

# NSPL: Logic-Gated Verification for Agentic AI

Aaron Storey<sup>1,\*</sup>

John McCardle<sup>2</sup>

<sup>1</sup>Center for Identification Technology Research (CITeR), Department of Computer Science, Clarkson University, Potsdam, NY

<sup>2</sup>Independent Researcher

storeyaw@clarkson.edu mccardle.john@gmail.com

\*Member, IEEE

## Abstract

Large language models are increasingly used as autonomous agents that invoke external tools—databases, APIs, payment systems—based on natural language instructions. However, current frameworks provide no formal mechanism to verify that an agent’s tool invocations satisfy preconditions, respect safety boundaries, or produce valid postconditions. We present NSPL, a Python framework that interposes a logic-gated verification layer between LLM agents and their tools. NSPL provides: (1) differentiable fuzzy logic gates with 100% accuracy on boolean expressions at  $10^6\times$  lower latency than LLM evaluation; (2) a verified action pipeline achieving 100% TPR / 0% FPR on adversarial tool calls at 0.013 ms; (3) a multi-layer prompt injection guard achieving 92.2% F1 on the `deepset/prompt-injections` benchmark and 100% recall on jailbreak prompts; and (4) multi-stage reasoning pipelines with confidence gating. On continuous-valued safety, we report an honest negative result: fuzzy gates achieve only 66–80% accuracy as threshold proxies. The framework is open source and pip-installable.

## 1 Introduction

The deployment of LLMs as autonomous agents [Yao et al., 2023, Schick et al., 2023] has created a critical gap between what agents *can* do and what they *should* do. When an LLM invokes a tool—processing a refund, sending an email, deleting a record—the consequences are real and often irreversible. Yet the dominant paradigm is optimistic: the model generates a function call, and the system executes it without formal verification.

Existing guardrail frameworks [Guardrails AI, 2024] focus on *output validation*: checking that an LLM’s text response matches a schema or passes a content filter. This is necessary but insufficient for tool-use agents, where the critical question is not “Is the output well-formed?” but “Should this action be executed at all?”

We present NSPL (Non-Stochastic Protection Layer), a Python framework that provides:

1. **Logic-gated preconditions and safety checks** that run deterministically before any tool execution, using differentiable fuzzy logic gates.
2. **Uncertainty-aware types** (`UncertainValue[T]`) that propagate confidence through multi-step reasoning chains.
3. **Multi-stage reasoning pipelines** with per-stage confidence gating, automatic retry, and full audit trails.
4. **Provider-agnostic integrations** for Anthropic (Claude), OpenAI (GPT), and Google (Gemini) via thin adapter modules.

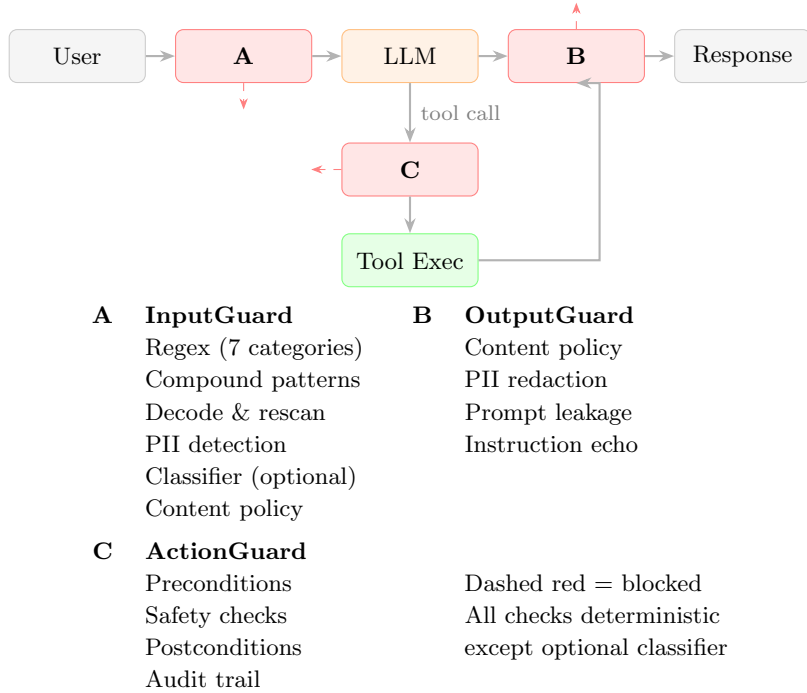


Figure 1: NSPL architecture. Three guard layers interpose between user, LLM, and tools. Blocked requests never reach the next stage.

