Integrating Answer Set Programming and Large Language Models for Reliable Structured Knowledge Extraction

MARIO ALVIANO*, University of Calabria, Italy
MATTEO CAPALBO, University of Calabria, Italy and TU Wien, Austria
GEORG GOTTLOB, University of Calabria, Italy
LORENZO GRILLO, University of Calabria, Italy
IRFAN KAREEM, University of Calabria, Italy
FABRIZIO LO SCUDO, University of Calabria, Italy
SEBASTIANO A. PICCOLO, University of Calabria, Italy
LUIS ANGEL RODRIGUEZ REINERS, University of Calabria, Italy

Answer Set Programming (ASP) and Large Language Models (LLMs) offer complementary strengths in symbolic reasoning and natural language understanding. In this work, we present an integrated framework for reliable structured knowledge extraction from natural language that combines the syntactic capabilities of LLMs with the semantic constraints provided by ASP. Structured representations in the form of ASP facts are obtained by guiding LLM generation through domain specific prompt templates and background knowledge specified in a declarative YAML configuration, and by validating and completing the extracted information using an ASP knowledge base.

Building on this framework, we introduce grammar constrained decoding as a principled mechanism to enforce structural and semantic alignment between LLM outputs and target symbolic representations. By leveraging formal grammars, we gain fine grained control over the generation process, substantially reducing hallucinations and limiting unnecessary verbosity, while preserving extraction quality. We study three grammar formats commonly used to represent structured knowledge, namely CSV, Datalog, and JSON, and analyze their impact on correctness, efficiency, and computational cost.

We evaluate the proposed approach on benchmarks derived from ASP Competitions. The results show that the integration of ASP consistently improves extraction accuracy over vanilla LLMs, particularly for smaller models. Moreover, grammar constrained decoding maintains these gains while significantly improving generation efficiency. Among the considered formats, CSV provides the best trade-off between accuracy and cost, JSON achieves the highest extraction quality at the expense of increased verbosity, and Datalog exhibits lower robustness in the extraction process. Overall, the results demonstrate that combining ASP with grammar constrained LLM output yields a reliable, scalable, and cost-effective approach to structured knowledge extraction.

JAIR Associate Editor: Martin Gebser

*Corresponding Author.

Authors' Contact Information: Mario Alviano, ORCID: 0000-0002-2052-2063, mario.alviano@unical.it, University of Calabria, Rende, Italy; Matteo Capalbo, ORCID: 0009-0001-0114-9304, matteo.capalbo@student.tuwien.ac.at, University of Calabria, Rende, Italy and TU Wien, Vienna, Austria; Georg Gottlob, ORCID: 0000-0002-2353-5230, georg.gottlob@unical.it, University of Calabria, Rende, Italy; Lorenzo Grillo, ORCID: 0009-0002-6099-3895, lorenzo.grillo@unical.it, University of Calabria, Rende, Italy; Irfan Kareem, ORCID: 0000-0002-9762-3954, irfan.kareem@unical.it, University of Calabria, Rende, Italy; Fabrizio Lo Scudo, ORCID: 0000-0001-9092-8918, fabrizio.loscudo@unical.it, University of Calabria, Rende, Italy; Sebastiano A. Piccolo, ORCID: 0000-0002-6986-3344, sebastiano.piccolo@unical.it, University of Calabria, Rende, Italy; Luis Angel Rodriguez Reiners, ORCID: 0009-0000-1808-9910, luis.reiners@unical.it, University of Calabria, Rende, Italy.

DOI: 10.1613/jair.1.21678

JAIR Reference Format:
Mario Alviano, Matteo Capalbo, Georg Gottlob, Lorenzo Grillo, Irfan Kareem, Fabrizio Lo Scudo, Sebastiano A. Piccolo, and Luis Angel Rodriguez Reiners. 2026. Integrating Answer Set Programming and Large Language Models for Reliable Structured Knowledge Extraction. *Journal of Artificial Intelligence Research* 87, Article 6 (September 2026), 47 pages. DOI: 10.1613/jair.1.21678

1 Introduction

Large Language Models (LLMs) and Answer Set Programming (ASP) represent two distinct but complementary paradigms in Artificial Intelligence (AI). On the one hand, LLMs including GPT (Brown et al. 2020), PaLM (Chowdhery et al. 2023), and LLaMA (Touvron et al. 2023), have revitalized natural language processing (NLP), achieving unprecedented levels of fluency and task generality. For instance, thanks to the power of LLMs, historically difficult NLP tasks such as language generation, summarization, and sentiment analysis can now be addressed, in a unified manner, with a remarkable degree of accuracy (Jin et al. 2024; W. Zhang et al. 2023). On the other hand, ASP (Marek and Truszczyński 1999; Niemelä 1999) is a declarative programming paradigm extending Datalog with answer set semantics (Gelfond and Lifschitz 1990). It has proven highly effective for problems that require robust inference or exact solutions such as patients scheduling (Cappanera et al. 2023; Cardellini et al. 2024), model based diagnosis (Wotawa 2020), and configuration problems (Taupe et al. 2021). That is, ASP excels precisely on those logical and relational reasoning tasks that remain challenging even for the most advanced LLMs (Li et al. 2024), highlighting the complementarity between the two paradigms.

The following example, in which an LLM is asked to find a maximum clique, illustrates both the fluency of the answers as well as the limits of LLMs on logical reasoning tasks.

Example 1. *Let us consider the following prompt:*

"You are organizing a networking event with several attendees, and you want to **form the largest possible group where every person in the group knows every other person.**
The attendees have provided the following information about who they know:

- Alice knows Bob, and Evan.
- Bob knows Charlie, Diana, Evan, and Fiona.
- Charlie knows Diana, Evan, Fiona, and George.
- Diana knows Evan, Fiona, and George.
- Evan knows George.
- Fiona knows George, and Henry.
- George knows Evan, and Henry.
- Henry knows George.

Question: What is the largest group of attendees where everyone knows every other person in the group?"

The answer provided by chatgpt.com (both by using GPT-4o from the web interface and gpt-4o-mini-2024-07-18 via API) is the following:

"The largest group where everyone knows everyone else is the group consisting of **Bob, Charlie, Diana, Evan, and Fiona**."

Although the answer is fluent and grammatically well-formed, it is factually incorrect because Evan does not know Fiona. We report that we obtained a wrong answer also from Meta LLaMA 3:70b (running locally on our servers). In contrast, modeling the problem in ASP yields a correct factual answer. However, the result would be an answer set, rather than a fluent natural-language explanation. ■

Recognizing the complementary strengths of the linguistic capabilities of Large Language Models (LLMs) and the reasoning power of Answer Set Programming (ASP), we propose LLMASP: a framework that integrates these

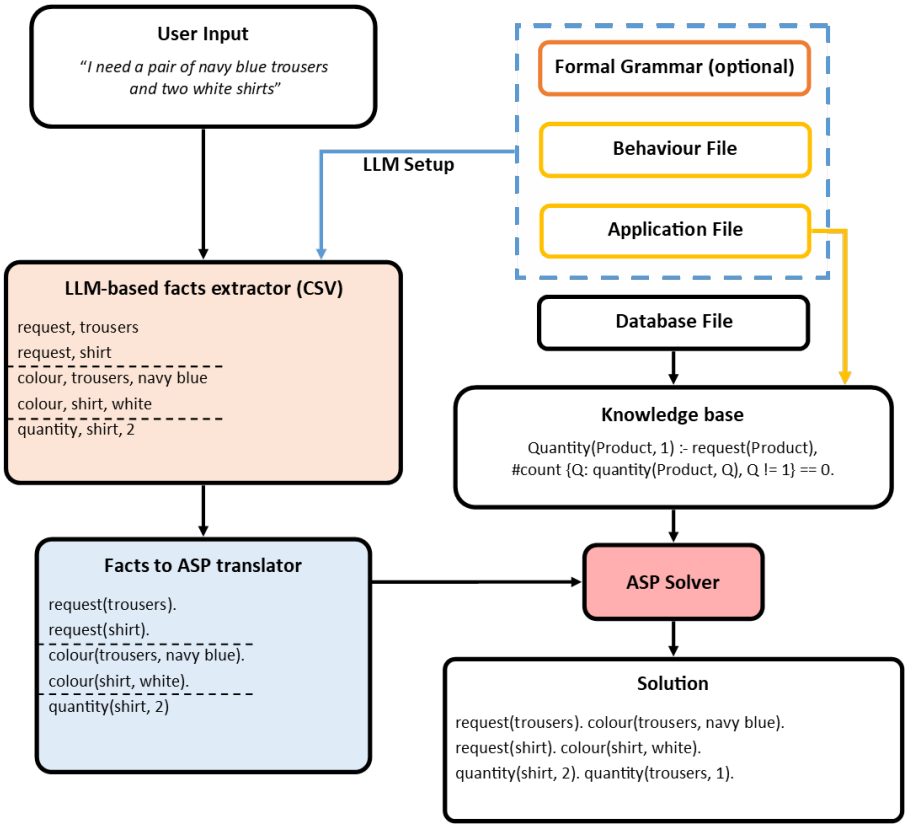

Fig. 1. Graphical depiction of the (grammar-constrained) LLMASP pipeline. The system takes as input a user query expressed in natural language, along with a behavior file defining general prompt construction logic, and an application file encoding domain-specific knowledge and extraction targets. In addition, a formal grammar, used to constrain the output format of the LLM, can be provided to enforce syntactic correctness during generation. The behavior and application files are combined to construct prompt templates that instruct the LLM on how to extract structured information from the user query, with outputs formatted according to the given grammar (if any). The resulting structured data, typically in the form of Datalog-style facts, is then merged with an optional input database and the logic program defined in the application file. The obtained logic program is processed by an ASP solver to compute one or more answer sets.

two paradigms into a unified system for structured knowledge extraction and reasoning (inspired by recent work in the literature (Basu, Varanasi, et al. 2021; Zeng et al. 2024)). Our objective is to seamlessly combine natural language processing with logical inference by strictly decoupling knowledge extraction from rule generation. Rather than relying on the LLM for end-to-end program synthesis, LLMASP uses the LLM exclusively to extract relational facts from unstructured text. These facts are then processed by static, expert-written ASP rules, guaranteeing determinism, safety, and correctness. The LLMASP pipeline is depicted in Figure 1. At its core, LLMASP receives as input a user request expressed in natural language, a database of relational facts, and two YAML [1] files: a *behavior file* specifying general system configuration, and an *application file* describing domain

[1]YAML is a human readable serialization language commonly used for configuration files and in applications where data is being stored or transmitted: http://yaml.org.

knowledge. The behavior and application files are combined to generate prompt templates, which are instantiated with the user input and submitted to the LLM. Each prompt is designed to extract instances of a specific predicate at a time. Subsequently, the extracted facts are parsed and combined with an ASP knowledge base forming a valid logic program that is then evaluated by an ASP solver to compute an answer set. Optionally, LLMASP can receive as input a specification of a formal grammar that is used to determine the output format of the extracted facts, and will be discussed later. For now, let us continue over the lines of Example 1 to illustrate how LLMASP functions.

Example 2 (Continuing Example 1). *LLMASP uses prompts that instruct the LLM to extract data rather than addressing directly the reasoning tasks. First, a system message instructs the LLM about the task to perform:*

System message
> *You are a Natural Language to Datalog translator.*
> ***I will provide text in the format [INPUT]input[/INPUT], and instructions on how to map the INPUT to facts. The expected OUTPUT format will be given by [OUTPUT]predicate(terms).[/OUTPUT].***
> *Here is some context that you MUST analyze and remember.*
> ***If there is a graph, represent it with the predicates node/1 and edge/2.***
> *For example, if a and b are connected, use node(a). node(b). edge(a,b).*

Subsequently, the LLM receives a series of prompts that focus on one single predicate at a time. In this example, we first extract nodes, then edges.

User mess.
> *[INPUT]You are organizing...[/INPUT]*
> *Process the INPUT according to the following instructions:*
> ***List all the edges from source to target.***
> *The output format is [OUTPUT]edge(source, target)[/OUTPUT]*

User mess.
> *[INPUT]You are organizing...[/INPUT]*
> *Process the INPUT according to the following instructions:*
> ***List all the nodes. Associate to every node its ID.***
> *The output format is [OUTPUT]node(id)[/OUTPUT]*

Therefore, LLMASP obtains a relational representation of the graph in input as a collection Datalog facts:

```
node("Alice"). node("Bob"). ...
edge("Alice","Bob"). edge("Alice","Evan"). ...
```

These facts are then coupled with an ASP program to compute a maximum clique:

```
{in(X)} :- node(X).
:- in(X), in(Y), X < Y, not edge(X,Y).
:~ node(X), not in(X). [1@1, X]
```

Finally, LLMASP returns the answer set in natural language:
The maximum clique consists of **Charlie, Diana, Evan, and George**.
(For this example, we coupled LLMASP with Meta LLaMA 3:70b and the behavior file EIN; see Section 5.) ■

Within this setting, the empirical evaluation of LLMASP is guided by two main research questions.

Q1: Can predicate-specific extraction, driven by structured prompt templates, improve the accuracy of LLM-based fact extraction compared to vanilla prompting strategies?
Q2: Can explicit domain knowledge encoded in Answer Set Programming be integrated with neural extraction to derive implicit facts and improve overall correctness?

Section 5 reports an empirical evaluation on benchmarks derived from ASP Competitions (Gebser, Maratea, et al. 2020) assessing the proposed integration of LLMs and ASP, which substantially improves fact extraction accuracy over vanilla LLM prompting. In particular, the F1-score increases from 72.1% to 86.3% when using Meta LLaMA 3:8b, and from 89.2% to 96.5% with Meta LLaMA 3:70b. These results demonstrate the effectiveness of ASP-guided extraction in improving correctness, especially for larger models.

Although these accuracy gains are an important contribution, they come at a non-negligible computational and energy cost. In particular, LLMASP relies on multiple predicate-specific prompts, issuing a separate generation request for each predicate to be extracted. When generation is unconstrained (i.e., when the LLM produces free-form text not guided by an explicit output grammar) this design can incur substantial overhead in terms of completion tokens, runtime, and inference cost. Moreover, unconstrained generation may yield outputs that deviate from the intended symbolic representation. These deviations include redundant facts, malformed structures, or spurious content not supported by the input. Such behaviors negatively affect extraction accuracy and, at the same time, increase computational and energy consumption by producing unnecessary tokens.

To better understand and mitigate these effects, we extend our analysis to explicitly account for the computational cost of the pipeline. In particular, we analyze generation behavior at the token level, distinguishing between prompt tokens and completion tokens, both of which directly impact runtime, API cost, and carbon footprint (Jegham et al. 2025). Completion tokens are especially critical, as they are generated sequentially and therefore dominate inference time, whereas prompt tokens can be processed more efficiently through parallelization in modern LLM architectures (Patel et al. 2023). This analysis highlights two key observations. First, the number of prompt tokens grows linearly with the number of predicates to be extracted, since the pipeline issues a separate completion request for each predicate. Second, while the number of completion tokens is typically stable when the model adheres to the expected output format, it increases noticeably when generation drifts outside the target symbolic structure, for instance due to redundant or malformed outputs (commonly referred to as hallucinations).

To illustrate the nature of such hallucinations, consider the following example in which LLMASP is used to support a virtual shop assistant in the fashion retail domain. Here, the role of the LLM is not to answer the user query directly, but to extract a symbolic representation of information needed by the user in the form of Datalog facts, which are then processed by an ASP solver.

Example 3 (Hallucinations due to Answer-Oriented Generation). *We consider a fashion retail domain with mappings to the predicates* `request/1`, `ask/2`, `color/2`, *and* `quantity/2`. *Given the user input*

"What colors are available for the blazers?"

the intended task of the LLM is to encode the user request as a symbolic representation, rather than to answer the query directly. Accordingly, when processing the predicate `ask/2`, *the correct extraction is the fact* `ask(blazer, color)`, *indicating that the user is requesting color-related information about blazers.*

However, under unconstrained generation, the LLM may embed the correct fact within additional explanatory or conversational text, for example:

```
The user is asking about the colors available for blazers.
    ask(blazer, color).
```

While the atom `ask(blazer, color)` *is correct, the surrounding natural-language text is extraneous with respect to the extraction task. Such mixed outputs violate the intended symbolic interface, increase the number of completion tokens, and complicate downstream parsing. In this setting, even correct facts become costly to recover reliably.*

A more severe issue arises when the unconstrained LLM is asked to process the predicate `color/2`. *In this case, the model may adopt an* answer-oriented *behavior, attempting to respond to the query rather than represent it. For instance, it may generate verbose, answer-like output such as:*

```
Based on the available information, blazers are commonly available in several colors.
```

```
    color(blazer, black).
    color(blazer, navy_blue).
    color(blazer, white).
These are some typical options customers may choose from.
```

These outputs are not entailed by the input and do not correspond to information explicitly stated or requested by the user. By interleaving answer-like natural language with inferred facts, the model departs from the intended task of request representation and introduces spurious domain knowledge. Such behavior results in hallucinated facts *and undermines both the correctness and the efficiency of the extraction pipeline.* ■

Beyond degrading extraction accuracy, answer-oriented generation also increases the number of generated completion tokens, as the model produces content that goes beyond the intended symbolic encoding of the request. As a result, unconstrained generation simultaneously harms correctness and computational efficiency. These observations motivate the following research question:

Q3: Can the generation phase be constrained in order to reduce hallucinations and unnecessary completion tokens without sacrificing extraction accuracy?

To address **Q3**, we extend the LLMASP pipeline with a grammar-constrained decoding mechanism (Geng et al. 2023). Grammar-constrained decoding restricts the output of a language model to strings that conform to a predefined formal grammar, such as grammars expressed in Backus–Naur Form (BNF) (Backus et al. 1963). During generation, the decoder consults the grammar at each step to determine the set of admissible next tokens, pruning all others from the search space. This enforces syntactic well-formedness by construction and prevents a wide class of hallucinations without requiring task-specific fine-tuning (Geng et al. 2023; Park et al. 2024). Grammar-constrained decoding is particularly well suited for structured output tasks such as information extraction and symbolic representation, where correctness depends not only on semantic content but also on adherence to a precise format. In our setting, grammars act as an explicit interface between LLM generation and ASP reasoning, enforcing alignment between generated text and the target symbolic representation.

Several modern LLM frameworks provide native support for grammar-constrained decoding using BNF. In our implementation, based on the Ollama framework, constrained generation is natively supported for JSON outputs, either as arbitrary JSON values or as instances of a given JSON Schema.[2] However, different grammars induce different trade-offs. While JSON is expressive and widely adopted, it introduces substantial syntactic overhead for simple relational data such as ASP facts, which are typically flat and homogeneous. The generation of verbose or nested JSON structures increases the number of completion tokens, leading to higher computational cost without a proportional gain in expressiveness for this task.

Example 4 (Continuing Example 3). *The list comprising* `ask(blazer, color)` *can be encoded in JSON as*

```
[{predicate: "ask", arguments: ["blazer", "color"]}]
```

or, alternatively, in a comma-separated values (CSV) format as

```
ask,blazer,color
```

After normalization by removing extraneous whitespace, the tokenizer associated with the LLaMA models[3] *produces 13 tokens for the JSON representation and 5 tokens for both the CSV representation and the corresponding Datalog fact. This difference illustrates the higher syntactic overhead introduced by JSON for simple relational data.* ■

These considerations motivate two additional research questions:

Q4: Do formal grammars effectively reduce hallucinations in structured fact extraction?

[2] https://github.com/Ollama/Ollama/pull/4525
[3] https://lunary.ai/llama3-tokenizer

Q5: Which grammar offers the best trade-off between extraction quality and generation efficiency?

To answer **Q4** and **Q5**, we investigate alternative grammar formats tailored to the structure of ASP facts. In addition to JSON and a Datalog-style representation, we introduce a lightweight CSV-like format designed for relational facts. This format is line-oriented, compact, and closely matches the structure expected by the ASP backend. We define a corresponding BNF grammar and employ grammar-constrained decoding to generate structured CSV rows representing facts.

We empirically evaluate the proposed grammars on benchmarks derived from ASP Competitions, considering 48 configurations obtained by combining two LLaMA models (8B and 70B), two application files, and four decoding strategies (unconstrained generation, Datalog grammar, JSON grammar, and CSV grammar). The results show that grammar-constrained decoding consistently improves reliability by reducing malformed outputs and duplicated facts. Among the considered formats, the CSV grammar achieves the best balance between extraction quality and generation efficiency, dominating the Pareto front in most settings. JSON yields the highest extraction accuracy but incurs substantial token overhead, while the Datalog grammar exhibits lower robustness during generation.

Overall, these results demonstrate that integrating grammar-constrained decoding into LLMASP substantially improves the reliability and scalability of structured knowledge extraction. Lightweight grammar formats such as CSV preserve or improve extraction quality while significantly reducing generation cost, making them particularly well suited for large-scale and cost-sensitive neuro-symbolic applications.

Contributions. This article makes the following contributions:

- We present a unified neuro-symbolic framework that integrates LLM-based natural language understanding with ASP-based reasoning through a declarative, YAML-driven extraction pipeline [4].
- We introduce grammar-constrained decoding in our framework as a principled mechanism to enforce structural alignment between LLM outputs and target symbolic representations, improving reliability without requiring model fine-tuning.
- We provide a systematic analysis of the computational and efficiency implications of multi-prompt structured extraction, focusing on token-level generation costs and their relationship to output format and predicate granularity.
- We empirically evaluate multiple grammar formats (JSON, Datalog, and a lightweight CSV-based representation) and show that CSV grammars offer the best trade-off between extraction accuracy and generation efficiency.
- We present an extensive experimental evaluation on benchmarks derived from ASP Competitions, demonstrating that grammar-constrained decoding improves robustness and scalability while preserving or improving extraction quality.

