LLMASP-LGX: A Lightweight Gateway to Logic-Guided Reasoning with LLMs and ASP

Mario Alviano[1], Lorenzo Grillo[1], Nicola Leone[1] and Fabrizio Lo Scudo[1,*]

[1]*University of Calabria, Italy*

Abstract

Answer Set Programming (ASP) is a mature formalism for non-monotonic reasoning, constraint satisfaction, and combinatorial problem solving, but its adoption beyond specialist circles is held back by the cost of writing and maintaining complete logic programs. Large Language Models (LLMs) are complementary in a specific operational sense: they are fluent at turning natural-language descriptions into candidate relational facts, but unreliable as exact symbolic reasoners.

Hybrid systems such as LLMASP combine the two, yet in their standard form they keep extraction and reasoning strictly sequential—the oracle is queried for every target predicate, and ASP is consulted only afterwards, filtering a database in which hallucinated facts already mix with well-supported ones. LLMASP-LGX closes this loop through a lightweight YAML-based configuration: the user declares predicates, natural-language extraction prompts, admissibility guards with monotonicity flags, and small ASP knowledge-base fragments, and the LLMASP-LGX engine reads these declarations as an interaction policy between the LLM oracle and the ASP solver.

The central claim of this extended abstract is that, despite its lightweight syntax, an LLMASP-LGX configuration specifies a *constructive* interaction between LLM and ASP: facts extracted so far, together with facts derived by ASP, decide which LLM queries are admissible at each step, and LLM responses update the logical state that drives subsequent reasoning. Reported experiments show that these lightweight annotations reduce oracle calls by more than 70% on a mixed-domain benchmark, while matching the quality of an *a posteriori* ASP filter and improving substantially over unguided extraction.

Keywords

Answer Set Programming, Large Language Models, End-User Logic Programming, Logic-Guided Extraction, Neural–Symbolic Integration

1. Motivation

Logic Programming has a long track record on problems that demand exact, declarative reasoning: configuration, scheduling, diagnosis, planning, and data integration [1, 2]. Yet outside specialist circles its adoption remains low. A domain expert may carry a precise mental model of the relevant constraints and concepts and still lack the training to render that model as a correct ASP program. This is the Knowledge Acquisition Bottleneck in its textbook form: the knowledge is there; the cost of formalising it is what is too high.

LLMs [3] look, at first, like a convenient way around the bottleneck. They read natural language fluently, extract structured information from heterogeneous descriptions, and rarely demand more tailoring than a prompt. They are not, however, substitutes for symbolic solvers: they hallucinate unsupported facts, drop global constraints, and break down on tasks that require exact non-monotonic or combinatorial reasoning [4, 5]. What they lack is precisely what ASP supplies.

This complementarity motivates hybrid systems such as LLMASP [6], in which an LLM *oracle* turns text into candidate facts and an ASP solver validates, completes, or reasons over them. The standard LLMASP pipeline, however, keeps the two layers strictly sequential: the oracle is queried independently for every target predicate, and ASP runs only after all candidate facts are in. Logic stays outside the extraction loop, with the oracle asked, predicate by predicate, about information the input may not

Joint Proceedings of the Workshops and Doctoral Consortium of the 42nd International Conference on Logic Programming (ICLP 2026), co-located with the Federated Logic Conference (FLoC 2026), July 18–19 and 24–25, 2026, Lisbon, Portugal

*Corresponding author.

mario.alviano@unical.it (M. Alviano); fabrizio.loscudo@unical.it (F. Lo Scudo)

© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

published 2026-11-03

CEUR-WS.org/Vol-4311/EULP_short_02.pdf

contain at all: an instance described as a Labyrinth puzzle is nonetheless probed for Sokoban-specific facts. This is precisely the regime in which LLMs are weakest. When no answer is available in the context, they may fabricate a plausible one rather than report absence; this is the failure mode the literature calls *hallucination* [7], and an unguided extraction pipeline invites it predicate by predicate. The resulting database mixes well-supported facts with hallucinated ones, and ASP, called only at the end, can do little more than filter the mixture. It cannot un-ask a question, and it cannot recover the oracle budget already spent; with end users who will eventually pay this extra cost. This late-stage use of ASP also makes failures harder to localise. Once extraction and reasoning have been performed in separate phases, an incorrect output may originate from an over-permissive prompt, a spurious oracle response, an inadequate logical constraint, or their interaction. For an end user, the absence of an explicit extraction policy obscures which questions were asked, why they were asked, and where corrective action should be applied.

