Declarative Communication Filters for Controlled LLM Integration in Logic-Based Multi-Agent Systems

Lorenzo De Lauretis[1], Stefania Costantini[1]

[1]DISIM — Department of Information Engineering, Computer Science and Mathematics, University of L'Aquila, L'Aquila, Italy

Abstract

The integration of Large Language Models (LLMs) into multi-agent systems promises to enhance agent decision-making with flexible natural-language reasoning, yet raises concerns about controllability, predictability, and safety. We present a principled approach to this problem implemented in DALI2, a logic-based multi-agent framework running on SWI-Prolog. Our key insight is that the *tell/told* communication filters—originally designed in the DALI language for inter-agent message filtering—can be uniformly extended to govern agent–LLM interactions. Tell rules declaratively constrain what queries an agent may send to an LLM oracle; told rules constrain which responses the agent will accept, with support for pattern matching, state-dependent conditions, and priority-based ordering. The mechanism is fully declarative, bidirectional, and operates within the standard agent reasoning cycle without architectural changes. We formalise the filtering protocol, evaluate it on three test scenarios (smart agriculture, emergency response, and a state-dependent suite), and discuss its properties in comparison with existing LLM guardrail approaches.

Keywords

multi-agent systems, large language models, communication filters, logic programming, agent-oriented programming, DALI

1. Introduction

Large Language Models (LLMs) such as GPT-4 [1] have demonstrated remarkable capabilities in natural-language understanding, reasoning, and generation. Their integration into autonomous agent systems is an active area of research, with recent proposals for LLM-based agents [2, 3, 4] and tool-using language models [5, 6]. However, the vast majority of these approaches treat the LLM as the *core* reasoning engine, delegating control flow, planning, and decision-making to the language model itself.

In logic-based multi-agent systems (MAS), agents are instead governed by *declarative rules* with well-defined semantics: reactive rules, goals, beliefs, constraints, and communication protocols. Frameworks such as DALI [7, 8], Jason/AgentSpeak [9], GOAL [10], and 2APL [11] provide formally grounded agent architectures within the Belief–Desire–Intention (BDI) paradigm [12]. These systems are deployed in domains where *verifiability*, *auditability*, and *predictable behaviour* are essential—e.g., critical infrastructure monitoring, healthcare protocols, and autonomous robotics—because their semantics allow formal reasoning about agent properties. Integrating LLMs into such systems could extend their capabilities with flexible natural-language understanding, commonsense reasoning, and the ability to interpret unstructured information, while retaining the logical guarantees of the host architecture. However, for this integration to be sound, the LLM should serve as an *auxiliary resource*—an oracle that enriches the agent's reasoning without subverting its logical control.

The challenge is then: *how can a logic-based agent interact with an LLM while maintaining declarative control over what it asks and what it accepts?* Existing guardrail approaches [13, 14] operate at the infrastructure level, external to the agent's reasoning cycle, and are not expressed in the agent's own declarative language.

In this paper we propose a solution implemented in DALI2 [15], a modern reimplementation of the DALI agent language on SWI-Prolog [16]. Our key insight is that DALI's *tell/told* communication filters—originally designed to control inter-agent message exchange [7]—can be uniformly extended

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

lorenzo.delauretis@univaq.it (L. D. Lauretis); stefania.costantini@univaq.it (S. Costantini)

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

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

published 2026-11-03

CEUR-WS.org/Vol-4311/RCRA_regular_06.pdf

to govern agent–LLM interactions. The same declarative rules that specify which messages an agent may send (`tell`) and receive (`told`) are reused, without modification, to filter queries to and responses from an LLM oracle.

Contributions. The main contributions of this paper are as follows, and the remainder of the paper is organised accordingly.

1. An architecture for integrating LLMs as oracles within the reasoning cycle of logic-based agents (Section 3).
2. A declarative, bidirectional filtering mechanism that reuses existing tell/told communication filters for LLM interaction control, with formal characterisation (Section 4).
3. A quantitative experimental evaluation on three scenarios using GPT-4o, demonstrating 100% filter accuracy, negligible computational overhead, and emergent protective behaviour of the whitelist model (Sections 5–6).

