TPLP **26** (1-2): 8–25, 2026. © The Author(s), 2026. Published by Cambridge University Press. This is an Open Access article, distributed under the terms of the Creative Commons Attribution licence (https://creativecommons.org/licenses/by/4.0/), which permits unrestricted re-use, distribution and reproduction, provided the original article is properly cited.

doi:10.1017/S1471068426100635 First published online 8 July 2026

*Logic-Guided Data Extraction with Answer Set Programming and Large Language Models**

MARIO ALVIANO
Department of Mathematics and Computer Science, University of Calabria, Italy
(*e-mail:* mario.alviano@unical.it)

LORENZO GRILLO
Università degli Studi della Campania Luigi Vanvitelli, Italy
(*e-mail:* lorenzo.grillo@unicampania.it)

NICOLA LEONE
Department of Mathematics and Computer Science, University of Calabria, Italy
(*e-mail:* nicola.leone@unical.it)

FABRIZIO LO SCUDO
Department of Mathematics and Computer Science, University of Calabria, Italy
(*e-mail:* fabrizio.loscudo@unical.it)

submitted 30 May 2026 UTC; revised 30 May 2026 UTC; accepted 1 June 2026 UTC

Abstract

When large language models (LLMs) are used for semantic data extraction from unstructured text, producing candidate relational facts from natural language, they may remain unreliable for tasks requiring complex combinatorial reasoning and global consistency. This paper proposes a logic-guided data extraction framework combining LLM-based extraction with answer set programming (ASP). The LLM produces candidate facts, whereas ASP performs validation, inference, consistency checking, and control. Unlike existing pipelines that query the LLM independently for all target predicates, the proposed approach uses ASP reasoning to identify which predicates are logically admissible at each stage and to guide extraction queries. By interleaving LLM calls with ASP derivation, the framework infers logically implied facts without further extraction and detects inconsistencies early. We formalize the pipeline and prove that, under

* 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).

https://doi.org/10.1017/S1471068426100635 Published online by Cambridge University Press

mild assumptions, it is equivalent to the baseline approach with respect to the final extracted facts, while requiring fewer LLM calls. We also introduce a caching mechanism for logic-based control queries, exploiting monotonicity of conjunctive queries over incrementally constructed fact sets to reduce solver invocations. Experiments on ASP-derived benchmarks show that the framework reduces LLM calls and improves extraction quality by mitigating spurious outputs, demonstrating the value of non-monotonic logic programming for controlled semantic extraction.

KEYWORDS: answer set programming, semantic data extraction, logic programming, large language models, neural–symbolic integration

1 Introduction

Large language models (LLMs), surveyed by Naveed *et al.* (1925), have recently been adopted for semantic data extraction and parsing from unstructured text in the works by Xu *et al.* (1923); Jin *et al.* (1924); Zhu *et al.* (1925); Zhang *et al.* (1923), enabling the automatic construction of structured representations such as relational facts or knowledge bases. In logic-based extraction pipelines, such as the [LLM]+ASP framework proposed by Yang *et al.* (1923) and the LLMASP framework by Alviano *et al.* (1925), extraction is commonly performed independently for each target predicate, with logical validation and inference applied only afterward. While effective, this design often requires issuing a large number of extraction queries and coping with incomplete or spurious outputs, especially in domains where predicates are logically interdependent. Crucially, this approach decouples extraction from *reasoning*: the LLM is used merely as a text parser, ignoring the deductive steps that could actively guide the information retrieval.

This work lies at the intersection of Neuro-Symbolic AI and Information Extraction. Unlike probabilistic neuro-symbolic frameworks such as DeepProbLog by Manhaeve *et al.* (1918) and NeurASP by Yang *et al.* (1920) that integrate neural outputs via probabilistic semantics, we focus on utilizing pre-trained LLMs as black-box oracles. In this modular setting, existing approaches typically employ logic only for post-hoc validation, as proposed by Eiter *et al.* (1923), or for iterative refinement, as explored by Madaan *et al.* (1923) and Shinn *et al.* (1923). Answer set programming (ASP), introduced by Marek and Truszczyński (1999) and Niemelä (1999), is particularly well-suited for this task due to its ability to represent complex relational structures and non-monotonic dependencies, as demonstrated by Calì *et al.* (2002); Leone *et al.* (2005). However, when ASP is used solely as a reasoning component after extraction (as in LLMASP), the logical structure of the domain is not exploited to guide the extraction process itself. As a consequence, logical dependencies that could inform extraction decisions are ignored during the extraction phase.

Example 1 (LLM Extraction with Logic-Based Reasoning).
Consider the *Crossing Minimization* problem, where one seeks an ordering of nodes within layers of a graph that minimizes edge crossings (see Figure 1). Solving such combinatorial reasoning tasks directly with a LLM is challenging, as it requires enforcing global consistency constraints and exploring complex structural dependencies. Rather than expecting the LLM to solve reasoning tasks implicitly, better results can be obtained through a clear division of the work: the LLM extracts candidate facts from text, and ASP

https://doi.org/10.1017/S1471068426100635 Published online by Cambridge University Press

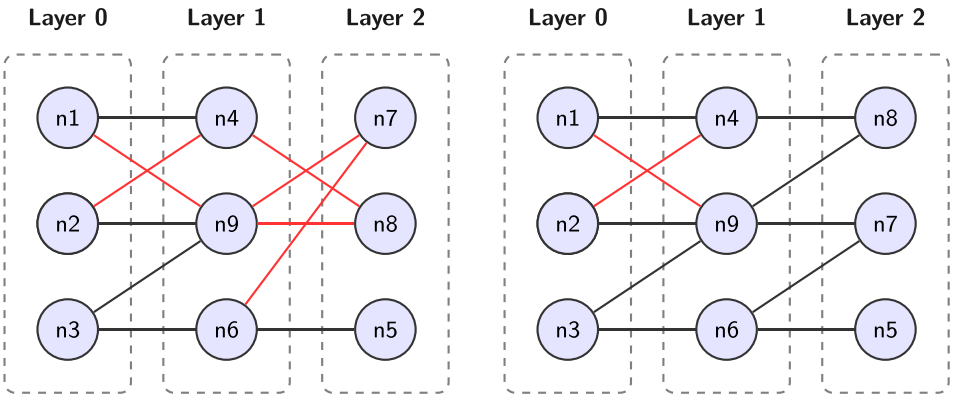


Figure 1. Comparison of a three-layer graph before (left) and after (right) applying a crossing minimization strategy. Intersecting edges are highlighted in red. Reordering the nodes, specifically in Layer 2, significantly reduces the total number of crossings and improves the readability of the layout.

performs reasoning and consistency checks. Here, the extraction task consists in building a structured graph representation from natural language descriptions. For example, a text may describe a (possibly layered) undirected graph by mentioning nodes, edges, and, when relevant, information about layers. LLM-based extractors can be queried to produce facts such as `edge(n1,n9)`, `layer_size(0,3)`, or `in_layer(0,n1)`. However, issuing extraction requests for all target predicates independently, as in LLMASP, can be both inefficient and error-prone: it may lead to redundant oracle calls and to spurious extractions, such as layer assignments when no layering information is present in the text. This motivates logic-based admissibility conditions that determine when extraction queries should be issued. We will use this layered-graph scenario as a running example throughout the paper (see Example 2).