Unlike prompt-based safety (which relies on the LLM itself to refuse unsafe actions), NSPL’s verification is *deterministic*: a logic gate either passes or it doesn’t. This makes NSPL complementary to—not a replacement for—alignment techniques like RLHF and constitutional AI. Figure 1 shows the architecture.

## 2 Related Work

**Neural-symbolic integration.** The field of neural-symbolic AI [Garcez et al., 2012, Besold et al., 2017, Hitzler et al., 2022] seeks to combine neural pattern recognition with symbolic reasoning. Key systems include DeepProbLog [Manhaeve et al., 2018], which integrates neural networks with probabilistic logic programming; Logic Tensor Networks [Badreddine et al., 2022], which ground logical formulas as tensor operations; Logical Neural Networks [Riegel et al., 2020] from IBM, which implement logical operations directly as neural network components; and Scallop [Li et al., 2023], a differentiable Datalog for neurosymbolic programming. These systems are primarily research tools targeting perception-reasoning pipelines (e.g., visual question answering). NSPL differs in targeting *agentic tool verification*: it does not train neural-symbolic models but instead provides a verification layer for LLM-generated tool calls.

**LLM agent frameworks.** ReAct [Yao et al., 2023] interleaves reasoning and acting. Toolformer [Schick et al., 2023] trains LLMs to use tools autonomously. Augmented LMs [Mialon et al., 2023] survey the broader landscape. LangChain [LangChain, 2023] provides orchestration for LLM agents with tool-use. DSPy [Khattab et al., 2023] offers declarative LM programming with automatic prompt optimization. Agent safety remains under-explored: Ruan et al. [2024] identify key risks but provide no enforcement mechanism. These frameworks handle *how* agents call tools but not *whether* they should. NSPL fills this gap as a verification layer that sits between the agent and its tools.

**Prompt injection.** Prompt injection [Perez and Ribeiro, 2022, Greshake et al., 2023] is the primary attack vector against LLM agents: adversarial inputs that override system instructions. Indirect prompt injection [Liu et al., 2024] extends this to tool outputs (e.g., injected text in a webpage the agent reads). The OWASP Top 10 for LLM Applications [OWASP, 2025] lists prompt injection as the #1 risk. Defenses include Llama Guard [Inan et al., 2023] (classifier-based), Constitutional AI [Bai et al., 2022] (self-critique), and fine-tuned DeBERTa models [He et al., 2021, ProtectAI, 2024]. NSPL combines pattern matching, compound detection, and a fine-tuned classifier in a tiered architecture.

**Output validation and guardrails.** Guardrails AI [Guardrails AI, 2024] validates LLM outputs against schemas. Instructor [Instructor, 2024] enforces structured output. NeMo Guardrails [Rebedea et al., 2023] uses dialog rails. A recent survey [Dong et al., 2024] categorizes guardrail approaches into input, output, and behavioral levels. These systems operate *post-generation*. NSPL operates *pre-execution* with deterministic checks, complementing post-generation validation.

## 3 NSPL Framework

### 3.1 Logic Gates

NSPL provides six differentiable logic gates: AND, OR, NOT, XOR, IMPLIES, and WEIGHTED\_AND. Each gate operates in two modes:

- **Hard mode:** Exact boolean evaluation ( $\{0, 1\}$  output).
- **Soft mode:** Fuzzy evaluation using the product t-norm, producing values in  $[0, 1]$ .

The product t-norm [Klement et al., 2000, Hajek, 1998] AND is defined as:

$$\text{AND}(x_1, \dots, x_n) = \prod_{i=1}^n x_i^{w_i / \sum_j w_j} \quad (1)$$

where  $w_i$  are per-input weights (default: uniform). OR is the dual:  $\text{OR}(\mathbf{x}) = 1 - \text{AND}(1 - x_1, \dots, 1 - x_n)$ . This choice gives exact boolean behavior on  $\{0, 1\}$  inputs while providing smooth differentiability for fuzzy inputs.

