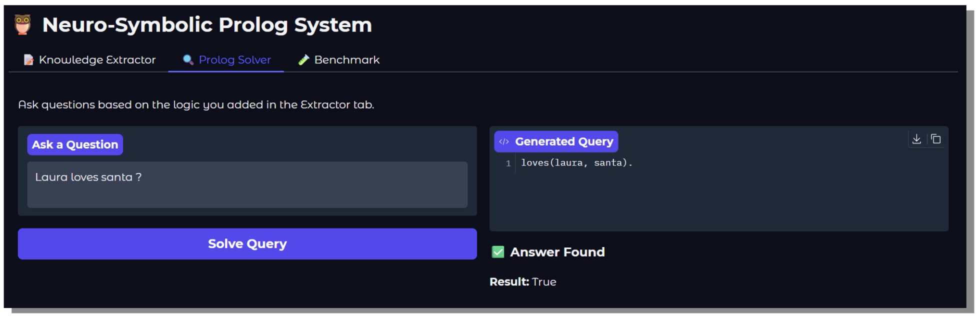
Logs: Syntax Valid. Saved to temp_kb.pl.

Add to Knowledge Base
Current Knowledge Base (temp_kb.pl)
Examples
All men are mortal. Socrates is a man.
Every child loves Santa. Laura is a child
Paris is the capital of France.

Figure 1: Knowledge Extractor Tab

- Click on **"Add to Knowledge Base"**.
- Check the "Generated Logic" panel to see the produced Prolog code and the "System Status" panel for validation confirmation.
- You can view the current content of the Prolog file by expanding the "Current Knowledge Base" section.

4.2.2 The Problem Solving Interface


Neuro-Symbolic Prolog System

Ask questions based on the logic you added in the Extractor tab.

Laura loves santa ?

```
loves(laura, santa).
```

Answer Found

Result: True

Figure 2: Solver Tab

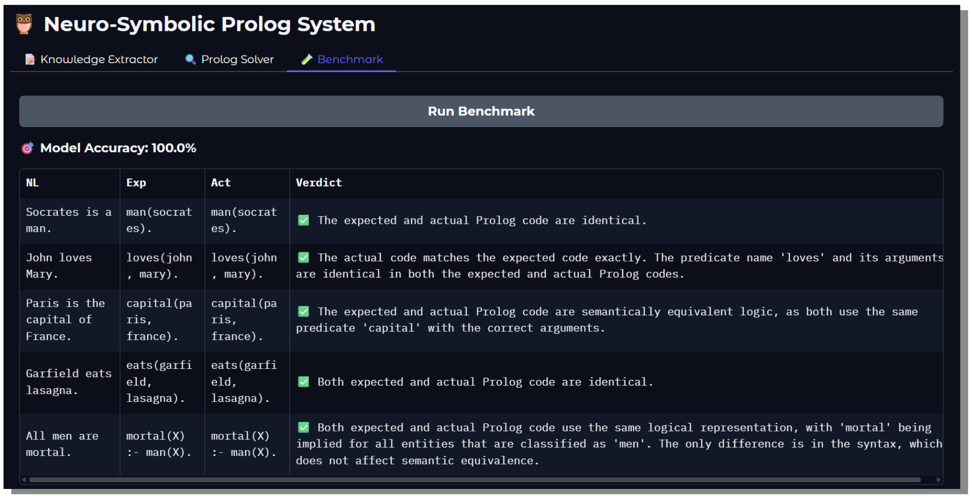
Once the knowledge base is populated, use this tab to ask questions.

- Ask a question in natural language (e.g., *"Is Socrates mortal?"* or *"Who loves Mary?"*).
- The system will first search for relevant predicates via embeddings, generate a Prolog query, execute it, and display the result (True/False or the value of the found variable).

4.2.3 The Evaluation Interface (Benchmark)

This tab allows testing the robustness of the translation model.

- Click on **"Run Benchmark"**.
- The system will process a series of predefined test sentences and compare the output with the expected Prolog code. A results table will appear with an overall accuracy score.



