Embedding Safe AI Assistants into Web Apps with LLMASP

Mario Alviano, Matteo Capalbo*, Georg Gottlob and Irfan Kareem

Abstract

We present a framework for embedding safe, Large Language Model (LLM)-driven assistants into web applications by separating the agent's reasoning from the user interface it acts upon. The interface is exposed to the agent as an abstract page state (a structured, machine-readable view of the designated UI elements) and the agent in turn expresses its intentions exclusively as ground Datalog facts, never as direct Document Object Model (DOM) operations. A pipeline of three components connects the two: an extractor that turns a natural-language request into facts via LLMASP; a validator that admits a fact set only if it is syntactically well-formed, typed, and logically consistent (under the stable-model semantics) with both a schema-derived knowledge base and a context-dependent layer that catches hallucinations disconnected from the user's input; and a planner that turns admitted facts into a sequence of DOM mutations gated by an explicit safety flag distinguishing autonomous operations from those that require user confirmation. We instantiate the framework on a tabular-data domain through a lightweight `data-ai-*` HTML5 annotation protocol, and discuss its portability: re-targeting the assistant to a different web stack, interaction medium, or domain reduces to replacing a scanner, an executor, and a behaviour specification, while the validator and the planner remain untouched. The separation turns LLM stochasticity into a feature: the agent is free to rephrase and generalise intent, while every effect on the world must first survive a symbolic check.

Keywords

Neuro-Symbolic AI, Answer Set Programming, Large Language Models, Intelligent Web Agents, Datalog Fact Generation, Logic-based Validation

1. Introduction

The integration of Large Language Models (LLMs) into interactive software has paved the way for *agentic* AI systems capable of executing tasks on behalf of users. However, deploying LLM agents over standard web interfaces remains a significant challenge. Raw Document Object Model (DOM) trees are notoriously noisy, unstructured, and saturated with layout details that distract the model. More critically, allowing a stochastic LLM to autonomously deduce and trigger actions (such as clicking buttons, submitting forms, or deleting records) poses severe safety and reliability risks. Without strict deterministic boundaries, agents are prone to hallucinations, potentially invoking non-existent UI elements or hallucinating invalid data values.

To address these challenges, we propose a Neuro-Symbolic (NeSy) framework for embedding safe AI assistants into web applications. The core intuition of our approach is to strictly decouple the web interface from the agent's reasoning process. Instead of forcing the LLM to parse raw HTML or write arbitrary execution code, we require page authors to explicitly declare the interface's affordances using a lightweight protocol of custom HTML5 `data-ai-*` attributes. These annotations are scanned to construct an *abstract page state*, which serves as the sole source of truth for the LLM. Furthermore, the agent is restricted to expressing its intentions exclusively as ground Datalog facts. This decoupling turns the stochasticity of the LLM into a feature rather than a liability: the agent is free to flexibly interpret, rephrase, and generalise the user's natural-language intent, but its output must survive a rigorous, deterministic symbolic check before any action is executed. To enforce this, we introduce a *multi-level validation architecture* built on top of LLMASP. Procedural layers handle syntax errors and perform fuzzy matching on categorical values, while an ASP solver performs symbolic reasoning

CILC 2026: The 41st Italian Conference on Computational Logic, June 23–25, 2026, Ferrara, Italy

mario.alviano@unical.it (M. Alviano); matteo.capalbo@student.tuwien.ac.at,matteo.capalbo@unical.it (M. Capalbo); georg.gottlob@unical.it (G. Gottlob); irfan.kareem@unical.it (I. Kareem)

0000-0002-2052-2063 (M. Alviano); 0009-0001-0114-9304 (M. Capalbo); 0000-0002-2353-5230 (G. Gottlob); 0000-0002-9762-3954 (I. Kareem)