2. Background

2.1. The DALI Agent Language

DALI (Dynamic Active Logic Interaction) [7, 8] is a logic-based agent-oriented programming language originally implemented on SICStus Prolog. Agents are defined by sets of reactive rules, internal events, goals, beliefs, past events, and communication filters. Key syntactic constructs include:

- **Reactive rules** (`eventE(X) :> Body`): respond to external events (messages, sensor data).
- **Internal events** (`eventI :> Body` + `internal_event/5`): proactive, periodic, or condition-triggered reasoning.
- **Condition-action rules** (`Cond :< Action`): edge-triggered state transitions.
- **Goals** (`obt_goal/test_goal`): BDI-style goal-directed behaviour with plan execution.
- **Tell/told filters**: declarative rules controlling outgoing (`tell`) and incoming (`told`) messages.

2.2. DALI2: Architecture Overview

DALI2 [15] reimplements DALI on SWI-Prolog with identical syntax and a modern infrastructure. Each agent runs as a separate OS process; inter-agent communication uses Redis pub/sub [17] in a star topology. A Prolog meta-interpreter implements a 13-phase reasoning cycle per agent (message processing, internal events, condition monitors, goal pursuit, constraint checking, past-event management, etc.). DALI2 preserves all original DALI constructs and adds new features including periodic tasks, condition monitors, ontological reasoning, learning rules, and the AI Oracle described in this paper.

2.3. Tell/Told Communication Filters

In DALI, *told rules* specify which incoming messages an agent is willing to accept, optionally with a priority and a body condition. *Tell rules* specify which outgoing messages an agent is allowed to send. Both mechanisms were originally described in DALI's `communication.con` configuration and operate as declarative guards on the agent's communication channel.

Listing 1: Tell/told filter syntax in DALI2.

```
% Told: accept pattern with priority and condition
told(_, Pattern, Priority) :- Condition.
told(_, Pattern). % no priority (default 0), unconditional

% Tell: allow sending pattern if condition holds
tell(_, _, Pattern) :- Condition.
tell(_, _, Pattern). % unconditional
```

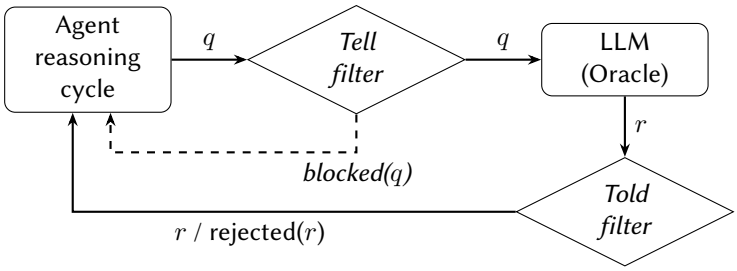


Figure 1: AI Oracle query protocol. The agent's query q passes through the tell filter; the LLM response r passes through the told filter. Rejected queries return `blocked(`q`)`; rejected responses return `rejected(`r`)`.

When an agent has told rules, only messages matching at least one pattern *and* whose body condition succeeds are accepted; others are rejected. When told rules assign priorities, incoming messages are sorted in decreasing priority order before processing, implementing a *priority queue*. Conditions may reference the agent's current beliefs (`believes/1`), past events (`has_past/1`), learned knowledge (`learned/2`), or any Prolog predicate, enabling *state-dependent* filtering. Tell rules operate symmetrically on outgoing messages.

3. The AI Oracle Architecture

The *AI Oracle* is a component that allows any DALI2 agent to query an LLM during rule execution. It is accessed through the body predicate `ask_ai/2` (or `ask_ai/3` with a custom system prompt). The oracle sends the agent's context to an LLM via an OpenAI-compatible API [18] and parses the response into a Prolog term.

Figure 1 shows the query protocol. Crucially, the oracle query passes through the *same* tell/told filters used for inter-agent communication. This design choice has several advantages:

1. **No new mechanism**: agent designers already familiar with tell/told rules need not learn a new API or configuration format.
2. **Uniformity**: the agent treats LLM interactions and inter-agent messages as instances of the same communication abstraction.
3. **State-dependent control**: filter conditions can depend on the agent's beliefs, past events, and learned knowledge, enabling dynamic, context-aware filtering.
4. **Priority integration**: LLM responses are assigned priorities by told rules and enter the agent's priority queue alongside regular messages.