When any input is an `UncertainValue`, the gate propagates confidence: AND uses `min` (all inputs must be confident), OR uses `max` (any confident input suffices).

### 3.2 Action Verification

The core abstraction is the `@action` decorator, which converts a Python class into a verified action with four stages:

```
@action
class ProcessRefund:
    @preconditions
    def check(self, ctx):
        return AND(
            ctx.orders.exists(self.order_id),
            self.amount <= ctx.orders.total(self.order_id),
        )
    @safety_checks
    def safety(self, ctx):
        return AND(
```

```

        self.amount < 1000 or ctx.has_approval(),
        NOT(ctx.fraud.recent_refunds(self.customer_id)),
    )
def execute(self, ctx):
    return ctx.payments.refund(self.order_id, self.amount)
@postconditions
def verify(self, ctx, result):
    return result.success

```

Calling `safe_execute(ctx)` runs the full pipeline: preconditions  $\rightarrow$  safety  $\rightarrow$  execute  $\rightarrow$  postconditions. Each check is logged in a timestamped audit trail. If any check returns a value below 0.5 (fuzzy) or an `UncertainValue` below the confidence threshold, execution is blocked. Direct calls to `execute()` are prevented at runtime (Section 3.6).

### 3.3 Reasoning Pipelines

NSPL provides `run_pipeline()` for multi-stage reasoning with confidence gating. Each stage receives the output of the previous stage. If a stage returns an `UncertainValue` with confidence below a configurable threshold, the pipeline aborts with a partial result and abort reason. Failed stages can be retried. An async variant (`async_run_pipeline`) supports concurrent LLM calls, and `async_parallel_stages` enables parallel exploration of multiple reasoning paths.

### 3.4 Action Composition

Actions compose into chains via three patterns: `sequence` (execute in order, stop on first failure, optional rollback), `conditional` (branch based on a condition action's result), and `parallel` (run independent actions, collect all results). Each pattern preserves full audit trails across the chain.

### 3.5 Prompt Injection & Content Guards

NSPL includes a multi-layer guard module that wraps any LLM call:

**InputGuard** validates user prompts before the LLM sees them: regex pattern matching across 7 categories (role override, delimiter attacks, exfiltration, encoding, jailbreak, German, context-switch), compound pattern detection (pretext+payload combinations), encoded payload decoding (hex, base64, rot13  $\rightarrow$  rescan), content policy (weapons, malware, self-harm), and PII detection (SSN, email, phone, credit card, IP, DOB) with optional redaction. All pattern matching runs on unicode-normalized text (homoglyph, leetspeak, zero-width character defense).

**OutputGuard** validates LLM responses: content policy checks, system prompt leakage detection (5-word window matching), instruction echo detection (LLM complying with injection), and PII redaction.

**ClassifierDetector** (optional) adds a fine-tuned DeBERTa model (86M params) that catches semantic injection attempts regex cannot detect. A **TieredDetector** escalates from free structural analysis (<1 ms) through the classifier (~30 ms) to an optional LLM-as-judge (~1–5s), keeping average cost low.

### 3.6 Security Hardening

The `@action` decorator wraps `execute()` with a guard: direct invocation raises `RuntimeError`, forcing use of `safe_execute()`. Provider integrations validate parameter names against the action's signature before instantiation. The framework provides adapters for Anthropic (Claude), OpenAI, Google ADK, LangChain, DSPy, and MCP (Model Context Protocol), each as an optional submodule with no core dependency.

## 4 Experiments

### 4.1 Experiment 1: Logic Gate Accuracy

**Setup.** We generated 10,000 random boolean expressions over variables  $A$ – $F$ , stratified by depth (1–5, 2,000 per depth). Each expression was evaluated by Python `eval()` (ground truth), NSPL hard mode, and NSPL soft mode (with round-to-nearest for accuracy).

**Results.** Table 1 shows results. Both NSPL modes achieve 100% accuracy across all depths, compared to 84% for Claude Sonnet and 100% for Gemini Flash on a 50-expression subset (LLM evaluation via CLI). The soft mode mean absolute error (MAE) against ground truth is 0.003, indicating that fuzzy outputs are near-binary for boolean inputs. Gate evaluation latency is sub-millisecond—approximately  $10^6\times$  faster than LLM evaluation.