In this paper, we propose a logic-guided data extraction framework in which ASP plays an active role in controlling and structuring the interaction with an LLM-based extractor. Rather than invoking the LLM independently for all target predicates (a bottom-up approach), our framework uses ASP reasoning to determine which predicates are logically admissible at each stage of the extraction process. Parallel to the reasoning-acting decomposition introduced in ReAct by Yao *et al.* (1923), and conceptually akin to the evaluation of external atoms in HEX-programs described by Eiter *et al.* (1918) and to the execution of external actions in ActHEX introduced by Fink *et al.* (1913), the extraction is interleaved with ASP reasoning and derivation, allowing logically implied facts to be inferred without any additional extraction and enabling early consistency checks on partial data. This query-driven behavior is similar in spirit to goal-directed ASP evaluation as developed by Arias *et al.* (1919), ensuring that extraction resources are spent only on facts that are consistent with the current partial model. We formalize the proposed pipeline and show that, under mild assumptions on the extraction oracle, it is equivalent to a baseline LLM-based extraction approach in terms of the final extracted facts, while never increasing the number of calls to the LLM and potentially reducing them.

https://doi.org/10.1017/S1471068426100635 Published online by Cambridge University Press

We further introduce a caching mechanism for logic-based control queries, exploiting monotonicity properties of conjunctive queries over incrementally constructed fact sets to reduce the number of ASP solver invocations. An experimental evaluation on benchmarks derived from standard ASP domains demonstrates that the proposed framework substantially reduces the number of LLM calls and, in practice, improves extraction quality by mitigating spurious outputs. These results show that non-monotonic logic programming can effectively serve as a control mechanism for semantic data extraction from text, complementing the generative capabilities of LLMs.

Contributions. This paper makes the following contributions: (i) we introduce a clean formal abstraction of LLM-based extraction pipelines, exemplified by LLMASP; (ii) we propose a logic-guided extraction framework that interleaves extraction and ASP reasoning via guard-based admissibility conditions; (iii) we establish equivalence with the baseline pipeline under a guard-respecting oracle assumption and prove efficiency properties; (iv) we present an execution algorithm with sound guard caching; (v) we experimentally evaluate the framework on ASP-derived benchmarks.

Structure. The remainder of the paper is organized as follows. Section 2 introduces the baseline extraction setting and notation. Section 3 presents the logic-guided extraction framework and its formal properties. Section 4 presents an execution algorithm for logic-guided extraction and its realization of the proposed semantics. Section 5 reports the experimental evaluation, and Section 7 concludes.

2 Preliminaries and baseline setting

This section introduces a formal abstraction of LLM-based extraction pipelines, including existing approaches such as LLMASP. While inspired by prior work, this abstraction is new and will serve as the foundation for the logic-guided framework introduced in Section 3. The presentation is intentionally minimal and focuses only on concepts required to formalize logic-guided extraction.

Text, predicates, and facts. Let T denote a finite unstructured text document, and $\mathscr{T}$ be the domain of text documents. Let $\mathscr{P}$ be a fixed finite set of predicates, each with a fixed arity. A *ground fact* is an atom of the form $p(c_1, \ldots, c_k)$, where $p \in \mathscr{P}$ has arity $k \geq 0$ and each c_i is a constant from a fixed finite domain of constants. Let $\mathscr{F}_p$ denote the set of all ground facts over predicate p, for all $p \in \mathscr{P}$, and $\mathscr{F}$ be the set of all ground facts, that is $\mathscr{F} = \bigcup_{p \in \mathscr{P}} \mathscr{F}_p$. A *database* D is a finite set of ground facts, that is, $D \subseteq \mathscr{F}$. Let $\mathscr{D}$ denote the domain of databases, defined as $2^{\mathscr{F}}$. In the following, we assume databases grow monotonically during an extraction process, that is, along a finite sequence $D_0 \subseteq \cdots \subseteq D_n$.

Queries. We consider *boolean conjunctive queries* over databases. A conjunctive query q is a conjunction of (possibly negated) atoms whose variables are implicitly existentially quantified. Given a database D, we say that q is *true* in D, written $D \models q$, if there exists a substitution σ mapping the variables of q to constants such that (i) for every positive atom a in q, $a\sigma \in D$, and (ii) for every negated atom *not* a in q, $a\sigma \notin D$. A query q is said to be *monotone* if for all databases $D \subseteq D'$, $D \models q$ implies $D' \models q$. In our setting,

https://doi.org/10.1017/S1471068426100635 Published online by Cambridge University Press

monotone queries are exactly those that do not contain negated atoms. In later sections, monotone queries will be exploited to optimize logic-based control checks via caching.

Logic programs. Let $\mathscr{LP}$ denote the set of logic programs over predicates in $\mathscr{P}$. Specifically, we consider normal logic programs under the stable model semantics. A *normal rule* is an expression *"head if body"* of the form

$$a \leftarrow b_1, \ldots, b_m,\ not\ c_1, \ldots, not\ c_n,$$

where a, b_i, and c_j are atoms. A *normal program* Π is a finite set of normal rules over predicates in $\mathscr{P}$. Programs are interpreted under their standard ground instantiation with respect to the fixed finite domain of constants. Given a ground program Π and a set of ground atoms I, the *Gelfond–Lifschitz reduct* Π^I is obtained by removing all rules whose body contains a negative literal *not* c with $c \in I$, and by deleting all remaining negative literals from the bodies of the remaining rules. A set of atoms I is a *stable model* (or *answer set*) of Π if I is the least model of the reduct Π^I. In the intended extraction setting, the programs are constructed so that $\Pi \cup D$ has a unique intended answer set. We denote it by $Ans_\Pi(D)$.