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

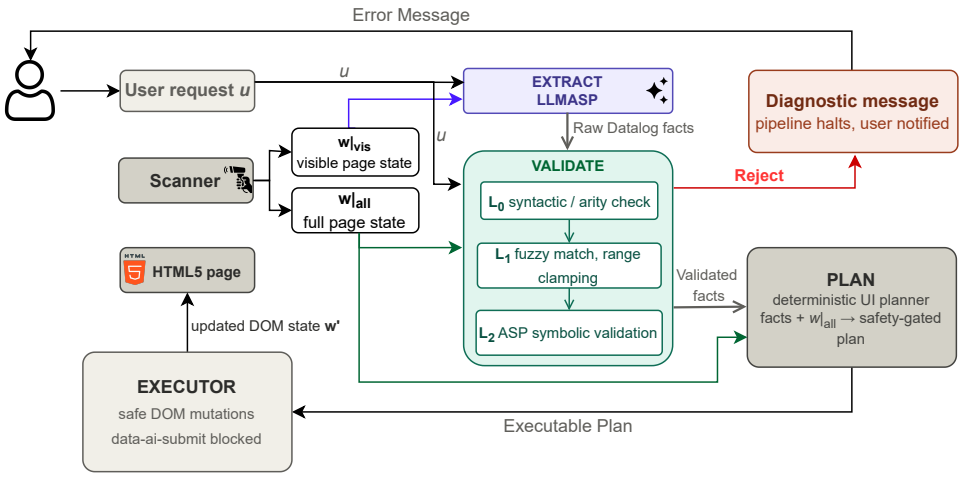

Figure 1: Three-stage pipeline of the framework. Extract converts the user request u and the visible page state $w|_{vis}$ into a fact set via LLMASP; Validate applies *L0*, *L1*, *L2* checks against a schema-derived KB and a context block, rejecting unsafe facts with a diagnostic message; Plan produces a safety-gated sequence of actions. The Executor (right) belongs to the concrete instantiation of Section 4 and applies the plan to yield the updated DOM state w'.

against a dynamically constructed knowledge base. This two-tier approach catches both schema-level incompatibilities (e.g., filtering by a non-existent column) and pragmatic hallucinations (e.g., inferring filter values not mentioned in the user's text or unavailable in the current UI state).

Consider a user interacting with an e-commerce dashboard who types: *"Find me blue cashmere coats under €1500 for a business meeting"*, as shown in Figure 2. In our framework, the LLM processes this request alongside the abstract page state and generates declarative facts such as `filter_by("color", "eq", "BLUE")`. Before interacting with the UI, the validation architecture checks these facts. The procedural layer discards syntax errors and auto-corrects string values (e.g., normalising "blue" to "BLUE"). The semantic validation layer then evaluates the facts against an automatically generated knowledge base, verifying that referenced columns exist, that categorical values are supported, and critically, that inferred filter tokens are grounded in the user's original request. For instance, if the LLM over-interprets "for a business meeting" and emits `filter_by("occasion", "eq", "business")` when no "occasion" column is exposed, this hallucination is caught and rejected. Only fully validated intents are passed to a planner, which safely sequences the required DOM mutations while respecting safety constraints (e.g., never autonomously triggering irreversible operations like deletions or form submissions).

While the underlying framework and validation pipeline are entirely domain-agnostic, we prove its viability by presenting a concrete instantiation tailored for tabular-data applications. The instantiation introduces a declarative annotation protocol, a three-level validation cascade (syntactic, procedural, symbolic), and an *open-and-fill* execution discipline that separates interface preparation from user-confirmed commitment.

In summary, this paper contributes an end-to-end, closed-loop framework for LLM-web interaction (see Figure 1). We introduce a declarative `data-ai-*` annotation protocol to securely map UI constraints, backed by a three-level syntactic, procedural, and ASP-based symbolic validation architecture designed to eliminate contextual hallucinations. These concepts are validated through a concrete HTML5 instantiation for tabular domains that ensures irreversible actions are safely gated and executed.

