

Embedding Safe AI Assistants into Web Apps with LLMASP: An Experimental Evaluation

Mario Alviano, Matteo Capalbo*, Georg Gottlob, Irfan Kareem and Luis Angel Rodriguez Reiners

Abstract

We present a framework for embedding safe, LLM-driven assistants into web applications by decoupling 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 UI elements, while the agent expresses its intentions exclusively as ground Datalog facts, never as direct DOM operations. A three-stage pipeline bridges the two: an extractor that converts a natural-language request into facts via LLMASP; a validator that accepts a fact set only if it is syntactically well-formed, type-correct, and logically consistent under stable-model semantics with both a schema-derived knowledge base and a context-dependent layer detecting hallucinations unsupported by input; and a planner that turns admitted facts into a sequence of DOM mutations guarded by a safety flag that separates autonomous actions from those requiring user confirmation. We instantiate the framework in a tabular-data setting using a lightweight `data-ai-*` HTML5 annotation protocol, and we discuss its portability: adapting the assistant to a different web stack, interaction medium, or domain reduces to replacing the scanner, the executor, and the behavior specification, while the validator and planner remain unchanged. This separation turns LLM stochasticity into an asset: the agent may rephrase and generalize intent, but every effect on the world must first pass a symbolic check. We empirically validate the framework on 13 open-weight LLMs (8B–8×22B) using a 39-query benchmark over a real fashion-retail catalogue. Extraction quality varies with model choice, with F1 ranging from 0.30 to 0.92, but the symbolic validator rejects all curated hallucinations (10/10) at a median cost of 23 ms, and no model, regardless of size, produces an executable unsafe action. These results show that model choice affects quality and cost, while safety remains an architectural guarantee independent of the LLM.

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) framework¹ 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

6th International Workshop on the Resurgence of Datalog in Academia and Industry (Datalog 2.0 2026), September 7, Klagenfurt, Austria

✉ 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); luis.reiners@unical.it (L. A. Rodriguez Reiners)

🆔 0000-0002-2052-2063 (M. Alviano); 0009-0001-0114-9304 (M. Capalbo); 0000-0002-2353-5230 (G. Gottlob); 0000-0002-9762-3954 (I. Kareem); 0009-0000-1808-9910 (L. A. Rodriguez Reiners)



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

¹A preliminary version of our framework was presented at CILC 2026 [1].

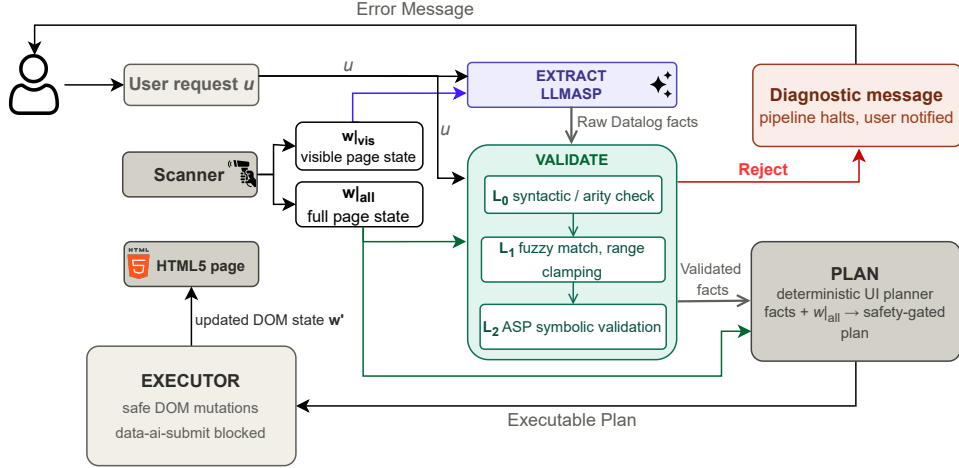


Figure 1: Three-stage pipeline of the framework. Extract converts the user request u and the visible page state $w|_{vis}$ into a fact set via LLMASP; Validate applies L_0 , L_1 , L_2 checks against a schema-derived KB and a context block, rejecting unsafe facts with a diagnostic message; Plan produces 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' .