We note that the benchmark dataset, as well as a preliminary version of the unconstrained pipeline and its evaluation, were previously presented in (Alviano, Grillo, et al. 2025).

Structure of the paper. The remainder of the paper is organized as follows. Section 2 introduces the necessary background on chat-completion–based large language models, grammar-constrained decoding, and Answer Set Programming. Section 3 formally presents the LLMASP framework, defining its configuration objects, execution pipeline, and semantics independently of any concrete implementation. Section 4 describes the concrete implementation used in our experiments, including the YAML-based configuration files, prompt templates, grammar definitions, and the LLM execution environment. Section 5 reports an extensive experimental evaluation on benchmarks derived from ASP Competitions, analyzing extraction accuracy, hallucinations, efficiency, and energy consumption, as well as the impact of grammar-constrained decoding and knowledge-base–assisted

[4]We release our framework, LLMASP, as open source software at https://github.com/Matteio/LLMASP-JAIR26

reasoning. Finally, Section 6 reviews related work, and Section 7 concludes the paper and discusses directions for future work.

2 Background

2.1 Chat Completion with Large Language Models

Large Language Models (LLMs) are neural architectures trained on massive corpora to model human language with remarkable fluency and generalization. In practical deployments, LLMs are commonly accessed through *chat completion* APIs, which treat models as black-box text transformers. Given an input *prompt*, the model produces a *completion*, that is, a textual response probabilistically generated from the input. Modern LLM APIs also expose token accounting mechanisms. Each interaction is measured in terms of:

- *Prompt tokens*, corresponding to the tokens consumed by the input prompt;
- *Completion tokens*, corresponding to the tokens generated in the output.

These quantities are relevant both for runtime and for inference cost, as completion tokens are typically generated sequentially.

Some LLM frameworks additionally support *grammar-constrained decoding*, allowing generation to be restricted so as to conform to a predefined formal grammar, such as JSON or a given JSON Schema. This capability enables fine-grained control over the structure of generated outputs without requiring model fine-tuning. Formally, let Σ be an alphabet (i.e., a set of symbols), Σ^* be the set of finite strings over Σ, and $\Sigma^{[*]}$ be the set of finite lists over Σ^*. A formal grammar is a 4-tuple $G = \langle N, \Sigma, P, S\rangle$, where

- N is the finite set of non-terminal symbols;
- Σ is a finite set of terminal symbols;
- P is a finite set of production rules of the form $\alpha \Rightarrow \beta$, where α and β are strings in $(N \cup \Sigma)^*$, with at least one non-terminal in α;
- $S \in N$ is the start symbol, from which production begins.

Let $\gamma\alpha\gamma'$ be a string in $(N \cup \Sigma)^*$. The application of $\alpha \Rightarrow \beta$ to $\gamma\alpha\gamma'$ (on the occurrence of α) results to $\gamma\beta\gamma'$. Let G^* be set of strings from Σ^* that can be obtained from S by applying production rules in P. Let $\mathcal{G}_\Sigma$ be the set of formal grammars (over the terminal symbols Σ), and let $\bot$ denote unconstrained generation.

We model *chat completion* as a (stochastic) function:

$$\begin{aligned} cc : \Sigma^* \times \Sigma^{[*]} \times \mathcal{G}_\Sigma &\to \Sigma^* \times \mathbb{N}^2 \\ (\alpha, G) &\mapsto (\beta, t_{in}, t_{out}) \end{aligned}$$

where α is an input text (the prompt), G is an optional grammar, β is the output text, t_{in} is the number of prompt tokens, and t_{out} is the number of completion tokens. In the following, the underling alphabet Σ is assumed to be fixed (and omitted to simplify the presentation).

Example 5 (Grammar-constrained completion). *Consider the alphabet $\Sigma = \{a, b\}$, and the formal grammar $G = \langle\{S\}, \Sigma, \{S \Rightarrow aS, S \Rightarrow b\}, S\rangle$. The language G^* is a^*b, consisting of all strings with zero or more occurrences of a followed by exactly one b.*

Assume that each symbol in Σ corresponds to one token. Let us consider the following grammar-constrained completion calls:

$$\begin{aligned} cc(\ \ a, G) &= (\ \ ab, 1, 2) \\ cc(aa, G) &= (aab, 2, 3) \end{aligned}$$

In the first call, the input text is $a \in \Sigma^*$, which consists of one prompt token, and the grammar-constrained completion produces $ab \in G^*$, consisting of two completion tokens. In the second call, the input text is aa, consisting of two prompt tokens, and the grammar-constrained completion produces aab, consisting of three completion tokens.

Note that the grammar G strictly constrains the space of admissible outputs: no string in $\{a, b\}^* \setminus a^*b$ can be generated by $cc(\cdot, G)$, independently of the prompt. Importantly, the outputs shown above are not deterministic. *Grammar-constrained decoding remains stochastic: at each generation step, the model samples among the admissible next tokens allowed by the grammar according to its learned probability distribution. The grammar restricts* which *tokens may be generated, but does not fix* which admissible token *is chosen. Different runs may therefore produce different strings in G^*, even under the same prompt.* ■

2.2 Answer Set Programming

We formally introduce the notion of fact and refer to the ASP-Core-2 standard (Calimeri et al. 2020) for the full language specification. Let **P** be a fixed nonempty set of *predicate names*, each associated with an *arity*, that is, a non-negative integer. Let **C** be the set of *constants*, consisting of integers and (identifier-like) strings. A *fact* is an expression of the form $p(\overline{c})$, where $p \in \mathbf{P}$ and $\overline{c}$ is a possibly empty sequence of (comma-separated) constants. Let $\mathcal{F}$ denote the set of all possible facts.

A *database* is a possibly empty set of facts, that is, a subset of $\mathcal{F}$. A *program* is a finite set of rules defining how new facts can be derived from an input database or how undesirable solutions can be eliminated. For the purposes of this paper, it is sufficient to treat a program as a function that associates an input database with zero or more output databases according to the stable model semantics (Gelfond and Lifschitz 1990). In the following, programs are also referred to as *knowledge bases*.

In many practical data extraction and post-processing scenarios (such as those considered in this work) ASP programs are used to deterministically derive implicit information from explicitly extracted facts, rather than to enumerate alternative solutions. Accordingly, throughout the paper we assume that the knowledge bases used in the LLMASP pipeline are written so as to yield a single stable model for any given input database. This restriction reflects a modeling choice, and serves to simplify the presentation without loss of generality for the extraction and reasoning tasks addressed here.

Let $\mathcal{D}$ be the set of finite databases and $\mathcal{P}$ the set of finite programs. Under the above assumption, we define the *stable model search* function as:

$$\begin{aligned} search : \mathcal{D} \times \mathcal{P} &\to \mathcal{D} \\ (D_{in}, \Pi) &\mapsto D_{out}, \end{aligned}$$

where D_{in} is the input database, Π is the program, and D_{out} is the stable model of $D_{in} \cup \Pi$.

Example 6. *Let us consider the following ASP program:*

```
request(trousers).   color(shirt,white).          quantity(shirt,2).
request(shirt).      color(trousers, navy_blue).
quantity(Product, 1) :- request(Product), #count{Q: quantity(Product,Q), Q!=1} = 0.
```

The first two lines define five facts, which form the input database D_{in}, while the last line is a rule that assigns a default quantity of 1 to requested products for which no other quantity is specified. The aggregate *#count returns the cardinality of the set specified in its argument. In this case, it provides the number of quantities different from 1 for a given product. In this case, the output database D_{out} is $D_{in} \cup$* `{quantity(trousers, 1)}`. ■

3 The LLMASP Framework

This section introduces *LLMASP*, a configuration-driven framework that integrates Large Language Models (LLMs) with Answer Set Programming (ASP) to extract structured knowledge from natural language and perform symbolic reasoning. LLMASP is defined independently of any specific LLM implementation or configuration serialization format, and operates by composing the chat-completion abstraction introduced in Section 2 with ASP-based stable model computation.

At a high level, LLMASP takes as input a natural language request, a database of facts, two configuration objects that specify how information must be extracted and how reasoning must be performed, and an optional grammars. LLMASP performs multiple predicate-specific calls to an LLM in order to extract facts from text, a technique that has proved effective for this kind of extraction tasks (Alviano and Grillo 2024; Alviano, Grillo, et al. 2025; Alviano, Lo Scudo, et al. 2024). LLMASP returns the extracted facts in the format specified by the optional grammar. Finally, the extracted facts are then combined with an ASP knowledge base into a valid ASP program that is used by an ASP solver to compute the final answer sets.

3.1 Configuration Objects

LLMASP is parameterized by two configuration objects: a *behavior configuration* and an *application configuration*. Conceptually, these configurations decouple prompt construction logic from domain-specific knowledge and reasoning rules. In the current implementation, both configurations are instantiated through YAML files; however, the framework definition presented here is independent of this concrete representation.

Behavior Configuration. The behavior configuration specifies global instructions governing how the LLM is prompted. It defines:

(1) an initialization message providing general instructions to the LLM;
(2) a context *template* used to inject application-specific background information into the LLM prompt;
(3) a mapping *template* used to construct predicate-specific extraction prompts.

Together, these components determine the structure and style of all LLM interactions performed by the framework.

Application Configuration. The application configuration encodes domain-specific extraction targets and symbolic reasoning knowledge. It specifies:

(1) a set of predicate-specific extraction instructions, possibly together with a global contextual description;
(2) an ASP knowledge base defining how extracted facts are combined and processed to derive conclusions.

The application configuration thus determines which predicates are extracted from natural language and how the extracted facts are interpreted by the ASP solver.

We emphasize that the behavior configuration object contains no domain- or predicate-specific information. Instead, it defines only the general prompt skeleton used to instruct the LLM. Whenever LLMASP executes a predicate-specific call, it dynamically populates this skeleton using the metadata provided in the application configuration object.

This declarative separation via configuration objects renders LLMASP both domain-agnostic and highly adaptable across diverse tasks. For instance, LLMASP has been successfully instantiated to power various virtual assistants, including a food marketplace assistant that matches user recommendations with recipes and a warehouse assistant that monitors inventory levels to issue automated replenishment alerts (Alviano, Lo Scudo, et al. 2024). Furthermore, it has been deployed for answer set verbalization (Alviano, Capalbo, et al. 2026), translating formal answer sets into natural language to support explainable AI applications (e.g., explaining Sudoku derivation graphs).

3.2 LLMASP Pipeline

The LLMASP pipeline maps a natural language request to ASP facts and reasoning results through four conceptual steps, denoted **P1**–**P4**. We describe these steps using the following notation. The input consists of a behavior configuration B, an application configuration A, an input database D_{in}, a natural language request T, and optionally a formal grammar G. When a grammar is provided, generation is constrained to strings accepted by G and decoded via an invertible function

$$f_G : G^* \to \mathcal{D},$$

which maps generated strings to databases. If no grammar is provided, generation is unconstrained and f_G is the identity function. For strings a, b, c, let $a[b \mapsto c]$ denote the string obtained by replacing all occurrences of b in a with c. Given a behavior configuration B, let $pre_B(\alpha)$ denote the component associated with identifier $\alpha \in \{\texttt{init}, \texttt{context}, \texttt{mapping}\}$. Given an application configuration A, let $pre_A(\alpha)$ denote the extraction instruction associated with predicate α or with the special symbol _, which represents global contextual information. Finally, let kb_A denote the ASP knowledge base specified by A.

P1. Initialization. The prompt of the LLM is initialized with the message $pre_B(\texttt{init})$, which provides global instructions governing the extraction process. This message forms the initial prefix of the prompt used in all subsequent LLM invocations.

P2. Context Injection. If a global context $pre_A(\_)$ is defined, the prompt prefix is extended with a context message obtained as

$$pre_B(\texttt{context})[\{\texttt{context}\} \mapsto pre_A(\_)].$$

This step injects application-specific background information that is shared across all predicate-specific extraction calls and remains fixed throughout the pipeline.

P3. Predicate-Specific Extraction. For each predicate α such that $pre_A(\alpha)$ is defined and $\alpha \neq \_$, the LLM is invoked with a prompt obtained by *extending the common prefix defined in Steps P1–P2* with a predicate-specific mapping message constructed as

$$\begin{aligned} pre_B(\texttt{mapping})\ &[\{\texttt{input}\} \mapsto T] \\ &[\{\texttt{atom}\} \mapsto f_G^{-1}(\{\alpha\})] \\ &[\{\texttt{instructions}\} \mapsto pre_A(\alpha)]. \end{aligned}$$

Note that the representation of predicate α is adjusted to the format accepted by G by applying the inverse function $f_G^{-1}(\alpha)$. Each invocation aims to extract instances of a single predicate from the input text T. The completion returned by the LLM is decoded (possibly under grammar constraints) into a set of facts, which are accumulated into the set F.

P4. Reasoning. The extracted facts F are combined with the input database D_{in} and the application knowledge base kb_A. An ASP solver is then used to compute the stable model of the resulting program, yielding the output database D_{out}.

Algorithm 1 implements the above steps explicitly. The set F of extracted facts is initialized as empty (line 1) and populated through one chat-completion call per predicate specified by the application configuration (lines 7–10). Throughout the process, the prompt prefix P accumulates the initialization and context messages (lines 2 and 5–6), while each extraction call uses a fresh predicate-specific prompt (line 8). Prompt and completion tokens generated by each call are accumulated in t_{in} and t_{out} (lines 3–4 and 11–12), respectively, enabling a fine-grained analysis of generation cost. Finally, symbolic reasoning is performed by computing the stable model of kb_A coupled with the input database extended by the extracted facts (line 13).

Example 7 (Conceptual execution of LLMASP). *Consider a request asking which colors are available for a given item:*

Algorithm 1: LLMASP

Input : a input text T, a behavior file B, an application file A, a database of ASP facts D_{in}, and an optional grammar G together with a function f_G mapping strings from G^* to facts.

Output : a database D_{out}, and two integers t_{in}, t_{out}.

$F := \emptyset$; // the set of extracted facts

$P := pre_B(\texttt{init})$; // the prompt prefix

$t_{in} := 0$; // number of prompt tokens

$t_{out} := 0$; // number of completion tokens

if $pre_A(\_)$ *is defined* **then**

  $P := P + pre_B(\texttt{context})[\{\texttt{context}\} \mapsto pre_A(\_)]$;

for *key* **such that** $pre_A(key)$ *is defined* **and** $key \neq \_$ **do**

  $prompt := pre_B(\texttt{mapping})[\{\texttt{input}\} \mapsto T]\ [\{\texttt{atom}\} \mapsto f_G^{-1}(\{key\})]\ [\{\texttt{instructions}\} \mapsto pre_A(key)]$;

  $(\beta, t'_{in}, t'_{out}) := cc(P + prompt, G)$;

  $F := F \cup f_G(\beta)$;

  $t_{in} := t_{in} + t'_{in}$;

  $t_{out} := t_{out} + t'_{out}$;

$D_{out} := search(D_{in} \cup F, kb_A)$;

return $(D_{out}, t_{in}, t_{out})$;

"What colors are available for the blazers?"

Assume that the application configuration specifies the predicates `request/1`, `color/2`, `quantity/2`, *and* `ask/2`, *together with an ASP knowledge base relating availability information to user queries. During execution, LLMASP initializes the prompt prefix using the behavior configuration (Step P1) and injects the global application context (Step P2). For example, if* $pre_B(\texttt{context})$ *is*

```
Here is some context that you MUST analyze and remember: {context}
```

and $pre_A(\_)$ *is*

```
The user may request to buy some items, or ask about their attributes.
```

the prompt used by Step P2 is

```
Here is some context that you MUST analyze and remember: The user may request to buy some items,
  or ask about their attributes.
```

After that, LLMASP issues four predicate-specific extraction calls (Step P3), one for each predicate. For example, if $pre_B(\texttt{mapping})$ *is*

```
[INPUT]{input}[/INPUT]

Process the INPUT according to the following instructions:
{instructions}

The output format is [OUTPUT]{atom}[/OUTPUT]

If the answer is not in the INPUT, reply NONE.
```

and $pre_A(\texttt{ask(item, attribute)})$ *is*

```
The user is asking for an "attribute" of an "item".
```

the prompt used by Step P3 for `ask(item, attribute)` *is*

```
[INPUT]What colors are available for the blazers?[/INPUT]

Process the INPUT according to the  following instructions:
The user is asking for an "attribute" of an "item".

The output format is [OUTPUT]ask(item,attribute)[/OUTPUT]

If the answer is not in the INPUT, reply NONE.
```

In this case, Step P3 is expected to extract a fact only for the predicate `ask/2`*, namely* `ask(blazer, color)`*, while the remaining extraction calls return no facts. Note that when G is a grammar accepting comma-separated values (being predicate names and constants), the OUTPUT section in the prompt used by Step P3 is adjusted to*

```
[OUTPUT]ask,item,attribute[/OUTPUT]
```

by the inverse function $f_G^{-1}(\{\texttt{ask(item, attribute)}\})$*, and the generated text*

```
ask,blazer,color
```

is mapped to the fact `ask(blazer,color)` *by the function* f_G *(line 10 of Algorithm 1).*

Finally, the extracted facts are combined with the input database and processed by the ASP solver (Step P4). For example, if D_{in} *is*

```
available_color(blazer, blue).
available_color(blazer, black).
available_color(trousers, navy_blue).
```

and kb_A *is*

```
answer(Item, color, Value) :- ask(Item, color), available_color(Item, Value).
```

the ASP solver yields $D_{out} = \{$ `answer(blazer, color, blue), answer(blazer, color, black)`$\}$. ■

We conclude this section by abstracting the behavior of Algorithm 1 as a function. Let $\mathcal{B}$ be the set of behavior configurations, and $\mathcal{A}$ be the set of application configurations. We define the *LLMASP* operator as

$$\begin{aligned} \mathit{LLMASP} : \Sigma^* \times \mathcal{B} \times \mathcal{A} \times \mathcal{D} \times \mathcal{G}_\Sigma &\to \mathcal{D} \times \mathbb{N}^2 \\ (T, B, A, D_{in}, G) &\mapsto (D_{out}, t_{in}, t_{out}), \end{aligned}$$

where $(D_{out}, t_{in}, t_{out})$ corresponds to the output produced by Algorithm 1 when executed with inputs T, B, A, D_{in}, and grammar G.

4 Implementation

This section describes the concrete realization of the LLMASP framework used in our experiments. We detail how abstract behavior and application configurations are serialized, how grammar-constrained decoding is implemented, and which prompt templates and configurations are employed. All implementation choices are orthogonal to the abstract definition of LLMASP given in Section 3, and are reported here for reproducibility.

4.1 Configuration Files

In our implementation, behavior and application configurations are serialized using YAML files. In particular, preprocessing specifications are represented as ordered collections of extraction directives, allowing the framework to support multiple independent extractions for the same predicate when required. This choice is purely pragmatic and does not affect the abstract definition of LLMASP. Behavior and application files are parsed into in-memory configuration objects that instantiate the parameters B and A used by Algorithm 1.

Example 8 (YAML configuration). *The configuration objects used in Example 7 can be serialized as the following behavior file:*

```
init: |
  You are a Natural Language to Datalog translator.
context: |
  Here is some context that you MUST analyze and remember:
  {context}
mapping: |
  [INPUT]{input}[/INPUT]

  Process the INPUT according to the following instructions:
  {instructions}

  The output format is [OUTPUT]{atom}[/OUTPUT]

  If the answer is not in the INPUT, reply NONE.
```

and the following application file:

```
preprocessing:
  - _: The user may request to buy some items, or ask about their attributes.
  - request(item): The request includes the "item".
  - color(item,value): The requested "item" must have the color "value".
  - quantity(item,value): The quantity of "item" requested is "value".
  - ask(item,attribute): The user is asking for an "attribute" of an "item".
knowledge base: |
  answer(Item, color, Value) :- ask(Item, color), available_color(Item, Value).
```

Note that the `preprocessing` *key of the application file is associated with a list of* atom–instruction *pairs.* ■

4.2 LLM Backend and Execution Environment

We implemented the LLMASP pipeline using the `/chat` completion endpoint of the Ollama framework[5], which provides local execution of large language models and fine-grained control over the generation process. In particular, Ollama natively supports grammar-constrained decoding and exposes detailed token accounting, including the number of prompt and completion tokens consumed by each generation call.

At the time of writing, the `/chat` endpoint supports constrained generation only when the target output is expressed as arbitrary JSON or conforms to a provided JSON Schema.[6] To support additional structured formats suitable for symbolic reasoning, such as CSV and Datalog-style facts, we extend the Ollama framework to accept

[5] https://ollama.com
[6] https://github.com/Ollama/Ollama/pull/4525

custom grammars expressed in BNF, beyond the native support for JSON and JSON Schema. These grammars are used to constrain decoding directly at generation time, while the expected output format in the prompt is adapted accordingly.

Chat completion in Ollama follows a role-based interaction protocol. Each message is associated with a role, namely *system*, *user*, or *assistant*. In our implementation, global instructions derived from the behavior configuration (e.g., initialization messages and contextual information) are injected as *system* messages, while predicate-specific extraction prompts generated by the LLMASP pipeline are issued as *user* messages. Previous model responses are stored as *assistant* messages, but our implementation does not use them (according to the execution flow of Algorithm 1).

In this article, all experiments were conducted using Meta LLaMA models provided by Ollama, specifically `llama3:8B` and `llama3:70B`, representing a small and a large model setting, respectively. However, the LLMASP framework is model-agnostic and can be instantiated with any language model supported by the Ollama framework that exposes a chat-completion interface. Unless otherwise stated, models were used with default decoding parameters, except for the optional grammar constraint when enabled. ASP knowledge bases are evaluated using CLINGO (Gebser, Kaminski, et al. 2011), which is invoked as an external solver to compute the stable model of the logic program resulting from the extraction tasks.