4. The Filtering Mechanism

We now formalise the filtering protocol applied to AI Oracle queries.

Definition 1 (Communication Filter). A *communication filter* is a finite set $F = \{(p_i, \pi_i, c_i)\}_{i=1}^{n}$ where each p_i is a Prolog term pattern, $\pi_i \in \mathbb{N}_0$ is a priority (default 0), and c_i is a Prolog goal (condition) evaluated against the agent's current state $\sigma = (\mathcal{B}, \mathcal{P}, \mathcal{L})$ comprising beliefs $\mathcal{B}$, past events $\mathcal{P}$, and learned knowledge $\mathcal{L}$.

Definition 2 (Filter Acceptance). Given a filter F and a ground term t:

$$accept(F, t, \sigma) \iff \exists\, (p_i, \pi_i, c_i) \in F : \; p_i \sqsupseteq t \;\wedge\; \sigma \models c_i$$

where $\sqsupseteq$ denotes Prolog subsumption (`subsumes_term/2`) and $\sigma \models c_i$ means condition c_i succeeds in the current agent state. The matching priority π_i is returned alongside the acceptance decision.

Definition 3 (AI Oracle Protocol). Let a be an agent with tell filter F_T (for outgoing communication) and told filter F_R (for incoming communication). Given a query context q, the oracle protocol proceeds as follows:

1. If $F_T = \emptyset$ or $accept(F_T, q, \sigma)$: the query is sent to the LLM, producing a response r.
   a) If $F_R = \emptyset$ or $accept(F_R, r, \sigma)$: the result is r, with associated priority π_r.
   b) Otherwise: the result is $rejected(r)$.
2. If $\neg\, accept(F_T, q, \sigma)$: the LLM is *not* called and the result is $blocked(q)$.

The implementation in DALI2 directly mirrors this definition. Listing 2 shows the core Prolog clause.

Listing 2: Implementation of filtered AI Oracle query (simplified from `engine.pl`).

```
execute_body(Name, ask_ai(Context, Result)) :-
  ( should_allow_send(Name, Context) ->
     ai_oracle:ask_ai(Context, RawResult),
     ( should_allow_receive(Name, RawResult, _) ->
        Result = RawResult
     ;  Result = rejected(RawResult) )
  ;  Result = blocked(Context) ).
```

The predicates `should_allow_send/2` and `should_allow_receive/3` evaluate the tell and told filters respectively. Each predicate first checks whether any filter rules are declared; if none exist, the message passes through (open-world default). Otherwise, it iterates over declared rules, applying `subsumes_term/2` for pattern matching and `call_condition/2` to evaluate body conditions against the agent's live state. If no rule matches, the message is blocked (tell) or rejected (told). The full implementation is available in the companion repository.[1]

4.1. Properties

The mechanism enjoys several desirable properties.

P1 — Uniformity. The same predicates, `should_allow_send/2` and `should_allow_receive/3`, are used for both inter-agent messages and AI Oracle queries, thereby ensuring that an agent's communication policy is defined once and enforced uniformly.

P2 — Declarativeness. Filters are standard Prolog rules. The agent designer specifies *what* is allowed, not *how* to enforce it. This aligns with the declarative philosophy of logic-based agent programming.

P3 — Bidirectionality. Outgoing queries and incoming responses are independently filtered. An agent can allow a broad set of queries while narrowly constraining accepted responses, or vice versa.

P4 — State-dependence. Filter conditions are evaluated against the agent's current cognitive state $(\mathcal{B}, \mathcal{P}, \mathcal{L})$. This enables dynamic policies such as "accept AI suggestions only when in active mode" or "block queries about topic X after learning that X is resolved".

P5 — Fail-safe default. If an agent declares *any* told rules, messages (including LLM responses) that do not match any pattern are automatically rejected. This is a *whitelist* model: only explicitly permitted interactions proceed.