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 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 domain-agnostic, we demonstrate its viability through a concrete HTML5 instantiation for tabular-data applications. This instantiation introduces a declarative `data-ai-*` annotation protocol, a three-level syntactic, procedural, and ASP-based symbolic validation cascade, and an *open-and-fill* execution discipline that separates interface preparation from user-confirmed commitment. The scanner never exposes the full DOM to the LLM: annotations let it collect only the relevant interface elements into a compact, page-specific prompt, and they declare which actions and fields exist without fixing how they are composed, leaving the agent free to sequence the interaction steps needed for a given request, much as a human user does on a page whose controls are already known. We validate the framework on a real fashion-retail catalogue using 13 open-weight LLMs and a 39-query benchmark, showing that extraction quality varies substantially

with model choice, while the symbolic validator rejects all curated hallucinations and prevents unsafe executable actions. In summary, this paper contributes an end-to-end, closed-loop framework for LLM-web interaction, as shown in Figure 1, ensuring that irreversible actions are safely gated before execution.

1.1. Related Work

Recent advances in large language models (LLMs) have enabled strong performance in generation and language understanding tasks, but reasoning over logic-heavy or safety-critical settings remains challenging, motivating neuro-symbolic approaches that combine LLMs with symbolic formalisms such as Answer Set Programming (ASP) [2, 3, 4, 5]. Several recent works explore this integration through prompt-based ASP translation, rule generation from natural language, and LLMASP-style pipelines that map text to facts, reason symbolically, and verbalize results [6, 7, 8, 9].

Agentic web systems extend LLMs with tool use and state management, with benchmarks and systems such as Mind2Web, AutoWebGLM, AgentOccam, WebShop, and WebMall pushing progress on web navigation and shopping tasks [10, 11, 12, 13, 14]. However, these approaches still rely mainly on data-driven policies, benchmark performance, or representation alignment, without explicit symbolic validation of extracted intent or UI actions. In contrast, our framework inserts a symbolic verification layer into the agent loop, enforcing structural and semantic correctness before execution.

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 $\Pi = \{p_1/n_1, \dots, p_k/n_k\}$ and a finite set of constants $\mathcal{C}$, the Herbrand base over Π and $\mathcal{C}$ is the set of all ground atoms:

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

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 [9] 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):

$$\text{EXTRACT: } \mathcal{W} \times \Sigma^* \rightarrow \mathbb{F}_{\Pi, \mathcal{C}} \quad (2)$$

$$\text{VALIDATE: } \mathbb{F}_{\Pi, \mathcal{C}} \times \mathcal{W} \times \Sigma^* \rightarrow \mathbb{F}_{\Pi, \mathcal{C}} \cup \Sigma^* \quad (3)$$

$$\text{PLAN: } \mathbb{F}_{\Pi, \mathcal{C}} \times \mathcal{W} \rightarrow (\mathcal{W} \rightarrow \mathcal{W})^* \quad (4)$$

where $(\mathcal{W} \rightarrow \mathcal{W})^*$ 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 $a: \mathcal{W} \rightarrow \mathcal{W}$. 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).

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

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 ===

```

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 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.

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>
```