Related Work. Recent NLP advances through transformer-based LLMs like GPT, LLAMA, Mistral, and Gemma have achieved unprecedented success in text generation, summarization, and sentiment analysis [1, 2, 3, 4, 5, 6]. However, LLMs struggle with complex reasoning tasks such as logic, causality, and probabilistic inference [7], motivating NeuroSymbolic (NeSy) AI that combines LLMs with symbolic reasoning frameworks like Answer Set Programming (ASP) [8, 9, 10]. Recent NeSy approaches integrate

LLMs with ASP through multiple strategies: prompt engineering for incremental ASP conversion [11], ASP rule generation from natural language [12], and story-based question answering [13]. The LLMASP framework [14] exemplifies this integration, using structured formats like YAML to encode domain knowledge, converting natural language to ASP facts for reasoning, and translating results back to natural language [15, 16]. Agentic AI extends LLMs by enabling tool invocation and state management. Recent systems include RAG-based e-commerce platforms [17, 18] and multilingual agent evaluation datasets for service-calling tasks [19]. Unlike RAG-driven systems lacking correctness guarantees and prior NeSy approaches applying symbolic reasoning in task-specific pipelines [20, 21, 22, 23], our framework introduces a two-tier symbolic validator: schema-level structural constraints and context-dependent semantic hallucination detection.

2. Background

Large Language Models. Large Language Models (LLMs) are neural sequence models trained on large corpora of text, capable of generating fluent natural-language output conditioned on an input prompt. LLMs excel at understanding and rephrasing natural-language requests, but are stochastic by nature: the same input may produce different outputs across invocations, and the model may generate plausible-looking but factually incorrect content (hallucinations). This makes LLMs unsuitable as direct executors of structured operations, but powerful as translators of natural-language intent into structured representations, provided that a deterministic verification layer is placed downstream. We abstract LLMs as functions from Σ^* (prompt) to Σ^* (generated text), where Σ^* is the set of finite strings over a fixed alphabet Σ.

Answer Set Programming. Answer Set Programming (ASP) is a declarative logic programming paradigm suited for knowledge representation and constraint-based reasoning. An ASP program consists of a set of definite clauses $\mathcal{R}$ (encoding domain rules) and a set of integrity constraints $\mathcal{IC}$ (denial rules of the form $\leftarrow$ *body*, which forbid certain combinations of facts). We refer to the pair $\mathcal{K} = \langle \mathcal{R}, \mathcal{IC} \rangle$ as a *knowledge base*. Given a finite set of predicate symbols with associated arities $\Pi = \{p_1/n_1, \ldots, p_k/n_k\}$ and a finite set of constants $\mathcal{C}$, the Herbrand base over Π and $\mathcal{C}$ is the set of all ground atoms:

$$\mathbb{F}_{\Pi,\mathcal{C}} = \{\, p_i(c_1, \ldots, c_{n_i}) \mid p_i/n_i \in \Pi,\ c_j \in \mathcal{C} \,\}. \tag{1}$$

A *fact set* F is any finite subset of $\mathbb{F}_{\Pi,\mathcal{C}}$. The logical consistency of F with respect to a knowledge base $\mathcal{K}$ is determined by checking whether the program $F \cup \mathcal{K}$ is satisfiable (*sat*) under the stable-model semantics; otherwise, the program is unsatisfiable (UNSAT).

LLMASP. LLMASP is a neuro-symbolic framework that bridges natural language and ASP. It operates by prompting an LLM according to a *behaviour specification* $\mathcal{B} = \langle init, ctx, map \rangle$, where $init$ fixes the system prompt and the agent's role, ctx provides background context, and map provides per-predicate templates instructing the model how to translate natural-language statements into ground facts for each predicate $p_i \in \Pi$. The output of LLMASP is a fact set $F \subseteq \mathbb{F}_{\Pi,\mathcal{C}}$, which can then be reasoned upon by an ASP solver.

The Document Object Model. The Document Object Model (DOM) is the standard tree-based representation of a web document maintained by the browser. Each node in the tree represents a structural element of the page (e.g., a button, an input field, a panel) and carries a set of *attributes*, which are key-value pairs encoding properties such as type, identifier, or current value. The browser exposes primitives to programmatically interact with DOM nodes. For example, dispatching a `click` event on a node triggers its associated handler, or filling a text field with a specific value. Let $\mathcal{W}$ (for *worlds*) denote the set of all possible DOM states.