Extraction oracle. We abstract the behavior of a LLM by means of an *extraction oracle*. Let $\mathscr{P}_E = \{p_1, \ldots, p_n\} \subseteq \mathscr{P}$ be a finite set of *target predicates*, that is predicates whose facts are to be extracted from text. When needed, we assume that the set of target predicates $\mathscr{P}_E$ is equipped with a fixed ordering. Formally, an extraction oracle is a function

$$\begin{aligned} \mathscr{O} : \mathscr{T} \times \mathscr{P}_E &\rightarrow \mathscr{D} \\ (T, p) &\mapsto D \end{aligned}$$

such that $D \subseteq \mathscr{F}_p$. Intuitively, $\mathscr{O}(T, p)$ returns a (possibly empty) set of candidate facts for predicate p extracted from the input text T. The oracle may be incomplete or return spurious facts, and no assumptions are made regarding determinism, completeness, or monotonicity of its behavior. Since at most one oracle call is performed per target predicate, the length of the extraction sequence is bounded by $|\mathscr{P}_E|$.

Prompt transformations. We model prompt construction abstractly as text transformations. Let $\Gamma : \mathscr{T} \rightarrow \mathscr{T}$ be a global transformation encoding macro-task or domain-level instructions, and let $\gamma_p : \mathscr{T} \rightarrow \mathscr{T}$ be predicate-specific transformation encoding instructions for extracting predicate $p \in \mathscr{P}_E$. The extraction oracle is always applied to transformed texts of the form $\gamma_p(\Gamma(T))$, that is, when we write $\mathscr{O}(T, p)$ we actually mean $\mathscr{O}(\gamma_p(\Gamma(T)), p)$. Stated differently, prompt transformations are modeled as text-to-text functions applied to the input text before oracle invocation; for clarity of presentation, they are omitted from the notation.

Example 2 (Running example).
Let Γ be the macro-task function mapping a text $\underline{t}$ to

```
Follow my instructions on how to map the INPUT to the given OUTPUT format.
The INPUT text describes a (possibly layered) undirected graph.
INPUT: t     OUTPUT:
```

https://doi.org/10.1017/S1471068426100635 Published online by Cambridge University Press

Let $\gamma_{\texttt{in_layer/2}}$ be the transformation associated with `in_layer/2`, mapping a text $\underline{t}$ to

```
t in_layer(layer, node).
For each layer, list every node that belongs to that layer.
```

Let T be the following unstructured text document:

```
The graph has three layers (0 to 2), each containing three nodes. Layer zero
contains nodes n1, n2, and n3, while layer 2 contains nodes n7, and n9.
There is an edge between n1 and n9, and node n4 is connected to n2 and n8.
```

So, if we call $\mathscr{O}(T, p)$, we are actually calling $\mathscr{O}(T', p)$, where $T' = \gamma_p(\Gamma(T))$ is

```
Follow my instructions [···] a (possibly layered) undirected graph.
INPUT: The graph has [···] n2 and n8.     OUTPUT: in_layer(layer, node).
For each layer, list every node that belongs to that layer.
```

The expected output are facts `in_layer(0,n1)`, `in_layer(0,n2)`, `in_layer(0,n3)`, `in_layer(2,n7)`, and `in_layer(2,n9)`, while facts like `in_layer(2,n8)`, and `in_layer(1,n5)` must not be produced in output as they are not explicitly mentioned in T. Specifically, we expect that the extraction process terminates reporting only facts that are actually present in the given input, avoiding any speculative interpolation of missing data.

Baseline extraction pipeline (LLMASP). As baseline, we consider a predicate-wise extraction pipeline, as exemplified by LLMASP-style approaches presented by Alviano *et al.* (1925). Given a fixed set of target predicates $\mathscr{P}_E$, we model the baseline extraction pipeline as a function

$$\begin{aligned} \mathsf{LLMASP} : \mathscr{T} \times \mathscr{D} \times \mathscr{LP} &\rightarrow \mathscr{D} \\ (T, D_0, \Pi) &\mapsto Ans_\Pi(D_0 \cup \textstyle\bigcup_{p \in \mathscr{P}_E} \mathscr{O}(T, p)) \end{aligned}$$

where $\mathsf{LLMASP}(T, D_0, \Pi)$ denotes the database produced by the baseline pipeline on input text T, program Π and initial database D_0. Operationally, the baseline invokes the extraction oracle independently for each target predicate $p \in \mathscr{P}_E$ on the (transformed) input text, collects all the extracted facts together with D_0 into a database $D' = D_0 \cup \bigcup_{p \in \mathscr{P}_E} \mathscr{O}(T, p)$, and then applies ASP reasoning once using program Π, yielding the final database $Ans_\Pi(D')$. Note that, in this baseline setting, logical reasoning is applied only after the extraction phase and does not influence which target predicates are selected for extraction. This formalization will allow us to precisely compare the baseline with the logic-guided extraction framework introduced next.

Example 3 (Continuing Example 2).
Let $\mathscr{P}_E$ be $\{\texttt{in}_{\texttt{layer/2}}, \texttt{layer}_{\texttt{size/2}}, \texttt{edge/2}\}$, $\mathscr{O}(T, \texttt{layer}_{\texttt{size/2}})$ be $\{\texttt{layer}_{\texttt{size(0,3)}}, \texttt{layer}_{\texttt{size(1,3)}}, \texttt{layer}_{\texttt{size(2,3)}}\}$ and $\mathscr{O}(T, \texttt{edge/2})$ be $\{\texttt{edge(n1,n9)}, \texttt{edge(n4,n2)}, \texttt{edge(n4,n8)}\}$. Let Π be the following program:

```
r1 : edge(X,Y) :- edge(Y,X).
r2 : warning(unmatched_layer_size, (Layer,Size,Cnt)) :-  layer_size(Layer,Size),
       Cnt = #count{Node : in_layer(Layer,Node)}, Size != Cnt.
```

https://doi.org/10.1017/S1471068426100635 Published online by Cambridge University Press

where r_1 computes the symmetric closure of `edge/2` (the graph is undirected), and r_2 detects inconsistencies or missing data in the input text. $\mathsf{LLMASP}(T, \emptyset, \Pi)$ returns a database including `warning(unmatched_layer_size, (1,3,0))` and `warning(unmatched_layer_size, (2,3,2))`. In an interactive setting, the derived `warning` facts may serve as explanations of inconsistencies or missing data, enabling the system to ask the user for clarification or further details.

3 Logic-guided extraction framework

In the baseline pipeline introduced in Section 2, extraction and reasoning are strictly separated: all target predicates are extracted independently by the oracle, and logical reasoning is applied only once as a post-processing step. While this approach is simple and effective, it does not exploit logical knowledge during the extraction process itself.

In this section, we introduce a logic-guided extraction framework in which logical reasoning actively controls the extraction process. The key idea is to interleave oracle calls with logic-based checks over the current database, allowing extraction decisions to depend on facts already obtained and on their logical consequences. As a result, oracle calls for certain target predicates can be safely skipped without affecting the final extracted information.