EXTRACT Phase. We instantiate the abstract $\text{EXTRACT} : \mathcal{W} \times \Sigma^* \rightarrow F_{\Pi, \mathcal{C}}$ 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 $\mathcal{K}$. LLASP receives a behaviour specification containing instruction templates, the visible page state $w|_{vis}$, and the user request u . On the running query, the extractor emits the raw fact set $w|_{vis}^u$:

```
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 $w|_{vis}^u$ undergoes a three-level cascade $L_0 \rightarrow L_1 \rightarrow L_2$. 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 $w|_{vis}^{u*}$ 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 $w|_{vis}^u \cup \mathcal{K} \cup \varphi$, 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. Here, a *token* denotes a domain-specific lexical unit derived from the catalogue’s controlled vocabulary of filter values, not a sub-word unit produced by the LLM’s own tokenizer. Three predicates carry tokens through validation: *user_token*/1 holds the normalised content n-grams of the user request, *extracted_filter_token*/4 holds the tokens of each extracted filter value, and *active_filter_token*/2 holds those of the filters already active in the session; the normalisation ν (case/diacritic folding, light IT/EN stemming, a small bounded synonym lexicon), is the only domain-specific ingredient and is itself declarative and auditable.

The context block φ mirrors each emitted *filter_by* fact as *extracted_filter_value*(C, O, V) and splits its value V into the tokens that compose it: *extracted_filter_token*(C, O, V, T) holds for each such token T , and *filter_arity*(C, O, V, N) records their number N . A token is *anchored* (the derived predicate *token_anchored*(C, O, V, T)) when it also occurs in the user request (*user_token*(T)) or in a filter already active in the session (*active_filter_token*(C, T)). Tokens are compared modulo a deterministic normalisation $\nu = \text{fold} \circ \text{stem} \circ \text{syn}$: case and diacritic folding, a light Italian/English inflectional stemmer (so *bianche* anchors the schema value BIANCO), and closure under a small, bounded lexicon of genuine lexical synonyms (so *vestiti* anchors Abiti). Crucially, the lexicon encodes lexical synonymy only, *not* world-knowledge inference, so the boundary of what may anchor a value is itself

Listing 3: Simplified excerpt of the ASP validation program. C_1 enforces structural schema validity; the value-scoped C_2 requires each extracted filter value to be anchored (by at least one token) in the user request or the active session filters.

```
% Schema-level rules from  $\mathcal{K}$ 
valid_column("category"). valid_column("price"). % ...
column_type("category", "categorical").

:- filter_by(C, _, _), not valid_column(C). %  $C_1$ 

% Context-level rules from  $\mathcal{K}$  and  $\varphi$  (value-scoped)
token_anchored(C, O, V, T) :-
    extracted_filter_token(C, O, V, T), user_token(T).
token_anchored(C, O, V, T) :-
    extracted_filter_token(C, O, V, T), active_filter_token(C, T).