4.3 Prompt Templates and Behavior Configurations

Behavior and application configurations are contingent to the task that LLMASP has to perform and have to be defined and tuned accordingly. In what follows, we describe the behavior and application configurations that we have used to perform information extraction into ASP predicates.

In order to obtain the best application configuration for our purposes, we studied a plethora of configurations by including one or more of the following features:

(E) a generic extraction example in the context;
(F) an explicit description of predicate and term formats in the context;
(I) repeated extraction instructions in the mapping;
(N) an instruction in the context to reply `NONE` when information is missing;
(R) repeated `NONE` instructions in the mapping.

Configurations are denoted by the concatenation of their features (e.g., `EF`).

We have empirically determined that the overall best performing behavior file consists of the EIN configuration; as such, we will be using the EIN behavior file in the following experiments. In appendix A, we provide the results of our empirical investigation (and ablation study) of behavior file configurations that has led us to select the EIN configuration.

EXAMPLE 9. *In the following, we provide an extract of the prompt we give to the LLM when it is constrained using the grammar for CSV. Specifically, we provide few-shot examples (E), including the case where the information required is not present in the input and the model has to answer with* `NONE` *(N):*

```
Here are a few examples:
- [INPUT]Joe is a developer[/INPUT], the instructions are "the job of a person", and the output
    format is [OUTPUT]job,person,value[/OUTPUT]: you MUST reply job,joe,developer
- [INPUT]Mario is a professor[/INPUT], the instructions are "where is the person", and the output
    format is [OUTPUT]location,person,value[/OUTPUT]: you MUST reply NONE
```

The above content is appended to the `init` *section of the behavior file introduced in Example 8.* ■

We also consider two classes of application configurations, following ASP Competition benchmarks:

(A1) minimal context specifying only the format of atoms (i.e., a description of how the predicates to be extracted have to be encoded);
(A2) extended context including both a problem description and atom formats.

To facilitate the comparison between the A1 and the A2 format, in Appendix C we report the A2 application file for the the Permutation Pattern Matching problem showing that A1 is strictly contained in A2.

All behavior configurations are adapted, when necessary, to reflect the output format enforced by the selected grammar (CSV, Datalog, or JSON), while preserving their original structure. Combined with grammar-constrained decoding, these prompt design choices reduce spurious generations and simplify post-processing.

4.4 Grammar-Constrained Decoding

Each grammar G is paired with a decoding function f_G mapping generated strings to sets of ASP facts, as described in Section 3. We consider three output formats: CSV, Datalog-style facts, and JSON.

CSV Grammar. We designed a lightweight grammar that constrains the model to output comma-separated values, where each line represents a relational fact:

```
root ::= line (newline line)* (newline)?
newline ::= "\n"
line ::= cell ("," cell)*
cell ::= integer | identifier
integer ::= "-"? [0-9]+
identifier ::= [a-z][a-zA-Z0-9_]*
```

Each line is parsed by interpreting the first field as the predicate name and the remaining fields as constant arguments. In other words, each generated line has the form $p, t_1, \ldots, p_n$, where $n \geq 0$, and encodes an atom $p(t_1, \ldots, t_n)$.

Datalog Grammar. As an alternative, we also consider a grammar that enforces standard Datalog syntax:

```
root ::= line (newline line)* (newline)?
newline ::= "\n"
line ::= predicate "(" args ")" "."?
predicate ::= identifier
args ::= arg ("," arg)*
arg ::= integer | identifier
integer ::= "-"? [0-9]+
identifier ::= [a-z][a-zA-Z0-9_]*
```

JSON Grammar. As a baseline (for grammar-constrained generation), we exploit the native support for JSON and JSON Schema in Ollama. While expressive, this format introduces additional syntactic overhead for simple relational data.

5 Experiments

To empirically evaluate LLMASP, we based our implementation on Ollama, focusing on two models, LLaMA 3.1 with 8 and LLaMA 3.3 with 70 billions parameters,[7] the OpenAI API, and CLINGO (Gebser, Kaminski, et al. 2011). We also defined a dataset using domains from ASP Competitions (Gebser, Maratea, et al. 2020). Specifically, we use the description of the domain and format of the facts available online, and systematically generate text

[7] The latest versions of these specific sizes in the LLaMA 3 family.

representations (of portions) of the instances. Our objective is to extract factual information from the (generated) text and to evaluate the ability of the model to produce problem-oriented, coherent structures using the F1-score as the primary quality metric.

We define the evaluation protocol as follows. First, we assess the performance of baseline LLMs that use single-call prompting techniques. We then apply LLMASP to evaluate a range of prompt configurations that combine behavior and application files with varying feature sets. Finally, we analyze the impact of grammar-constrained generation within all our baseline models, considering its effects on extraction performance, energy consumption, and the mitigation of hallucinations.

We evaluate three primary approaches:

(1) **LLM (Baseline):** The model is instructed to extract all predicates at once.
(2) **LLMASP:** Our multi-prompt strategy targeting specific predicates.
(3) **LLMASP+KB:** LLMASP extended with an ASP knowledge base to handle implicit rule-based derivations, to address problems whose instances allow portions of the target representation to be derived by ASP once a subset of facts has been extracted.

Each of these approaches is evaluated with and without grammar constrained generation (all the three grammars are used) and with both application files. In the following, to indicate configurations where a grammar is used to constrain the generation, we use the prefix [Grammar]. The application file is indicated as a suffix. For instance, the string '[CSV] LLM ^{A1}' indicates that we are using the approach LLM, constrained by the CSV grammar, prompted with the application file A1.

5.1 Benchmark Dataset

The dataset contains 512 problem instances distributed across 16 domains, with 32 instances per domain. Domains range from graph coloring to incremental scheduling, each requiring the extraction of structured facts from natural language. Table 1 lists the domains and reports the average number of facts to be extracted for each problem instance, thereby giving an indication of the complexity of each domain. This complexity, however, is also influenced by additional factors, including the richness of the textual descriptions and the characteristics of the generated instances.

Text generation is performed by randomly applying a set of alternative templates to randomly selected facts. The templates are designed to promote linguistic diversity and to ensure variability in the resulting natural language descriptions as described in the following example.

Example 10. *The application of some templates of Incremental Scheduling (IS) to the facts*

```
job(1). job(3). job(10). job(17). job(19).
deadline(3,14). deadline(18,36).
job_len(1,14). importance(1,2).
precedes(13,14). precedes(10,11).
precedes(17,18). precedes(19,20).
device(1). job_device(1,1).
offline_instance(1,1).
```

may produce the following text:

> *The job 3 has a deadline 14. The completion time of job 13 must occur before the start time of 14. The job 1 has an importance of 2, must be executed by device 1, needs 14 timesteps to be performed. Job 19 should end before 20 starts. Device 1 has instance one offline. Job 17 must finish before the start time of 18. Job 10 must finish before the start time of 11. The job 18 has a deadline 36.*

5.2 The Impact of Structured Extraction (RQ1 & RQ2)

To evaluate the proposed LLMASP framework by isolating the contributions of the different application files (A1 and A2) and the per-predicate multi-call strategy, we first establish strong baselines by pushing the performance of the single-call LLM as far as possible. We begin by evaluating the prompt recommended by OpenAI for parsing unstructured data (https://platform.openai.com/docs/examples/default-parse-data), which we adapt to conform to our specific data format:

> **User message**: "You will be provided with unstructured data, and your task is to parse it into Datalog facts format."

This represents the LLM (No Info) configuration in Table 2, where no information about the format of the extracted facts is provided. In this configuration, both models exhibit a low F1-score (0.201 for 8B and 0.128 for 70B). A

Table 1. Average number of facts in the instances of each tested domain (overall, and per arity).

ID	*Problem Name*	*Facts* (all arities)	*Arity 1*	*Arity 2*	*Arity 3*
CMSL	Connected Max.-density Still Life	1	1	0	0
CM	Crossing Minimization	53	1	52	0
GG	Graceful Graphs	8	0	8	0
GC	Graph Colouring	109	54	55	0
IS	Incremental Scheduling	77	24	53	0
KT	Knight Tour	3	1	0	0
KTH	Knight Tour with Holes	9	1	8	0
L	Labyrinth	290	1	257	32
MCP	Maximal Clique Problem	19	11	8	0
NM	No Mystery	31	8	15	8
PU	Partner Units	24	9	15	0
PPM	Permutation Pattern Matching	17	1	16	0
RR	Ricochet Robots	22	9	0	13
S	Sokoban	44	32	4	8
SM	Stable Marriage	16	0	0	16
VLP	Valves Location Problem	28	12	8	8

Table 2. Progression of extraction performance (F1-score) from vanilla LLM baselines to the fully integrated LLMASP framework. Results for LLMASP utilize the optimal 'EIN' behavior configuration (see Appendix A for a full ablation).

Configuration	**LLaMA 3 8B**	**LLaMA 3 70B**
LLM (No Info)	0.201	0.128
LLM (Description Only)	0.237	0.273
LLM (Encoding)	0.491	0.806
LLM ^{A1} (Format Only)	0.686	0.832
LLM ^{A2} (Format + Description)	0.721	0.892
LLMASP ^{A1}	0.768	0.912
LLMASP ^{A2}	0.827	0.963
LLMASP+KB ^{A1}	0.823	0.927
LLMASP+KB ^{A2}	0.863	0.965

slight improvement over the LLM (No Info) baseline is obtained by providing the prompt with a description of the domain for which the extraction is being performed (F1-scores of 0.237 for 8B and 0.273 for 70B). A better strategy (Encoding), inspired by prior work exploring the use of LLMs for ASP program synthesis (Ishay et al. 2023; Pan et al. 2023)[8], consists of including the official ASP program from the ASP Competitions in the prompt (note that this does not contain the problem instance). This provides the LLM with a light form of guidance and improves extraction performance (F1-scores reaching 0.491 for 8B and 0.806 for 70B). A more substantial improvement is achieved by providing the exact format of the facts to extract, mimicking the core idea behind the A1 application file (F1-scores of 0.686 and 0.832). Finally, including both descriptions and formats—mimicking the A2 application file—further improves extraction performance (F1-scores of 0.721 and 0.892).

Moving on to the extraction performance of our framework, we find that all configurations outperform the strongest baselines. LLMASP^{A1} achieves F1-scores of 0.768 and 0.912 for the 8B and 70B models, respectively. The application file A2 in LLMASP^{A2} pushes performance even further, exhibiting F1-scores of 0.827 and 0.963.

Finally, we obtain further improvements by integrating a knowledge base into LLMASP (LLMASP+KB). The knowledge base contains ASP rules used to symbolically generate specific portions of the output. Specifically, the domains *Labyrinth* (**L**), *No Mystery* (**NM**), *Ricochet Robots* (**RR**), and *Sokoban* (**S**) contain instances that allow portions of the target representation to be derived via ASP once a minimal subset of facts has been extracted. For instance, the domain *Labyrinth* describes grids of size $N \times N$. Rather than forcing the LLM to produce all N^2 predicates encoding the grid cells, we generate them symbolically as soon as LLMASP successfully extracts the value of N. In Appendix B, we explain in detail how we produced the knowledge base for each of these four domains. The integration of this knowledge base with LLMASP yields an additional increase in performance, with F1-scores reaching 0.823 and 0.927 for LLMASP+KB^{A1}, and 0.863 and 0.965 for LLMASP+KB^{A2}.

As shown in Appendix B, the increase in the F1-score provided by the knowledge base integration is primarily driven by a noticeable reduction in false negatives (i.e., predicates that LLMASP previously failed to extract). The domain-wise analysis (Table 8 in Appendix B) reveals that the most difficult instances for the LLMs, particularly for the 8B model, are those within the *Ricochet Robots* domain. Alongside this domain-wise analysis, Appendix B also offers an examination of extraction performance across different behavior files. Performance under LLMASP+KB exhibits much narrower variability than in the non-KB models, especially for the 70B LLM. This suggests that the inclusion of a knowledge base might render extensive prompt engineering techniques less relevant when deploying large models within the LLMASP framework.

5.3 The impact of Grammar-constrained Generation on LLMASP Performance (RQ3)

We next assess the impact of grammar-constrained generation within LLMASP on both predictive performance and energy consumption. The experiments evaluate grammar-constrained and unconstrained generation across multiple dimensions. We employ the LLMASP pipeline, enforcing grammar constraints through three formal grammars: CSV, Datalog, and JSON.

Performance is evaluated here using the following metrics:

(i) macro-averaged F1-score, computed as the average of the F1-scores across all domains, and micro F1-score, computed by aggregating true positives (TP), false positives (FP), and false negatives (FN) across all instances;

(ii) the percentage of instances that are correctly extracted and directly executable by an ASP solver (*Perfect Rate*);

[8]We have performed a preliminary test of the approach in (Ishay et al. 2023) on our benchmark dataset and obtained an F1-scores of 0.24 for the 8B model and 0.55 for the 70B model. This stems from a fundamental task mismatch: program synthesis is designed to generate problem-solving rules and solve instance logic end-to-end, whereas information extraction requires faithfully mapping unstructured text into ground facts conforming strictly to a predefined schema. Forcing synthesis models into an extraction role yields malformed programs or invalid predicate formats.

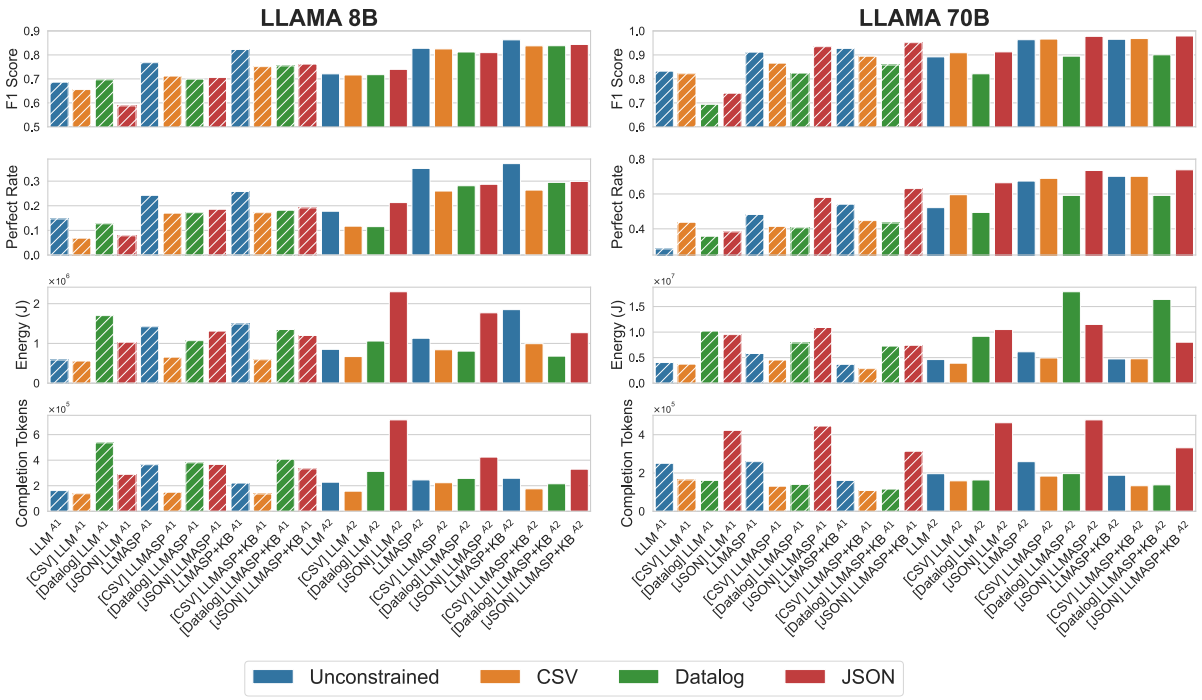

Fig. 2. F1-score, prompt token usage, and completion token usage measured on LLM, LLMASP, using LLaMA 3 8B (left) and 70B (right). Results are ordered by increasing F1-score. Grammars are color-coded, with hatched bars for the application file A1, and full bars for the application file A2.

(iii) energy consumption, measured in Joules; and
(iv) the number of completion tokens.

Figure 2 summarizes the main results across all 48 experimental configurations, where we evaluate how the usage of different grammars impacts both extraction performance and energy expenditure.

As shown in Figure 2, imposing output grammars (thus enforcing structured decoding) acts as a high-leverage control mechanism that improves both reliability (higher F1-score and perfect extraction rate) and efficiency (lower or stable token usage and energy consumption). This effect is particularly evident for non-JSON grammars, which maintain strong extraction quality while substantially reducing the number of generated completion tokens. This suggests that tighter admissible output spaces better support predicate-specific, slot-filling behavior with limited generative effort. In contrast, JSON-style constrained generation with LLaMA 3 70B achieves the highest F1-scores and perfect rates overall, but at the cost of significantly higher completion token counts and, consequently, increased energy consumption. A detailed analysis of the extraction performance of each configuration is reported in Table 3.

On LLaMA 8B, introducing an explicit output grammar generally results in performance degradation for LLMASP variants. Under A1, grammar constraints reduce the macro F1-score of LLMASP from 0.768 to as low as 0.699 (with Datalog) and lower the perfect extraction rate from 0.242 to 0.170 (with CSV). Under A2, the degradation in macro F1 is smaller (from 0.828 to 0.810 with JSON), yet the perfect rate still drops substantially: from 0.352 to 0.260 for LLMASP configurations without a knowledge base, and from 0.371 to 0.264 for configurations supported

by a knowledge base. The vanilla (non-LLMASP) baselines exhibit a similar overall tendency, though not uniformly across grammars. For A1, CSV and JSON exhibit a reduction in macro F1-score (to 0.656 and 0.588, respectively), whereas Datalog yields a slight macro-F1 improvement (0.696 vs. 0.686) but with a decrease in the perfect rate. For A2, JSON is the only grammar that improves both macro F1 (0.740 vs. 0.721) and the perfect extraction rate (0.213 vs. 0.178). Overall, these results indicate that, on smaller models, the benefits of grammar-constrained decoding are neither systematic nor guaranteed, but instead depend strongly on the interaction between grammar choice, configuration, and extraction strategy.

For LLaMA 70B, the picture reverses: grammar constraints (especially JSON) are consistently beneficial in tighter extraction regimes and often translate into higher perfect extraction rates. For LLMASP with the application file A1, the use of a JSON grammar increases the macro F1 from 0.912 to 0.934 and boosts the perfect extraction rate from 0.482 to 0.580. We observe that the micro F1-score decreases slightly (from 0.952 to 0.942), signaling an increase in the total number of false negatives and false positives. We analyze errors and hallucinations in the next section. Under A2, JSON attains the best overall results. For LLMASP without knowledge base support, JSON

Table 3. Extraction performance: Comparison between grammar constrained generation and unconstrained generation. **P**: Precision. **R**: Recall. **F1-macro**: F1-score macro average. **F1-micro**: F1-score micro average. **Perfect**: Percentage of perfectly extracted problem instances. The best results are highlighted in **blue** for 8B and in **bold** for 70B (underlined for KB models).

LLM	LLaMA 8B					LLaMA 70B				
Configuration	P	R	F_{mac}	F_{mic}	Perfect	P	R	F_{mac}	F_{mic}	Perfect
LLM ^{A1}	0.727	0.690	0.686	0.610	0.146	0.862	0.858	0.832	0.709	0.285
[CSV] LLM ^{A1}	0.697	0.657	0.656	0.669	0.068	0.825	0.828	0.822	0.888	0.438
[Datalog] LLM ^{A1}	0.750	0.677	0.696	0.714	0.127	0.720	0.682	0.694	0.723	0.357
[JSON] LLM ^{A1}	0.667	0.563	0.588	0.572	0.078	0.789	0.726	0.740	0.825	0.383
LLMASP ^{A1}	0.775	0.810	0.768	0.707	0.242	0.927	0.911	0.912	0.952	0.482
[CSV] LLMASP ^{A1}	0.724	0.759	0.712	0.635	0.170	0.895	0.867	0.866	0.825	0.414
[Datalog] LLMASP ^{A1}	0.734	0.718	0.699	0.603	0.172	0.861	0.816	0.824	0.763	0.406
[JSON] LLMASP ^{A1}	0.755	0.708	0.706	0.621	0.186	0.961	0.925	0.934	0.942	0.580
LLMASP+KB ^{A1}	0.810	0.868	0.823	0.835	0.258	0.937	0.930	0.927	0.958	0.541
[CSV] LLMASP+KB ^{A1}	0.754	0.795	0.751	0.801	0.172	0.900	0.901	0.894	0.940	0.447
[Datalog] LLMASP+KB ^{A1}	0.772	0.772	0.754	0.791	0.182	0.875	0.855	0.859	0.879	0.434
[JSON] LLMASP+KB ^{A1}	0.786	0.766	0.762	0.812	0.191	0.964	0.948	0.951	**0.972**	0.631
LLM ^{A2}	0.729	0.777	0.721	0.645	0.178	0.891	0.907	0.892	0.830	0.521
[CSV] LLM ^{A2}	0.738	0.740	0.716	0.767	0.117	0.914	0.912	0.909	0.902	0.596
[Datalog] LLM ^{A2}	0.749	0.742	0.718	0.724	0.115	0.827	0.822	0.821	0.860	0.494
[JSON] LLM ^{A2}	0.799	0.725	0.740	0.639	0.213	0.921	0.912	0.913	0.896	0.664
LLMASP ^{A2}	0.812	**0.903**	**0.828**	0.749	**0.352**	0.944	0.991	0.963	0.912	0.674
[CSV] LLMASP ^{A2}	0.804	0.886	0.825	0.774	0.260	0.958	0.975	0.966	0.936	0.689
[Datalog] LLMASP ^{A2}	0.815	0.842	0.812	**0.793**	0.281	0.881	0.914	0.895	0.890	0.592
[JSON] LLMASP ^{A2}	**0.831**	0.828	0.810	0.772	0.287	**0.969**	**0.992**	**0.977**	**0.958**	**0.734**
LLMASP+KB ^{A2}	0.832	**0.941**	**0.863**	0.825	**0.371**	0.946	0.990	0.965	0.939	0.701
[CSV] LLMASP+KB ^{A2}	0.822	0.894	0.838	0.787	0.264	0.950	**0.993**	0.968	0.937	0.701
[Datalog] LLMASP+KB ^{A2}	0.833	0.872	0.838	0.822	0.295	0.889	0.918	0.900	0.893	0.592
[JSON] LLMASP+KB ^{A2}	**0.851**	0.864	0.844	**0.861**	0.299	**0.968**	**0.993**	**0.978**	0.968	**0.738**

scores $F1_{\text{macro}} = 0.977$ and *Perfect* = 0.734 (vs. $F1_{\text{macro}} = 0.963$ and *Perfect* = 0.674 for unconstrained LLMASP). For LLMASP+KB, JSON scores $F1_{\text{macro}} = 0.978$ and *Perfect* = 0.738 (vs. $F1_{\text{macro}} = 0.965$ and *Perfect* = 0.701 for unconstrained LLMASP+KB). CSV also achieves remarkable performance: [CSV] LLMASP^{A2} scores $F1_{\text{macro}} = 0.966$ and *Perfect* = 0.689; [CSV] LLMASP+KB^{A2}, instead, scores $F1_{\text{macro}} = 0.968$ and *Perfect* = 0.701. For the vanilla A2 setting, both CSV and JSON also improve macro F1 (to 0.909 and 0.913) and substantially raise the perfect rate (to 0.596 and 0.664, respectively). In contrast, Datalog is the least reliable grammar across settings, producing sizable macro-F1 drops (e.g., 0.895 vs. 0.963 for LLMASP A2, and 0.694 vs. 0.832 for LLM A1), even when the perfect rate does not always degrade proportionally.