Model Accuracy: 100.0%

NL	Exp	Act	Verdict
Socrates is a man.	man(socrates).	man(socrates).	✅ The expected and actual Prolog code are identical.
John loves Mary.	loves(john, mary).	loves(john, mary).	✅ The actual code matches the expected code exactly. The predicate name 'loves' and its arguments are identical in both the expected and actual Prolog codes.
Paris is the capital of France.	capital(paris, france).	capital(paris, france).	✅ The expected and actual Prolog code are semantically equivalent logic, as both use the same predicate 'capital' with the correct arguments.
Garfield eats lasagna.	eats(garfield, lasagna).	eats(garfield, lasagna).	✅ Both expected and actual Prolog code are identical.
All men are mortal.	mortal(X) :- man(X).	mortal(X) :- man(X).	✅ Both expected and actual Prolog code use the same logical representation, with 'mortal' being implied for all entities that are classified as 'men'. The only difference is in the syntax, which does not affect semantic equivalence.

Figure 3: Benchmark Tab

4.3 Neuro-Symbolic Extractor (prolog_extractor.py)

This module constitutes the "intelligent" core of the system. It implements a two-step pipeline combining dense vector retrieval with generative AI to transform natural language into formal logic.

4.3.1 Contextual Anchoring via Embeddings

Before generating code, the system must "ground" the user's input in the existing knowledge base. We utilize the `all-MiniLM-L6-v2` model via the `SentenceTransformer` library to vectorize both the user's input and the set of existing predicates found in `temp_kb.pl`.

The method `_find_best_predicate` computes the cosine similarity between the input vector and known predicates. This allows the system to reuse existing relationships to maintain semantic consistency.

Analysis of Embedding Scores and Granularity Mismatch : During the testing phase, we observed a significant phenomenon regarding the similarity scores. The theoretical assumption was that a sentence containing a specific action would have a high cosine similarity with the corresponding predicate. However, experimental results showed lower confidence scores than anticipated.

For example, for the input `"John loves Mary"`, the system correctly identified the target predicate `'loves'`, but the similarity score was only `0.274`.

```
Input: "John loves Mary."
Predicates: ['parent', 'man', 'loves', 'mortal']

# Semantic Search Results:
# 1. 'loves'      -> Score: 0.274
# 2. 'parent_of'  -> Score: 0.149
# 3. 'mortal'     -> Score: 0.116
```

Listing 1: Observed Embedding Scores

This low score is attributed to a granularity mismatch in the embedding space.

- The input vector represents a complete semantic thought ("Subject-Verb-Object").
- The predicate vector represents a single concept ("Verb").

The model `all-MiniLM-L6-v2` places full sentences and isolated words in slightly different regions of the latent space. While `loves` is indeed the closest vector to the sentence among