:- extracted_filter_value(C, O, V), column_type(C, "categorical"),
    filter_arity(C, O, V, N), N > 0,
    #count { T : token_anchored(C, O, V, T) } < 1. %  $C_2$ 
```

declarative and auditable. Constraint C_2 makes the program unsatisfiable for any categorical value that has at least one token ($N > 0$) yet none anchored; requiring a single anchored token is enough to admit legitimate single-token references to multi-word catalogue values, while still rejecting values invented wholesale from context. Text columns are handled analogously. 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 $w_{vis}^{u*} = w_{vis}^u$. The instantiated knowledge base $\mathcal{K} \cup \varphi$ is not the excerpt shown in Listing 3. It comprises 69 schema-derived facts (12 valid columns, 15 valid fields, 15 column-type declarations, 19 valid operator-per-type facts, 1 aggregate target, 2 sort directions, and 5 aggregate functions), 7 derivation rules, and 11 integrity constraints: 9 schema-level constraints (the C_1 – C_6 families), plus two contextual anchoring constraints, IC-S1 for categorical and multi-value columns and IC-S2 for text columns. None of this is hand-written: the whole knowledge base is generated from the table schema by a compiler, and the context block φ is regenerated per request from the user input and the active session state; porting the validator to a new table or domain means regenerating facts, not rewriting code. The program is entirely stratified, with negation and the single `#count` aggregate used by IC-S1/IC-S2 applied only over predicates computed in an earlier stratum, so a stratified-Datalog fixpoint procedure suffices in principle; ASP (via Clingo) is used instead for the convenience of native negation-as-failure, aggregate literals, and `#show` directives that project answer sets onto the predicates of interest for the surrounding LLMASP pipeline. ASP is used here as a global consistency checker rather than to explore multiple stable models: the ground program $w_{vis}^u \cup \mathcal{K} \cup \varphi$ admits at most one answer set or none, so `VALIDATE` accepts or rejects the extracted fact set as a whole rather than repairing individual facts, ensuring that a single hallucinated fact cannot be silently dropped while the rest of the request is executed. Expressing this check declaratively, rather than as procedural code, keeps the schema-level facts and the contextual anchoring rules auditable and extensible as a single artifact; future work will associate choice rules with extracted facts and weak constraints with preference criteria, letting the solver favour fact subsets that retain the most validated information while respecting all hard safety constraints.

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

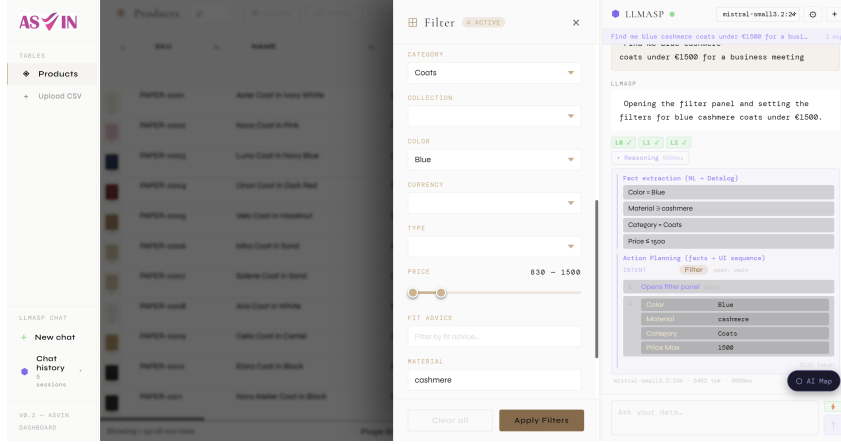


Figure 2: ASVIN dashboard responding to “Find me blue cashmere coats under €1500 for a business meeting”. The LLMASP panel shows extracted facts, L_0 – L_2 validation, and the resulting filter plan.

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):

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

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. The planner that follows symbolic validation is thus guarded structurally rather than by the model’s own judgement: `data-ai-safe` gates which actions the executor may click autonomously, `data-ai-submit` elements can never be clicked by the agent under any circumstance, and a deterministic reference plan is used as a fallback whenever the LLM’s output is unusable.

5. Empirical Evaluation

Our framework uses an LLM for intent extraction and action planning, while symbolic reasoning ensures correctness and safety. We evaluate this design along four questions: (RQ1) How accurately do LLMs from different families and sizes extract intent facts through the pipeline?, (RQ2) Does the symbolic validator reject genuine hallucinations while preserving legitimate requests?, (RQ3) What is the per-interaction cost in tokens, latency, and energy, and how much of that cost is due to symbolic checking?, (RQ4) How faithful and how safe is LLM-driven planning compared with a deterministic reference? All RQs are evaluated on the end-to-end `EXTRACT` $\rightarrow$ `VALIDATE` $\rightarrow$ `PLAN` pipeline running on a shared server with three AMD Instinct MI210 GPUs, recording prompt, answer, and estimated reasoning-trace tokens, wall-clock latency, and GPU energy for each stage.