Another important insight from Figure 2 is that CSV is the only grammar that supports a consistent reduction in energy costs and completion tokens (compared to the unconstrained configurations) across all model sizes and configurations. We will analyze this trade-off between performance and energy consumption in Section 5.5.

Overall, the results indicate that the effectiveness of grammar-constrained generation depends jointly on model capacity, the choice of the grammar, and the application file adopted. In general, grammar-constrained generation does not pay off for the 8B model. For the 70B model, the JSON grammar provides the most robust improvements across all configurations. If we limit our considerations to the A2 application file, both JSON and CSV grammars offer performance improvements, with CSV also providing the lowest energy costs.

5.4 Hallucinations and Errors (RQ4)

Although the micro-F1 and macro F1-scores from our previous experiments are highly correlated ($r = 0.90$, $p < 0.001$), we observe some minor misalignments. This observation calls for a deeper look into the extraction errors to understand how grammar-constrained generation affects extraction hallucinations in terms of spurious and redundant facts.

Concretely, we treat false positives (FP) as incorrect facts not supported by the gold annotations, and duplicates (D) as repeated extractions of the same fact (a practical failure mode that inflates the produced fact base and often correlates with repeated hallucinated content). We define D_{uniq} as the number of distinct facts that were extracted more than once. False negatives (FN) quantify omissions, and true positives (TP) quantify correctly extracted facts. Table 4 reports these values aggregated across all domains, directly capturing the overall "error mass" produced by each configuration.

On configurations powered by LLaMA 8B, enforcing structured outputs is generally beneficial for suppressing hallucinations, particularly in configurations employing the A2 application file. The A2 baseline (LLM^{A2}) exhibits 12,739 false positives and 3,549 duplicates. Constraining it with a JSON grammar ([JSON] LLM^{A2}) yields a sharp reduction in hallucinations (FP: from 12,739 to 3,237; D: from 3,549 to 530; and D_{uniq} from 1,067 to 419), albeit with a clear precision-recall trade-off (TP decreases and FN increases). CSV, by contrast, achieves an overall performance improvement: it reduces hallucinations substantially (FP: from 12,739 to 5,414; D: from 3,549 to 773; and D_{uniq} from 1,067 to 659) while also improving coverage (TP: 17,364 to 18,147; and FN: 6,396 to 5,613). With A1, the gains are smaller but still visible: JSON and Datalog reduce FP relative to LLM^{A1}, whereas CSV increases FP, indicating that not all structured formats uniformly constrain the generator under weaker prompting.

A particularly salient pattern emerges under LLMASP. Without an output grammar, LLMASP^{A1} on 8B exhibits substantial redundancy (D = 4,556) alongside elevated FP (7,103), suggesting that when the generator is noisy, downstream reasoning can amplify the volume of candidate facts and their repetitions. Output grammars mitigate this failure mode decisively: JSON reduces duplicates by more than an order of magnitude (from 4,556 down to 368) and also lowers FP (from 7,103 to 5,143). Similar (though slightly weaker) duplicate suppression is achieved by Datalog (D = 400), while CSV offers only a partial improvement (D = 581) and does not reduce FP relative to the LLMASP^{A1} baseline. In A2, the same trend persists: JSON yields the lowest hallucination burden among the LLMASP variants (FP: from 8,789 to 3,867; D: from 1,105 to 345), with the expected cost in FN.

When we turn to configurations supported by a knowledge base, the impact of grammar-constrained generation in reducing hallucinations and duplicates is less noticeable, and it often comes with a reduction in the number of true positives. Indeed, as a symbolic tool for the exact generation of facts, the knowledge base inherently helps to suppress hallucinations and duplicates. In Appendix B, we show that the knowledge base greatly improves performance in the LLaMA 8B powered configurations, which are more error-prone than the 70B ones.

The 70B model is inherently less prone to redundancy, yet structured constraints remain impactful. For single-call configurations, CSV achieves the best overall performance, followed by JSON, with both the A1 and A2 application files. JSON typically achieves a better reduction in false positives and duplicates, but at the same time, it exhibits a higher number of false negatives and a lower number of true positives compared to CSV.

LLMASP improves over single-call baselines across almost all metrics, increasing extraction performance and decreasing the hallucination rate. Yet, grammar-constrained generation still modulates both extraction performance and hallucination rate. For LLMASP^{A1}, JSON reduces FP to 870, and both D and D_{uniq} to 65, at the cost of a lower number of TP and a higher number of FN. For LLMASP^{A2}, CSV yields the highest coverage (TP = 23,533 and FN = 227), whereas JSON yields the lowest D and D_{uniq} (= 122). Interestingly, [CSV] LLMASP^{A2} exhibits a high discrepancy between D (1,986) and D_{uniq} (422). While the number of unique duplicates is reduced compared to the unconstrained LLMASP^{A2} (645), the total number of duplicates more than doubled. This suggests

Table 4. Comparison of the total number of true positive (TP), false positive (FP), false negative (FN), duplicated (D) and distinct duplicated (D_{uniq}) facts extracted by each configuration. The best results are highlighted in **blue** for 8B and in **bold** for 70B (underlined for KB models).

LLM	LLaMA 8B					LLaMA 70B				
Configuration	TP	FP	FN	D	D_{uniq}	TP	FP	FN	D	D_{uniq}
LLM^{A1}	12281	4234	11479	778	409	14781	3172	8979	1686	1600
[CSV] LLM^{A1}	14761	5576	8999	919	747	21782	3534	1978	363	334
[Datalog] LLM^{A1}	15193	3578	8567	922	757	17332	6840	6428	177	154
[JSON] LLM^{A1}	10881	3397	12879	734	470	18952	3207	4808	118	95
LLMASP^{A1}	16893	7103	6867	4556	652	22895	1444	865	135	128
[CSV] LLMASP^{A1}	14493	7379	9267	581	488	18108	2022	5652	313	309
[Datalog] LLMASP^{A1}	13028	6443	10732	400	**297**	16797	3464	6963	154	149
[JSON] LLMASP^{A1}	13003	5143	10757	368	307	21947	**870**	1813	**65**	**65**
LLMASP+KB^{A1}	21841	6725	1919	404	362	23108	1338	652	114	107
[CSV] LLMASP+KB^{A1}	20569	7049	3191	446	387	22844	1980	916	284	280
[Datalog] LLMASP+KB^{A1}	19523	6099	4237	316	**221**	21212	3319	2548	150	145
[JSON] LLMASP+KB^{A1}	19572	4891	4188	**244**	**221**	23264	**840**	496	**62**	**62**
LLM^{A2}	17364	12739	6396	3549	1067	21507	6562	2253	2174	459
[CSV] LLM^{A2}	18147	5414	5613	773	659	21863	3306	1897	713	384
[Datalog] LLM^{A2}	16710	5665	7050	24758	986	20778	3804	2982	731	322
[JSON] LLM^{A2}	12673	**3237**	11087	530	419	21603	2846	2157	423	419
LLMASP^{A2}	19501	8789	4259	1105	491	23460	4252	300	864	645
[CSV] LLMASP^{A2}	**21073**	9609	**2687**	852	647	**23533**	2983	**227**	1986	422
[Datalog] LLMASP^{A2}	19550	6026	4210	541	338	21978	3661	1782	265	261
[JSON] LLMASP^{A2}	17379	3867	6381	**345**	304	23152	1414	608	122	122
LLMASP+KB^{A2}	**22692**	8551	**1068**	754	402	23460	2772	300	616	573
[CSV] LLMASP+KB^{A2}	21480	9347	2280	691	506	23543	2935	217	1983	419
[Datalog] LLMASP+KB^{A2}	20712	5910	3048	479	279	22026	3503	1734	264	260
[JSON] LLMASP+KB^{A2}	20752	**3666**	3008	271	233	**23634**	1422	**126**	122	122

that there are certain instances in our dataset where the LLM struggles particularly under the CSV grammar. In Appendix D, we provide a deep qualitative analysis of these failure modes. In particular, we find that the most challenging domains for CSV-based configurations are Graph Colouring (GC) and Incremental Scheduling (IS)—accounting for 90% of false positives and 93% of duplicates generated by [CSV] LLMASP^{A2}.

With LLaMA 70B, the integration of a knowledge base has a less pronounced impact on reducing hallucinations and duplicates compared to its effect on the 8B model. This is because the 70B model is inherently more precise and less error-prone; furthermore, through the LLMASP framework, it already exhibits strong performance on the four domains coded in the knowledge base (see Appendix B for details). Another interesting observation from Table 4 regards the number of duplicates for [CSV] LLMASP^{A2} and [CSV] LLMASP+KB^{A2}: they remain basically the same, showing that the root of the error does not lie in any of the four domains supported by the knowledge base.

As detailed in Appendix D, our qualitative error analysis reveals that the vast majority of false positives across all LLMASP and LLMASP+KB configurations are concentrated in five domains: Sokoban (S), Graph Colouring (GC), Incremental Scheduling (IS), Stable Marriage (SM), and the Valves Location Problem (VLP). Crucially, these errors are superficial extraction artifacts rather than systemic reasoning failures. A large portion of false positives stem from symmetry canonicalization issues (such as argument order in `link/2` predicates for GC) or ungrounded placeholder generations (such as outputting `deadline(job,NONE)`) in IS.

We demonstrate that these failure modes can be easily mitigated through lightweight post-processing. By applying targeted rules to (i) enforce symmetry for `link/2`, (ii) ensure arguments of `node/1` are integer IDs, and (iii) discard out-of-schema predicates and placeholder strings in IS, we reduced the total false positives for [CSV] LLMASP^{A2} from 2,983 to 1,053 (a 65% reduction). Consequently, this simple cleanup elevated the overall F1-score of [CSV] LLMASP^{A2} from 0.966 to 0.980 across the entire benchmark.

Taken together, these findings suggest that (i) strong models benefit from grammar-constrained generation in that it reduces redundancy and sharpens extraction accuracy; (ii) CSV and JSON lead to distinct error patterns: CSV typically maintains or boosts extraction performance while markedly cutting hallucinations, whereas JSON is usually the most aggressive at removing duplicates and false positives, occasionally resulting in a higher number of false negatives, but frequently reaching the overall best performance.

5.5 Energy Consumption and Performance Trade-offs (RQ5)

Operating cost in our experiments is driven by a combination of model scale, application file configuration, and the strictness of the applied output grammar. In Table 5, we report efficiency indicators, input tokens ($\#t_{in}$), completion tokens ($\#t_{out}$), energy consumption, and runtime, for each configuration. A primary observation is that energy consumption and computation time are virtually interchangeable metrics in this context; they exhibit a near-perfect correlation ($r = 0.998, p < 0.001$). Therefore, optimizing for energy efficiency simultaneously yields equivalent reductions in runtime latency.

Table 5 clearly demonstrates that the choice of output grammar radically shapes the quality-cost plane. Compact grammars, specifically CSV, consistently reduce completion tokens, directly translating to lower energy consumption and faster processing. Conversely, verbose formats like JSON heavily inflate output token volume, resulting in substantial energy and latency penalties despite improving structural reliability. To formally separate the formatting overhead from pure token-volume effects, we fit four log-linear regression models predicting

energy consumption, $\log(E)$, and computation time, $\log(T)$:

Model 1: $\log(E) = \beta_0 + \beta_1 \mathbb{I}[\texttt{llama70b}] + \beta_2 \mathbb{I}[\texttt{A2}] + \beta_3 \mathbb{I}[\texttt{KB}] + \beta_4 \mathbb{I}[\texttt{CSV}] + \beta_5 \mathbb{I}[\texttt{Datalog}] + \beta_6 \mathbb{I}[\texttt{JSON}] + \varepsilon,$

Model 2: $\log(E) = \text{Model 1 terms} + \gamma_1 \texttt{input\_tokens} + \gamma_2 \texttt{completion\_tokens} + \varepsilon$

Model 3: $\log(T) = \beta_0 + \beta_1 \mathbb{I}[\texttt{llama70b}] + \beta_2 \mathbb{I}[\texttt{A2}] + \beta_3 \mathbb{I}[\texttt{KB}] + \beta_4 \mathbb{I}[\texttt{CSV}] + \beta_5 \mathbb{I}[\texttt{Datalog}] + \beta_6 \mathbb{I}[\texttt{JSON}] + \varepsilon,$

Model 4: $\log(T) = \text{Model 3 terms} + \gamma_1 \texttt{input\_tokens} + \gamma_2 \texttt{completion\_tokens} + \varepsilon$

Log-linear models are appropriate in this setting: first, energy values and running times are large (on the order of 10^5 Joules and 10^4 seconds), and second, coefficients can naturally be interpreted as multiplicative effects[9]. All non-token regressors are binary indicators. The reference configuration is expressed by the intercept and

[9]Because the outcome is in log scale, coefficients can be interpreted as percentage changes. More precisely, for a coefficient c, the implied multiplicative change is e^c, and the associated percent change is $(e^c - 1) \times 100$.

Table 5. # t_{in}: Number of tokens in input ($\times 10^6$). # t_{out}: Number of token in output ($\times 10^6$). **Energy**: Energy consumption in Joules ($\times 10^5$). **Time**: Computation time in seconds ($\times 10^3$). The best results are highlighted in **blue** for 8B and in **bold** for 70B (underlined for KB models).

LLM	LLaMA 8B				LLaMA 70B			
Configuration	$\#t_{in}$	$\#t_{out}$	Energy	Time	$\#t_{in}$	$\#t_{out}$	Energy	Time
LLM ^{A1}	**0.239**	0.162	5.8	**5.3**	**0.239**	0.250	40.4	16.7
[CSV] LLM ^{A1}	0.267	**0.138**	**5.6**	5.4	0.267	0.164	**37.2**	**15.8**
[Datalog] LLM ^{A1}	0.267	0.536	17.0	11.1	0.264	0.161	102.0	37.5
[JSON] LLM ^{A1}	0.288	0.289	10.3	6.7	0.289	0.422	95.1	34.6
LLMASP ^{A1}	1.392	0.365	14.2	9.9	1.392	0.259	58.1	23.5
[CSV] LLMASP ^{A1}	1.617	0.149	6.5	5.9	1.617	**0.131**	45.3	18.7
[Datalog] LLMASP ^{A1}	1.618	0.380	10.7	8.1	1.618	0.140	78.7	30.3
[JSON] LLMASP ^{A1}	1.725	0.367	13.1	9.0	1.724	0.445	109.0	40.2
LLMASP+KB ^{A1}	**1.393**	0.220	14.9	9.3	**1.393**	0.161	36.8	15.8
[CSV] LLMASP+KB ^{A1}	1.617	**0.136**	**5.9**	**5.8**	1.617	**0.108**	**28.4**	**13.2**
[Datalog] LLMASP+KB ^{A1}	1.620	0.407	13.5	9.1	1.619	0.116	72.6	28.6
[JSON] LLMASP+KB ^{A1}	1.726	0.332	12.0	8.6	1.726	0.313	74.1	28.3
LLM ^{A2}	0.376	0.228	8.5	6.5	0.376	0.196	46.2	18.4
[CSV] LLM ^{A2}	0.403	0.157	6.7	6.0	0.397	0.164	39.1	16.3
[Datalog] LLM ^{A2}	0.404	0.312	10.6	7.8	0.399	0.163	91.8	34.3
[JSON] LLM ^{A2}	0.423	0.715	23.0	13.0	0.425	0.462	105.0	38.0
LLMASP ^{A2}	2.644	0.245	11.3	8.1	2.644	0.259	61.4	24.4
[CSV]LLMASP ^{A2}	2.868	0.223	8.4	6.6	2.869	0.184	49.3	20.4
[Datalog] LLMASP ^{A2}	2.871	0.258	8.1	6.6	2.855	0.197	179.0	64.8
[JSON] LLMASP ^{A2}	2.970	0.424	17.7	11.1	2.975	0.477	115.0	43.0
LLMASP+KB ^{A2}	2.616	0.259	18.5	11.4	2.616	0.188	47.4	19.4
[CSV] LLMASP+KB ^{A2}	2.840	0.176	9.9	7.7	2.841	0.133	47.7	19.6
[Datalog]LLMASP+KB ^{A2}	2.844	0.215	6.8	6.1	2.821	0.138	164.0	62.1
[JSON] LLMASP+KB ^{A2}	2.948	0.330	12.7	8.9	2.949	0.332	80.2	31.1

consists of LLaMA 8B with the application file A1, unconstrained generation (thus no active grammar), and no knowledge base.

Table 6 summarizes the regression coefficients. The results establish that energy and running time are almost completely determined by the number of tokens processed during inference, with model size (LLaMA 70B) acting as the primary fixed multiplier. Crucially, Model 2 and Model 4 indicate that completion tokens ($\#t_{out}$) are a dominant predictor of both energy consumption ($\gamma_2 = 0.1274, p < 0.001$) and running time ($\gamma_2 = 0.0812, p < 0.01$). Putting these numbers into perspective, given that token counts are scaled by 10^5, an increase of $100,000$ completion tokens causes a 13.6% increase in energy consumption and an 8.5% increase in running time.

Conversely, focusing on reducing the number of completion tokens as a mitigation strategy, a reduction of $100,000$ completion tokens yields a 12.0% reduction in energy consumption and a 7.8% faster execution time.

This reveals that focusing on reducing the number of output tokens is a highly effective strategy for curbing resource demands. Furthermore, even after adjusting for token volume, some residual variation is accounted for by the grammars: CSV maintains a statistically significant energy reduction ($\beta_4 = -0.2367$), whereas the massive

Table 6. Energy consumption and running time as determined by the different configurations of LLM, application file, knowledge base, grammar, and token totals. Heteroskedasticity robust standard errors in parentheses. $^{*}\ p < 0.05$, $^{**}\ p < 0.01$, $^{***} p < 0.001$.

	Energy consumption		Running time	
	Model 1	Model 2	Model 3	Model 4
Intercept	13.6679***	13.2608***	8.7750***	8.4975***
	(0.1447)	(0.1300)	(0.1106)	(0.1002)
llama70b	1.8497***	1.9271***	1.2134***	1.2628***
	(0.0938)	(0.0853)	(0.0762)	(0.0706)
A2	0.1625*	0.0802	0.1338*	0.0592
	(0.0938)	(0.0811)	(0.0762)	(0.0686)
KB	-0.0391	-0.0356	-0.0257	-0.0470
	(0.1122)	(0.1234)	(0.0891)	(0.1004)
CSV	-0.3232**	-0.2367*	-0.2003*	-0.1491
	(0.1254)	(0.1116)	(0.0970)	(0.0881)
Datalog	0.3730*	0.3365*	0.3571**	0.3298*
	(0.1716)	(0.1684)	(0.1368)	(0.1354)
JSON	0.4582***	0.2152	0.3700***	0.2090
	(0.1282)	(0.1315)	(0.0992)	(0.1085)
input_tokens		0.0078		0.0075
		(0.0045)		(0.0039)
completion_tokens		0.1274***		0.0812**
		(0.0369)		(0.0313)
R-squared	0.9147	0.9273	0.8794	0.8944
R-squared Adj.	0.9022	0.9124	0.8618	0.8727
AIC	34.4	30.7	14.5	12.1
F	97.1	106.2	60.3	61.5
F p-val	< 0.001	< 0.001	< 0.001	< 0.001

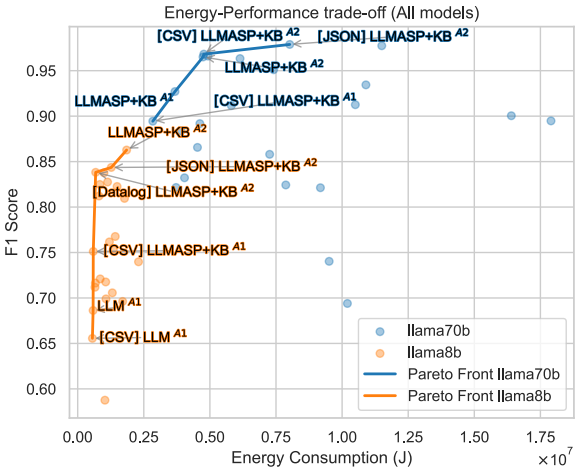

Fig. 3. Energy-Performance trade-off for all the 48 configurations. The Pareto front is largely dominated by configurations powered by knowledge base, which addresses the most difficult problems in the dataset.