3. Framework

We present a framework for enabling an AI assistant to interact with a web application on behalf of a user. The framework is organised as a three-stage pipeline (Figure 1):

$$\textsc{Extract}: \quad \mathcal{W} \times \Sigma^* \to \mathbb{F}_{\Pi,\mathcal{C}} \tag{2}$$

$$\textsc{Validate}: \mathbb{F}_{\Pi,\mathcal{C}} \times \mathcal{W} \times \Sigma^* \to \mathbb{F}_{\Pi,\mathcal{C}} \cup \Sigma^* \tag{3}$$

$$\textsc{Plan}: \mathbb{F}_{\Pi,\mathcal{C}} \times \mathcal{W} \quad \to (\mathcal{W} \to \mathcal{W})^* \tag{4}$$

where $(\mathcal{W} \to \mathcal{W})^*$ is the set of *plans* (i.e., finite sequences of functions from $\mathcal{W}$ to $\mathcal{W}$). Given a natural-language user request and the current DOM state, the pipeline first applies Extract to produce the user's intent as a structured fact set, then Validate to clean it, and finally Plan to produce a concrete sequence of actions to be executed on the page. The framework is agnostic to any particular rendering technology, domain vocabulary, or UI toolkit. Concrete implementation choices are deferred to Section 4.

Fields, Actions, and Page State. The full DOM tree is too verbose for reliable reasoning: it contains deep hierarchies of layout and styling nodes irrelevant to interaction. We therefore assume that a subset of DOM nodes is *designated* as meaningful to the agent, each annotated with a natural-language description of its role. Designated nodes are of two kinds. A *field* is a designated DOM node that holds a value the agent can read or write. Each field carries a unique identifier, an element type drawn from a fixed set of primitives (e.g., text, number, categorical), and an admissible domain (either a finite set of valid values or a numeric interval). An *action* is a designated DOM node that the agent can trigger. Each action carries a unique identifier and is associated with a total function $a\colon \mathcal{W} \to \mathcal{W}$. Triggering an action transitions the current DOM state $w \in \mathcal{W}$ to a successor state $a(w)$. The *page state* of w, denoted $w|_{all}$, is the collection of all designated fields and actions. We partition $w|_{all}$ into two disjoint subsets: $w|_{vis}$, containing elements rendered on screen, and $w|_{hid}$, containing hidden elements (e.g., inside closed panels or modals).

Intent Extraction. Given a user request $u \in \Sigma^*$ and the visible page state $w|_{vis}$, Extract builds a behaviour specification $\mathcal{B}$ from the descriptions of the designated nodes in $w|_{vis}$ and a set of *intent type* predicates (e.g., filtering, sorting, inserting), and invokes LLMASP to produce a raw fact set representing the user's intent. Constants are drawn from $w|_{vis}$ and u. Let $w|^{u}_{vis}$ denote $\textsc{Extract}(w|_{vis}, u)$.

Sanitisation and Validation. Validate checks and cleans the raw fact set $w|^{u}_{vis}$, which may contain malformed, inconsistent, or hallucinated facts. Validation is driven by page state and user request, and is guaranteed to return a safe fact set to plan over (or a diagnostic human-interpretable message in case the pipeline must be interrupted). Let $w|^{u*}_{vis}$ be short for $\textsc{Validate}(w|^{u}_{vis}, w|_{all}, u)$.

Planning. A validated fact set $w|^{u*}_{vis}$ expresses *what* the user wants. Plan determines *how* to achieve it by mapping $w|^{u*}_{vis}$ and $w|_{all}$ to a concrete *plan* $[a_1, \ldots, a_n]$: a finite, ordered sequence of actions drawn from those available in $w|_{all}$. Executing the plan transitions the DOM through the sequence of states $w \xrightarrow{a_1} w_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} w_n$. The planner may reference $w|_{hid}$ to schedule navigation actions (e.g., opening a panel) that must precede interactions with currently hidden fields.

4. Tabular-data Domain Instantiation

This section instantiates the abstract pipeline of Section 3 (Extract, Validate, Plan) on a concrete HTML5 application targeting a tabular-data domain. The UI exposes a product table with a toolbar and a collapsible *filter panel* containing per-column filter fields; opening or closing this panel controls which fields lie in $w|_{vis}$ or $w|_{hid}$. Throughout we use as running example the user query *"Find me blue*

Table 1
The `data-ai-*` annotation vocabulary used by the scanner to construct $w|_{all}$. `data-ai-context` groups elements into panels or forms; `data-ai-safe` is enforced by the executor at plan time; and `data-ai-submit` marks commit operations the agent may never trigger autonomously.