The UI schema and catalogue are derived from a deterministically anonymised version of a European fashion retailer’s product catalogue (5,446 items, 20 columns), preserving all fields, i.e.(column names, types, categorical domains, numeric ranges) required for symbolic validation. We built a benchmark of 39 Italian natural-language queries covering the data-oriented fragment of Π_{tab} (*filter_by*/3, *sort_by*/2,

Table 2

Extraction F1 (set-based, vs. gold facts) per intent and overall; mean over 39×3 runs per model, $\pm$ std across runs. Models sorted by (quantised) size; † marks models that emit a reasoning trace.

Model	filter	sort	aggr.	add	delete	remove	overall F1
llama3.1:8b	0.26	0.45	0.11	0.70	0.67	0.31	0.30 ± 0.30
gemma3:12b	0.70	1.00	0.90	0.83	1.00	0.50	0.77 ± 0.34
ministral-3:14b	0.80	1.00	0.50	1.00	1.00	0.50	0.79 ± 0.36
gpt-oss:20b †	0.86	0.92	0.98	1.00	1.00	0.50	0.88 ± 0.27
mistral-small3.2:24b	0.77	1.00	0.80	1.00	1.00	0.50	0.81 ± 0.34
gemma3:27b	0.58	1.00	0.80	0.89	1.00	0.50	0.69 ± 0.39
gemma4:31b †	0.87	1.00	1.00	1.00	1.00	1.00	0.92 ± 0.21
deepseek-r1:32b †	0.80	1.00	0.80	1.00	1.00	0.83	0.84 ± 0.31
qwen3.5:35b †	0.87	1.00	1.00	1.00	1.00	0.83	0.91 ± 0.21
llama3.3:70b	0.44	1.00	0.90	1.00	1.00	1.00	0.65 ± 0.40
gpt-oss:120b †	0.91	1.00	1.00	1.00	1.00	0.50	0.92 ± 0.21
qwen3.5:122b †	0.84	1.00	0.93	1.00	1.00	0.67	0.87 ± 0.27
mixtral:8x22b	0.59	1.00	0.67	1.00	1.00	0.50	0.68 ± 0.40

aggregate/2, add_value/2, delete_row/2, remove_filter/1). The benchmark is divided into two families, and every extracted fact was manually verified against a symbolic gold set. Family A contains 25 legitimate requests covering all six intents, each with a gold fact set; three queries additionally include an active-filter session state. Family B contains 14 adversarial requests annotated with the inconsistency they encode, divided into traps that a sound validator must reject (occasion-to-category inference, unmentioned colours or materials, non-existent columns, type-incompatible operators, invalid sort directions and aggregate targets, and out-of-range numerics) and negative controls that a sound validator must not reject (legitimate session refinements, single-token anchoring of multi-word schema values, and synonym/morphology variants). We evaluated 13 instruction-tuned open-weight LLMs from six families (Llama, Gemma, Mistral, Qwen, DeepSeek, GPT-OSS, spanning (8B–8×22B parameters) served locally with Ollama at temperature 0. Intent extraction was repeated three times per query to account for residual non-determinism (1,521 runs), while planning was evaluated once on the 25 legitimate requests (325 runs).

5.1. Extraction Quality Across Models

Table 2 reports set-based extraction F1 against the gold facts, per intent and overall; the large $\pm$ reflects the per-query variance of set-based F1, not measurement noise. The results yield three main insights. First, most mid-size models are already strong extractors: every model from 12B upwards reaches overall F1 between 0.65 and 0.92, the best being gemma4:31b and gpt-oss:120b (0.92) and qwen3.5:35b (0.91). Second, performance is not strictly scale-driven: llama3.3:70b (0.65) and mixtral:8x22b (0.68) are outperformed by gemma3:12b (0.77), indicating family and training matter more than parameter count. In contrast, llama3.1:8b drops to 0.30, representing the lower bound of the approach. Third, the per-intent breakdown highlights the difficulty: single-fact intents (*sort_by*, *add_value*, *delete_row*) are solved nearly perfectly by all models ≥ 12 B, whereas *filter_by*, which composes several facts and normalises wording to schema constants, shows widest variance (0.26–0.91) and is the main quality discriminator. Finally, *remove_filter* errors (0.50 for most models) stem from a convention mismatch: models alternate between explicitly naming the filtered column and emitting the “clear-all” sentinel “_all”, reflecting an interface-specification issue rather than a failure in language understanding.