LLMASP-LGX [8] closes this loop. The user writes a lightweight configuration, and the framework engine, acting as an orchestrator, reads it as an interaction policy between the oracle and the ASP solver: facts extracted so far, together with facts derived symbolically by ASP, decide which predicates are admissible at each step. ASP no longer corrects the oracle after the fact; it directs the conversation as it unfolds. It is important to note that, as LLMASP, the LLMASP-LGX's focus is on the fact extraction task, and the oracle is never requested to generate individual rules or complete programs.

The contribution of this extended abstract is to distil LLMASP-LGX as an end-user logic programming interface, emphasizing the configuration artefact through which non-specialists specify the LLM–ASP interaction. We present LLMASP-LGX as a lightweight formalism that makes ASP-guided reasoning easy to specify, reuse, and adapt, with the full machinery of LLM–ASP orchestration encapsulated by a compact configuration layer. In a nutshell, LLMASP-LGX automates the extraction of a problem *instance*—a set of ground facts for a problem whose ASP encoding already exists—from natural-language text under logical control. The output is thus a logical database of facts ready for an existing ASP encoding, not a synthesised ASP program; the user writes only the small rule fragments needed to derive auxiliary facts and to constrain admissibility.

By guarding extraction with logic, the configuration suppresses queries the input cannot answer, cutting hallucinated facts and oracle calls together; and by making the extraction policy explicit, it lets errors be localised to a specific predicate, guard, or rule rather than lost in an opaque pipeline.

2. A Lightweight LLMASP Configuration

An LLMASP-LGX application lives in two YAML files. The *behaviour file*—typically written once and reused—defines how the oracle is contacted: initialisation instructions, output format, and the prompt template. The *application file* is the artefact a domain expert actually edits: an ordered list of predicates to extract, each carrying a natural-language description, optional admissibility guards, and an optional ASP fragment evaluated after extraction. Listing 1 shows a compact example.

Listing 1: A compact LLMASP-LGX application fragment.

```
- problem_name(name):
    prompt: |
      Tell me the name of the problem that best matches the text.
      Choose from: [labyrinth, nomystery, ricochetrobots, sokoban]
- size(value):
    prompt: |
      value is the size of the quadratic grid (an integer).
    extraction_condition:
      - condition: problem_name(labyrinth)
        monotone: true
    knowledge_base: |
      field(X,Y) :- size(N), X = 1..N, Y = 1..N.
      #show field/2.
```

The fragment declares two predicates: `problem_name/1` is queried unconditionally, while `size/1` is queried only when `problem_name(labyrinth)` holds in the current logical state. The `monotone: true` flag asserts that this guard, once satisfied, cannot be invalidated by any later fact, so the LLMASP-LGX engine caches its positive outcome and never re-evaluates it. The knowledge base attached to `size/1` then materialises every grid cell, and any later predicate that mentions `field/2` can reuse this derivation without an additional oracle call.

The user thus supplies three kinds of knowledge: prompts that say *what* to extract, guards that say *when* extraction is meaningful, and ASP fragments that say *what follows* once a predicate has been extracted. Everything else—prompt construction, oracle invocation, JSON parsing, guard evaluation, ASP solving, fact propagation, caching—is the job of the LLMASP-LGX engine.