The framework presented here generalizes the baseline pipeline and subsumes it as a special case. It provides a formal basis for reducing the number of oracle invocations while preserving the extracted facts under suitable assumptions, and it enables additional optimizations such as caching of logic-based checks.

3.1 Intuition and overview

The logic-guided extraction framework builds on a simple observation: in many domains, the extraction of certain predicates becomes unnecessary once sufficient information has already been obtained and logically processed. In the baseline pipeline, however, such dependencies are ignored, as all target predicates are extracted independently before any reasoning takes place. In contrast, our framework interleaves extraction and reasoning. After each oracle invocation, the newly extracted facts are combined with the current database and processed by a logic program, yielding additional derived facts. These derived facts can then be used to decide whether further oracle calls are needed. Intuitively, if the truth of a target predicate can already be established (or ruled out) based on the current database and logical knowledge, invoking the oracle for that predicate becomes redundant.

To support such decisions, the framework relies on logic-based checks expressed as queries over the current database. These queries act as *guards* that determine whether a target predicate is eligible for extraction at a given stage. Crucially, guards may depend both on facts explicitly extracted from the text and on facts derived by logical reasoning. By construction, this approach may skip oracle calls that would be performed in the baseline pipeline. Under suitable assumptions on the extraction oracle, however, skipping such calls does not affect the final set of extracted facts. As a result, the logic-guided framework can reduce the number of oracle invocations while preserving the outcome

https://doi.org/10.1017/S1471068426100635 Published online by Cambridge University Press

of the baseline pipeline, and in practice may also mitigate the generation of spurious or hallucinated facts.

3.2 Extraction conditions and guards

The logic-guided extraction framework is based on the notion of *extraction conditions*, which determine whether invoking the extraction oracle for a target predicate is necessary at a given stage of the process. These conditions are expressed declaratively as queries over the current database and its logical consequences.

Guards. Let $\mathscr{P}_E$ be the set of target predicates, equipped with a fixed ordering. For each predicate $p \in \mathscr{P}_E$, let $\mathscr{G}_p$ be a finite set of conjunctive queries, called the *guards* for p. Given a database D, predicate p is said to be *admissible* for extraction in D if all guards in $\mathscr{G}_p$ are satisfied in D, that is, if $D \models g$ for every $g \in \mathscr{G}_p$. If at least one guard in $\mathscr{G}_p$ is not satisfied in D, the extraction of p is skipped at that stage. Guards are not required to be unique to a predicate and may appear in the guard sets of multiple target predicates.

Monotone and non-monotone guards. A guard $g \in \mathscr{G}_p$ is said to be *monotone* if it does not contain negated atoms, as defined in Section 2. Monotone guards depend only on the presence of facts in the database and, once satisfied, remain satisfied as the database grows. Non-monotone guards may depend on the absence of facts and can change their truth value as new information becomes available. This distinction is important for optimization purposes. In particular, monotone guards admit persistent caching of positive outcomes, while non-monotone guards generally require re-evaluation when the database is extended.

3.3 Logic-guided extraction semantics

We now define the semantics of logic-guided extraction using the guard-based admissibility conditions introduced in Section 3.2. Extraction proceeds over a fixed ordering $\mathscr{P}_E = \langle p_1, \ldots, p_n \rangle$ of target predicates. Each predicate p_i is associated with a logic program Π_{p_i}, which is applied after processing p_i. We collectively refer to the ordering $\mathscr{P}_E$, the guards $\{\mathscr{G}_{p_i}\}_{i=1}^n$, and the programs $\{\Pi_{p_i}\}_{i=1}^n$ as a *logic-guided extraction configuration*. Let $\mathscr{C}$ denote the domain of all such configurations, and let $C \in \mathscr{C}$ be a fixed configuration.

Incremental database construction. The logic-guided extraction process constructs a sequence of databases $D_0 \subseteq D_1 \subseteq \cdots \subseteq D_n$, where D_0 is the initial database. For each $i \in \{1, \ldots, n\}$, the database D_i is obtained from D_{i-1} by processing predicate p_i according to its admissibility in D_{i-1}. If predicate p_i is admissible in D_{i-1}, then the extraction oracle is invoked for p_i, the extracted facts are added to the current database, and logical reasoning is applied using program Π_{p_i}. If predicate p_i is not admissible in D_{i-1}, the oracle invocation is skipped and no logical reasoning is applied. Formally,

$$D_i = \begin{cases} Ans_{\Pi_{p_i}}(D_{i-1} \cup \mathscr{O}(T, p_i)) & \text{if } D_{i-1} \models g \text{ for all } g \in \mathscr{G}_{p_i}, \\ D_{i-1} & \text{otherwise.} \end{cases} \tag{1}$$

https://doi.org/10.1017/S1471068426100635 Published online by Cambridge University Press

Logic-guided extraction function. The logic-guided extraction semantics defined above induces a function

$$\begin{aligned} \mathsf{LGX} : \mathscr{T} \times \mathscr{D} \times \mathscr{C} &\to \mathscr{D} \\ (T, D_0, C) &\mapsto D_n \end{aligned}$$

which maps an input text T, an initial database D_0, and a configuration $C \in \mathscr{C}$ to the final database D_n produced according to the semantics defined above.

Example 4 (Continuing example 3).
Let g be the guard `layer_size(_,_)`, g' be `not warning(_,_)`, and C be the configuration

$$\big(\langle \texttt{layer}_{\texttt{size/2}}, \texttt{in}_{\texttt{layer/2}}, \texttt{edge/2} \rangle, \langle \emptyset, \{g\}, \{g'\} \rangle, \langle \emptyset, \{r_2\}, \{r_1\} \rangle\big)$$

The target predicates are ordered so that `in_layer/2` is extracted only if some layer information has been found, as enforced by the guard g. Moreover, the program Π is split so that warnings about unmatched layer sizes (rule r_2) are derived immediately after extracting `in_layer/2`. If such warnings arise, the guard g' fails, thereby preventing the extraction of `edge/2` at later stages.

Baseline as a special case. In the proposed framework, a logic program can be seen as a degenerate logic-guided configuration in which no admissibility constraints are imposed and all logical reasoning is deferred to the final step. From this perspective, the baseline extraction pipeline of Section 2 is recovered as a special case of the above semantics. Specifically, for an arbitrary ordering $\mathscr{P}_E = \langle p_1, \ldots, p_n \rangle$, we associate an empty set of guards to every predicate, that is, $\mathscr{G}_{p_i} = \emptyset$ for all i. As a consequence, every predicate is admissible for extraction and the oracle is invoked for each p_i. Moreover, we associate the empty program with all predicates except the last one. Formally, we set $\Pi_{p_i} = \emptyset$ for all $i < n$ and $\Pi_{p_n} = \Pi$, where Π is the baseline logic program. With this choice, no logical reasoning is applied during extraction. All reasoning is deferred to the processing of the last predicate p_n. In this baseline setting, the order of predicates in $\mathscr{P}_E$ does not affect the final result. Accordingly, the input expected by LGX coincides exactly with the program parameter of the LLMASP function defined in Section 2.