5.2. Hallucination Rejection

We evaluate the validator from two angles: a controlled-injection ablation that isolates it from model noise, and the behaviour observed on the facts the models actually produced.

Table 3

Cumulative validator ablation on the ten curated hallucination sets and the 39 legitimate gold sets; deterministic, no LLM in the loop. Only contextual anchoring catches the schema-valid hallucinations; the two residual false rejects are the morphology/synonym cases left to future work.

Configuration	Hallucination detection	False rejects (legit)
L_0 only (syntactic)	0/10	0/39
+ L_1 (procedural, fuzzy+clamp)	0/10	0/39
+ L_2 schema-only (C_1 – C_6)	4/10	0/39
+ L_2 full (contextual anchoring)	10/10	0/39

Controlled injection. For each of the ten annotated traps we push the gold facts *plus* the curated hallucination facts through the L_0 – L_2 cascade (no LLM involved), under four cumulative configurations (Table 3). Syntactic and procedural checks catch nothing, every injected fact is well-formed and type-correct. Schema-level symbolic validation (C_1 – C_6 of Listing 3) catches exactly the four schema-*invalid* traps (non-existent column, wrong operator, invalid direction, invalid aggregate target). The six schema-*valid* hallucinations (e.g. “meeting di lavoro” $\rightarrow$ filter_by("category", "eq", "Completi e tailleur")) are caught *only* by the contextual anchoring layer, which reaches full 10/10 detection. The layer reaches full 10/10 detection with *zero* false rejects over the 39 legitimate gold sets, and all 49 benchmark expectations are matched. This includes every negative control: session refinements, single-token anchoring of multi-word values, and the synonym/morphology variants admitted by the normalisation ν (“vestiti” $\rightarrow$ Abiti, “bianche” $\rightarrow$ BIANCO) are all accepted, while their inferential look-alikes (“meeting di lavoro” $\rightarrow$ a category) are still rejected: the anchoring separates lexical variation from genuine inference.

Behaviour on real model output. Every model hallucinates under adversarial pressure, from 10% of runs (gpt-oss:20b) to 98% (llama3.1:8b), so the phenomenon the validator targets is real across the whole quality spectrum, not an artefact of weak models. Where a model’s spurious facts happen to be schema-valid, enabling the contextual layer raises their rejection substantially (e.g. from 0.20 to 0.70 for deepseek-r1:32b) mirroring the controlled ablation. Crucially, the validator rarely rejects correct output: on runs with a fully correct extraction the false-reject rate stays at 0–2% for eleven of thirteen models. The two exceptions are benign: gemma3:12b’s rejects trace to a quoting deviation (aggregate(count, ...) instead of the string "count" the interface contract requires) that a more lenient L_0 could repair, and llama3.1:8b has only four fully-correct extractions to begin with.

5.3. Cost: Tokens, Latency, and Energy

The cost structure is highly asymmetric. Table 4 reports per-query cost of the extraction stage alongside F1. The prompt remains nearly constant ($\approx$ 3.3–3.5k tokens, dominated by the serialized schema context), while the *visible* output is minimal (19–48 tokens), as validated Datalog facts form a compact representation. The main source of variation is *reasoning overhead*: models with traces consume an additional 180–1,850 tokens per query, increasing median latency from 1–4 s (non-reasoners) to 11–26 s and energy from 0.3–1.8 kJ to 5.8–12.4 kJ. Despite this overhead, performance gains remain limited. Model gpt-oss:120b matches gemma4:31b in F1 (0.92) while using roughly $6\times$ less energy, and the best non-reasoning model (mistral-small3.2:24b) remains within 0.11 F1 of the best overall model while consuming about $22\times$ less energy. This suggests that fact extraction over an explicit schema is a constrained translation task, where extended deliberation yields diminishing returns. Finally, the symbolic stage is negligible in the end-to-end pipeline. Across 1,521 runs, the L_2 ASP solve—grounding the schema knowledge base, context program, and extracted facts, and checking satisfiability—takes a median of 23 ms (90th percentile 31 ms), accounting for only 0.1–2% of the total latency of the fastest model. Thus, the correctness guarantee comes at negligible runtime cost compared to a single LLM call.