P6 — Priority integration. Told rules assign priorities to accepted LLM responses. These responses enter the agent's message queue alongside regular messages and are processed in priority order. This allows the designer to express that, e.g., an emergency alert from another agent (priority 200) should be processed before an AI suggestion (priority 50).

[1] https://github.com/lollix91/DALI2-Tell-Told

5. Illustrative Scenarios

We illustrate the mechanism on three test scenarios included in the open-source repository [15].

5.1. Smart Agriculture with Filtered AI Oracle

In this scenario, a `crop_advisor` agent receives soil and weather reports from sensor agents and must decide on irrigation actions. When an AI Oracle is available, the advisor queries it for analysis.

Listing 3: Crop advisor with AI Oracle query and communication filters.

```
:- agent(crop_advisor, [cycle(1)]).

% Tell rules: only these query types may be sent
tell(_, _, soil_analysis(_,_,_)) :- true.
tell(_, _, weather_analysis(_,_,_)) :- true.

% Told rules: only accept specific response patterns
told(_, suggestion(_), 80) :- true.
told(_, recommendation(_,_), 90)
   :- believes(status(active)).

% Reactive rule with filtered AI query
soil_reportE(Moisture, PH, Field) :>
  ( ai_available ->
     ask_ai(
       soil_analysis(
         moisture(Moisture),ph(PH),field(Field)),
       Advice),
     log("AI recommends: ~w", [Advice])
  ; true ),
  ( Moisture < 30 ->
     send(irrigation_controller, irrigate(Field))
  ; Moisture > 80 ->
     send(irrigation_controller, reduce_water(Field))
  ; true ).
```

Table 1 shows the filtering behaviour for various query–response pairs.

The tell filter prevents the agent from sending queries unrelated to its domain (e.g., market prices), even if the underlying LLM could answer them. The told filter rejects responses that do not match expected patterns (e.g., financial advice), and the state-dependent condition on `recommendation/2` ensures that high-priority recommendations are only accepted when the agent is in active mode.

Table 1
Filtering behaviour in the smart agriculture scenario.

Event	Filter	Result
Query: `soil_analysis(m(25), ph(6.5), f(n))`	tell ✓	Sent to LLM
Query: `market_price(wheat)`	tell ×	`blocked(...)`
Response: `suggestion(irrigate)`	told ✓	Accepted (pri. 80)
Response: `recommendation(irrigate, urgent)` (status=active)	told ✓	Accepted (pri. 90)
Response: `recommendation(irrigate, urgent)` (status=idle)	told ×	`rejected(...)`
Response: `buy_stock(wheat_futures)`	told ×	`rejected(...)`

5.2. Emergency Response with Priority-Based Filtering

In the emergency response scenario, a `coordinator` agent manages the response to detected emergencies. It communicates with sensor, responder, manager, and evacuator agents, and optionally queries the AI Oracle for situation analysis.

Listing 4: Coordinator with priority-based told rules and AI Oracle.

```
:- agent(coordinator, [cycle(2)]).

% Told rules with priority queue
told(_, emergency(_, _), 200) :- true.
told(_, alert(_, _), 100) :- true.
told(_, sensor_data(_), 30)
   :- believes(status(active)).
told(_, calibration_request, 10) :- true.

% Tell rules
tell(_, _, analyze(_)) :- true.
tell(_, _, log_event(_, _, _)) :- true.

% Emergency handler with AI analysis
emergencyE(Type, Value) :>
 log("EMERGENCY: ~w = ~w", [Type, Value]),
 ( ai_available ->
    ask_ai(analyze(emergency(Type, Value)),
         Advice),
    log("AI advice: ~w", [Advice])
 ; true ).
```

In this configuration, the coordinator's told rules establish a priority hierarchy: emergencies (200) are processed before alerts (100), which are processed before sensor data (30) and calibration requests (10). When the AI Oracle returns a response, it is matched against the same told rules. If the response is, e.g., `suggestion(evacuate_building_a)`, it will be rejected unless a told rule for `suggestion/1` is present. This ensures that the AI Oracle cannot introduce unexpected message types into the agent's processing queue.