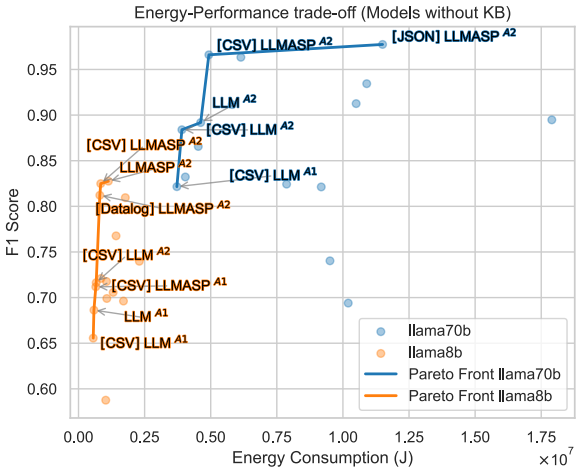

Fig. 4. Energy-Performance trade-off for all the configurations which are not powered by the knowledge base. The Pareto front is largely dominated by the CSV constrained configurations.

+58.1% energy inflation caused by JSON ($\beta_6 = 0.4582$ in Model 1) vanishes entirely in Model 2 once its token expansion is controlled for.

The coefficient for the integration of a knowledge base (β_3) is negative for all models, suggesting that the knowledge base impacts energy consumption and execution time by reducing them, though this effect is not statistically significant. This might be due to a lack of sufficient data, as the knowledge base is employed on only four domains out of sixteen. Therefore, in Appendix B, we offer a qualitative analysis of the knowledge base's impact on energy consumption and execution time, limited to the four domains on which it was employed. Such an analysis shows that the use of a knowledge base helps mitigate energy consumption and execution time, although the effect is mediated by the grammar employed. In sum, the preceding analysis confirms that constraining LLM generation via a compact grammar, such as the CSV grammar, is an effective design choice for mitigating energy consumption and, consequently, speeding up execution time.

Combining these results with our previous analysis of extraction performance yields a critical insight: employing the CSV grammar is the most cost-effective approach. It maintains performance metrics on par with unconstrained LLMASP while keeping computational overhead comparable to single-call LLM strategies. We visualize this balance in the *energy-performance* plane via Pareto efficiency frontiers (Figures 3 and 4). In Figure 3, the frontier is predictably dominated by KB-supported configurations, as offloading extraction to symbolic logic structurally shrinks completion length while preserving precision. However, when isolating purely neural capabilities (Figure 4), the Pareto front is decisively dominated by CSV-constrained setups. For instance, unconstrained LLMASP^{A2} (LLaMA 3 70B) achieves $F1 = 0.963$ at an energy cost of $E = 61.4 \times 10^5$ J, whereas [CSV] LLMASP^{A2} achieves comparable performance ($F1 = 0.966$) while reducing energy expenditure by nearly 20% ($E = 49.3 \times 10^5$ J). Even more striking is the comparison against the single-call baseline LLM^{A2} (LLaMA 3 70B), which yields $F1 = 0.892$ at $E = 46.2 \times 10^5$ J. In effect, [CSV] LLMASP^{A2} delivers state-of-the-art extraction performance for an energy increase of just 6.7% over the single-call baseline. In contrast, the marginal 1.1% gain in extraction

performance offered by JSON-constrained configurations comes at a steep price, consuming up to 133% more energy than CSV-constrained setups.

Ultimately, lightweight serialization constraints like CSV successfully suppress redundant LLM generation, lowering operational overhead through strict completion truncation without compromising logical extraction fidelity.

5.6 Answer to Research Questions

We answer our research questions:

Q1: Can predicate-specific extraction, driven by structured prompt templates, improve the accuracy of LLM-based fact extraction compared to vanilla prompting strategies?

Answer: Yes. Across all evaluated models and domains, predicate-specific extraction as implemented in LLMASP consistently outperforms vanilla prompting. By decomposing the extraction task into focused, predicate-level queries guided by structured prompt templates, LLMASP significantly improves both precision and recall. In particular, we observe substantial gains in F1-score over vanilla prompting, with improvements of up to 10.6 percentage points for LLaMA 3 8B and up to 8.0 percentage points for LLaMA 3 70B, confirming that structured, predicate-aware prompting is a key driver of extraction accuracy.

Q2: Can explicit domain knowledge encoded in Answer Set Programming be integrated with neural extraction to derive implicit facts and improve overall correctness?

Answer: Yes. Integrating explicit domain knowledge encoded as ASP rules with neural extraction yields further improvements in overall correctness. By delegating the derivation of implicit or structurally regular facts to an ASP solver, LLMASP reduces the burden on the LLM and recovers facts that are logically entailed but often omitted during extraction. Empirically, this integration leads to consistent gains in recall without degrading precision, particularly in domains where structural constraints or transitive relationships are prominent, demonstrating the effectiveness of combining neural extraction with symbolic reasoning.

Q3: Can the generation phase be constrained in order to reduce hallucinations and unnecessary completion tokens without sacrificing extraction accuracy?

Answer: Yes. Grammar-constrained configurations based on the CSV format consistently achieve performance comparable to unconstrained LLMASP while significantly reducing completion tokens and energy consumption.

Q4: Do formal grammars effectively reduce hallucinations in structured fact extraction?

Answer: Yes. JSON-style grammar constraints yield the lowest number of hallucinated and duplicated facts across all settings, while CSV constraints are also effective, particularly with the larger LLaMA 70B model and richer application descriptions (A2).

Q5: Which grammar offers the best trade-off between extraction quality and generation efficiency?

Answer: While JSON-based decoding achieves the highest extraction accuracy, CSV emerges as the most balanced solution overall, offering strong accuracy with substantially lower energy consumption and dominating the Pareto front in most experimental settings.

All in all, these results demonstrate that grammar-constrained decoding is an effective and principled mechanism for controlling both the accuracy and the computational footprint of LLMASP.

6 Related Work

Large Language Models (LLMs), such as GPT-3 (Brown et al. 2020), PaLM (Chowdhery et al. 2023), and LLaMA (Touvron et al. 2023), excel at sequence-to-sequence (seq2seq) tasks where text inputs are mapped to generated text outputs. Applications of seq2seq modeling range from machine translation (Dabre et al. 2020) and factual question answering (Petroni et al. 2019) to arithmetic reasoning (Hoffmann et al. 2022), text summarization (H. Zhang et al. 2019), and dialogue systems (Liu et al. 2022). According to (Nye et al. 2021; Ren et al. 2025), LLMs are better suited for data extraction than for complex reasoning, particularly in scenarios requiring grounded statements and logical inference.

ASP Program Synthesis and Knowledge Extraction. While LLMs can effectively predict plausible text continuations, they often lack the necessary understanding of logic, causality, and probabilistic inference to output provably factual text. This limitation has fueled a growing interest in neurosymbolic AI, which aims to combine the unmatched language fluency of LLMs with the rigor and exactness of symbolic solvers.

Because ASP is an excellent language for knowledge representation and non-monotonic reasoning, several efforts have focused on integrating ASP with LLMs. For example, AQuA (Basu, Shakerin, et al. 2020) (ASP-based Question Answering) combines ASP and YOLO to answer natural language questions about visual inputs; to provide factual answers, the input query is translated into ASP through NLP techniques.

More broadly, LLMs have become standard tools for translating natural language into ASP or other logic-based representations amenable to symbolic inference (Kaur et al. 2025; Rajasekharan et al. 2023; Z. Wang et al. 2024; Yang et al. 2024). Some approaches are tailored toward specific tasks, such as verifying claims with logic solvers by translating them into first-order logic propositions (Olausson et al. 2023; H. Wang and Shu 2023). Similarly, the STAR framework (Rajasekharan et al. 2023) generates ASP code to solve Natural Language Understanding (NLU) tasks with the assistance of a dedicated commonsense reasoner.

Other studies aim for broader approaches capable of solving a wider array of tasks. This has been achieved through various techniques, including (i) sequential prompts that generate ASP programs by extracting constants first, then predicates, and finally generating the rules (Ishay et al. 2023); (ii) the fine-tuning of specialized LLMs to support diverse logic tasks (Coppolillo et al. 2024); and (iii) solver-in-the-loop methodologies that use feedback from the solver either to refine the extracted ASP code on the fly (Pan et al. 2023) or to guide the LLM fine-tuning stage (Schrader et al. 2026).

These prior approaches, however, primarily focus on end-to-end ASP program synthesis: mapping a natural language specification directly to a complete ASP program. In this paper, we tackle a distinct and highly modular problem: *knowledge extraction*. Historically, mapping text to structured data fell under the umbrella of classical Information Extraction (IE) and Named Entity Recognition (NER), which relied on rigid, multi-stage pipelines involving tokenization, dependency parsing, and rule-based relation extraction. Here, we demonstrate how LLMs bypass these traditional, often fragile pipelines by translating unstructured natural language directly into a rigorous set of ASP facts.

This extracted representation effectively acts as a factual database over which custom, pre-defined ASP programs can operate. In industrial and enterprise applications (e.g., warehouse inventory monitoring or automated recipe order-fulfillment), the domain logic and operational constraints remain static; it is the unstructured input data that varies. By decoupling extraction from reasoning, the underlying ASP rules can be written once by domain experts—guaranteeing determinism, safety, and correctness—while the LLM handles the unstructured extraction directly. Our proposed LLMASP framework fills exactly this void, as demonstrated in our prior work (Alviano, Capalbo, et al. 2026; Alviano and Grillo 2024; Alviano, Lo Scudo, et al. 2024).

The Energy-Performance Trade-Off. The exceptional performance of LLM-based systems comes at a steep computational and environmental price. As inference demands scale, reducing the energy expenditure of these

models without compromising their accuracy has become a challenge of paramount importance. Because autoregressive generation scales linearly with sequence length, the number of generated completion tokens serves as an excellent proxy for energy consumption, as demonstrated by our experiments. Consequently, minimizing completion tokens offers a direct, operational strategy for cost reduction.

Recent literature has explored various techniques to mitigate these inference costs, though often by trading off performance or adding pipeline complexity. For instance, (Ruiz et al. 2024) employs a two-stage pipeline where a distilled LLM produces a compact intermediate representation that a smaller model subsequently expands. In the context of Chain-of-Thought (CoT) prompting (Wei et al. 2022), (Xia et al. 2025) propose TokenSkip to selectively omit less informative reasoning tokens. Other approaches impose strict generation limits: Concise Thoughts (CCoT) (Nayab et al. 2024) enforces a fixed global token budget, while (J. Wang et al. 2024) dynamically estimates a token budget using an auxiliary LLM, a method that unfortunately introduces its own latency and computational overhead.

Unlike these methods, which attempt to compress outputs post-generation or rely on heuristic token budgets, our approach structurally enforces efficiency *at generation time*. By utilizing grammar-constrained decoding, we physically prevent the model from generating superfluous tokens, enabling predictable, low-cost, and syntactically well-formed outputs without requiring auxiliary models.

Grammar-constrained decoding limits generation strictly to outputs that conform to a specified formal language. Prior work has successfully applied this concept to semantic role labeling (Deutsch et al. 2019), entity disambiguation (Cao et al. 2021), and code generation (Scholak et al. 2021). Recently, SynCode (Ugare et al. 2024) introduced a framework for context-free grammars (CFGs) that incrementally parses LLM outputs to filter invalid tokens on the fly.

While general-purpose frameworks like SynCode are powerful for complex, hierarchical CFGs, our method targets a deliberately narrower and highly efficient use case. By constraining the output to regular-language formats such as CSV (where the structure is flat, repetitive, and inherently finite-state), we achieve lightweight syntactic filtering with negligible computational overhead. Although our constraints are expressed using standard BNF notation, the grammars strictly define regular languages devoid of recursion or deep nesting. As such, our constraints can be enforced using finite-state automata (FSA).

Consequently, our grammar-constrained approach provides a robust mechanism to guarantee syntactic correctness while drastically minimizing completion token usage and latency. This successfully reconciles the traditionally opposing objectives of reducing energy expenditure and maintaining high performance, making it highly effective for scalable symbolic pipelines like LLMASP.

7 Conclusions

In this article, we presented LLMASP, a modular neuro-symbolic framework that integrates Large Language Models with Answer Set Programming to support reliable extraction of structured facts from natural language and their subsequent use for logical reasoning. LLMASP is designed around predicate-specific extraction, declarative configuration, and a clean separation between linguistic processing and symbolic inference.

We enriched the LLMASP architecture by introducing a grammar-constrained decoding module that enforces syntactic validity of generated outputs through formal grammars. We investigated three output formats (CSV, Datalog, and JSON) and analyzed their impact on extraction quality, token usage, runtime, and energy consumption. We performed a systematic and fine-grained empirical evaluation across 48 configurations, covering all possible combinations of model sizes, application settings, extraction strategies, and decoding formats.

Our results show that grammar-constrained decoding is an effective mechanism for balancing extraction accuracy and costs within LLMASP. Compact grammars such as CSV substantially reduce completion tokens and energy consumption while preserving, and in some cases improving, extraction accuracy. JSON-based

configurations, although more verbose and energy consuming, consistently yield among the highest accuracy and the lowest hallucination rates, suggesting that stronger structural constraints can guide large models toward more faithful and stable completions. Overall, CSV emerges as the most cost-effective format, dominating the energy-performance Pareto frontier in most settings.

Beyond grammar-constrained decoding, our experiments confirm the effectiveness of predicate-specific extraction combined with the selective use of explicit symbolic knowledge encoded as ASP rules. Delegating the derivation of implicit knowledge to ASP improves recall without sacrificing precision, highlighting the mutually reinforcing relation between neural extraction and symbolic reasoning.

Taken together, these results demonstrate that LLMASP provides a practical, extensible, and energy-aware neuro-symbolic pipeline with the potential to serve as a foundational backbone for complex AI applications. By combining declarative configuration, structured decoding, and logical inference, LLMASP offers several benefits in terms of robustness, scalability, and interpretability over monolithic prompting strategies. To foster reproducibility and further research, we release the LLMASP implementation, grammar definitions, and experimental scripts as open-source software.

As future work, we plan to investigate the complementary direction of the pipeline, namely the generation of natural language explanations from symbolic outputs. In particular, we aim to study how LLMs can be used to post-process answer sets produced by an ASP solver, translating them into faithful, concise, and user-oriented natural language responses. This direction represents the natural completion of LLMASP, enabling fully bidirectional interactions between natural language processing and symbolic reasoning, thereby combining language fluency with the guarantees offered by logic-based inference. Such an extension can position LLMASP as a general framework for explainable and interactive neuro-symbolic systems.

Acknowledgments

This work was supported by the Italian Ministry of University and Research (MUR) under PRIN project PRODE "Probabilistic declarative process mining", CUP H53D23003420006, under PNRR project FAIR "Future AI Research", CUP H23C22000860006, under PNRR project Tech4You "Technologies for climate change adaptation and quality of life improvement", CUP H23C22000370006, and under PNRR project SERICS "SEcurity and RIghts in the CyberSpace", CUP H73C22000880001; by the Italian Ministry of Health (MSAL) under POS projects CAL.HUB.RIA (CUP H53C22000800006) and RADIOAMICA (CUP H53C22000650006); by the Italian Ministry of Enterprises and Made in Italy under project STROKE 5.0 (CUP B29J23000430005); under PN RIC project ASVIN "Assistente Virtuale Intelligente di Negozio" (CUP B29J24000200005); by Regione Calabria under PR CALABRIA FESR FSE 2021–2027 projects NextGenGuides "Innovazione Tecnologica per le Guide Turistiche del Futuro tramite l'ausilio di tecnologie innovative AI e sistemi di geolocalizzazione avanzata" (CUP J39I24002040005); and MOZART "Modelli e tecniche di mOdernizzazione di applicaZioni e data silos a supporto di clienti esterni, field force e impiegati di backoffice basati su AI GeneRativa e apprendimento auTomatico" (CUP J49I24001740005); and by the LAIA lab (part of the SILA labs). Mario Alviano is a member of Gruppo Nazionale Calcolo Scientifico-Istituto Nazionale di Alta Matematica (GNCS-INdAM).

References

M. Alviano, M. Capalbo, and S. A. Piccolo. 2026. "A Map–Summarize Framework for Answer Set Verbalization." In: *Proceedings of the 23rd International Conference on Principles of Knowledge Representation and Reasoning (KR 2026)*. Recipe ASP Chef: https://asp-chef.netlify.app/s/KR2026/verbalizing-asp.

M. Alviano and L. Grillo. 2024. "Answer Set Programming and Large Language Models Interaction with YAML: Preliminary Report." In: *CILC 2024, Rome, Italy, June 26-28, 2024* (CEUR Workshop Proceedings). Ed. by E. D. Angelis and M. Proietti. Vol. 3733. CEUR-WS.org. https://ceur-ws.org/Vol-3733/short2.pdf.

M. Alviano, L. Grillo, F. Lo Scudo, and L. A. Rodriguez Reiners. 2025. "Integrating Answer Set Programming and Large Language Models for Enhanced Structured Representation of Complex Knowledge in Natural Language." In: *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2025, Montreal, Canada, August 16-22, 2025*. ijcai.org, 4330–4338. doi:10.24963/IJCAI.2025/482.

M. Alviano, F. Lo Scudo, L. Grillo, and L. A. R. Reiners. 2024. "Answer Set Programming and Large Language Models interaction with YAML: Second Report." In: *KoDis+CAKR+SYNERGY@KR 2024, Hanoi, Vietnam, November 2-8, 2024* (CEUR Workshop Proceedings). Ed. by L. G. Á. et al. Vol. 3876. CEUR-WS.org. https://ceur-ws.org/Vol-3876/paper3.pdf.

J. W. Backus et al.. 1963. "Revised report on the algorithmic language ALGOL 60." *Comput. J.*, 5, 4, 349–367. doi:10.1093/COMJNL/5.4.349.

K. Basu, F. Shakerin, and G. Gupta. 2020. "AQuA: ASP-Based Visual Question Answering." In: *Practical Aspects of Declarative Languages - 22nd International Symposium, PADL 2020, New Orleans, LA, USA, January 20-21, 2020, Proceedings* (Lecture Notes in Computer Science). Ed. by E. Komendantskaya and Y. A. Liu. Springer, 57–72. doi:10.1007/978-3-030-39197-3_4.

K. Basu, S. C. Varanasi, F. Shakerin, J. Arias, and G. Gupta. 2021. "Knowledge-driven Natural Language Understanding of English Text and its Applications." In: *AAAI 2021, February 2-9, 2021*. AAAI Press, 12554–12563. doi:10.1609/AAAI.V35I14.17488.

T. B. Brown et al.. 2020. "Language Models are Few-Shot Learners." In: *NeurIPS 2020, December 6-12, 2020, virtual*. Ed. by H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin. https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html.

F. Calimeri et al.. 2020. "ASP-Core-2 Input Language Format." *TPLP*, 20, 2, 294–309. doi:10.1017/S1471068419000450.

N. D. Cao, G. Izacard, S. Riedel, and F. Petroni. 2021. "Autoregressive Entity Retrieval." In: *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=5k8F6UU39V.

P. Cappanera, M. Gavanelli, M. Nonato, and M. Roma. 2023. "Logic-Based Benders Decomposition in Answer Set Programming for Chronic Outpatients Scheduling." *TPLP*, 23, 4, 848–864. doi:10.1017/S147106842300025X.

M. Cardellini, P. D. Nardi, C. Dodaro, G. Galatà, A. Giardini, M. Maratea, and I. Porro. 2024. "Solving Rehabilitation Scheduling Problems via a Two-Phase ASP Approach." *TPLP*, 24, 2, 344–367. doi:10.1017/S1471068423000030.

A. Chowdhery et al.. 2023. "PaLM: Scaling Language Modeling with Pathways." *J. Mach. Learn. Res.*, 24, 240:1–240:113. http://jmlr.org/papers/v24/22-1144.html.

E. Coppolillo, F. Calimeri, G. Manco, S. Perri, and F. Ricca. 2024. "LLASP: fine-tuning large language models for answer set programming." *arXiv preprint arXiv:2407.18723*.

R. Dabre, C. Chu, and A. Kunchukuttan. 2020. "A survey of multilingual neural machine translation." *ACM Computing Surveys (CSUR)*, 53, 5, 1–38.

D. Deutsch, S. Upadhyay, and D. Roth. 2019. "A General-Purpose Algorithm for Constrained Sequential Inference." In: *Proceedings of the 23rd Conference on Computational Natural Language Learning, CoNLL 2019, Hong Kong, China, November 3-4, 2019*. Ed. by M. Bansal and A. Villavicencio. Association for Computational Linguistics, 482–492. doi:10.18653/V1/K19-1045.

M. Gebser, R. Kaminski, and T. Schaub. 2011. "Complex optimization in answer set programming." *TPLP*, 11, 4-5, 821–839.

M. Gebser, M. Maratea, and F. Ricca. 2020. "The Seventh Answer Set Programming Competition: Design and Results." *TPLP*, 20, 2, 176–204. doi:10.1017/S1471068419000061.

M. Gelfond and V. Lifschitz. 1990. "Logic programs with classical negation." In: *Logic Programming: Proc. of the Seventh International Conference*. Ed. by D. Warren and P. Szeredi, 579–597.