Attribute	Component	Domain
`data-ai-field`	field identifier	Σ^*
`data-ai-type`	element type	$\mathcal{T}$
`data-ai-options`	categorical domain	comma-separated list
`data-ai-range`	numeric domain	"`min-max`"
`data-ai-description`	natural-language hint	Σ^*
`data-ai-action`	action identifier	Σ^*
`data-ai-safe`	safety flag	{`true`, `false`}
`data-ai-context`	enclosing panel/form	Σ^*
`data-ai-submit`	user-only commit action	Σ^*
`data-ai-option`	option in multi-value controls	Σ^*
`data-ai-guard`	action precondition	Σ^*
`data-ai-param`	optional action parameter	Σ^*
`data-ai-sort-state`	current sort state	{`asc`, `desc`, `none`}

Listing 1: Visible scan $w|_{vis}$ at the moment the running query is issued. The filter panel is closed, so no per-column filter field appears in $w|_{vis}$; those fields belong to $w|_{hid}$.

```
=== UI CONTEXT: main-table-view ===
AVAILABLE ACTIONS:
  [action: navigate-products]   "Switch to table Products"
  [action: open-filters]        "Opens the filter panel"
  [action: clear-filters]       "Removes all active filters (requires user confirmation)"
  [action: open-add-row]        "Opens the add-row form (does not submit)"
FILLABLE FIELDS:
  [field: global_search | type: text]
    description: "Full-text search across all columns"
=== END UI CONTEXT ===
```

cashmere coats under €1500 for a business meeting", issued in a session where the filter panel is initially closed.

Designating Fields and Actions. We start from the abstract notion of *designated nodes* from Section 3 and show how page authors declare them in HTML. A lightweight vocabulary of custom `data-ai-*` attributes (see Table 1) is added so that a scanner can process the DOM to construct $w|_{all}$. The scanner performs two traversals: a *visible scan* capturing what the user currently sees ($w|_{vis}$, see Listing 1), and a *full scan* collecting all annotated elements regardless of visibility ($w|_{all}$). The visible scan is serialised as a text block to feed Extract, while the full scan provides the richer schema details (e.g., concrete enumerations in `data-ai-options`) needed later by Validate. In the running example, the DOM elements shown in Listing 2 contribute an action `open-filters` and a numeric field `price_max` in the `filter-panel` context, since the panel is closed at query time, both elements reside in $w|_{hid}$, not in $w|_{vis}$. The context of each element is determined as the nearest `data-ai-context` ancestor.

Extract Phase. We instantiate the abstract Extract : $\mathcal{W} \times \Sigma^* \to F_{\Pi,\mathcal{C}}$ on our tabular domain. The intent predicate signature Π_{tab} includes data-oriented predicates (*filter_by*/3, *sort_by*/2, *aggregate*/2) and UI-level intents (*open_form*/1, *navigate_to*/1). At this stage, Extract only asserts *what* should happen, not *how* it is realised on the UI. At request time, the UI metadata (columns, data types, and admissible operators) is automatically serialised into a static Datalog knowledge base $\mathcal{K}$. LLMASP receives a behaviour specification containing instruction templates, the visible page state $w|_{vis}$, and the

Listing 2: Annotated DOM fragment exposing `open-filters`, `price_max`, and the `filter-panel` submit control.

```
<button data-ai-action="open-filters" data-ai-safe="true"
        data-ai-description="Opens the filter panel">Filters</button>

<aside data-ai-context="filter-panel">
  <input data-ai-field="price_max" data-ai-type="number"
         data-ai-range="830-5200">
  <button data-ai-submit="filter-panel"
          data-ai-description="Apply filters">Apply Filters</button>
</aside>
```