Furthermore, the tell rule for `analyze/1` constrains the coordinator to only send analysis-related

queries to the LLM. Queries such as `generate_report(summary)` would be blocked at the tell filter stage, preventing the LLM from being used outside its intended scope.

5.3. State-Dependent Filtering

A third scenario isolates the *state-dependent* behaviour of told rules. A `state_test` agent declares a single tell rule for `suggestion_request/1` and told rules whose body conditions reference the agent's current beliefs:

Listing 5: State-dependent told filter.

```
:- agent(state_test, [cycle(1)]).

tell(_, _, suggestion_request(_)) :- true.
told(_, suggestion(_), 50) :- believes(status(active)).
told(_, recommendation(_,_), 90)
   :- believes(status(active)).
```

The same query `suggestion_request(optimise_irrigation)` is *accepted* when `status(active)` holds and *rejected* when `status(idle)` is believed—even though the LLM returns the same response pattern in both cases. This isolates Property P4 and confirms that body conditions are evaluated against live agent beliefs at the time of filter evaluation.

5.4. Discussion

The three scenarios illustrate complementary aspects of the mechanism. In the agriculture scenario, the emphasis is on *domain-specific filtering*: the agent is restricted to soil/weather queries and accepts only suggestion/recommendation responses. In the emergency scenario, the emphasis is on *priority integration*: LLM responses are prioritised alongside inter-agent messages, and the whitelist model prevents unexpected response types. In the state-dependent scenario, the emphasis is on *context-sensitive control*: the same response is accepted or rejected depending on the agent's current belief state, confirming live evaluation of filter conditions.

A key advantage of the approach is that filter rules are co-located with the agent's other rules in the same source file. The agent designer specifies communication policies as part of the agent definition, not in a separate configuration file or middleware layer. This promotes coherence and makes the agent's communication behaviour auditable by inspection of its source code.

The whitelist semantics (Property P5) is particularly important for LLM integration: since LLMs can produce arbitrary output, a default-deny policy ensures that only expected response patterns are processed. Unexpected responses are gracefully handled by returning `rejected(RawResponse)`, which the agent can log or react to.

A potential limitation of the whitelist model is that it may prevent the agent from usefully adapting to genuinely novel circumstances: if the LLM returns a semantically appropriate response whose syntactic form was not anticipated by the designer, it will be rejected. This rigidity is the price of predictability. Mitigating this trade-off is the motivation for the *adaptive filters* direction outlined in Section 8, where learning rules could dynamically extend the whitelist based on accumulated experience while preserving safety invariants.

6. Experimental Evaluation

We evaluate the filtering mechanism quantitatively across three test suites mirroring the scenarios of Section 5, targeting four research questions: (RQ1) filter correctness, (RQ2) LLM response parseability, (RQ3) state-dependent accuracy, (RQ4) computational overhead.

Table 2
Filter evaluation over 60 interactions (GPT-4o, 3 repetitions per test case). *Blocked* = tell filter; *Rejected* = told filter; *Acc.* = filter accuracy; $\bar{t}$ = mean LLM latency.

Scenario	Tests	Blocked	Accepted	Rejected	Acc.	$\bar{t}$ (ms)
Agriculture	21	9 (43%)	12 (57%)	0	100%	1 545
Emergency	21	9 (43%)	0	12 (57%)	100%	2 546
State-dependent	18	6 (33%)	6 (33%)	6 (33%)	100%	1 629
All	60	24 (40%)	18 (30%)	18 (30%)	100%	1 907

Setup. We evaluated the filtering mechanism on the actual DALI2 system running in Docker [15]. Three agent definitions were loaded into DALI2 (`experiment_agents.pl`): `crop_advisor` (scenario A), `coordinator` (scenario B), and `state_test` (scenario C), each declaring the tell/told rules shown in Listings 3, 4, and 5. A Python driver script injected test events via the DALI2 REST API (`POST /api/inject`) and observed results by subscribing to the Redis `LOGS` pub/sub channel, to which agents publish belief-assertion log entries upon asserting facts.[2] Outcomes were read from `test_result(Id, Outcome)`, where `Outcome` $\in \{$*blocked*, *rejected*, *accepted*$\}$. LLM calls were made through GPT-4o (temperature 0.3, max 100 tokens) via the OpenRouter API [18], with a system prompt instructing the model to return a single Prolog fact. Agent beliefs were set programmatically via `set_status` events to exercise state-dependent body conditions. Three test suites cover: (A) the `crop_advisor` with domain and off-domain queries; (B) the `coordinator` similarly; (C) a state-dependent suite whose told filter only accepts responses when `status(active)` is believed. Each suite provides seven (A, B) or six (C) test cases repeated three times (60 executions total). Source code and raw results are available in the companion repository.[3]

Results. Table 2 reports aggregate statistics.