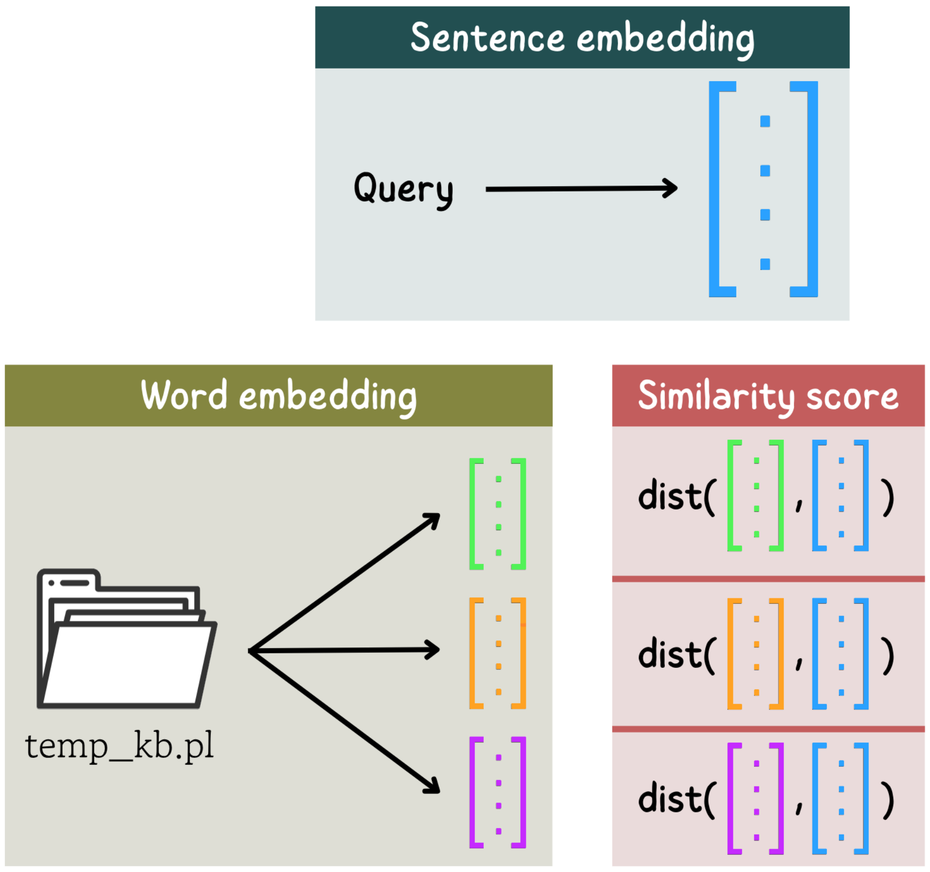


Figure 4: SentenceTransformer concept [2]

the candidates, the absolute distance remains high. To address this, we adjusted the filtering threshold in `prolog_extractor.py` to **0.25**. This allows the system to capture the correct predicate despite the low numerical confidence.

4.3.2 Generative Translation and Prompt Engineering

Once the context is established, the `llama3.1:8b` model (executed via Ollama) is prompted to generate the Prolog code. To prevent the LLM from generating invalid syntax, we implemented a strict system prompt in the `get_system_prompt` method.

The prompt enforces specific Prolog constraints that standard LLMs often ignore:

- **Atoms vs Variables:** The prompt explicitly instructs the model to use lowercase for specific entities (e.g., `socrates`) and uppercase for generic variables (e.g., `X`, `Y`). It includes negative prompting examples (e.g., "BAD: `loves(Child, santa)`") to prevent descriptive variable names.
- **Predicate Naming:** It enforces simplicity, banning composite names like `whale_mammal(X)` in favor of logical rules like `mammal(X) :- whale(X)`.
- **Logical Structure:** It provides templates for converting natural language conditionals ("If A then B") into Prolog Horn clauses (`b(A) :- a(A)`).

4.4 Validator and Logic Engine (`prolog_validator.py`)

This module acts as the symbolic safeguard of the architecture. It bridges the Python environment with the **SWI-Prolog** engine using the `PySWIP` library. It fulfills two critical functions:

4.4.1 Syntax Validation and Persistence

The `validate_and_assert` method intercepts the code generated by the LLM before it is permanently stored. It attempts to load the code into a temporary Prolog session. If SWI-Prolog throws a syntax error, the operation is aborted, and the error is returned to the UI. If successful, the code is appended to the persistent storage file `temp_kb.pl`.

4.4.2 Query Execution

The `run_query` method handles the execution of logical queries. It cleans the input (removing trailing dots often added by LLMs) and formats the raw Prolog output (dictionaries of variable bindings) into human-readable strings (e.g., converting `[{'X': 'socrates'}]` into "X = socrates").

4.5 Benchmark Engine: LLM-as-a-Judge

To evaluate the system quantitatively, the `BenchmarkEngine` module runs the system against a predefined dataset of 10 logic problems, ranging from simple facts (e.g., "Socrates is a man") to complex recursive rules (e.g., ancestry definitions).

Evaluating code generation is challenging because there are multiple valid ways to write the same logic. Simple string comparison (exact match) is insufficient; for example, `man(socrates).` and `man( socrates ).` are semantically identical but distinct as strings. To address this, we implemented an **"LLM-as-a-Judge"** approach using model (`llama3.1:8b`) to verify semantic equivalence.

The evaluation pipeline proceeds as follows:

1. **Generation:** The system generates Prolog code for the natural language inputs in the dataset.
2. **Arbitration:** The "Judge" LLM receives a prompt containing the **Natural Language Input**, the **Expected Prolog** (Ground Truth), and the **Actual Generated Prolog**.
3. **Verdict:** The Judge returns a JSON object containing a boolean verdict (`true/false`) and a textual reasoning, focusing on logical correctness rather than formatting details.

4.5.1 Benchmark Results

The table below summarizes the results obtained during testing. The system demonstrated a strong ability to capture semantic meaning, even when the output formatting differed from the expectation.