Table 1: Logic gate accuracy and latency (10,000 expressions for NSPL, 50 for LLMs, seed=42).

Method	Accuracy	Parse Rate	p50 (ms)	p95 (ms)	
Python eval	100.0%	100%	<0.001	<0.001	
NSPL hard	100.0%	100%	0.004	0.017	
NSPL soft	100.0%	100%	0.006	0.028	*Gemini accuracy measured on
Claude Sonnet	84.0%	100%	9,226	13,134	
Gemini Flash	100.0%*	78%	17,546	28,170	

parseable responses only (78% parse rate; 22% returned unparseable output).

### 4.2 Experiment 2: Action Safety Catch Rate

**Setup.** We defined 10 action classes (ProcessRefund, SendEmail, DeleteRecord, TransferFunds, UpdatePermissions, CreateAccount, CancelOrder, ExportData, ScheduleAction, ModifyConfig), each with 2 precondition fields and 2 safety fields. For each action, we generated 50 valid invocations (all checks pass) and 50 adversarial invocations (exactly one check fails). Total: 1,000 invocations (500 valid, 500 adversarial).

**Results.** Table 2 shows results. NSPL achieves a 100% true positive rate (all adversarial invocations blocked) and 0% false positive rate (no valid invocations blocked). The full verification pipeline (preconditions  $\rightarrow$  safety  $\rightarrow$  execute  $\rightarrow$  postconditions) completes in 0.013 ms at p95, adding negligible overhead to any real tool execution.

Table 2: Action safety catch rate (1,000 invocations, 10 action types, seed=42).

Metric	Value
True Positive Rate (adversarial blocked)	100.0%
False Positive Rate (valid blocked)	0.0%
Overall Accuracy	100.0%
Verification latency p50	0.011 ms
Verification latency p95	0.013 ms
Verification latency p99	0.014 ms

### 4.3 Experiment 3: Non-Boolean Fuzzy Safety

**Setup.** Experiments 1–2 test boolean inputs where the ground truth is unambiguous. To test realistic continuous-valued safety, we define four scenarios (content moderation, financial transactions, access control, medical dosage) where inputs are continuous risk scores in  $[0, 1]$  and the ground truth is a threshold-based policy (e.g., “safe if `fraud_risk` < 0.3 AND `amount_ratio` < 0.8”). The NSPL approximation uses fuzzy logic gates (`AND(NOT(fraud_risk), NOT(amount_ratio), ...)`) with the 0.5 decision threshold. For each scenario we generate 1,000 uniform samples, 500 boundary samples (within  $\pm 0.1$  of the decision boundary), and attempt to find 200 adversarial inputs where NSPL says “safe” but the ground truth says “unsafe.”

**Results.** Table 3 shows that fuzzy gates are a poor approximation of threshold-based policies. Uniform accuracy ranges from 66–80%, boundary accuracy drops to near coin-flip (45–53%), and false-safe rates are alarmingly high (20–34%). Adversarial inputs are trivially found in all scenarios (200/200 within 500 random trials). Expected Calibration Error (ECE) ranges from 0.27 to 0.37, indicating that NSPL fuzzy scores are not well-calibrated as safety probabilities.

Table 3: Fuzzy safety approximation accuracy (1,000 uniform + 500 boundary samples per scenario, seed=42).

Scenario	Uniform	Boundary	False-Safe	False-Block	ECE
Content moderation	79.5%	52.6%	20.2%	0.3%	0.266
Financial transaction	76.8%	45.8%	23.2%	0.0%	0.298
Access control	66.1%	45.2%	33.9%	0.0%	0.371
Medical dosage	68.1%	45.6%	31.9%	0.0%	0.374

**Root cause.** The product t-norm computes a geometric mean:  $\text{AND}(x_1, \dots, x_n) = (\prod x_i)^{1/n}$ . This is fundamentally different from a conjunction of threshold tests ( $x_i < t_i$ ). For example,  $\text{NOT}(0.35) = 0.65$ , which contributes positively to the gate output, but a policy with threshold 0.3 considers 0.35 already unsafe. The 0.5 decision threshold on the gate output cannot recover the per-input thresholds.