S. Geng, M. Josifoski, M. Peyrard, and R. West. 2023. "Grammar-Constrained Decoding for Structured NLP Tasks without Finetuning." In: *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*. Ed. by H. Bouamor, J. Pino, and K. Bali. Association for Computational Linguistics, 10932–10952. doi:10.18653/V1/2023.EMNLP-MAIN.674.

J. Hoffmann et al.. 2022. "Training compute-optimal large language models." *arXiv preprint arXiv:2203.15556*.

A. Ishay, Z. Yang, and J. Lee. 2023. "Leveraging large language models to generate answer set programs." In: *Proceedings of the 20th International Conference on Principles of Knowledge Representation and Reasoning*, 374–383.

N. Jegham, M. Abdelatti, L. Elmoubarki, and A. Hendawi. 2025. *How Hungry is AI? Benchmarking Energy, Water, and Carbon Footprint of LLM Inference*. (2025). https://arxiv.org/abs/2505.09598 arXiv: 2505.09598 (cs.CY).

H. Jin, Y. Zhang, D. Meng, J. Wang, and J. Tan. 2024. "A Comprehensive Survey on Process-Oriented Automatic Text Summarization with Exploration of LLM-Based Methods." *CoRR*, abs/2403.02901. arXiv: 2403.02901. doi:10.48550/ARXIV.2403.02901.

N. Kaur, L. McPheat, A. Russo, A. G. Cohn, and P. Madhyastha. 2025. *An Empirical Study of Conformal Prediction in LLM with ASP Scaffolds for Robust Reasoning*. (2025). https://arxiv.org/abs/2503.05439 arXiv: 2503.05439 (cs.CL).

Z. Li, Y. Cao, X. Xu, J. Jiang, X. Liu, Y. S. Teo, S. Lin, and Y. Liu. 2024. "LLMs for Relational Reasoning: How Far are We?" In: *LLM4CODE@ICSE*, 119–126. doi:10.1145/3643795.3648387.

Z. Liu, M. Patwary, R. Prenger, S. Prabhumoye, W. Ping, M. Shoeybi, and B. Catanzaro. 2022. "Multi-Stage Prompting for Knowledgeable Dialogue Generation." In: *ACL 2022*, 1317–1337.

V. Marek and M. Truszczyński. 1999. "Stable models and an alternative logic programming paradigm." In: *The Logic Programming Paradigm: a 25-year Perspective*, 375–398. doi:10.1007/978-3-642-60085-2_17.

S. Nayab, G. Rossolini, G. C. Buttazzo, N. Manes, and F. Giacomelli. 2024. "Concise Thoughts: Impact of Output Length on LLM Reasoning and Cost." *CoRR*, abs/2407.19825. arXiv: 2407.19825. doi:10.48550/ARXIV.2407.19825.

I. Niemelä. 1999. "Logic programming with stable model semantics as a constraint programming paradigm." *Annals of Mathematics and Artificial Intelligence*, 25, 3,4, 241–273. doi:10.1023/A:1018930122475.

M. Nye, M. Tessler, J. Tenenbaum, and B. M. Lake. 2021. "Improving coherence and consistency in neural sequence models with dual-system, neuro-symbolic reasoning." *Advances in Neural Information Processing Systems*, 34, 25192–25204.

T. Olausson, A. Gu, B. Lipkin, C. Zhang, A. Solar-Lezama, J. Tenenbaum, and R. Levy. 2023. "LINC: A neurosymbolic approach for logical reasoning by combining language models with first-order logic provers." In: *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 5153–5176.

L. Pan, A. Albalak, X. Wang, and W. Wang. Dec. 2023. "Logic-LM: Empowering Large Language Models with Symbolic Solvers for Faithful Logical Reasoning." In: *Findings of the Association for Computational Linguistics: EMNLP 2023*. Ed. by H. Bouamor, J. Pino, and K. Bali. Association for Computational Linguistics, Singapore, (Dec. 2023), 3806–3824. doi:10.18653/v1/2023.findings-emnlp.248.

K. Park, J. Wang, T. Berg-Kirkpatrick, N. Polikarpova, and L. D'Antoni. 2024. *Grammar-Aligned Decoding*. (2024). https://arxiv.org/abs/2405.21047 arXiv: 2405.21047 [cs.AI].

P. Patel, E. Choukse, C. Zhang, Í. Goiri, A. Shah, S. Maleki, and R. Bianchini. 2023. "Splitwise: Efficient generative LLM inference using phase splitting." *CoRR*, abs/2311.18677. arXiv: 2311.18677. doi:10.48550/ARXIV.2311.18677.

F. Petroni, T. Rocktäschel, P. Lewis, A. Bakhtin, Y. Wu, A. H. Miller, and S. Riedel. 2019. "Language models as knowledge bases?" *arXiv preprint arXiv:1909.01066*.

A. Rajasekharan, Y. Zeng, P. Padalkar, and G. Gupta. 2023. "Reliable Natural Language Understanding with Large Language Models and Answer Set Programming." In: *ICLP 2023, Imperial College London, UK, 9th July 2023 - 15th July 2023* (EPTCS). Ed. by E. Pontelli et al. Vol. 385, 274–287. doi:10.4204/EPTCS.385.27.

L. Ren, G. Xiao, G. Qi, Y. Geng, and H. Xue. Oct. 2025. "Can LLMs Solve ASP Problems? Insights from a Benchmarking Study." In: *Proceedings of the 22nd International Conference on Principles of Knowledge Representation and Reasoning*. (Oct. 2025), 621–631. doi:10.24963/kr.2025/60.

A. G. Ruiz, T. de la Rosa, and D. Borrajo. 2024. "TRIM: Token Reduction and Inference Modeling for Cost-Effective Language Generation." *CoRR*, abs/2412.07682. arXiv: 2412.07682. doi:10.48550/ARXIV.2412.07682.

T. Scholak, N. Schucher, and D. Bahdanau. 2021. "PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models." In: *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*. Ed. by M. Moens, X. Huang, L. Specia, and S. W. Yih. Association for Computational Linguistics, 9895–9901. doi:10.18653/V1/2021.EMNLP-MAIN.779.

T. P. Schrader, L. Lange, T. Kaminski, S. Razniewski, and A. Friedrich. 2026. "A solver-in-the-loop framework for improving LLMs on answer set programming for logic puzzle solving." In: *Proceedings of the AAAI Conference on Artificial Intelligence* 30. Vol. 40, 25226–25234.

R. Taupe, G. Friedrich, K. Schekotihin, and A. Weinzierl. 2021. "Solving Configuration Problems with ASP and Declarative Domain Specific Heuristics." In: *Proceedings of the 23rd International Configuration Workshop (CWS/ConfWS 2021), Vienna, Austria, 16-17 September, 2021* (CEUR Workshop Proceedings). Ed. by M. Aldanondo, A. A. Falkner, A. Felfernig, and M. Stettinger. Vol. 2945. CEUR-WS.org, 13–20. https://ceur-ws.org/Vol-2945/21-RT-ConfWS21_paper_4.pdf.

H. Touvron et al.. 2023. "LLaMA: Open and Efficient Foundation Language Models." *CoRR*, abs/2302.13971. arXiv: 2302.13971. doi:10.48550/ARXIV.2302.13971.

S. Ugare, T. Suresh, H. Kang, S. Misailovic, and G. Singh. 2024. "SynCode: LLM generation with grammar augmentation." *arXiv preprint arXiv:2403.01632*.

H. Wang and K. Shu. 2023. "Explainable claim verification via knowledge-grounded reasoning with large language models." In: *Findings of the association for computational linguistics: EMNLP 2023*, 6288–6304.

J. Wang, S. Jain, D. Zhang, B. Ray, V. Kumar, and B. Athiwaratkun. 2024. "Reasoning in Token Economies: Budget-Aware Evaluation of LLM Reasoning Strategies." In: *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024*. Ed. by Y. Al-Onaizan, M. Bansal, and Y. Chen. Association for Computational Linguistics, 19916–19939. https://aclanthology.org/2024.emnlp-main.1112.

Z. Wang, J. Liu, Q. Bao, H. Rong, and J. Zhang. 2024. "ChatLogic: Integrating Logic Programming with Large Language Models for Multi-Step Reasoning." In: *International Joint Conference on Neural Networks, IJCNN 2024, Yokohama, Japan, June 30 - July 5, 2024*. IEEE, 1–8. doi:10.1109/IJCNN60899.2024.10650138.

J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou. 2022. "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models." In: *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*. Ed. by S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh. http://papers.nips.cc/paper%5C_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html.

F. Wotawa. 2020. "On the Use of Answer Set Programming for Model-Based Diagnosis." In: *IEA/AIE 2020, Kitakyushu, Japan, September 22-25, 2020, Proceedings* (LNCS). Ed. by H. Fujita, P. Fournier-Viger, M. Ali, and J. Sasaki. Vol. 12144. Springer, 518–529. doi:10.1007/978-3-030-55789-8_45.

H. Xia, Y. Li, C. T. Leong, W. Wang, and W. Li. 2025. "TokenSkip: Controllable Chain-of-Thought Compression in LLMs." *CoRR*, abs/2502.12067. arXiv: 2502.12067. doi:10.48550/ARXIV.2502.12067.

X. Yang, B. Chen, and Y. Tam. 2024. "Arithmetic Reasoning with LLM: Prolog Generation & Permutation." In: *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Short Papers, NAACL 2024, Mexico City, Mexico, June 16-21, 2024*. Ed. by K. Duh, H. Gómez-Adorno, and S. Bethard. Association for Computational Linguistics, 699–710. doi:10.18653/V1/2024.NAACL-SHORT.61.

Y. Zeng, A. Rajasekharan, P. Padalkar, K. Basu, J. Arias, and G. Gupta. 2024. "Automated Interactive Domain-Specific Conversational Agents that Understand Human Dialogs." In: *PADL 2024, London, UK, January 15-16, 2024, Proceedings* (LNCS). Ed. by M. Gebser and I. Sergey. Vol. 14512. Springer, 204–222. doi:10.1007/978-3-031-52038-9_13.

H. Zhang, J. Xu, and J. Wang. 2019. "Pretraining-based natural language generation for text summarization." *arXiv preprint arXiv:1902.09243*.

W. Zhang, X. Li, Y. Deng, L. Bing, and W. Lam. 2023. "A Survey on Aspect-Based Sentiment Analysis: Tasks, Methods, and Challenges." *IEEE Trans. Knowl. Data Eng.*, 35, 11, 11019–11038. doi:10.1109/TKDE.2022.3230975.

A LLMASP: Domain-wise Analysis and Behavior File Ablation Study

In this appendix, we report the ablation study that we performed to select the best performing behavior file configuration for LLMASP. The structure of behavior files, along with their features, is described in section 4.3. Here, we investigated the different configurations resulting by the concatenation of their features. We also considered sets of application files whose context provide the following information (as given in the ASP Competitions websites):

(A1) format of atoms (i.e., a description of how the predicates to be extracted have to be encoded);
(A2) description of the problem and format of atoms.

Results are shown in Table 7 and commented below. Note that for this ablation study, we utilized LLaMA 3.1 70B instead of LLaMA 3.3 70B to ensure a completely uniform comparison with the 8B model size.

In line with the previous experiments, Table 7 shows that LLMASP consistently improves over the vanilla LLaMA baselines across both model sizes. Under A1, the 8B model improves from the best vanilla setting (Descr.+Format, $F1 = 0.737$) to $F1 = 0.823$ with EIN, while the 70B model rises from $F1 = 0.904$ to $F1 = 0.968$ with EIN, indicating that the benefit of predicate-focused prompting extends to both capacity regimes. Under A2, the same trend persists and culminates in the overall best configuration, namely EIN on the 70B model, which attains the highest aggregate performance ($P = 0.974$, $R = \mathbf{0.988}$, $F1 = \mathbf{0.981}$). Notably, EIN remains competitive (and often best) across A1/A2 and model scales. But, while we acknowledge a small improvement in using A2 instead of A1 (possibly thanks to some contextualized examples), we also notice that F is not necessary and may even confuse the 8b model; N helps the LLMs to handle missing answers, while R does not; I is also beneficial, likely because it simplifies the attention mechanism of the LLMs; E tends to improve the F1-score. The problem-wise results across the different configurations settings are reported in Figure 5. For reference, Figure 6 reports the per-domain performance of the baselines, including also the performance of ChatGPT 4o–mini, which are similar to those obtained for 70b: 12.3% when no information is provided; 36.5% when the domain is described; 82.9% when the format is detailed; 87.6% when both the domain and the format are given; 76.0% when the ASP program is included.

Overall, these results indicate that the EIN behavior file yields the most effective and stable performance within LLMASP, and for this reason we adopted EIN as the default configuration throughout the paper.

B Knowledge Base Integration for Implicit Knowledge Generation in LLMASP

In this appendix, we extend our primary evaluation by analyzing the impact of incorporating domain knowledge bases into LLMASP. Specifically, we examine the effect of restricting the LLM to extracting only explicit facts from

Table 7. Aggregate results in terms of precision, recall, and F1-scode for the vanilla LLaMA, and the two strategies employed by LLMASP: (A1) format of atoms, and (A2) description of the problem and format of atoms. The best results are highlighted in **blue** for 8b and in **bold** for 70b.

	LLaMA (no LLMASP)					
	llama3.1-8b			llama3.1-70b		
Info	*P*	*R*	*F1*	*P*	*R*	*F1*
Format	0.731	0.700	0.694	0.911	0.826	0.841
Descr.+Format	0.756	0.779	0.737	0.934	0.891	0.904
Descr.	0.258	0.244	0.237	0.278	0.286	0.273
None	0.231	0.196	0.201	0.130	0.134	0.128
	LLMASP A1					
	llama3.1-8b			llama3.1-70b		
Conf.	*P*	*R*	*F1*	*P*	*R*	*F1*
F	0.791	0.824	0.787	0.946	0.944	0.940
EF	0.809	0.854	0.811	0.960	0.959	0.954
FN	0.765	0.831	0.772	0.966	0.949	0.952
EFN	0.807	0.866	0.816	0.969	0.951	0.953
I	0.785	0.884	0.801	0.958	0.973	0.960
EI	0.799	0.889	0.813	0.964	0.970	0.964
IN	0.781	0.874	0.794	0.973	0.962	0.963
EIN	0.810	0.892	0.823	0.973	0.968	0.968
INR	0.796	0.857	0.796	0.970	0.952	0.953
EINR	0.813	0.876	0.819	0.974	0.954	0.955
	LLMASP A2					
	llama3.1-8b			llama3.1-70b		
Conf.	*P*	*R*	*F1*	*P*	*R*	*F1*
F	0.786	0.826	0.789	0.954	0.976	0.962
EF	0.806	0.853	0.804	0.966	0.984	0.975
FN	0.777	0.829	0.784	0.967	0.978	0.971
EFN	0.810	0.853	0.807	0.972	0.988	0.980
I	0.786	0.892	0.809	0.961	0.986	0.971
EI	0.812	0.896	0.823	0.968	0.986	0.977
IN	0.794	0.894	0.811	0.968	0.988	0.976
EIN	0.818	**0.904**	**0.828**	0.974	**0.988**	**0.981**
INR	0.820	0.879	0.823	0.973	0.987	0.977
EINR	**0.828**	0.881	0.821	**0.976**	0.972	0.967

the input, while delegating the generation of deterministically derivable output to explicit ASP rules. To maintain a focused evaluation, this analysis is restricted exclusively to the four domains where target representations can be partially derived via ASP and for which domain knowledge was explicitly encoded in the application files. Specifically, we only consider the domains *Labyrinth* (**L**), *No Mystery* (**NM**), *Ricochet Robots* (**RR**), and *Sokoban* (**S**), whose instances allow portions of the target representation to be derived by ASP once a subset of facts has been extracted. These problems, can also be hard for LLMs. For instance, the domain *Labyrinth* contains problems which describe a field as a set of coordinates (x, y). Since the field is basically a grid $N \times N$, instead of asking an LLM to directly extract the coordinates from the textual description, we can use an ASP knowledge base (*KB*

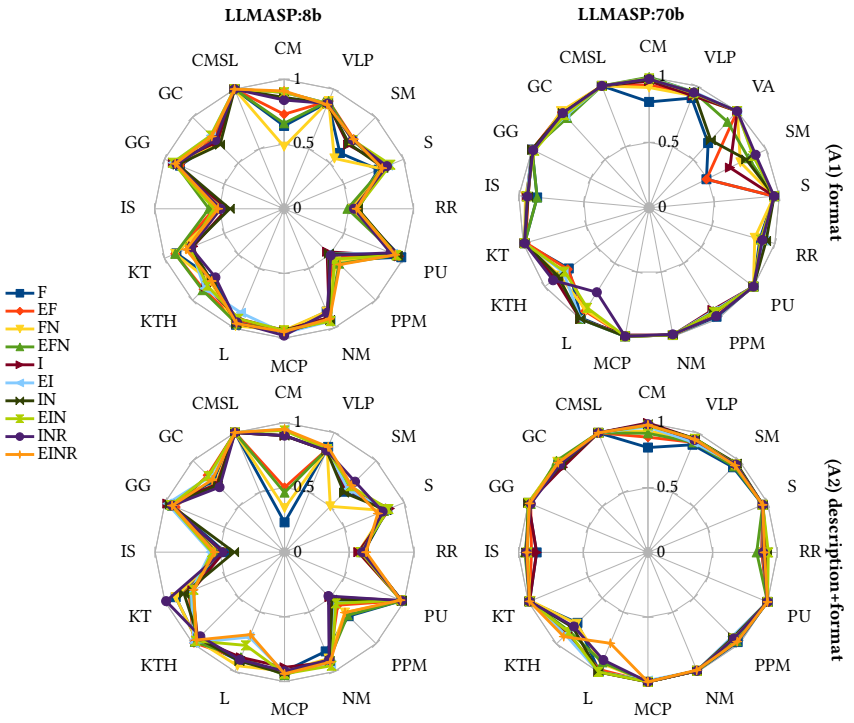

Fig. 5. F1-score obtained by generating facts with LLMASP using LLaMA 3.1 models (8b and 70b), and application files providing the format of atoms and possibly the description of the domain.

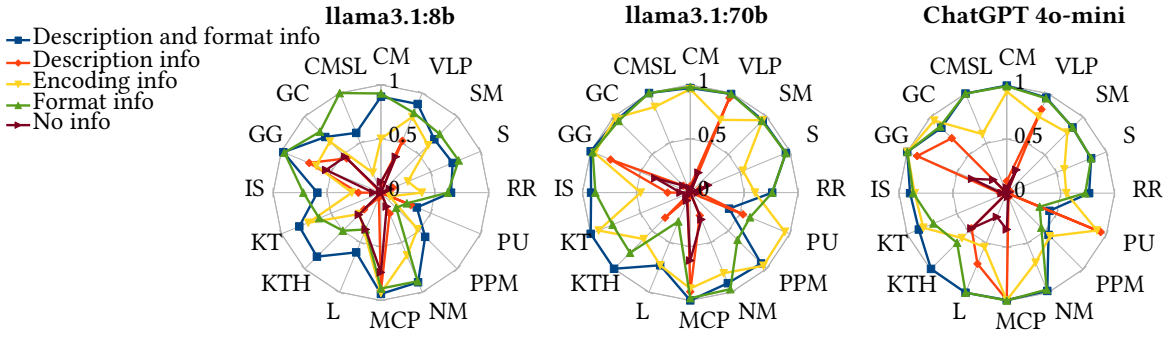

Fig. 6. F1-score obtained by LLaMA 3.1 models (8b and 70b) and ChatGPT (4o–mini) single-call baselines, under different prompting strategies.

hereafter) replacing the predicate `field/2` with `size/1` (asking the size of the grid instead of listing its fields) and the following ASP rule:

```
field(X,Y) :- size(N), X = 1..N, Y = 1..N.
```

Similarly, in *No Mystery* and Sokoban we replace `step/1` with `steps/1` and the ASP rule:

```
step(I) :- steps(N), I = 1..N.
```

Finally, in *Ricochet Robots* the extraction of `dim/1` can be replaced with

```
dim(I-1) :- dim(I), I > 1.
```

while predicate `barrier/3` can be replaced with `barrier/5` (asking to extract pairs of cells separated by barriers and the orientation of such barriers) together with the following ASP rules (to obtain the representation used in ASP Competitions):

```
barrier(X,Y,A,B,D) :- barrier(A,B,X,Y,D).
barrier(X,Y,D) :- barrier(X,Y,X-1,Y,D), D == west.
barrier(X,Y,D) :- barrier(X,Y,X+1,Y,D), D == east.
barrier(X,Y,D) :- barrier(X,Y,X,Y-1,D), D == north.
barrier(X,Y,D) :- barrier(X,Y,X,Y+1,D), D == south.
```

In the program above, the definition of barrier/3 is of particular interest. In fact, we observed that often the tested LLMs have difficulties to associate a barrier between $(x-1, y)$ and (x, y) toward direction west with location $(x-1, y)$. We therefore expect better results by first extracting a more syntactic fact such as `barrier(x-1,y,x,y,west)` and then employing ASP to generate the correct fact `barrier(x-1,y,west)`.

Domain-wise results are shown in Table 8, under the two prompt strategies (A1 vs. A2). and two model sizes, restricted to the four selected domains. We observe a sensible improvement for RR, especially for the 8b model.

Table 8. F1-score difference due to the use of ASP in the generation of facts. Positive offsets are improvements (values in percentage).