Listing 3: Simplified excerpt of the ASP validation program. C_1 enforces structural schema validity; C_2 enforces contextual anchoring to the user request to catch hallucinations.

```
% Schema-level rules from 𝒦
valid_column("category"). valid_column("price"). % ...
column_type("category", "categorical").

:- filter_by(C, _, _), not valid_column(C). % C1

% Context-level rules from 𝒦 and φ
token_anchored(C, O, T) :-
    extracted_filter_token(C, O, T), user_token(T).

:- extracted_filter_value(C, O, _), column_type(C, "categorical"),
   filter_arity(C, O, N), N > 0,
   #count { T : token_anchored(C, O, T) } < N. % C2
```

user request u. On the running query, the extractor emits the raw fact set $w|^{u}_{vis}$:

```
filter_by("category", "eq", "Coats").    filter_by("material", "contains", "cashmere").
filter_by("color",    "eq", "Blue").        filter_by("price",    "lte", 1500).
```

The extractor translates and normalises the user's wording into schema-level constants (e.g. *coats* is mapped to `Coats`, and *blue* to `Blue`). Conversely, the phrase *for a business meeting* produces no fact, because the schema exposes no column for occasion.

Validate Phase. The raw fact set $w|^{u}_{vis}$ undergoes a three-level cascade $L_0 \to L_1 \to L_2$. Each level acts as a gate: a fact set that fails at L_i is not forwarded; instead, Validate returns a human-interpretable diagnostic string $m \in \Sigma^*$. Only a fact set that clears all three levels is returned as the sanitised intent $w|^{u*}_{vis}$ and forwarded to Plan. L_0 performs syntactic validation, discarding malformed text or wrong-arity facts. L_1 performs type-directed procedural validation. Categorical values are fuzzily matched against admissible schema options (e.g., normalising spelling), while numeric filters are checked against exposed ranges. For our running query, `Coats`, `Blue`, and `1500` are all verified as procedurally admissible. L_2 performs symbolic validation via an ASP solver on $w|^{u}_{vis} \cup \mathcal{K} \cup \varphi$, where $\mathcal{K}$ provides the static schema rules and φ provides a dynamic context block containing tokens extracted from the user request and active filters. An excerpt of $\mathcal{K} \cup \varphi$ is shown in Listing 3. Contextual validation (C_2 in Listing 3) relies on lexical anchoring to catch semantic errors that structural schema validation alone cannot detect. Suppose the extractor over-interprets the phrase *"for a business meeting"* and emits `filter_by("category", "eq", "Suits")`. Although `Suits` is a valid category in the database and passes C_1, the tokens `suits` do not appear in the user request u. Thus, C_2 renders the program unsatisfiable, and L_2 rejects the hallucination. For our actual running query, all constraints are satisfied, yielding the validated intent $w|^{u*}_{vis} = w|^{u}_{vis}$.

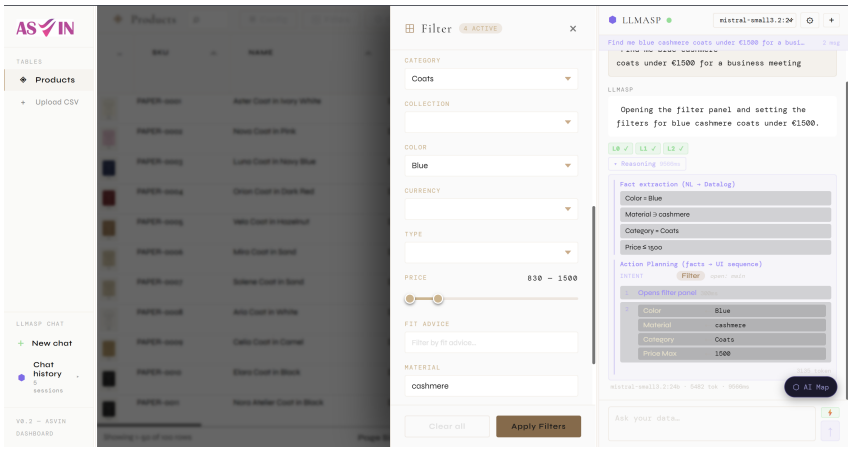

Figure 2: ASVIN dashboard responding to *"Find me blue cashmere coats under €1500 for a business meeting"*. The LLMASP panel shows extracted facts, L_0–L_2 validation, and the resulting filter plan.