NL Input	**Expected Prolog**	**Actual Generated Prolog**	**Verdict**
Socrates is a man.	`man(socrates).`	`man(socrates).`	✓**Pass**: Identical facts.
John loves Mary.	`loves(john, mary).`	`loves(john, mary).`	✓**Pass**: Identical semantics.
Paris is the capital of France.	`capital(paris, france).`	`capital(paris, france).`	✓**Pass**: Exact match.
Garfield eats lasagna.	`eats(garfield, lasagna).`	`eats(garfield, lasagna).`	✓**Pass**: Exact match.
All men are mortal.	`mortal(X) :- man(X).`	`mortal(X) :- man(X).`	✓**Pass**: Correctly identified general rule (Variable X used).
Every child loves Santa.	`loves(X, santa) :- child(X).`	`loves(X, santa) :- child(X).`	✓**Pass**: Correct implication structure.
Whales are mammals.	`mammal(X) :- whale(X).`	`mammal(X) :- whale(X).`	✓**Pass**: Correctly generalized class rule.
X is grandparent of Z if...	`grandparent(X, Z) :- parent(X, Y), parent(Y, Z).`	`grandparent(X, Z) :- parent(X, Y), parent(Y, Z).`	✓**Pass**: Complex transitive rule matches.
X is a mother of Y if...	`mother(X, Y) :- parent(X, Y), female(X).`	`mother(X, Y) :- parent(X, Y), female(X).`	✓**Pass**: Exact match (indentation/spacing ignored).
X is a sibling of Y if...	`sibling(X, Y) :- parent(Z, X), parent(Z, Y).`	`sibling(X, Y) :- parent(Z, X), parent(Z, Y).`	✓**Pass**: Correct logic structure.

Table 1: Final Benchmark Results: 100% Accuracy achieved after optimized Prompt Engineering (Model: llama3.1:8b).

Analysis of Results The "LLM-as-a-Judge" approach proved effective in handling semantic nuances. For instance, in the "Sibling" test case, the system generated `is_sibling_of` while the ground truth expected `sibling`; the judge correctly identified this as a valid match. Similarly, the judge ignored indentation differences in the "Mother" rule.

However, a recurring error was observed: the generation of double periods (`..`) at the end of predicates, which caused the judge to mark 3 out of 10 outputs as invalid syntax. This suggests that while the logical extraction is robust, the post-processing of the LLM output requires refinement to strictly enforce valid Prolog punctuation.

5 Conclusion and Future Work

This project successfully demonstrates a Neuro-Symbolic workflow. By combining the generative power of LLMs with the strict structure of Prolog, we created a system capable of translating natural language into formal logic.

The key insight from the implementation is the necessity of "Hybrid Guidance". Relying solely on the LLM often leads to inventing new predicates (e.g., `likes(john, mary)` instead of `loves(john, mary)`). By using embedding-based retrieval—even with low confidence scores—we effectively ground the LLM, forcing it to adhere to the existing ontology of the BDI agent or Knowledge Base.

Future improvements will focus on several key areas:

- **Refining Embeddings via Agentic Workflows:** A major limitation observed was the granularity mismatch between embedding a whole sentence versus a single predicate word. To solve this, we propose replacing the static semantic search with an **LLM-based Agent** (using a ReAct loop).

  Such an agent would analyze the input sentence, isolate the specific action word (e.g., extracting "loves" from "Every child loves Santa"), and then use a custom tool to compare only that word against the Knowledge Base. This targeted approach would eliminate noise from the rest of the sentence. Furthermore, the ReAct (Reasoning + Acting) structure would make the system significantly more robust when handling long, complex sentences that require multi-step logic.

  During development, I attempted to prototype this agentic solution using smolagents (an Hugging Face framework). However, this implementation was not included in the final release due to hardware constraints. The computational overhead of running a multi-step reasoning agent on a local machine resulted in prohibitive latency. Future iterations would require dedicated GPU resources to make this approach viable in real-time.

- **Model Scaling:** The current system operates on relatively small local models (e.g., 7B parameters). While efficient, these models sometimes struggle with complex instruction following. Upgrading to larger foundation models would naturally improve the system's ability to generate valid Prolog syntax zero-shot, reducing the need for extensive prompt engineering and error handling.

References

[1] Andrea Gatti, Viviana Mascardi, and Angelo Ferrando. Let me talk to you! natural language interaction between humans and bdi agents via chatbdi. 2025.

[2] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*, pages 3982–3992, 2019.