3.4 Baseline equivalence

We now establish that the logic-guided extraction framework is equivalent to the baseline extraction pipeline under a degenerate configuration and a natural assumption on the extraction oracle.

Oracle assumption. We assume that the extraction oracle is *guard-respecting*, in the following sense. Let $p \in \mathscr{P}_E$ be a target predicate and let D be a database. If $D \not\models g$ for some guard $g \in \mathscr{G}_p$, then $\mathscr{O}(T, p) = \emptyset$. Intuitively, when the admissibility conditions for a predicate are not satisfied, invoking the oracle for that predicate would not produce any extracted facts.

Theorem 1 (Equivalence theorem).
Let $T \in \mathscr{T}$ be an input text and let $D_0 \in \mathscr{D}$ be an initial database. Let Π be a logic program, seen as a degenerate logic-guided configuration as described in Section 3.3. Let

https://doi.org/10.1017/S1471068426100635 Published online by Cambridge University Press

$\mathsf{LGX}(T, D_0, \Pi)$ denote the result of logic-guided extraction under this configuration, and let $\mathsf{LLMASP}(T, D_0, \Pi)$ denote the result of the baseline extraction pipeline. Under the oracle assumption above, it holds that $\mathsf{LGX}(T, D_0, \Pi) = \mathsf{LLMASP}(T, D_0, \Pi)$.

Proof (sketch).
Let $D_0 \subseteq \cdots \subseteq D_n$ be the sequence of databases produced by $\mathsf{LGX}(T, D_0, \Pi)$. For all $i \in \{1, \ldots, n\}$, we have $\mathscr{G}_{p_i} = \emptyset$ by assumption. Hence, $D_{i-1} \models g$ for all $g \in \mathscr{G}_{p_i}$ holds vacuously, and the update rule (1) reduces to $D_i = Ans_{\Pi_{p_i}}(D_{i-1} \cup \mathscr{O}(T, p_i))$.

By construction of the degenerate configuration, $\Pi_{p_i} = \emptyset$ for all $i < n$, and $\Pi_{p_n} = \Pi$. Moreover, $Ans_\emptyset(D) = D$ for any database D. Therefore, for all $i \in \{1, \ldots, n\}$,

$$D_i = \begin{cases} D_{i-1} \cup \mathscr{O}(T, p_i) & \text{if } i < n, \\ Ans_\Pi(D_{n-1} \cup \mathscr{O}(T, p_n)) & \text{if } i = n. \end{cases}$$

Unfolding the construction of D_n, we obtain

$$\mathsf{LGX}(T, D_0, \Pi) = D_n = Ans_\Pi\left(D_0 \cup \bigcup_{i=1}^{n} \mathscr{O}(T, p_i)\right).$$

By definition of LLMASP, the right-hand side coincides with $\mathsf{LLMASP}(T, D_0, \Pi)$. □

3.5 Extraction efficiency and guard-based optimization

A key advantage of logic-guided extraction is that admissibility checks allow the process to avoid unnecessary oracle invocations while preserving the final outcome. In LLMASP, the oracle is invoked once for every target predicate, whereas LGX invokes it only when guards are satisfied in the current database. Guard outcomes can be cached to avoid redundant guard evaluations (and thus solver calls) without affecting correctness.

Theorem 2 (Efficiency and sound guard reuse).
Let $D_0 \subseteq \cdots \subseteq D_n$ be the database sequence constructed by $\mathsf{LGX}(T, D_0, C)$. Then LGX performs at most as many oracle calls as LLMASP, and strictly fewer whenever some target predicate is not admissible during the extraction process. Moreover, guard evaluations can be safely reused: (i) if $D_i = D_j$ with $i < j$, then $D_j \models g$ if and only if $D_i \models g$; and (ii) if g is monotone and $D_i \models g$ for some $i \in \{0, \ldots, n-1\}$, then $D_j \models g$ for all $j \geq i$.

Proof (sketch).
Oracle call reduction is immediate: LLMASP invokes the oracle once for every target predicate in $\mathscr{P}_E$, whereas LGX invokes it only when the corresponding guards are satisfied. For guard reuse, (i) follows from the fact that conjunctive queries have the same truth value on identical databases, and (ii) is a consequence of the monotonicity of g and because $D_i \subseteq D_j$ by construction. □

The above theorem justifies execution strategies that cache guard outcomes according to these two principles. Such caching always agrees with fresh guard evaluation, and therefore preserves admissibility decisions and yields the same final database as the uncached semantics of LGX. Consequently, caching affects only efficiency, not correctness.

https://doi.org/10.1017/S1471068426100635 Published online by Cambridge University Press

Algorithm 1 LGX-EXEC(T, D_0,C): Logic-guided extraction with guard caching

Input : Text T, initial database D_0, configuration $C = (\langle p_1, \ldots, p_n \rangle, \{\mathscr{G}_{p_i}\}, \{\Pi_{p_i}\})$
Output: Final database D

$(D, \mathscr{K}_m^\top, \mathscr{K}_m^\bot, \mathscr{K}_n^\top, \mathscr{K}_n^\bot) := (D_0, \emptyset, \emptyset, \emptyset, \emptyset)$ // result and cache initialization
for $i := 1$ **to** n **do** // process target predicates
  $admissible := \top$
  foreach *guard* $g \in \mathscr{G}_{p_i}$ **do**
    if $g \in \mathscr{K}^\top_{mono(g)} \cup \mathscr{K}^\bot_{mono(g)}$ **then** // cache hit
      $hold := g \in \mathscr{K}^\top_{mono(g)}$ // retrieve answer from cache
    else // cache miss
      $hold := q \in Ans_{\{q \leftarrow g\}}(D)$ // answer query
      $\mathscr{K}^{hold}_{mono(g)} := \mathscr{K}^{hold}_{mono(g)} \cup \{g\}$ // cache answer
    if $hold = \bot$ **then** // if some guard is false
      $admissible := \bot$ // p_i is not admissible
  if *admissible* **then**
    $D' := Ans_{\Pi_{p_i}}(D \cup \mathscr{O}(T, p_i))$ // extract and derive facts
    if $D' \neq D$ **then** // if there are new facts
      $(D, \mathscr{K}_m^\bot, \mathscr{K}_n^\top, \mathscr{K}_n^\bot) := (D', \emptyset, \emptyset, \emptyset)$ // update DB, invalidate cache
return D

4 Implementation

This section presents a concrete realization of the logic-guided extraction framework defined in Section 3, given in Algorithm 1, and shows how its declarative semantics can be operationalized in practice. Implementation choices such as guard caching affect only efficiency and do not alter the formal semantics of the framework.