The two ASP-bearing fields differ in expressivity. An `extraction_condition` is a guard: a conjunction of (possibly partially ground) ASP literals, evaluated as a query over the current database, each tagged with a `monotone` flag. A `knowledge_base`, attached to a single predicate or declared globally, accepts an arbitrary ASP program fragment, including ordinary rules, choice rules, aggregates, and integrity constraints; a global integrity constraint can, for instance, reject any `problem_name/1` value outside the declared set, surfacing an inconsistency as soon as the offending fact is extracted. ASP thus appears in the configuration only where symbolic precision is meaningful, and the rest reads as natural language.

3. Constructive Interaction Between LLM and ASP

A configuration induces a *dynamic* message exchange: the sequence of oracle queries is not fixed in advance but generated from the evolving logical state. The LLMASP-LGX engine walks the predicate list one entry at a time. If the guards of the current predicate hold, it instantiates the prompt template, queries the oracle, parses the returned facts into the database, and evaluates the predicate's local knowledge base; otherwise the call is skipped. Either way, the database that drives the next guard check has changed. The result is a feedback loop:

$$\text{oracle extraction} \longrightarrow \text{ASP derivation} \longrightarrow \text{guard evaluation} \longrightarrow \text{next oracle message}$$

in which each component does what the other cannot: the LLM parses arbitrary text into candidate facts; ASP reasons exactly over those facts. Neither alone is enough, and the configuration specifies their alternation.

One simple pattern that LLMASP-LGX makes easy to implement is the *domain router*. An early extraction classifies the input, and later predicates are guarded by the resulting classification, so that predicates belonging to unrelated categories are skipped automatically. The mixed-domain benchmark of [8] instantiates this pattern: `problem_name/1` is extracted first against four planning categories (Labyrinth, NoMystery, Ricochet Robots, Sokoban), and each domain-specific predicate carries an `extraction_condition` referencing it. A Labyrinth input therefore does not trigger Sokoban or Ricochet queries, yielding fewer oracle calls and avoiding opportunities for cross-domain hallucination.

This is *constructive* reasoning, not merely corrective. A post-hoc design lets the oracle produce a possibly noisy database and asks ASP to filter it afterwards; LLMASP-LGX uses ASP to shape the database as it grows—deciding which questions are admissible, deriving intermediate facts, and letting those facts steer later queries. Each response may change the logical state, and each change in the logical state may change the next message.

For the end user, this means a clean split: the configuration captures the *what* and *when* of the interaction; the engine handles the rest (Table 1).

4. Empirical Indications

The full evaluation appears in [8]; we here recall the aspects most relevant to the end-user argument. LLMASP-LGX is compared against an unguided LLMASP baseline on two benchmarks—a graph bench-

Table 1
Division of responsibilities between the user and the LLMASP-LGX engine.

	User	**LLMASP-LGX engine**
Extraction	Writes a natural-language prompt per predicate.	Builds oracle messages, parses and validates JSON output.
Ordering	Lists predicates in the intended evaluation order.	Processes predicates in that order; enforces guards.
Guards	Declares admissibility conditions as ASP atoms.	Evaluates guards against the current database.
Caching	Marks each guard monotone or non-monotone.	Maintains and invalidates guard caches accordingly.
Reasoning	Writes local and global ASP fragments where useful.	Grounds, solves, and propagates derived facts.
Orchestration	None.	Interleaves oracle calls, guard evaluation, and ASP solving.

mark and the mixed-domain planning benchmark of Section 3—each run with three LLM sizes (8B, 70B, and 120B parameters). To isolate the effect of guarding, the baseline is simulated with the same engine under a degenerate, guard-free configuration; oracle responses are cached and the LLM temperature is fixed at zero.

The annotations pay off on both benchmarks, and the gains scale with the structural richness of the domain. On the graph benchmark, guards cut oracle calls by roughly 15–23% and match the quality of the best hand-crafted post-hoc filter, lifting the F1-score from 0.800 to 0.893 on the Small model and reaching 1.000 on the Large one. On the mixed-domain benchmark the domain-router pattern is more dramatic: oracle calls drop by more than 70% (from up to 3,072 down to about 864 on the Medium and Large configurations), and quality again matches the corresponding *a posteriori* filter—inadmissible facts are avoided by construction, not discarded after the fact. Guard caching, activated by the single `monotone: true` flag, further saves between 61.4% and 82.6% of the ASP solver calls that would otherwise be redundant.