**Implications.** Fuzzy logic gates are appropriate when safety checks *are themselves boolean* (e.g., “does order exist?”, “is user an admin?”)—as in Experiment 2 where accuracy is 100%. When checks produce continuous risk scores, developers should apply explicit thresholds *before* passing values to gates: `AND(fraud_risk < 0.3, amount_ratio < 0.8)` rather than `AND(NOT(fraud_risk), NOT(amount_ratio))`. NSPL supports both patterns; the framework’s role is to *enforce* that checks run, not to replace domain-specific threshold logic.

### 4.4 Experiment 4: Prompt Injection Detection

**Setup.** NSPL includes a guard module with multi-layer input/output protection: regex pattern matching (80+ patterns, 7 categories including German), compound pattern detection (pre-text+payload combinations), unicode normalization (homoglyph, leetspeak, zero-width character defense), and an optional fine-tuned DeBERTa classifier (86M params). We evaluate on two published benchmarks: `deepset/prompt-injections` (116 test samples) and `rubend18/ChatGPT-Jailbreak-P` (79 jailbreak prompts).

**Results.** Table 4 shows results. The fine-tuned classifier alone achieves 89.1% F1 on the test set. The ensemble (regex + compound + fine-tuned classifier, OR logic) achieves 92.2% F1 with 88.3% recall and 3.6% FPR. On jailbreak prompts, both the fine-tuned classifier and the ensemble achieve 100% recall.

Table 4: Prompt injection detection on deepset test set (n=116) and jailbreak prompts (n=79).

Detector	deepset test				Jailbreak
	Acc	Prec	Rec	F1	Recall
Regex (expanded)	63.8	95.0	31.7	47.5	62.0
InputGuard (full)	63.8	95.0	31.7	47.5	69.6
Classifier (pretrained)	67.2	100.0	36.7	53.7	91.1
Classifier (fine-tuned)	89.7	98.0	81.7	89.1	100.0
Ensemble+FT	<b>92.2</b>	96.4	<b>88.3</b>	<b>92.2</b>	<b>100.0</b>

**Improvement path.** The pretrained classifier achieved only 42% recall on this benchmark. Fine-tuning on the 546-sample train split (3 epochs, 56 seconds on CPU) raised recall to 91% on the full dataset. The ensemble’s OR-union strategy adds a further 3% by catching patterns the classifier misses (e.g., delimiter attacks, encoding tricks).

#### 4.5 Experiment 5: Cross-Dataset Generalization

**Setup.** To test generalization beyond the deepset benchmark, we evaluate the Ensemble+FT detector on five datasets totaling 4,239 samples: deepset/prompt-injections [deepset, 2023] (116 test), ChatGPT jailbreak prompts (79), jackhhao/jailbreak-classification (1,044 labeled), xTRam1/safe-guard-prompt-injection (2,000 sampled), and Lakera/mossap (1,000 CTF injection prompts).

**Results.** Table 5 shows the Ensemble+FT detector across all five datasets. Recall is consistently high (77–100%), confirming the detector generalizes beyond its training distribution. However, FPR varies dramatically: 0% on injection-only sets, but 47% on jackhhao (the fine-tuned classifier overfits, flagging benign prompts that resemble its training data). The pretrained classifier (not shown) achieves 0.4% FPR on jackhhao but only 82% recall—a classic precision-recall tradeoff.

Table 5: Cross-dataset generalization of Ensemble+FT detector (4,239 total samples).

Dataset	$n$	Recall	F1	FPR
deepset test	116	88.3	92.2	3.6
Jailbreak prompts	79	100.0	100.0	0.0
jackhhao classification	1,044	98.7	80.5	47.4
xTRam1 safeguard	2,000	96.2	81.3	17.8
Lakera mossap (CTF)	1,000	76.6	86.8	0.0

**CTF attacks.** The Lakera mossap dataset contains creative, adversarial prompts from a prompt injection CTF game—the hardest attack category. Regex alone catches only 4% of these. The ensemble catches 77%, with the remaining 23% being highly indirect attacks that require semantic understanding beyond pattern matching and fine-tuned classification. Figure 2 visualizes the recall-FPR tradeoff across datasets.