Guard evaluation. For a predicate p_i, admissibility is determined by evaluating the guards in $\mathscr{G}_{p_i}$ against the current database D. A guard g is evaluated by checking whether a fresh atom q is derivable from the program $\{q \leftarrow g\}$ together with the facts in D, that is, whether $q \in Ans_{\{q \leftarrow g\}}(D)$. Guard evaluation conceptually corresponds to the conjunction of the guards in $\mathscr{G}_{p_i}$. As soon as one guard is found to be false, the predicate p_i is deemed not admissible and no further guard evaluations are required. If all guards hold, p_i is admissible, the extraction oracle is invoked, and the facts returned by the oracle are combined with the current database and processed by the associated logic program Π_{p_i}.

Guard caches. In what follows, we use $\top$ and $\bot$ to denote the Boolean outcomes *true* and *false*. To optimize guard evaluation, the algorithm maintains a family of cache sets $\mathscr{K}^\tau_\mu$, where $\tau \in \{\top, \bot\}$ denotes the truth value of a guard and $\mu \in \{m, n\}$ denotes whether the guard is monotone or non-monotone. A guard g belongs to $\mathscr{K}^\tau_\mu$ if it has previously been evaluated with truth value τ and $mono(g) = \mu$, where *mono* determines the monotonicity

https://doi.org/10.1017/S1471068426100635 Published online by Cambridge University Press

of guards. Cached results are reused whenever possible. When the database is extended, cached results for non-monotone guards and cached negative results for monotone guards are invalidated, while cached positive results for monotone guards are preserved. This policy is sound by Theorem 2 and affects only the number of guard evaluations performed, without altering admissibility decisions or the final database produced.

5 Experimental evaluation

In this section, we empirically evaluate the LGX framework to assess its effectiveness in reducing LLM oracle calls, its impact on extraction quality, and the efficiency of the proposed caching strategies. Source files and datasets are available online at https://github.com/Xiro28/LGX.

Implementation and setup. From an implementation standpoint, we observe how the two cache sets $\mathscr{K}_m^{\perp}$ and $\mathscr{K}_n^{\perp}$ can be safely combined, since entries of both sets require re-evaluation once the database has changed. Our framework is implemented using CLINGO 5.8 by Gebser *et al*. (1911) as the underlying ASP solver, and META LLAMA 3.1 proposed by Meta (1924) as the extraction oracle, deployed in two configurations: a lightweight version (`8B`, hereafter referred to as *Small*) and a medium-capacity version (`70B`, referred to as *Medium*). Additionally, we performed robustness tests with a third high-capacity configuration based on GPT-OSS presented by OpenAI (1925), denoting its `120B` version as *Large*, to evaluate the consistency of the framework's performance across model architectures.

To ensure a fair and rigorous comparison, we simulated the LLMASP baseline using the LGX engine itself, adopting the degenerate configuration described in Section 3.4. This approach eliminates potential implementation biases, ensuring that performance deltas are solely attributable to the logic-guided strategy. Furthermore, we minimized stochastic variance by setting the LLM sampling temperature to 0 and employing a persistent cache for oracle calls. This guarantees that identical queries yield the same candidate facts across different system runs, effectively isolating the impact of the logic-guided control flow on both extraction efficiency and quality.

Benchmarks. Our evaluation relies on two primary benchmarks designed to assess different aspects of the logic-guided extraction.

Graph benchmark (G). Based on our running example of layered graphs, this benchmark compares LGX against the LLMASP baseline under two *guard enforcement* strategies (see Table 1): (i) *None*, representing a naïve, blind extraction; and (ii) *A posteriori*, where invalid results are pruned post-hoc using manually injected constraints.

Logic puzzle benchmark (LNRS). Adapted from the work by Alviano *et al*. (1925), this dataset aggregates four ASP domains (Labyrinth, Nomystery, Ricochet Robots and Sokoban) to evaluate mixed-domain handling. In this setting, an initial *domain-identifier* extraction acts as a hierarchical guard, restricting all subsequent oracle calls to the specific puzzle schema and preventing cross-domain noise.

https://doi.org/10.1017/S1471068426100635 Published online by Cambridge University Press

Table 1. *Experimental results on Graph (G) and Logic Puzzle (LNRS) benchmarks. We report the number of Oracle interactions (𝒪 Calls), the enforcement strategy, and accuracy metrics (F1-Score, Perfect Rate)*

Bench.	𝒪	Pipeline	𝒪 Calls	Guard enforcement	F1-score	Perfect rate
G	Small (8B)	LLMASP	240	None	0.800	0.317
				A posteriori	**0.893**	**0.617**
		LGX	**203**	By design	**0.893**	**0.617**
	Medium (70B)	LLMASP	240	None	0.870	0.417
				A posteriori	**0.952**	**0.850**
		LGX	**193**	By design	**0.952**	**0.850**
	Large (120B)	LLMASP	240	None	0.944	0.667
				A posteriori	**1.000**	**1.000**
		LGX	**186**	By design	**1.000**	**1.000**
LNRS	Small (8B)	LLMASP	2998	None	0.440	0.000
				A posteriori	**0.842**	**0.023**
		LGX	**876**	By design	**0.842**	**0.023**
	Medium (70B)	LLMASP	3072	None	0.572	0.000
				A posteriori	**0.963**	**0.469**
		LGX	**864**	By design	**0.963**	**0.469**
	Large (120B)	LLMASP	3072	None	0.632	0.000
				A posteriori	**0.947**	**0.555**
		LGX	**864**	By design	**0.947**	**0.555**

Performance and efficiency analysis. Table 1 summarizes the experimental results on the Graph (G) and Logic Puzzle (LNRS) benchmarks. The empirical data supports three key observations regarding the impact of LGX:

- **Drastic reduction in traffic.** LGX significantly lowers the number of interactions with the oracle compared to the baseline. In the LNRS benchmark, calls are reduced by approximately 72% (dropping from 3072 to 864 calls with the medium oracle). This confirms that the logic-guided pruning effectively identifies and discards inconsistent branches before expensive queries are made.
- **Accuracy equivalence.** Consistent with Theorem 1, the accuracy of LGX (enforcement by design) is identical to the *a posteriori* baseline. This empirically verifies that our pruning strategy is sound: it optimizes efficiency without compromising the quality of the solution.
- **Scalability with complexity.** The efficiency gap widens significantly as the logical structure deepens. While the reduction is moderate for the structurally simpler G benchmark (∼15–20%), it becomes substantial for the hierarchically structured LNRS benchmark. This confirms that LGX excels in rich domains, where guard dependencies enable aggressive pruning of oracle calls.

https://doi.org/10.1017/S1471068426100635 Published online by Cambridge University Press

Quality analysis (Benchmark G). Naïve extraction generates spurious facts, such as invalid layers not explicitly mentioned in the input text, or facts that are not expected to be extracted due to failing guards. These structural inconsistencies severely impact F1-score and Perfect Rate. While LLMASP can recover quality via *a posteriori* filtering, LGX ensures this consistency *by design*. This allows LGX to achieve optimal quality with strictly fewer oracle calls (Table 1), unlike the fixed-cost baseline.