Table 4

Per-query cost of EXTRACT. Prompt and generated tokens (generated = visible answer + estimated reasoning trace); median latency and median GPU energy per query (all three cards, idle baseline 121 W). F1 repeated from Table 2 for reference.

Model	tok _{prompt}	tok _{gen}	latency (s)	energy (J)	F1
llama3.1:8b	3,328	48	0.97	317	0.30
gemma3:12b	3,369	19	2.36	827	0.77
ministral-3:14b	3,489	22	1.11	369	0.79
gpt-oss:20b [†]	3,603	338	3.57	1,847	0.88
mistral-small3.2:24b	3,489	25	1.57	559	0.81
gemma3:27b	3,369	21	4.26	1,540	0.69
gemma4:31b [†]	3,377	987	21.6	12,380	0.92
deepseek-r1:32b [†]	3,692	356	11.4	6,862	0.84
qwen3.5:35b [†]	3,699	1,857	26.4	5,821	0.91
llama3.3:70b	3,328	21	3.38	1,389	0.65
gpt-oss:120b [†]	3,486	203	4.15	2,101	0.92
qwen3.5:122b [†]	3,899	942	26.3	11,937	0.87
mixtral:8x22b	3,891	21	4.23	1,364	0.68

Table 5

LLM action planning vs. the deterministic gold plan on the 25 legitimate queries. *plan match* = same action sequence *and* same set of filled fields; *stripped/q* = mean per-query steps referencing unsafe or non-existent actions, all removed by the gate (hence never executed); *fallback* = share of queries where the LLM output was unusable and the deterministic plan was used instead.

Model	plan match	action seq.	fill keys	stripped/q	fallback	(s)
llama3.1:8b	0.44	0.52	0.72	0.00	0.00	1.3
gemma3:12b	0.80	0.96	0.80	0.04	0.00	7.4
ministral-3:14b	0.76	0.96	0.76	0.04	0.00	2.4
gpt-oss:20b [†]	0.68	0.92	0.76	0.00	0.05	5.2
mistral-small3.2:24b	0.80	0.96	0.80	0.04	0.00	3.7
gemma3:27b	0.84	0.96	0.88	0.04	0.00	6.1
gemma4:31b [†]	0.72	0.88	0.80	0.00	0.10	35.7
deepseek-r1:32b [†]	0.72	0.88	0.80	0.00	0.10	16.1
qwen3.5:35b [†]	0.68	0.92	0.76	0.00	0.05	16.5
llama3.3:70b	0.84	0.96	0.84	0.04	0.00	8.0
gpt-oss:120b [†]	0.68	0.92	0.76	0.00	0.05	8.6
qwen3.5:122b [†]	0.72	0.96	0.76	0.04	0.00	44.0
mixtral:8x22b	0.80	0.92	0.80	0.00	0.00	9.6

5.4. Planning Fidelity and Safety

The planner of Section 4 is a hybrid: a deterministic translation of the validated facts into a structured intent, an LLM sequencer that turns the intent and the page capabilities into concrete steps, and a safety gate that filters the result. The deterministic path doubles as a free *gold plan*: for each legitimate query we feed the gold facts to both paths against a frozen page-capability fixture and compare (Table 5).

Two findings stand out. First, *the safety invariant holds identically* across all 325 plans: no model—including the weakest—produces an executable unsafe action. Rare unsafe or non-existent steps (≤ 0.04 per query) and occasional fallback cases ($\leq 10\%$) are fully absorbed by the safety gate and recovery mechanism. This confirms that the *open-and-fill* discipline is architectural: model scale affects plan quality, not safety. Second, plan *content* quality degrades smoothly and is not improved by reasoning. Exact-match scores range from 0.68 to 0.84 for models ≥ 12 B, with most errors in fill-key granularity while action sequences remain correct in 0.88–0.96 of cases. Reasoning models incur 16–44 s latency without outperforming gemma3:27b (0.84 at 6.1 s), indicating that constrained sequencing over

fixed capabilities does not benefit from extended deliberation.

5.5. Discussion and Threats to Validity