Plan Phase. The planner uses LLMs to determine *how* the validated intent $w|_{vis}^{u*}$ is realised. A plan is made of steps $s_i = \langle \alpha_i, \phi_i, \delta_i \rangle$, comprising an optional action identifier α_i ($\perp$ for no action), a dictionary of field assignments ϕ_i, and a DOM-settling delay δ_i. In the running example, the filter panel is initially closed, so the planner must schedule opening it before any filter field can be written (those fields reside in $w|_{hid}$ until s_1 completes):

$$
\begin{aligned}
s_1 &= \langle \texttt{open-filters},\ \emptyset,\ 250 \rangle, \\
s_2 &= \langle \perp,\ \{\texttt{category} \mapsto \texttt{Coats},\ \texttt{color} \mapsto \texttt{Blue},\ \texttt{material} \mapsto \texttt{cashmere}, \\
&\qquad \texttt{price_max} \mapsto 1500\},\ 0 \rangle.
\end{aligned}
$$

Note that in s_2 `filter_by("price", "lte", 1500)` in $w|_{vis}^{u*}$ maps to the upper-bound field `price_max` (see Listing 2), illustrating how the planner bridges abstract intent predicates to concrete UI fields. Execution is guarded at the DOM level. The executor clicks `open-filters` autonomously (it carries `data-ai-safe="true"`), waits 250 milliseconds for the UI to settle, then writes the values in s_2 to the corresponding `data-ai-field` elements and dispatches browser events. Crucially, the executor does *not* click the Apply Filters button, because it carries `data-ai-submit` and is therefore an irreversible, user-only action. This realises the intended *open-and-fill* discipline: the agent prepares the interface state safely, and the verbalisation stage prompts the user to confirm submission.

5. Conclusions and Future Work

In this paper, we presented a neuro-symbolic framework for embedding safe, LLM-driven assistants into web applications. By decoupling the interface from the agent's reasoning process through a lightweight HTML5 annotation protocol (`data-ai-*`), we restrict the agent to expressing its intentions exclusively as declarative Datalog facts. Our three-stage pipeline (Extract, Validate, Plan) ensures that the stochastic nature of LLMs is rigorously constrained. Crucially, the multi-level ASP-based validation layer guarantees that all proposed UI actions are syntactically correct, schema-compliant, and contextually anchored, effectively neutralising hallucinations before they reach the execution phase. Furthermore, by structurally blocking `data-ai-submit` operations, the framework enforces an open-and-fill discipline that keeps irreversible actions safely under user control.

Future work will focus on extending the extraction and validation logic beyond tabular domains to support more complex, nested web interfaces. Additionally, we plan to investigate multi-turn dialogue management within the LLMASP framework to handle follow-up constraints, and to conduct comprehensive empirical evaluations on real-world web applications to assess the system's extraction accuracy and end-to-end latency.

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 under project NextGenGuides "Innovazione Tecnologica per le Guide Turistiche del Futuro tramite l'ausilio di tecnologie innovative AI e sistemi di geolocalizzazione avanzata" (CUP J39I24002040005); under project 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); 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).

Declaration on Generative AI

During the preparation of this work, the authors used GPT-5.4 for grammar and spelling check. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] H. Jin, Y. Zhang, D. Meng, J. Wang, J. Tan, A comprehensive survey on process-oriented automatic text summarization with exploration of llm-based methods, CoRR abs/2403.02901 (2024). doi:10.48550/ARXIV.2403.02901. arXiv:2403.02901.

[2] W. Zhang, X. Li, Y. Deng, L. Bing, W. Lam, A survey on aspect-based sentiment analysis: Tasks, methods, and challenges, IEEE Trans. Knowl. Data Eng. 35 (2023) 11019–11038. doi:10.1109/TKDE.2022.3230975.

[3] T. B. Brown, et al., Language models are few-shot learners, in: H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, H. Lin (Eds.), NeurIPS 2020, December 6-12, 2020, virtual, 2020. URL: https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html.

[4] H. Touvron, et al., Llama: Open and efficient foundation language models, CoRR abs/2302.13971 (2023). doi:10.48550/ARXIV.2302.13971. arXiv:2302.13971.

[5] M. AI, Ministral 3, CoRR abs/2601.08584 (2026).