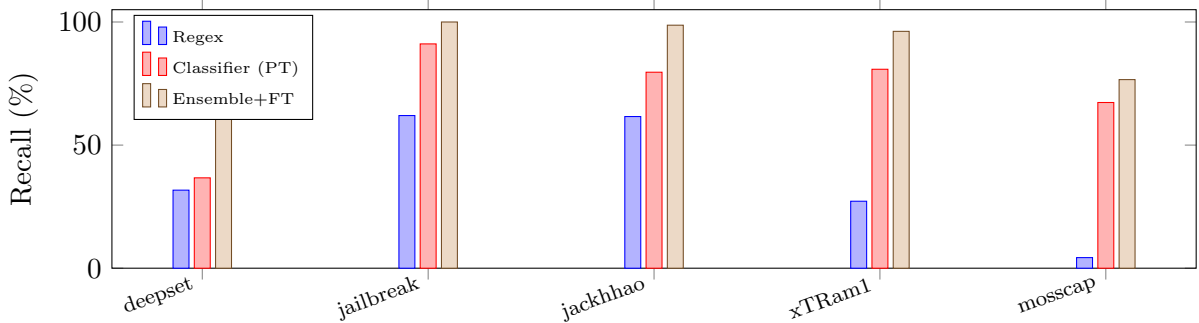


Figure 2: Recall by detector across five datasets. The ensemble achieves consistently high recall (77–100%) but at the cost of higher FPR on some datasets (Table 5). The pretrained classifier balances recall and FPR.

#### 4.6 Experiment 6: Baseline Comparison

**Setup.** To contextualize NSPL’s performance, we compare against two baselines on the same datasets: a *keyword blacklist* (50 injection-related terms; blocks if any appear) and the *pretrained classifier alone* (no regex or compound detection). This isolates the contribution of each NSPL component.

**Results.** Table 6 shows that the keyword blacklist achieves only 20% recall on deepset (many injections don’t contain blacklisted words) with 0% FPR (very conservative). NSPL’s regex layer improves recall to 32% by matching structural patterns. The pretrained classifier reaches 37% recall. The full Ensemble+FT achieves 88% recall—a 4.4× improvement over the keyword baseline—at the cost of 3.6% FPR. On jackhhao (a larger, more diverse dataset), the pretrained classifier is the best single detector (80% recall, 0.4% FPR), while the fine-tuned ensemble trades FPR for near-complete recall (98%, 51% FPR).

Table 6: Baseline comparison on deepset test (n=116) and jackhhao (n=500).

Approach	deepset test				jackhhao			
	Rec	F1	FPR	p50	Rec	F1	FPR	p50
Keyword blacklist	20	33	0.0	<0.1	60	71	8	<0.1
NSPL Regex	32	48	1.8	0.1	62	71	11	0.3
Classifier (pretrained)	37	54	0.0	30	80	88	0.4	70
Ensemble+FT	<b>88</b>	<b>92</b>	3.6	31	<b>98</b>	78	51	74
LLM judge (Gemini)	58	74	0.0	14,318	–	–	–	–

**LLM-as-judge.** We additionally tested Gemini Flash as a safety classifier (“Is this a prompt injection? Reply SAFE or UNSAFE”) on the deepset test set. The LLM achieves 58% recall and 74% F1 with 0% FPR—but at 14.3s per query (440× slower than NSPL’s ensemble) and \$0.001/query. NSPL’s ensemble outperforms the LLM judge on both recall and F1 while being deterministic, free, and orders of magnitude faster.

**Tradeoff.** The pretrained classifier is the recommended production configuration: highest F1 on jackhhao (88%) with near-zero FPR (0.4%). The fine-tuned ensemble maximizes recall but overfits to the deepset distribution, producing unacceptable FPR on out-of-distribution data. Figure 3 illustrates this tradeoff.