The user-facing configuration is therefore not syntactic sugar: each annotation has a measurable operational consequence, and that consequence flows from the very declaration the domain expert can read and write. Accessibility and formal control reinforce each other here rather than trading off.

5. Conclusion

LLMASP-LGX takes a clear position in the design space of neural–symbolic systems: it neither hides logic nor demands a full ASP encoding from the user. It surfaces only the constructs needed to specify a constructive LLM–ASP interaction—predicates, guards, monotonicity flags, and small knowledge bases—and delegates the rest to the LLMASP-LGX engine. The resulting configuration is simpler than a full ASP encoding and more reliable than a prompt-only LLM pipeline, and both properties follow from the same lightweight annotations.

Several extensions are natural. Inferring guard monotonicity automatically would remove one of the few annotations users still have to provide. Letting prompt templates depend on facts derived during execution—in the direction of the logic-driven templating used in ASP Chef [9]—would tighten the loop between symbolic reasoning and prompt construction. Beyond knowledge extraction, the same feedback loop applies wherever an LLM can turn informal input into candidate facts and ASP can derive consequences over them: product configuration, compliance checking, planning, and diagnosis are immediate next targets.

Declaration on Generative AI

During the preparation of this work, the authors employed a Generative AI tool for grammar and spelling checks and to improve narrative fluency. After using the tool, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

Acknowledgments

This work was supported 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 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 member of Gruppo Nazionale Calcolo Scientifico-Istituto Nazionale di Alta Matematica (GNCS-INdAM).

References

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

[2] I. Niemelä, Logic programming with stable model semantics as a constraint programming paradigm, 1999. doi:10.1023/A:1018930122475.

[3] H. Naveed, A. U. Khan, S. Qiu, M. Saqib, S. Anwar, M. Usman, N. Akhtar, N. Barnes, A. Mian, A comprehensive overview of large language models, 2025. URL: https://doi.org/10.1145/3744746. doi:10.1145/3744746.

[4] Z. Yang, A. Ishay, J. Lee, Neurasp: Embracing neural networks into answer set programming, in: Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020, ijcai.org, 2020, pp. 1755–1762. URL: https://doi.org/10.24963/ijcai.2020/243. doi:10.24963/IJCAI.2020/243.

[5] R. Manhaeve, S. Dumancic, A. Kimmig, T. Demeester, L. D. Raedt, Deepproblog: Neural probabilistic logic programming, in: NeurIPS, 2018, pp. 3753–3763.

[6] M. Alviano, L. Grillo, F. Lo Scudo, L. A. R. Reiners, Integrating answer set programming and large language models for enhanced structured representation of complex knowledge in natural language, in: Proceedings of IJCAI 2025, 2025, pp. 4330–4338.

[7] H. Orgad, M. Toker, Z. Gekhman, R. Reichart, I. Szpektor, H. Kotek, Y. Belinkov, Llms know more than they show: On the intrinsic representation of llm hallucinations, in: Y. Yue, A. Garg, N. Peng, F. Sha, R. Yu (Eds.), International Conference on Learning Representations, volume 2025, 2025, pp. 66880–66913. URL: https://proceedings.iclr.cc/paper_files/paper/2025/file/a712d461e57201efe35d429a6f1731c1-Paper-Conference.pdf.

[8] M. Alviano, L. Grillo, N. Leone, F. Lo Scudo, Logic-guided data extraction with answer set programming and large language models, in: 42nd International Conference on Logic Programming (ICLP 2026), 2026.

[9] M. Alviano, L. A. R. Reiners, W. Faber, ASP chef grows mustache to look better, Theory and Practice of Logic Programming 25 (2025) 417–436.