Efficiency analysis (Benchmark LNRS). Efficiency is critical in the mixed-domain LNRS benchmark. While the baseline indiscriminately queries all predicates, incurring high costs, LGX matches the optimal accuracy of the *a posteriori* configuration at a fraction of the expense. By proactively pruning irrelevant queries rather than filtering results post-hoc, LGX avoids the prohibitive overhead of the standard pipeline.

ASP solver overhead. Our caching strategy significantly reduces internal reasoning costs. In LNRS, shared guards allow LGX to avoid between 61.4% and 82.6% of redundant ASP solver calls. Crucially, even where caching is less effective (Benchmark G), the latency introduced by guard checks is negligible compared to the massive runtime savings achieved by reducing oracle interactions.

6 Related work

Existing work on combining LLMs with logic-based reasoning can be categorized according to the role assigned to the symbolic component. In some approaches, logic is the target representation generated from text; in others, it is used as a validation layer, or as part of a probabilistic neuro-symbolic semantics. We discuss these lines of work with emphasis on their relation to extraction from natural language and to the use of ASP as a reasoning component.

Logic as a target language. A first class of approaches uses natural-language processing, and more recently LLMs, to construct a symbolic representation that is then given to a logic-based solver. This view extends earlier work in which natural-language content was encoded directly in ASP or related declarative languages. For example, Pendharkar *et al*. (1922) represent English passages through a Neo-Davidsonian formalism and use common-sense knowledge from WordNet, a lexical database presented by Miller (1995), to answer questions over the resulting program. With the advent of LLMs, part of the symbolic encoding can be delegated to neural generation. Ishay *et al*. (1923) introduce a Generate–Define–Test methodology to obtain entities, relations, and rules from text; Ishay and Lee (1925) extend this direction to action languages for temporal and causal reasoning. Similarly, the STAR framework of Rajasekharan *et al*. (1923) translates natural-language knowledge into a representation processed by s(CASP), supporting goal-directed and explainable derivations. These approaches differ from ours because the LLM is primarily used to construct the symbolic theory itself. In our setting, the symbolic schema and the control knowledge are fixed by the framework, while the LLM is used to produce candidate facts.

Logic-based extraction with fixed schemas. A second line of work assumes a fixed relational or logical schema and uses the LLM to populate it from natural language.

https://doi.org/10.1017/S1471068426100635 Published online by Cambridge University Press

The [LLM]+ASP framework of Yang *et al.* (1923) and LLMASP by Alviano *et al.* (1925) are representative of this direction and motivate the predicate-wise abstraction adopted earlier in the paper. Related ideas also appear in systems such as AutoCompanion by Zeng *et al.* (1924), where s(CASP) is used as a symbolic component in a dialogue setting. These approaches provide an important reference point for our work because they separate extraction from reasoning through a fixed schema. The distinction is that, in our framework, part of the symbolic reasoning is moved from post-processing to extraction control: logical conditions are evaluated during the construction of the database and can affect which extraction requests are issued.

Probabilistic neuro-symbolic reasoning. A distinct family of neuro-symbolic systems integrates neural predictions directly into a probabilistic logical semantics. DeepProbLog by Manhaeve *et al.* (1918) extends probabilistic logic programming with neural predicates, and NeurASP by Yang *et al.* (1920) combines ASP with probability distributions associated with atoms generated by neural classifiers. These frameworks provide a principled account of uncertain neural outputs and support forms of learning or probabilistic inference. Our setting is different in two respects. First, the LLM is treated as a pre-trained black-box oracle rather than as a trainable neural component integrated into the semantics. Second, extracted atoms are handled as crisp candidate facts, and the role of ASP is to derive consequences, check consistency, and control admissibility rather than to assign or combine probabilities.

Post-hoc validation and refinement. Several approaches use symbolic reasoning, execution feedback, or self-reflective procedures only after a neural model has generated an output. For instance, Eiter *et al.* (1923) use ASP to compute contrastive explanations for visual question-answering results. In the LLM literature, Self-Refine by Madaan *et al.* (1923) and Reflexion by Shinn *et al.* (1923) introduce iterative feedback loops in which generated answers are critiqued and revised. These methods can improve output quality by detecting or correcting problematic outputs after generation. The framework proposed here addresses a complementary problem: it uses logic before selected extraction calls are made, so that some irrelevant or inadmissible queries are never issued to the oracle.

7 Conclusion

We introduced a logic-guided data extraction framework that interleaves LLM-based extraction with Answer Set Programming, using logic-based admissibility conditions to actively control the extraction process. By exploiting logical dependencies during extraction, the proposed framework reduces the number of oracle invocations while preserving the final extracted information under mild assumptions. We formally characterized the framework, established its equivalence with a baseline extraction pipeline, and identified efficiency properties that enable practical optimizations. We presented an execution algorithm that realizes the proposed semantics and exploits guard caching based on monotonicity and unchanged database states. An experimental evaluation on ASP-derived benchmarks confirms that the framework substantially reduces the number of LLM calls and, in practice, mitigates the generation of spurious extracted facts.

https://doi.org/10.1017/S1471068426100635 Published online by Cambridge University Press

Several directions for future work remain. On the implementation side, guard monotonicity is currently treated as an external property provided by the configuration. An interesting extension would be to automatically infer monotonicity properties of guards, enabling more aggressive and transparent caching strategies without additional user annotation. More broadly, we plan to investigate dynamic guard evaluation strategies, incremental reasoning techniques, and tighter integrations between logic-based control and prompt construction. In particular, logic-driven templating approaches for prompt generation (such as the Mustache-based mechanisms adopted in ASP Chef by Alviano *et al.* (1925)) could be leveraged to dynamically adapt extraction prompts based on derived facts and admissibility conditions. These directions further strengthen the role of non-monotonic reasoning as a principled control mechanism for LLM-based data extraction. Furthermore, exploring the integration of goal-directed engines (e.g., PROLOG systems with incremental tabling) for guard evaluation presents a promising path toward more granular and reactive extraction strategies, effectively combining local dependency tracking with ASP global constraint-solving capabilities.