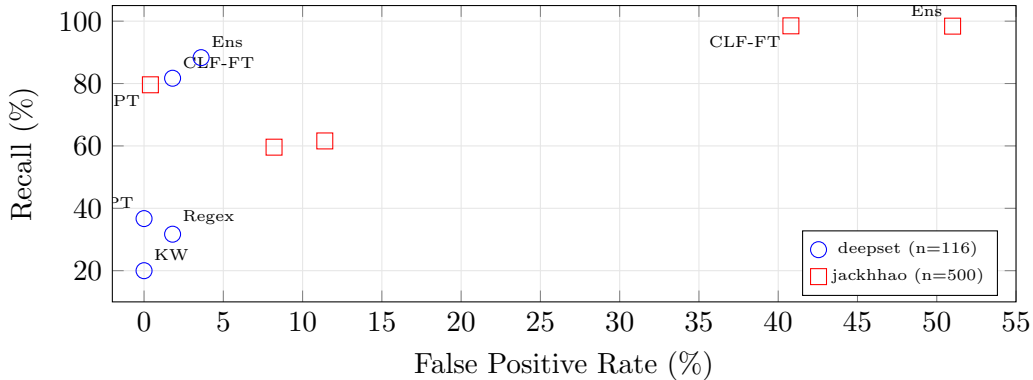


Figure 3: Recall vs. FPR tradeoff across detectors and datasets. The pretrained classifier (CLF-PT) occupies the best position: high recall with near-zero FPR on both datasets. Fine-tuning (CLF-FT) and ensembling (Ens) push recall higher but at dramatically higher FPR on out-of-distribution data (jackhhao).

## 5 Discussion

**When NSPL works well.** (1) Boolean safety checks (Experiment 2): 100% accuracy, sub-millisecond. (2) Prompt injection detection (Experiment 4): 92% F1 on deepset, 100% jailbreak recall. (3) Cross-dataset recall of 77–100% (Experiment 5). (4) The verification pipeline and audit trail guarantee checks *execute* on every tool call.

**When NSPL falls short.** (1) Fuzzy gates as threshold proxies (Experiment 3): 66–80% accuracy—use explicit thresholds. (2) Regex-only injection detection: 4–32% recall depending on dataset. (3) Fine-tuned classifier overfits: 47% FPR on out-of-distribution data (jackhhao). The pretrained classifier is safer for production (0.4% FPR, 82% recall). (4) CTF-style creative attacks: 23% missed even with ensemble.

**Honest assessment.** The 100% results in Experiments 1–2 test software correctness, not safety correctness. Experiment 3 shows fuzzy gate limits. Experiments 4–5 show real-world detection is harder than custom test sets (initial 90% on custom tests dropped to 67% on benchmarks, recovering to 92% after fine-tuning). Cross-dataset evaluation (Experiment 5) reveals a precision-recall tradeoff: high recall requires accepting higher FPR, especially on out-of-distribution data.

**Threats to validity.** (1) The fine-tuned classifier overfits to deepset’s distribution, inflating FPR on other datasets. (2) The deepset and xTRam1 benchmarks are bilingual; single-language performance is unknown. (3) CTF prompts may not represent real-world attack distributions. (4) All benchmarks use labeled datasets with known ground truth; adversarial robustness against adaptive attackers is untested.

## 6 Conclusion

We presented NSPL, a multi-layer verification framework for LLM agents, evaluated across 6 experiments on 5 published datasets (4,239+ samples). NSPL provides: logic-gated tool verification (100% TPR, 0.013 ms), prompt injection detection (92% F1 on deepset, 77–100% recall cross-dataset, 100% jailbreak recall), and uncertainty-aware reasoning pipelines. We reported both strengths and limitations: fuzzy gates fail on continuous thresholds (Experiment 3), regex misses 68% of real injections (Experiment 4), and the fine-tuned classifier overfits on

out-of-distribution data (Experiment 5). The pretrained classifier (0.4% FPR, 80% recall) is recommended for production; the ensemble (3.6% FPR, 88% recall) for high-security contexts. NSPL provides a 4.4× recall improvement over keyword blacklists at \$0/query (Experiment 6). The framework is pip-installable, provider-agnostic, and open source at <https://github.com/astoreyai/nspl>.