RQ1 – Filter correctness. All 60 executions yielded the expected outcome, giving 100% filter accuracy. The tell filter intercepted all 24 off-domain queries before any LLM call, eliminating 40% of potential API invocations.

RQ2 – LLM parseability. GPT-4o returned a syntactically valid Prolog term in all 36 LLM-invoked cases (parse rate 100%), confirming that a structured system prompt reliably elicits parseable Prolog output.

RQ3 – State-dependent accuracy. In suite C, the same query with `status(active)` was accepted in all three repetitions, while the identical query with `status(idle)` was correctly rejected, confirming live evaluation of body conditions against agent beliefs.

RQ4 – Filter overhead. Filter evaluation took less than 0.1 ms in all cases, versus a mean LLM latency of 1,545–2,546 ms. The overhead ratio is of order 10^{-4}: filtering is computationally negligible.

Emergent protective behaviour (suite B). All 12 LLM-reachable queries in the emergency suite were *rejected* by the told filter. GPT-4o, given queries of the form `analyze(emergency(`T`,` V`))`, consistently returned terms such as `suggestion(evacuate)` or `action(deploy_responders)` that do not match the coordinator's narrow whitelist (`emergency/2`, `alert/2`, ...). This demonstrates Property P5 in practice: the told filter prevents unanticipated LLM output types from entering the agent's processing queue, even when the outgoing query itself was authorised by the tell filter.

[2] The REST `GET /api/beliefs` endpoint does not access agent subprocess beliefs, since each DALI2 agent runs as a separate OS process with an isolated Prolog database; the Redis `LOGS` channel is the correct observation interface.
[3] https://github.com/lollix91/DALI2-Tell-Told

Table 3
Comparison of LLM control approaches.

Approach	Decl.	Agent-internal	State-dep.	Bidir.
NeMo Guardrails	Partial	No	No	Yes
Prompt engineering	No	No	No	No
API-level filtering	No	No	No	Partial
Tell/told (ours)	**Yes**	**Yes**	**Yes**	**Yes**

Discussion: scenarios with lower accuracy. In our experiments the filters achieved 100% accuracy because patterns were chosen to align with GPT-4o's structured-output behaviour under a constrained system prompt. In practice, filter accuracy could degrade in several ways: (i) if the LLM returns a syntactically malformed term (e.g., missing parenthesis), parsing fails and the response is treated as rejected—a safe default; (ii) if the LLM wraps a valid answer in an unexpected functor (e.g., `result(suggestion(irrigate))` instead of `suggestion(irrigate)`), the told filter will reject it even though the content is semantically correct; (iii) with weaker or less instruction-following models, the rate of unparseable responses may increase, effectively reducing the agent's ability to exploit the oracle. In all these cases the whitelist model errs on the side of *safety*: no invalid or unexpected data enters the agent's reasoning cycle. The cost is a potential loss of *utility* when legitimate answers are rejected due to form rather than content. Strategies to address this include broader pattern matching (e.g., using Prolog unification variables for wrapper functors), semantic matching via the ontology subsystem, or retry logic with rephrased prompts—all of which are compatible with the current architecture.

7. Related Work

LLM-based multi-agent systems. Recent work has explored using LLMs as the primary reasoning engine for agents. Generative Agents [3] simulate human-like behaviour through LLM-driven memory and planning. MetaGPT [4] assigns roles to LLM agents for collaborative software development. CAMEL [19] studies multi-agent communication via role-playing LLMs. A comprehensive survey is provided by [2, 20]. These approaches treat the LLM as the agent's "brain"; in contrast, our approach treats it as a *controlled oracle* within a logic-based reasoning cycle, where the agent retains full declarative control.