References

ALVIANO, M., GRILLO, L., LO SCUDO, F. and REINERS, L. A. R. 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*, ijcai.org, Montreal, Canada, 4330–4338.

ALVIANO, M., REINERS, L. A. R. and FABER, W. 2025. ASP chef grows mustache to look better. *Theory and Practice of Logic Programming* 25, 4, 417–436.

ARIAS, J., CARRO, M., CHEN, Z. and GUPTA, G. 2019. Constraint answer set programming without grounding and its applications. In *Datalog 2.0 2019 - 3rd International Workshop on the Resurgence of Datalog in Academia and Industry co-located with the 15th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR. 2019) at the Philadelphia Logic Week 2019*, CEUR Workshop Proceedings, Vol. 2368, June 4-5 1919, M. Alviano and A. Pieris, Eds. CEUR-WS.org, Philadelphia, PA, 22–26.

CALÌ, A., CALVANESE, D., GIACOMO, G. D. and LENZERINI, M. 2002. On the role of integrity constraints in data integration. *IEEE Data Engineering Bulletin* 25,3, 39–45.

EITER, T., GEIBINGER, T., HIGUERA, N. and OETSCH, J. 2023. A logic-based approach to contrastive explainability for neurosymbolic visual question answering. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence*, ijcai.org, 3668–3676.

EITER, T., GERMANO, S., IANNI, G., KAMINSKI, T., REDL, C., SCHÜLLER, P. and WEINZIERL, A. 2018. The DLVHEX system. *KI - Künstliche Intelligenz* 32, 2–3, 187–189.

FINK, M., GERMANO, S., IANNI, G., REDL, C. and SCHÜLLER, P. 2013. Acthex: Implementing HEX programs with action atoms. In *Logic Programming and Nonmonotonic Reasoning, 12th International Conference, LPNMR*, Proceedings-Lecture Notes in Computer Science, Vol. 8148, September 15-19 1913, P. Cabalar and T. C. Son, Eds. Springer, 317–322,

GEBSER, M., KAMINSKI, R. and SCHAUB, T. 2011. Complex optimization in answer set programming.

ISHAY, A. and LEE, J. 2025. LLM+AL: Bridging large language models and action languages for complex reasoning about actions. In *Proceedings of the AAAI Conference on Artificial Intelligence*, AAAI Press, 24212–24220.

ISHAY, A., YANG, Z. and LEE, J. 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.

https://doi.org/10.1017/S1471068426100635 Published online by Cambridge University Press

Jin, H., Zhang, Y., Meng, D., Wang, J. and Tan, J. 2024. A comprehensive survey on process-oriented automatic text summarization with exploration of llm-based methods.

Leone, N., Eiter, T., Faber, W., Fink, M., Gottlob, G., Granata, L., Greco, G., Kalka, E., Ianni, G., Lembo, D., Lenzerini, M., Lio, V., Nowicki, B., Rosati, R., Ruzzi, M., Staniszkis, W. and Terracina, G. 2005. Data integration: A challenging ASP application. In *LPNMR. Lecture Notes in Computer Science*, Vol. 3662, Springer, 379–383.

Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., Gupta, S., Majumder, B. P., Hermann, K., Welleck, S., Yazdanbakhsh, A. and Clark, P. 2023. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS*, M. Hardt, S. Levine, A. Oh, T. Naumann, A. Globerson and K. Saenko, Eds. December 10 - 16 1923, NeurIPS, New Orleans, LA, USA.

Manhaeve, R., Dumancic, S., Kimmig, A., Demeester, T. and Raedt, L. D. 2018. Deepproblog: Neural probabilistic logic programming. In *NeurIPS*, 3753–3763.

Marek, V. and Truszczyński, M. 1999. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: A 25-Year Perspective*, 375–398.

Meta. 2024. The llama 3 herd of models. *CoRR abs/2407.21783*.

Miller, G. A. 1995. Wordnet: A lexical database for English. *Communications of the ACM* 38, 11, 39–41.

Naveed, H., Khan, A. U., Qiu, S., Saqib, M., Anwar, S., Usman, M., Akhtar, N., Barnes, N. and Mian, A. 2025. A comprehensive overview of large language models.

Niemelä, I. 1999. Logic programming with stable model semantics as a constraint programming paradigm.

OpenAI. 2025. gpt-oss-120b & gpt-oss-20b model card.

Pendharkar, D., Basu, K., Shakerin, F. and Gupta, G. 2022. An ASP-based approach to answering natural language questions for texts. *Theory and Practice of Logic Programming* 22, 3, 419–443.

Rajasekharan, A., Zeng, Y., Padalkar, P. and Gupta, G. (2023) Reliable natural language understanding with large language models and answer set programming. In *ICLP Technical Communications. EPTCS*, Vol. 385, 274–287.

Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K. and Yao, S. (2023) Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems 36: Annual Conference On Neural Information Processing Systems 2023*, December 10 - 16 1923, M. Hardt, S. Levine, A. Oh, T. Naumann, A. Globerson and K. Saenko, Eds. NeurIPS, New Orleans, LA, USA.

Xu, D., Chen, W., Peng, W., Zhang, C., Xu, T., Zhao, X., Wu, X., Zheng, Y. and Chen, E. (2023). Large language models for generative information extraction: a survey. *CoRR abs/2312.17617*.

Yang, Z., Ishay, A. and Lee, J. 2020. Neurasp: Embracing neural networks into answer set programming. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI*, C. Bessiere, Ed. ijcai.org, 1755–1762.

Yang, Z., Ishay, A. and Lee, J. 2023. Coupling large language models with logic programming for robust and general reasoning from text. In *ACL (Findings). Association for Computational Linguistics*, A. Rogers, J. L. Boyd-Graber and N. Okazaki, Eds, 5186–5219.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K. R. and Cao, Y. 2023. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations, ICLR*, OpenReview.net, Kigali, Rwanda.

https://doi.org/10.1017/S1471068426100635 Published online by Cambridge University Press

Zeng, Y., Rajasekharan, A., Basu, K., Wang, H., Arias, J. and Gupta, G. 2024. A reliable common-sense reasoning socialbot built using LLMs and goal-directed ASP. *Theory and Practice of Logic Programming* 24, 4, 606–627.

Zhang, W., Li, X., Deng, Y., Bing, L. and Lam, W. 2023. A survey on aspect-based sentiment analysis: Tasks, methods, and challenges.

Zhu, Y., Yuan, H., Wang, S., Liu, J., Liu, W., Deng, C., Chen, H., Liu, Z., Dou, Z. and Wen, J.-R. 2025. Large language models for information retrieval: A survey.

https://doi.org/10.1017/S1471068426100635 Published online by Cambridge University Press