	Domain/ Conf.	(A1) format				(A2) description+format			
		L	NM	RR	S	L	NM	RR	S
llama3.1-8b	**F**	97+2	86	54+37	79−1	91+7	83+5	61+32	86−2
	EF	94+6	91+1	52+43	83+9	89+10	91+2	58+38	86+5
	FN	98+1	86	53+38	81−2	95+3	89+1	57+36	84−1
	EFN	96+3	91	49+46	83+7	88+11	90+2	59+38	83+9
	I	92+7	91−2	56+36	86	88+11	92+2	57+37	88
	EI	87+13	94	55+41	87+3	71+29	94	60+38	85+5
	IN	96+3	93−2	57+36	83	91+8	91+4	61+33	85+2
	EIN	92+8	94	56+39	89−1	78+22	95	59+39	87−2
	INR	95+4	87+3	54+40	86+3	90+9	90	60+35	82+3
	EINR	96+4	92+1	56+40	82+7	69+31	92	63+33	78+12
llama3.1-70b	**F**	99+1	99	88+10	96	97+3	98+1	92+7	96
	EF	98+2	99	88+12	96	100	99	93+6	96
	FN	93+7	99	89+8	96	97+3	98+1	91+7	96
	EFN	92+8	99	84+16	96	100	99	93+6	96
	I	100	99−1	91+8	96	100	98	93+5	96
	EI	100	99	91+9	96	100	99	91+8	96
	IN	100	99	92+7	96	100	99	94+3	96
	EIN	100	99	93+6	96	100	99	92+7	96
	INR	90+10	99	89+10	96	97+3	99	94+5	96
	EINR	76+24	98.5+1	90+9	96	77+23	99	94+5	96

Table 9. Aggregate results for LLMASP computed on the four domains (*Labyrinth*, *No Mystery*, *Ricochet Robots*, *Sokoban*). We compare the baseline (LLMASP A2) against *KB*–augmented variants that use ASP to derive implicit consequences (A1 +*KB* and A2 +*KB*), for `llama3.1-8b` and `llama3.1-70b` and multiple configurations. The best results are highlighted in **blue** for 8B and in **bold** for 70B.

	LLMASP A2			LLMASP A1 +*KB*						LLMASP A2 +*KB*					
	llama3.1-70b			llama3.1-8b			llama3.1-70b			llama3.1-8b			llama3.1-70b		
Conf.	*P*	*R*	*F1*	*P*	*R*	*F1*	*P*	*R*	*F1*	*P*	*R*	*F1*	*P*	*R*	*F1*
F	0.960	0.962	0.958	0.917	0.865	0.885	0.975	0.992	0.982	0.932	0.885	0.907	0.978	0.992	0.985
EF	0.960	0.982	0.970	**0.963**	0.928	0.948	0.978	**0.998**	**0.988**	0.960	0.943	0.948	0.978	0.995	0.985
FN	0.960	0.960	0.955	0.910	0.873	0.888	0.975	0.988	0.980	0.927	0.895	0.910	0.978	0.988	0.982
EFN	0.960	0.982	0.970	0.955	0.925	0.937	**0.980**	**0.998**	**0.988**	0.960	0.940	0.950	0.978	0.992	0.985
I	0.960	0.980	0.968	0.915	0.927	0.915	0.973	**0.998**	0.982	0.938	0.948	0.938	0.970	0.992	0.980
EI	0.958	0.975	0.965	0.953	0.955	0.950	**0.980**	**0.998**	**0.988**	0.947	**0.970**	**0.955**	0.978	0.995	0.985
IN	0.965	0.982	0.972	0.917	0.930	0.915	0.975	0.995	0.985	0.935	0.943	0.938	0.970	0.992	0.980
EIN	0.960	0.978	0.968	0.948	0.945	0.942	0.975	0.995	0.985	0.940	0.965	0.945	0.975	0.995	0.985
INR	0.965	0.975	0.965	0.947	0.917	0.930	0.978	0.992	0.985	0.935	0.925	0.922	0.978	0.992	0.985
EINR	0.965	0.920	0.915	0.953	0.940	0.945	0.978	0.990	0.985	0.953	0.945	0.945	0.978	0.995	0.985

With a few exceptions, the use of ASP is beneficial for the generation process. Aggregate results are reported in Table 9, which isolates the contribution of injecting *explicit domain knowledge* in the form of ASP rules and delegating the derivation of *implicit knowledge* to symbolic reasoning. Across all configurations, the inclusion of a knowledge base leads to consistent improvements in overall extraction performance. These gains are largely driven by increased recall, while precision remains essentially unchanged. This indicates that ASP-based inference primarily recovers facts that are logically entailed by the extracted information but are not explicitly mentioned in the input text, rather than introducing spurious or hallucinated facts.

For LLaMA 70B, the non-KB A2 baseline achieves $F1 \in [0.915, 0.972]$ (best at IN). Augmenting the pipeline with the KB pushes performance close to saturation under both prompt strategies: A1+KB reaches $F1 \in [0.980, 0.988]$ and A2+KB reaches $F1 \in [0.980, 0.985]$. The strongest result is obtained by A1+KB in configurations such as EFN and EI, reaching ($P = 0.980, R = 0.998, F1 = 0.988$). Notably, relative to the baseline, while both precision and recall increase, the recall reaches values close to one in almost all 70B configurations (often $R \geq 0.995$), reinforcing the interpretation that the KB predominantly addresses omission errors by completing derivable structure.

For LLaMA 8B, KB augmentation also yields substantial gains, reaching a best $F1 = 0.955$ under both A1+KB and A2+KB (configuration EI). Moreover, for the 8B model, A2+KB is systematically competitive with (and in several configurations slightly above) A1+KB, suggesting that the additional domain description in A2 remains valuable when the underlying LLM has limited capacity. In this regime, the A2 prompt appears to stabilize extraction of the "seed" facts required for KB closure, thereby enabling ASP to reconstruct the remaining consequences more reliably.

A further practical benefit is increased robustness with respect to configuration choice. With KB support, the spread of F1 values across configurations is markedly narrower than in the non-KB A2 baseline, consistent with the view that symbolic completion mitigates variability arising from incomplete extraction and yields more stable aggregate behavior across domains and prompting variants.

Figure 7 summarizes the corresponding quality–cost trade-offs when combining KB support with different serialization grammars. As in the non-KB analysis, we report, for each configuration, the F1-score, perfect extraction rate, energy consumption, and completion-token usage.

Injecting a domain KB and delegating fact completion to ASP reshapes this trade-off in two ways. First, when missing facts are ASP-derivable, quality improves by reducing omission errors without requiring the LLM to

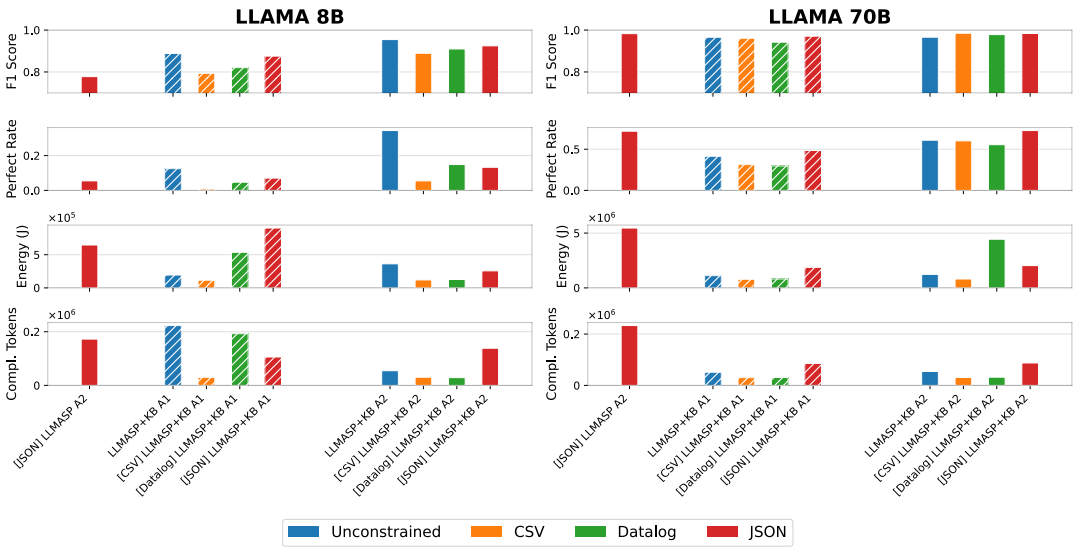

Fig. 7. F1-score, Perfect Rate, Energy and completion token usage measured on LLM, LLMASP, using LLaMA 3 8B (left) and 70B (right).

explicitly enumerate all consequences. Second, operational cost can decrease because the model is tasked with producing only a compact set of explicit seed facts, while the ASP component deterministically reconstructs the implied closure.

For the smaller model, KB support induces a substantial upward shift in extraction quality. The strongest improvement is obtained by the unconstrained A2+KB configuration: F1 increases from 0.778 (baseline and larger model) to 0.955, while the perfect extraction rate rises from 0.055 to 0.344.

Table 10 clarifies the mechanism behind this gain. A2+KB improves both precision and recall (P: from 0.871 to 0.949, R: from 0.740 to 0.963), and the improvements are consistent across predicate arities, including higher-arity facts ($F1_{\text{ar3}}$: from 0.747 to 0.902). This pattern matches the intended role of ASP-based completion: once the LLM extracts a compact set of parameters (e.g., grid size, step bounds, or structural constraints), ASP reconstructs the expanded relational structure deterministically, with limited semantic ambiguity.

To complement the aggregate results, Table 11 reports the total number of TP, FP, FN, and duplicates (D) over the four domains where KB-guided ASP completion is applicable. As in Section 5.4, we interpret FP as *spurious* extracted facts (i.e., unsupported by the gold annotations) and D as *redundant* repetitions, while FN captures omissions. This view directly characterizes how KB support reshapes the mass of extraction errors beyond what is visible from F1 alone.

Across both model sizes, a first salient effect of KB support is the near-complete elimination of redundancy: all KB configurations yield $\mathrm{D}_F = 0$ on these domains. This indicates that, once the pipeline is organized around extracting a compact seed set and delegating closure to ASP, the resulting fact bases become structurally more regular and free of repeated outputs: an important practical property when the downstream solver cost scales with the size of the produced instance.

For LLaMA 8B, KB support fundamentally changes the error profile from omission-dominated to near-complete coverage. The non-KB reference point ([JSON] LLMASP^{A2}) still misses a large portion of facts (FN= 4048) despite a relatively small FP cost (FP= 488). In contrast, the unconstrained A2+KB configuration attains substantially

Table 10. Qualitative assessment of the influence of employing constraint generation and ASP support across four domains (*Labyrinth*, *No Mystery*, *Ricochet Robots*, *Sokoban*). **F1-ar1**: F1-score (macro) for unary predicates. **F1-ar2**: F1-score for binary predicates. **F1-ar3**: F1-score (macro) for ternary predicates. **Perfect**: Percentage of perfectly extracted problem instances. The best results are highlighted in **blue** for 8B and in **bold** for 70B.

LLM	Configuration	P	R	F1-macro	F1-micro	F1-ar1	F1-ar2	F1-ar3	Perfect
llama8b	(Baseline) [JSON] LLMASP ^{A2}	0.871	0.740	0.778	0.785	0.823	0.866	0.747	0.055
	LLMASP+KB ^{A1}	0.899	0.894	0.888	0.935	0.949	0.950	0.750	0.125
	[CSV] LLMASP+KB ^{A1}	0.877	0.740	0.793	0.920	0.800	0.892	0.674	0.008
	[Datalog] LLMASP+KB ^{A1}	0.909	0.769	0.821	0.894	0.918	0.851	0.691	0.047
	[JSON] LLMASP+KB ^{A1}	0.932	0.838	0.876	0.939	0.959	0.879	0.745	0.070
	LLMASP+KB ^{A2}	**0.949**	**0.963**	**0.955**	**0.979**	**0.978**	**0.980**	**0.902**	**0.344**
	[CSV] LLMASP+KB ^{A2}	0.926	0.862	0.889	0.947	0.932	0.967	0.759	0.055
	[Datalog] LLMASP+KB ^{A2}	0.944	0.884	0.910	0.960	**0.978**	0.947	0.780	0.148
	[JSON] LLMASP+KB ^{A2}	0.934	0.921	0.925	0.963	0.967	0.945	0.838	0.133
llama70b	(Baseline) [JSON] LLMASP ^{A2}	**0.981**	0.991	0.983	0.975	**0.987**	0.989	**0.998**	0.719
	LLMASP+KB ^{A1}	0.968	0.968	0.966	0.985	0.978	0.950	0.993	0.414
	[CSV] LLMASP+KB ^{A1}	0.972	0.954	0.961	0.986	0.979	0.960	0.971	0.312
	[Datalog] LLMASP+KB ^{A1}	0.967	0.925	0.942	0.983	0.986	0.954	0.941	0.297
	[JSON] LLMASP+KB ^{A1}	0.978	0.966	0.971	0.990	**0.987**	0.956	0.987	0.484
	LLMASP+KB ^{A2}	0.947	0.990	0.966	0.985	0.971	0.999	0.978	0.609
	[CSV] LLMASP+KB ^{A2}	0.976	**0.996**	**0.985**	0.993	0.986	0.998	0.989	0.602
	[Datalog] LLMASP+KB ^{A2}	0.961	0.920	0.937	0.982	0.986	0.951	0.932	0.273
	[JSON] LLMASP+KB ^{A2}	0.976	0.993	0.984	**0.994**	0.986	**1.000**	0.989	**0.727**

higher coverage (TP increases from 8300 to 12160) while collapsing omissions to FN=188, and it does so without inflating spurious facts (FP=337). Within A1+KB, grammatical constraints primarily act as a precision mechanism: FP drops monotonically from 980 (unconstrained) to 366 (JSON), but this comes with a recall penalty (FN rises from 637 to 1086), consistent with the interpretation that stricter serialization can make seed extraction more brittle for a low-capacity model. Under A2+KB, this sensitivity is even more apparent: adding CSV/Datalog/JSON increases FN relative to the unconstrained KB setting, with no compensating reduction in FP for 8B. Hence, on

Table 11. Comparison of the total number of true positive (TP), false positive (FP), false negative (FN), and duplicated (D) facts extracted by each configuration, across the four problem domains. The best results are highlighted in **blue** for 8B and in **bold** for 70B.

LLM	LLaMA 8B				LLaMA 70B			
Configuration	TP	FP	FN	D	TP	FP	FN	D
[JSON] LLMASP ^{A2}	8300	488	4048	74	11858	114	490	**0**
LLMASP+KB ^{A1}	11711	980	637	**0**	12214	233	134	**0**
[CSV] LLMASP+KB ^{A1}	11003	569	1345	**0**	12177	165	171	**0**
[Datalog] LLMASP+KB ^{A1}	10350	446	1998	**0**	12092	165	256	**0**
[JSON] LLMASP+KB ^{A1}	11262	366	1086	**0**	12218	**129**	130	**0**
LLMASP+KB ^{A2}	**12160**	337	**188**	**0**	12313	338	35	**0**
[CSV] LLMASP+KB ^{A2}	11535	481	813	**0**	**12324**	145	**24**	**0**
[Datalog] LLMASP+KB ^{A2}	11717	**336**	631	**0**	12293	153	55	**0**
[JSON] LLMASP+KB ^{A2}	11845	409	503	**0**	**12324**	133	**24**	**0**

Table 12. # t_{in}: Number of tokens in input ($\times 10^6$). # t_{out}: Number of token in output ($\times 10^6$). *Energy*: Energy consumption in Joules ($\times 10^5$). *Time*: Computation time in seconds ($\times 10^3$). The best results are highlighted in **blue** for 8B and in **bold** for 70B.

LLM	Configuration	$\#t_{in}$	$\#t_{out}$	Energy ($J \cdot 10^5$)	Time ($s \cdot 10^3$)
llama8b	[JSON] LLMASP ^{A2}	1.133	0.172	6.462	3.433
	LLMASP+KB ^{A1}	**0.504**	0.223	1.937	1.213
	[CSV] LLMASP+KB ^{A1}	0.585	0.030	**1.153**	**0.796**
	[Datalog] LLMASP+KB ^{A1}	0.586	0.193	5.355	3.186
	[JSON] LLMASP+KB ^{A1}	0.583	0.106	9.017	4.138
	LLMASP+KB ^{A2}	0.992	0.055	3.625	2.041
	[CSV] LLMASP+KB ^{A2}	1.073	0.030	1.202	0.836
	[Datalog] LLMASP+KB ^{A2}	1.072	**0.029**	1.268	0.853
	[JSON] LLMASP+KB ^{A2}	1.068	0.138	2.567	1.528
llama70b	[JSON] LLMASP ^{A2}	1.137	0.233	54.606	20.579
	LLMASP+KB ^{A1}	**0.504**	0.051	11.190	4.089
	[CSV] LLMASP+KB ^{A1}	0.585	**0.030**	**7.840**	**3.056**
	[Datalog] LLMASP+KB ^{A1}	0.586	0.031	8.140	3.123
	[JSON] LLMASP+KB ^{A1}	0.583	0.085	18.727	6.858
	LLMASP+KB ^{A2}	0.992	0.054	12.348	4.620
	[CSV] LLMASP+KB ^{A2}	1.073	0.031	8.121	3.238
	[Datalog] LLMASP+KB ^{A2}	1.033	0.032	44.392	16.597
	[JSON] LLMASP+KB ^{A2}	1.070	0.087	20.341	7.515

these domains, the dominant benefit for the smaller model is the KB/ASP completion itself; additional output grammars can be counterproductive once the KB already provides a strong structural regularizer.

For LLaMA 70B, the qualitative picture differs. Here, KB support primarily strengthens coverage while preserving low hallucination rates, and grammars can further tighten precision without degrading recall. Compared to [JSON] LLMASP^{A2} (FP=114, FN=490), A1+KB reduces omissions sharply (FN=134) at the cost of higher FP (233); imposing a grammar, especially JSON, largely removes this cost (FP drops to 129 while keeping FN near the A1+KB level). Under A2+KB, grammars are particularly beneficial: CSV and JSON reduce FP from 338 to 145/133 and simultaneously reduce FN from 35 to 24, with a slight increase in TP (to 12324). Thus, for a high-capacity model, structured constraints appear to suppress residual spurious seed facts while maintaining (and marginally improving) completeness.

Overall, Table 11 supports the main interpretation of the KB results: ASP-based completion primarily reduces omission errors by deterministically reconstructing derivable structure from a compact seed set, and it yields highly non-redundant outputs. However, the interaction with output grammars is model-dependent: for 8B, strict serialization can increase omissions in the KB regime, whereas for 70B it provides an additional lever for reducing hallucinations (FP) while preserving near-saturated recall.

KB support also yields efficiency gains. Relative to the baseline, A2+KB reduces completion length from 172470 to 54652 tokens and decreases energy consumption from 6.46×10^5 J to 3.63×10^5 J, while also lowering runtime (Table 12). A1+KB attains even lower energy (approximately 1.94×10^5 J), largely due to shorter prompts, but at a clear cost in accuracy and executability (Table 10). This highlights a key constraint for low-capacity models: the A2 description remains important to ensure accurate extraction of the seed facts on which KB closure critically depends.

When grammar constraints are introduced on top of KB support, the efficiency frontier shifts further. Under A2+KB, CSV and Datalog are particularly economical (completion tokens ≈ 30k and energy ≈ $1.2–1.3 \times 10^5$ J), but they underperform the unconstrained KB setting in executability (Perfect Rate is 0.055 for CSV; 0.148 for Datalog; versus 0.344 without grammar). JSON under A2+KB yields a stronger quality point ($F1_{\text{macro}} = 0.925$, Perfect Rate = 0.133) but is less efficient than CSV/Datalog due to longer completions. Overall, for LLaMA 8B, the KB already acts as a strong structural regularizer; additional grammatical constraints can reduce redundancy and enforce serialization discipline, but may simultaneously increase omission or formatting fragility because the LLM must output a minimal, correctness-critical set of seed facts.

For the larger model, baseline quality is already near saturation on the four selected domains (LLMASP^{A2} with JSON: $F1_{\text{macro}} = 0.983$, Perfect Rate= 0.719; Table 10). Accordingly, KB support does not systematically increase F1; instead, its principal benefit is efficiency.

Figure 7 and Table 12 show that KB-supported configurations can reduce completion tokens dramatically (e.g., from 233164 to 30725 under A2+KB with CSV) and, correspondingly, reduce energy from 5.46×10^6 J to 8.12×10^5 J (an 85% decrease), while cutting runtime from 2.06×10^4 s to 3.24×10^3 s. Thus, even when F1 cannot improve materially because performance is already close to the ceiling, ASP-based completion remains a strong computational lever by avoiding generative enumeration of implicit consequences.

As in the non-KB setting, grammars induce differentiated operating points. Under A2+KB, CSV attains the highest macro F1 (0.985) with the lowest energy (8.12×10^5 J), but reduces the perfect rate to 0.602 (Table 10). JSON under A2+KB yields the best perfect extraction rate (0.727), slightly surpassing the baseline (0.719), while maintaining essentially the same F1 (0.984); however, this comes at higher energy than CSV due to longer completions (2.03×10^6 J). Finally, Datalog is the least attractive option in this regime: under A2+KB it combines a substantial quality drop ($F1_{\text{macro}} = 0.937$, Perfect Rate = 0.273) with a pronounced energy/time increase relative to the other KB grammars, suggesting that more complex constraint regimes may impose overheads beyond raw token counts and may be unstable when the target representation is poorly aligned with the model's preferred serialization patterns.

Taken together, these results support two conclusions. First, KB-supported ASP completion is a high-impact mechanism for smaller models: it substantially improves recall and executability by recovering logically entailed facts that would otherwise be omitted, directly addressing a dominant error mode in low-capacity settings. Second, for larger models, the KB functions primarily as an efficiency instrument: quality is already high, but ASP-based completion enables large reductions in completion length, energy, and runtime without materially degrading F1, provided that the grammar/prompt combination does not hinder extraction of the seed facts required by the KB. Within this design space, CSV and JSON emerge as complementary operating points: CSV is typically the most energy-efficient, whereas JSON more consistently maximizes perfect extraction, particularly on LLaMA 70B.