Taken together, the four experiments support the central design claim: model choice affects *quality* and *cost*, but not *safety*. For this workload, a mid-size non-reasoning model (14–24B), achieving within 0.11 F1 of the best model at roughly $20\times$ lower energy with identical safety. Several limitations qualify these findings. The evaluation is limited to a single tabular domain, one language, and 39 curated queries, prioritising internal validity over breadth. The synonym lexicon behind ν is small and hand-curated for the catalogue taxonomy: porting to a new domain or language requires regenerating it, and enlarging it trades paraphrase recall against the risk of reintroducing inference-like mappings. The per-model hallucination counts rely on a beyond-gold proxy that over-counts benign anchored over-specifications, which is why the controlled injection is reported separately as ground truth. Ollama generation is not perfectly deterministic at temperature 0; we mitigate with three repetitions and report dispersion. Finally, energy is measured on a shared multi-GPU host: we report per-card counter deltas with a recorded idle baseline and medians, but co-location noise cannot be fully excluded. Reasoning-trace tokens are estimated from character lengths, and all results use Ollama’s quantised models, so absolute latency and energy may differ from full-precision deployments.

6. 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-*`), the agent is restricted to expressing its intentions exclusively as declarative Datalog facts. Our three-stage pipeline (EXTRACT, VALIDATE, PLAN) constrains the stochastic nature of LLMs through a multi-level ASP-based validation layer that ensures proposed UI actions are syntactically correct, schema-compliant, and contextually anchored, effectively neutralising hallucinations before execution. In addition, by structurally blocking `data-ai-submit` operations, the framework enforces an open-and-fill discipline that keeps irreversible actions under user control. Our evaluation over 13 open-weight LLMs corroborates the design: extraction quality and cost vary widely with the model (F1 0.30–0.92, per-query energy 317 J–12.4 kJ), while every curated hallucination is rejected, no unsafe step is ever executable, and the symbolic guarantee costs a median of 23 ms per interaction.

Future work will extend the extraction and validation logic beyond tabular domains to more complex nested web interfaces. We also plan to investigate multi-turn dialogue management within the LLASP framework for handling follow-up constraints, and to broaden the anchoring lexicon beyond the single Italian catalogue studied here, extending the evaluation to further domains, languages, and live deployments.

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).

Data and Code Availability

The full knowledge base, driver code, anonymised catalogue, benchmark queries, and evaluation scripts are available at GitHub https://github.com/IrfanKareem/asvin_ui_datalog2.0.git.

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] M. Alviano, M. Capalbo, G. Gottlob, I. Kareem, Embedding safe AI assistants into web apps with LLMASP, in: A. Bertagnon, D. Azzolini, F. Riguzzi, M. Gavanelli, M. Vespa (Eds.), Proceedings of the 41st Italian Conference on Computational Logic, Ferrara, Italy, June 23-25, 2026, CEUR Workshop Proceedings, CEUR-WS.org, 2026.
- [2] 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.
- [3] 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.
- [4] 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.
- [5] 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.
- [6] A. Ishay, Z. Yang, J. Lee, Leveraging large language models to generate answer set programs, in: KR, 2023, pp. 374–383.
- [7] 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.
- [8] 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.
- [9] 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.
- [10] X. Deng, Y. Gu, B. Zheng, S. Chen, S. Stevens, B. Wang, H. Sun, Y. Su, Mind2web: Towards a generalist agent for the web, in: NeurIPS, 2023.
- [11] H. Lai, X. Liu, I. L. Iong, S. Yao, Y. Chen, P. Shen, H. Yu, H. Zhang, X. Zhang, Y. Dong, J. Tang, Autowebglm: A large language model-based web navigating agent, in: KDD, ACM, 2024, pp. 5295–5306.
- [12] K. Yang, Y. Liu, S. Chaudhary, R. Fakoor, P. Chaudhari, G. Karypis, H. Rangwala, Agentoccam: A simple yet strong baseline for llm-based web agents, in: ICLR, OpenReview.net, 2025.
- [13] S. Yao, H. Chen, J. Yang, K. Narasimhan, Webshop: Towards scalable real-world web interaction with grounded language agents, in: NeurIPS, 2022.

- [14] R. Peeters, A. Steiner, L. Schwarz, J. Y. Caspary, C. Bizer, Webmall - A multi-shop benchmark for evaluating web agents, CoRR abs/2508.13024 (2025).