[6] G. Team, T. Mesnard, C. Hardin, R. Dadashi, S. Bhupatiraju, S. Pathak, L. Sifre, M. Rivière, M. S. Kale, J. Love, et al., Gemma: Open models based on gemini research and technology, arXiv preprint arXiv:2403.08295 (2024).

[7] M. I. Nye, M. H. Tessler, J. B. Tenenbaum, B. M. Lake, Improving coherence and consistency in neural sequence models with dual-system, neuro-symbolic reasoning, in: NeurIPS, 2021, pp. 25192–25204.

[8] 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.

[9] I. Niemelä, Logic programming with stable model semantics as a constraint programming paradigm, Annals of Mathematics and Artificial Intelligence 25 (1999) 241–273. doi:10.1023/A:1018930122475.

[10] M. Gelfond, V. Lifschitz, Logic programs with classical negation, in: D. Warren, P. Szeredi (Eds.), Logic Programming: Proc. of the Seventh International Conference, 1990, pp. 579–597.

[11] A. Ishay, Z. Yang, J. Lee, Leveraging large language models to generate answer set programs, in: KR, 2023, pp. 374–383.

[12] M. A. B. Santana, I. Kareem, F. Ricca, Towards automatic composition of ASP programs from natural language specifications, in: IJCAI, ijcai.org, 2024, pp. 6198–6206.
[13] I. Kareem, K. Gallagher, M. A. Borroto, F. Ricca, A. Russo, Using learning from answer sets for robust question answering with LLM, in: LPNMR, Lecture Notes in Computer Science, Springer, 2024, pp. 112–125.
[14] M. Alviano, L. Grillo, F. L. Scudo, L. A. R. Reiners, Integrating answer set programming and large language models for enhanced structured representation of complex knowledge in natural language, in: IJCAI, ijcai.org, 2025, pp. 4330–4338.
[15] M. Alviano, L. Grillo, Answer set programming and large language models interaction with YAML: preliminary report, in: CILC, CEUR Workshop Proceedings, CEUR-WS.org, 2024.
[16] M. Alviano, F. L. Scudo, L. Grillo, L. A. R. Reiners, Answer set programming and large language models interaction with YAML: second report, in: KoDis+CAKR+SYNERGY@KR, CEUR Workshop Proceedings, CEUR-WS.org, 2024.
[17] K. Rodrigues, G. Dsouza, O. Phansopkar, H. Siddiqui, V. Kushwaha, V. Hole, Outfitai: Novel architecture to leverage generative ai for immersive fashion e-commerce solutions, in: 2025 International Conference on Applications of Machine Intelligence and Data Analytics (ICAMIDA), IEEE, 2025, pp. 1–6.
[18] P. J. V. Vicente, J. S. Castejón-Garrido, C. Caparrós-Laiz, R. Pan, J. A. García-Díaz, R. Valencia-García, Retaillm: A multilingual, optimised llm-based chatbot system for retail management, in: SEPLN-PD, CEUR Workshop Proceedings, CEUR-WS.org, 2025, pp. 91–97.
[19] A. Maronikolakis, A. Peleteiro-Ramallo, W. Cheng, T. Kober, What should I wear to a party in a greek taverna? evaluation for conversational agents in the fashion domain, CoRR abs/2408.08907 (2024).
[20] Z. Wang, J. Liu, Q. Bao, H. Rong, J. Zhang, Chatlogic: Integrating logic programming with large language models for multi-step reasoning, in: 2024 International Joint Conference on Neural Networks (IJCNN), IEEE, 2024, pp. 1–8.
[21] L. Pan, A. Albalak, X. Wang, W. Wang, Logic-lm: Empowering large language models with symbolic solvers for faithful logical reasoning, in: Findings of the Association for Computational Linguistics: EMNLP 2023, 2023, pp. 3806–3824.
[22] Y. Su, K. Xu, Y. Gao, F. Yang, C. Li, M. Yang, T. Xu, Neuro-symbolic verification on instruction following of llms, arXiv preprint arXiv:2601.17789 (2026).
[23] Y. Feng, N. Weir, K. Bostrom, S. Bayless, D. Cassel, S. Chaudhary, B. Kiesl-Reiter, H. Rangwala, Vericot: Neuro-symbolic chain-of-thought validation via logical consistency checks, CoRR abs/2511.04662 (2025).