Tool-using LLMs. ReAct [5] and Toolformer [6] enable LLMs to invoke external tools. In these architectures, the LLM decides *when and how* to use tools; in ours, the *agent* decides when to consult the LLM, and its queries/responses are filtered by declarative rules. The control flow is inverted.

LLM guardrails. NeMo Guardrails [13] provides a toolkit for constraining LLM behaviour through programmable "rails" defined in a domain-specific language (Colang). [14] surveys guardrail techniques including input/output filtering. These approaches operate at the *infrastructure level*, external to any agent architecture. Our approach integrates filtering *within* the agent's own declarative language, reusing an existing mechanism (tell/told) rather than introducing a new one. Table 3 summarises the differences.

Neuro-symbolic integration. The broader neuro-symbolic AI programme [21, 22] seeks to combine neural and symbolic reasoning. Our work contributes a concrete, practical integration point: logic-based

agents that use LLMs as oracles, with the interaction governed by the agent's declarative communication policy.

Agent communication in BDI systems. Communication filtering has a long history in agent-oriented programming [23, 24]. DALI's tell/told mechanism [7] was designed for inter-agent messaging; GOAL [10] and Jason [9] also provide message-handling constructs. Our contribution is the observation that these existing mechanisms can be repurposed for LLM interaction control without modification, a form of *architectural reuse*.

8. Conclusions and Future Work

We have presented a principled approach to integrating LLMs in logic-based multi-agent systems via declarative communication filters. By reusing the tell/told mechanism from DALI within DALI2's AI Oracle architecture, we achieve controlled, bidirectional, state-dependent filtering of agent–LLM interactions without introducing new constructs or external middleware. The approach is fully declarative, auditable, and operates within the agent's standard reasoning cycle.

The mechanism has been implemented in DALI2, evaluated across three test scenarios, and experimentally verified across 60 filter evaluations (36 involving LLM calls) achieving 100% filter accuracy and negligible overhead. The DALI2 implementation is available at [15]; experiment source code and raw results are archived in the companion repository.[4]

Future work includes: (i) extending the filtering mechanism to support *semantic* matching via the ontology subsystem (e.g., accepting `recommendation` when `suggestion` is declared, if they are ontologically equivalent); (ii) integration with structured output formats (e.g., JSON mode) to further improve LLM response parseability; and (iii) exploring *adaptive filters* that evolve based on the agent's learning rules, dynamically tightening or relaxing LLM interaction constraints based on accumulated experience.

Acknowledgments

The authors acknowledge the DALI research group at the University of L'Aquila (DISIM) for the foundational work on the DALI agent language.

Declaration on Generative AI

During the preparation of this work, the authors used LLM-based tools for assistance with drafting and editing. The authors reviewed and edited all content and take full responsibility for the publication's content.

References

[1] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, et al., GPT-4 technical report, arXiv preprint arXiv:2303.08774 (2023).

[2] Z. Xi, W. Chen, X. Guo, W. He, Y. Ding, B. Hong, M. Zhang, J. Wang, S. Jin, E. Zhou, R. Zheng, X. Fan, X. Wang, L. Xiong, Y. Zhou, W. Wang, C. Jiang, Y. Zou, X. Liu, Z. Yin, S. Dou, R. Weng, W. Qin, Y. Zheng, X. Qiu, X. Huang, Q. Zhang, T. Gui, The rise and potential of large language model based agents: A survey, Science China Information Sciences (2023).

[3] J. S. Park, J. C. O'Brien, C. J. Cai, M. R. Morris, P. Liang, M. S. Bernstein, Generative agents: Interactive simulacra of human behavior, in: Proceedings of the 36th Annual ACM Symposium

[4] https://github.com/lollix91/DALI2-Tell-Told

on User Interface Software and Technology (UIST '23), 2023, pp. 1–22. doi:10.1145/3586183.3606763.

[4] S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, C. Zhang, J. Wang, Z. Wang, S. K. S. Yau, Z. Lin, et al., MetaGPT: Meta programming for a multi-agent collaborative framework, arXiv preprint arXiv:2308.00352 (2023).