C Example of an Application File for the Permutation Pattern Matching

In the following we report the A2 application file for the extraction of the predicates of the instances in the Permutation Pattern Matching domain. For the A2 application file (i.e., Format+Description), the context (i.e., the first item marked with the underscore '_') contains both a *description* of the Permutation Pattern Matching problem, and a description of the *format* of the predicates to be extracted from an input string. For the A1 application file (Format), the context contains only the format of the predicates to be extracted.

```
preprocessing:
- _: |
Problem description:
Permutation Pattern Matching (PPM) is an NP-complete pattern matching problem.
Given a permutation T (the text) and a permutation P (the pattern)  the question is
whether there exists a matching of P into T. A matching is a subsequence of T that
```

```
has the same relative order as P. For example the permutation T = 53142
(written in one-line representation) contains the pattern 231, since the subsequence
342 of T is order-isomorphic to 231 (i.e. the smallest element is in the third position,
the second smallest in the first position and the largest in the second position).
Note that: 1. Both text and pattern consist only of numbers.
2. A text of length m is a permutation on {1,...,m} , i.e., every element appears exactly once.
3. A pattern of length n is a permutation on {1,...,n} . Hence 241 is not a pattern.
But note that 241 might be matching!
(For example: 241 is a matching from the pattern 231 into the text 32451.)

Format description of the predicates to be extracted:
The permutations T and P are encoded as binary predicates, e.g. T=132 is encoded as
t(1,1). t(2,3). t(3,2). Additionally, patternlength/1 describes the length of the pattern P.
- t(index, number): |
Represent the permutation text T by a sequence of facts, each fact associating an index
(an integer starting from 1) to the number (an integer) at that index.
- p(p,n): |
Represent the permutation pattern P by a sequence of facts, each fact associating an index
(an integer starting from 1) to the number (an integer) at that index.
- patternlength(value): |
value is the length of the pattern (an integer).
```

The remainder of the application file contains a list of atom–instruction pairs that are used by LLMASP to instruct predicate-specific LLM calls.

D Qualitative Analysis of LLMASP Failure Modes

While aggregated quantitative metrics such as the F1-score provide a high-level overview of extraction performance, they obscure the underlying mechanics of model failure. To effectively guide future research and practical applications, it is crucial to understand not just how often these models fail, but *why*. In this section, we conduct a qualitative analysis to identify systematic error patterns, pinpoint the most challenging problem domains, and categorize the specific types of hallucinations generated by different configurations.

We begin our analysis by looking for systematic error patterns related to the arity of the predicates to be extracted. In particular, we break down the F1-score by predicate arity, model size, application file, and grammar constraints (Figure 8). The first notable trend is that single-call baselines (LLM, shown in green in Figure 8) exhibit a lower F1-score for unary predicates (arity 1), independent of model size and application files. The introduction of the multi-call LLMASP strategy (blue for LLMASP configurations and red for LLMASP+KB) leads to a decisive increase in the F1-score regardless of arity, model size, application file, and grammar.

Additionally, LLMASP reverses the pattern observed in single-call baselines: in most configurations, the highest F1-score is obtained for unary predicates, followed by binary and ternary predicates. Configurations paired with A2 application files exhibit, *ceteris paribus*, higher performance than their A1 counterparts. Finally, A2 configurations not only reach higher absolute F1-scores, but they also reduce the variance in performance across different arities, effectively stabilizing extraction accuracy for the 70B model. Across grammars, CSV and JSON exhibit the lowest performance variability across the LLMASP and LLMASP+KB configurations.

Having established these global patterns, we now focus our error analysis on the LLaMA 70B-based configurations, as they are the best-performing models. Our goal is to uncover the most common failure modes, including the specific nature of false positives (i.e., hallucinations not present in the ground truth), and to identify the most challenging problem domains. We begin with a highly restrictive performance metric: the number of perfectly extracted instances per problem domain (Table 13).

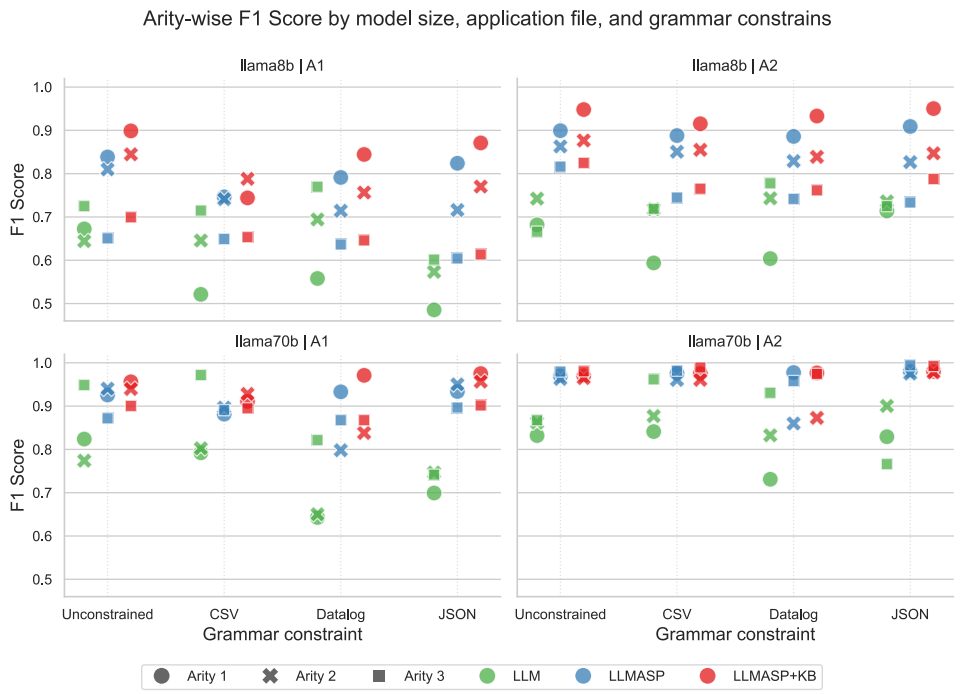

Fig. 8. Breakdown of the F1-score by predicate arity, model size, application file, and grammar constraints.

The data indicates that Connected Maximum-density Still Life (CMSL) is a relatively simple domain, where the LLM baseline already achieves perfect extraction. Crossing Minimization (CM) and Partner Units (PU) are nearly completely resolved by LLMASP, succeeding where the baseline LLM fails. A notable exception is the baseline LLM with JSON and the A2 application file, which correctly extracts 27 out of 32 instances for PU. Conversely, Sokoban (S) and Incremental Scheduling (IS) are the toughest domains for perfect extraction, despite our approaches obtaining remarkably high overall F1-scores on them. Interestingly, the LLM A2 baseline manages to perfectly extract up to 20 out of 32 instances for Sokoban. Incremental Scheduling remains the most resistant to perfect extraction.

Graph Colouring (GC) exhibits an unusual pattern: LLMASP with A1 can fully extract up to 27/32 instances, while LLMASP with A2 fails entirely on this strict metric. Upon contrasting the extracted predicates with the ground truth, we found this failure is primarily driven by symmetry resolution errors: extracting `link(82, 10)` instead of `link(10, 82)`. While logically valid within the context of an undirected graph, strict information extraction requires absolute adherence to the positional constants of the canonical ground truth.

For Sokoban, the primary source of error is the hallucination of the `stone/1` predicate. While not explicitly mentioned in the input text, its existence is logically implied by other contextual clues. Here, the multi-call strategy of LLMASP inadvertently forces the model to over-analyze, deriving implicit entailments rather than sticking to strict textual extraction. It is revealing that the single-pass LLM A2 outperforms LLMASP on extraction completeness in this domain. A similar pattern occurs in the Valves Location Problem (VLP), where the best-performing configuration is LLM A2 (CSV). This suggests that while complex domains benefit from multi-call reasoning, highly literal domains may be more amenable to direct, single-pass extraction.

Table 13. Number of perfectly extracted instances, per each domain, across configurations powered by LLaMA 70B.

Configuration	L	NM	RR	S	CMSL	CM	GG	GC	IS	KT	KTH	MCP	PU	PPM	SM	VLP
LLM^{A1}	1	19	0	0	**32**	10	12	0	0	17	2	0	10	11	18	14
[CSV] LLM^{A1}	27	15	0	9	**32**	18	29	0	0	10	5	31	0	7	23	18
[Datalog] LLM^{A1}	27	15	0	7	15	1	27	0	0	**32**	6	30	0	5	18	0
[JSON] LLM^{A1}	9	26	0	5	0	25	25	1	0	**32**	6	**32**	7	6	22	0
LLMASP^{A1}	19	4	2	0	**32**	14	**32**	21	0	20	9	31	**32**	20	10	1
[CSV] LLMASP^{A1}	16	7	1	0	16	8	31	21	0	29	11	29	**32**	0	11	0
[Datalog] LLMASP^{A1}	15	6	1	0	**32**	0	**32**	12	0	26	11	30	**32**	0	10	1
[JSON] LLMASP^{A1}	26	9	3	0	**32**	30	**32**	**27**	0	28	13	31	**32**	24	10	0
LLMASP+KB^{A1}	22	2	31	0	**32**	14	**32**	21	0	20	9	31	**32**	20	10	1
[CSV] LLMASP+KB^{A1}	26	8	7	0	16	8	31	21	0	29	11	29	**32**	0	11	0
[Datalog] LLMASP+KB^{A1}	30	5	1	0	**32**	0	**32**	12	0	26	11	30	**32**	0	10	1
[JSON] LLMASP+KB^{A1}	31	7	26	0	**32**	30	**32**	**27**	0	28	13	31	**32**	24	10	0
LLM^{A2}	8	21	0	12	**32**	16	**32**	3	1	30	**32**	21	0	19	20	20
[CSV] LLM^{A2}	27	17	0	13	17	23	**32**	0	0	**32**	**32**	30	0	12	24	**25**
[Datalog] LLM^{A2}	28	14	0	17	11	11	**32**	1	0	**32**	**32**	30	0	17	22	6
[JSON] LLM^{A2}	9	25	0	**20**	**32**	30	**32**	4	**2**	**32**	**32**	31	27	28	**27**	9
LLMASP^{A2}	23	26	16	0	**32**	26	**32**	7	0	**32**	**32**	29	**32**	26	20	12
[CSV] LLMASP^{A2}	27	22	24	0	**32**	30	**32**	2	0	**32**	**32**	31	**32**	27	22	8
[Datalog] LLMASP^{A2}	30	24	10	0	**32**	10	31	4	0	31	**32**	31	**32**	0	23	13
[JSON] LLMASP^{A2}	31	30	**31**	0	**32**	**31**	31	9	0	**32**	**32**	31	**32**	**29**	18	7
LLMASP+KB^{A2}	24	28	27	0	**32**	26	**32**	7	0	**32**	**32**	29	**32**	26	20	12
[CSV] LLMASP+KB^{A2}	29	24	26	0	**32**	30	**32**	2	0	**32**	**32**	31	**32**	27	22	8
[Datalog] LLMASP+KB^{A2}	29	25	10	0	**32**	10	31	4	0	31	**32**	31	**32**	0	23	13
[JSON] LLMASP+KB^{A2}	**32**	**31**	**31**	0	**32**	**31**	31	9	0	**32**	**32**	31	**32**	**29**	18	7

In the Incremental Scheduling domain, we observe similar failure modes: the model hallucinates job/1 predicates when the text entails a job without explicitly declaring it, and invents ungrounded predicates like `deadline(job, NONE)`.

Fortunately, these failure modes are largely predictable and structurally sound. By implementing a post-processing stage involving simple pattern matching, most of these obvious hallucinations could be easily filtered out, thereby increasing precision and, in turn, F1-score. Indeed, this represents a promising avenue for future research. As shown in Table 4, LLMASP A2 configurations (70B) already successfully extract over 23,500 true positives out of the 23,760 total facts in our database. This suggests that recall has effectively plateaued; the remaining frontier for LLMASP is the strict reduction of hallucinations, a goal that our grammar-constrained generation has already significantly advanced.

To complete the picture, we report the performance of all LLaMA 70B configurations on the 5 most difficult problems in Table 14 (informally defined as domains where most LLMASP configurations fail to perfectly extract at least 75% of instances).

On Sokoban, LLMASP+KB^{A2} correctly extracts all predicates (TP=1400), achieving perfect recall. However, this is accompanied by a persistent rate of false positives—at least one per instance—which drops its perfect recovery rate to zero. In constrast, [JSON] LLM^{A2}, despite retrieving almost 100 fewer true positives, manages to perfectly recover 20/32 instances. Understanding why this precision degrades when moving from single-call to multi-call architectures remains a key question.

Table 14. Performance on difficult problems for LLaMA 70B powered configurations.

Problem Domain	S					GC					IS					SM					VLP				
Configuration	TP	FP	FN	D	D_{uniq}	TP	FP	FN	D	D_{uniq}	TP	FP	FN	D	D_{uniq}	TP	FP	FN	D	D_{uniq}	TP	FP	FN	D	D_{uniq}
LLM^{A1}	1211	444	189	50	41	3127	1439	104	478	442	2382	273	68	2	2	504	30	8	69	67	877	56	3	20	20
[CSV] LLM^{A1}	1383	76	17	10	10	2791	2092	440	295	274	2358	381	92	0	0	499	13	13	0	0	871	16	9	31	30
[Datalog] LLM^{A1}	1357	68	43	4	4	199	4055	3032	86	86	1560	1040	890	12	12	489	22	23	1	1	117	705	763	37	34
[JSON] LLM^{A1}	1178	277	222	1	1	3098	1318	133	94	71	2278	116	172	12	12	500	11	12	0	0	0	885	880	10	10
LLMASP^{A1}	1396	210	4	1	1	3223	58	8	0	0	2410	512	40	10	10	466	33	46	0	0	713	86	167	104	97
[CSV] LLMASP^{A1}	1397	130	3	1	1	3038	273	193	32	32	2262	1103	188	102	101	472	25	40	0	0	717	111	163	148	145
[Datalog] LLMASP^{A1}	1367	152	33	0	0	2710	561	521	18	18	2019	1138	431	25	20	470	21	42	0	0	717	75	163	105	105
[JSON] LLMASP^{A1}	1391	139	9	2	2	3224	0	7	4	4	2409	517	41	0	0	471	19	41	0	0	719	108	161	58	58
LLMASP+KB^{A1}	1391	187	9	0	0	3223	58	8	0	0	2410	512	40	10	10	466	33	46	0	0	713	86	167	104	97
[CSV] LLMASP+KB^{A1}	1396	130	4	0	0	3038	273	193	32	32	2262	1103	188	102	101	472	25	40	0	0	717	111	163	148	145
[Datalog] LLMASP+KB^{A1}	1381	113	19	0	0	2710	561	521	18	18	2019	1138	431	25	20	470	21	42	0	0	717	75	163	105	105
[JSON] LLMASP+KB^{A1}	1393	133	7	0	0	3224	0	7	4	4	2409	517	41	0	0	471	19	41	0	0	719	108	161	58	58
LLM^{A2}	1381	58	19	4	4	2802	2450	429	127	122	2174	340	276	1014	119	495	16	17	0	0	869	7	11	23	23
[CSV] LLM^{A2}	1380	54	20	2	2	2830	2061	401	250	250	2356	338	94	12	12	501	11	11	0	0	874	3	6	24	24
[Datalog] LLM^{A2}	1383	48	17	5	5	2163	2169	1068	125	125	1847	643	603	12	12	500	12	12	0	0	714	41	166	23	23
[JSON] LLM^{A2}	1307	99	93	2	2	3006	1400	225	126	122	2297	83	153	4	4	507	5	5	0	0	273	603	607	16	16
LLMASP^{A2}	1399	275	1	4	4	3036	1654	195	367	353	2414	631	36	18	13	499	12	13	0	0	876	75	4	100	92
[CSV] LLMASP^{A2}	1398	121	2	0	0	3170	1639	61	1815	255	2326	1033	124	30	26	501	4	11	0	0	879	89	1	138	138
[Datalog] LLMASP^{A2}	1352	153	48	0	0	3025	1250	206	142	138	2096	967	354	3	3	502	2	10	0	0	876	57	4	112	112
[JSON] LLMASP^{A2}	1396	112	4	0	0	3169	867	62	32	32	2419	326	31	0	0	494	5	18	0	0	879	93	1	57	57
LLMASP+KB^{A2}	1400	278	0	0	0	3036	1654	195	367	353	2414	631	36	18	13	499	12	13	0	0	876	75	4	100	92
[CSV] LLMASP+KB^{A2}	1396	129	4	0	0	3170	1639	61	1815	255	2326	1033	124	30	26	501	4	11	0	0	879	89	1	138	138
[Datalog] LLMASP+KB^{A2}	1382	108	18	0	0	3025	1250	206	142	138	2096	967	354	3	3	502	2	10	0	0	876	57	4	112	112
[JSON] LLMASP+KB^{A2}	1395	120	5	0	0	3169	867	62	32	32	2419	326	31	0	0	494	5	18	0	0	879	93	1	57	57

Stable Marriage (SM) reinforces this phenomenon; extraction performance degrades slightly when moving from the baseline [JSON] LLM^{A2} to LLMASP^{A2}. For VLP, the baseline [CSV] LLM^{A2} performs best, while its JSON counterpart performs worst. The JSON configuration introduces a unique failure mode here: it hallucinates the arity integer directly into the predicate name (e.g., `pipe/2(101,103)` and `junction/1(10)`). Without this syntax confusion, the extractions would be largely correct. The zero in the true positives for [JSON] LLM^{A1} in this problem domain is driven by the same failure mode. It is interesting to understand why this specific failure mode takes place in VLP only and not in other domains. We leave this to future research.

In Graph Colouring, the A1 applications are highly effective, with the JSON specification yielding zero false positives. The [CSV] LLMASP^{A1} configuration struggles slightly more with unary predicates, occasionally outputting string variables like `node(node_12)` or `node(node27)` instead of the required integer ID. It is interesting that this error mode happens in less than 10 instances. Additionally, this is a different error mode than the one observed, on the same problem domain, for [CSV] LLMASP^{A2}, where the primary error mode consists of the symmetry resolution of the `link(node_1,node_2)` predicate. Finally, Incremental Scheduling exhibits the most chaotic error patterns, particularly with the CSV grammar, which generates out-of-schema artifacts `like 1(1,one)` or `instances(device,quantity)`.

Notably, errors are highly localized. For example, [CSV] LLMASP^{A2} generates a total of 2,983 false positives across all 16 domains; yet, Graph Colouring (FP=1,639) and Incremental Scheduling (FP=1,033) account for roughly 89.6% of them. Summing over the false positives of S, GC, IS, and VLP accounts for 96.7% of the total false positives generated by [CSV] LLMASP^{A2}.

To demonstrate that the failure modes identified above are superficial rather than systemic reasoning failures, we implemented a lightweight, deterministic post-processing script tailored to the Graph Colouring and Incremental Scheduling domains. For Graph Colouring, we applied a simple canonicalization rule to enforce symmetry (sorting the arguments of `link/2`) and a regular expression to isolate integer IDs for `node/1` predicates. For Incremental Scheduling, we implemented a strict schema whitelist to drop out-of-signature predicates (e.g., `1(1,one)` or `instances(device,quantity)`) and filtered out constants containing placeholder strings (e.g., `NONE`). When applied to [CSV] LLMASP^{A2}, these few simple rules decreased the number of false positives from 1,639 to 433 in

the Graph Colouring domain and from 1,033 to 309 in the Incremental Scheduling domain, increasing the overall F1-score from 0.966 to 0.980. Similarly, on [JSON] LLMASP^{A2} the F1-score increases from 0.977 to 0.985.

In summary, this error analysis demonstrates that (i) LLMASP systematically improves over single-call baselines, and grammar constraints are broadly beneficial; (ii) the resulting errors are predictable and largely solvable via standard post-processing; and (iii) future iterations of LLMASP must prioritize precision and strict literal grounding to tackle the final challenge of perfect extraction.

E Research Methods

E.1 Experimental Methodology

The research methodology follows an empirical evaluation of the proposed LLMASP framework and its grammar-constrained extensions. We assess structured fact extraction from natural language descriptions across multiple domains derived from ASP Competition benchmarks. The study systematically varies the extraction strategy (single-prompt vs. predicate-specific), the presence of grammar constraints, the use of a symbolic knowledge base, and the underlying language model. Performance is evaluated using precision, recall, F1-score (macro and micro), perfect extraction rate, token usage, runtime, and energy consumption. All experiments are conducted under controlled conditions, and comparisons are made across identical datasets and configurations.

E.2 Implementation and Evaluation Protocol

The LLMASP pipeline is implemented using the Ollama framework for LLM inference and CLINGO for Answer Set Programming. We employ Meta LLaMA 3.1 and LLaMA 3.3 models (8B and 70B parameters) and evaluate grammar-constrained decoding using CSV, Datalog, and JSON grammars. Custom grammars are enforced through a patched grammar interface. Energy consumption and runtime are measured during execution, and regression analysis is used to study the relationship between token usage, grammar choice, knowledge base integration and energy cost. All reported results aggregate multiple problem instances per domain and configuration.

F Online Resources

All artifacts required to reproduce the experimental results, including the LLMASP implementation, grammar specifications, configuration files, and experimental scripts, are made publicly available at https://github.com/Matteio/LLMASP-JAIR26 . The repository includes detailed documentation describing the execution pipeline, dependencies, and instructions for reproducing the experiments. Links to the source code and supplementary materials will be provided upon publication.

Received 19 January 2026; accepted 27 August 2026