[5] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, Y. Cao, ReAct: Synergizing reasoning and acting in language models, in: International Conference on Learning Representations (ICLR 2023), 2023.

[6] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, E. Hambro, L. Zettlemoyer, N. Cancedda, T. Scialom, Toolformer: Language models can teach themselves to use tools, in: Advances in Neural Information Processing Systems 36 (NeurIPS 2023), 2024.

[7] S. Costantini, A. Tocchio, A logic programming language for multi-agent systems, in: Logics in Artificial Intelligence, 8th European Conference (JELIA 2002), volume 2424 of *Lecture Notes in Computer Science*, Springer, 2002, pp. 1–13. doi:10.1007/3-540-45757-7_1.

[8] S. Costantini, A. Tocchio, The DALI logic programming agent-oriented language, Logics in Artificial Intelligence 3229 (2004) 685–688. doi:10.1007/978-3-540-30227-8_57.

[9] R. H. Bordini, J. F. Hübner, M. Wooldridge, Programming Multi-Agent Systems in AgentSpeak using Jason, Wiley, 2007.

[10] K. V. Hindriks, Programming rational agents in GOAL, in: Multi-Agent Programming, Springer, 2009, pp. 119–157. doi:10.1007/978-0-387-89299-3_4.

[11] M. Dastani, 2APL: A practical agent programming language, Autonomous Agents and Multi-Agent Systems 16 (2008) 214–248. doi:10.1007/s10458-008-9036-y.

[12] A. S. Rao, M. P. Georgeff, BDI agents: From theory to practice, in: Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95), MIT Press, 1995, pp. 312–319.

[13] T. Rebedea, R. Dinu, M. N. Sreedhar, C. Parisien, J. Cohen, NeMo guardrails: A toolkit for controllable and safe LLM applications with programmable rails, Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing: System Demonstrations (2023).

[14] Y. Dong, R. Mu, G. Jin, Y. Qi, J. Hu, X. Zhao, J. Meng, W. Ruan, X. Huang, Building guardrails for large language models, arXiv preprint arXiv:2402.01822 (2024).

[15] AAAI-DISIM-UnivAQ, AAAI-DISIM-UnivAQ/DALI2, 2026. URL: https://doi.org/10.5281/zenodo.19858727. doi:10.5281/zenodo.19858727.

[16] J. Wielemaker, T. Schrijvers, M. Triska, T. Lager, SWI-Prolog, Theory and Practice of Logic Programming 12 (2012) 67–96. doi:10.1017/S1471068411000494.

[17] Redis Ltd., Redis – the real-time data platform, 2024. URL: https://redis.io/.

[18] OpenRouter, OpenRouter – unified API for LLMs, 2024. URL: https://openrouter.ai/.

[19] G. Li, H. A. A. K. Hammoud, H. Itani, D. Khizbullin, B. Ghanem, CAMEL: Communicative agents for "mind" exploration of large language model society, arXiv preprint arXiv:2303.17760 (2023).

[20] T. Guo, X. Chen, Y. Wang, R. Chang, S. Pei, N. V. Chawla, O. Wiest, X. Zhang, Large language model based multi-agents: A survey of progress and challenges, IJCAI (2024).

[21] A. d'Avila Garcez, L. C. Lamb, Neurosymbolic AI: The 3rd wave, Artificial Intelligence Review 56 (2023) 12387–12406. doi:10.1007/s10462-023-10448-w.

[22] L. De Raedt, S. Dumanvcić, R. Manhaeve, G. Marra, From statistical relational to neuro-symbolic artificial intelligence, Artificial Intelligence 310 (2020) 103552. doi:10.1016/j.artint.2020.103552.

[23] M. Wooldridge, An Introduction to MultiAgent Systems, 2nd ed., Wiley, 2009.

[24] Foundation for Intelligent Physical Agents, FIPA ACL Message Structure Specification, Technical Report SC00061G, FIPA, 2000. URL: https://web.archive.org/web/20080828231221/http://www.fipa.org/specs/fipa00061/SC00061G.html.