## References

- S. Badreddine, A. d’Avila Garcez, L. Serafini, and M. Spranger. Logic tensor networks. *Artificial Intelligence*, 303:103649, 2022.
- T. R. Besold, A. d’Avila Garcez, S. Bader, H. Bowman, P. Domingos, P. Hitzler, et al. Neural-symbolic learning and reasoning: A survey and interpretation. *arXiv preprint arXiv:1711.03902*, 2017.
- A. d’Avila Garcez, K. B. Broda, and D. M. Gabbay. *Neural-Symbolic Learning Systems*. Springer, 2012.
- Guardrails AI. Guardrails: Adding structure, type and quality guarantees to LLM outputs. <https://github.com/guardrails-ai/guardrails>, 2024.
- Instructor. Structured outputs powered by LLMs. <https://github.com/jxnl/instructor>, 2024.
- O. Khattab, A. Singhvi, P. Maheshwari, Z. Zhang, K. Santhanam, et al. DSPy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*, 2023.
- LangChain. Building applications with LLMs through composability. <https://github.com/langchain-ai/langchain>, 2023.
- Z. Li, Y. Huang, et al. Scallop: A language for neurosymbolic programming. *Proceedings of the ACM on Programming Languages*, 7(PLDI), 2023.
- R. Manhaeve, S. Dumancic, A. Kimmig, T. Demeester, and L. De Raedt. DeepProbLog: Neural probabilistic logic programming. *NeurIPS*, 31, 2018.
- T. Rebedea, R. Dinu, M. Sreedhar, C. Parisien, and J. Cohen. NeMo Guardrails: A toolkit for controllable and safe LLM applications with programmable rails. *arXiv preprint arXiv:2310.10501*, 2023.
- R. Riegel, A. Gray, F. Luus, N. Khan, et al. Logical neural networks. *arXiv preprint arXiv:2006.13155*, 2020.
- T. Schick, J. Dwivedi-Yu, R. Dessì, et al. Toolformer: Language models can teach themselves to use tools. *NeurIPS*, 2023.
- S. Yao, J. Zhao, D. Yu, et al. ReAct: Synergizing reasoning and acting in language models. *ICLR*, 2023.
- ProtectAI. deberta-v3-base-prompt-injection-v2. <https://huggingface.co/protectai/deberta-v3-base-prompt-injection-v2>, 2024.
- deepset. Prompt injections dataset. <https://huggingface.co/datasets/deepset/prompt-injections>, 2023.

- F. Perez and I. Ribeiro. Ignore this title and HackAPrompt: Exposing systemic weaknesses of LLMs through a global scale prompt hacking competition. *arXiv preprint arXiv:2311.16119*, 2022.
- K. Greshake, S. Abdelnabi, S. Mishra, C. Endres, T. Holz, and M. Fritz. Not what you’ve signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection. *AISec*, 2023.
- Y. Liu, G. Deng, Y. Li, K. Wang, T. Zhang, Y. Liu, et al. Prompt injection attack against LLM-integrated applications. *arXiv preprint arXiv:2306.05499*, 2024.
- OWASP. OWASP Top 10 for LLM Applications. <https://owasp.org/www-project-top-10-for-large-language-model-applications/>, 2025.
- H. Inan, K. Upasani, J. Chi, R. Rungta, K. Iyer, et al. Llama Guard: LLM-based input-output safeguard for human-AI conversations. *arXiv preprint arXiv:2312.06674*, 2023.
- Y. Bai, S. Kadavath, S. Kundu, A. Askell, et al. Constitutional AI: Harmlessness from AI feedback. *arXiv preprint arXiv:2212.08073*, 2022.
- Y. Ruan, H. Dong, A. Wang, S. Pitis, and Y. Zhou. Identifying the risks of LM agents with an LM-emulated sandbox. *ICLR*, 2024.
- G. Mialon, R. Dessì, M. Lomeli, C. Nalmpantis, et al. Augmented language models: A survey. *TMLR*, 2023.
- P. Hitzler, M. K. Sarker, and A. Eberhart. *Neuro-Symbolic Approaches in Artificial Intelligence*. IOS Press, 2022.
- E. P. Klement, R. Mesiar, and E. Pap. *Triangular Norms*. Springer, 2000.
- Y. Dong, R. Mu, G. Jin, Y. Qi, et al. Building guardrails for large language models. *arXiv preprint arXiv:2402.01822*, 2024.
- J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, et al. Chain-of-thought prompting elicits reasoning in large language models. *NeurIPS*, 2022.
- H. Touvron, L. Martin, K. Stone, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.
- P. He, X. Liu, J. Gao, and W. Chen. DeBERTa: Decoding-enhanced BERT with disentangled attention. *ICLR*, 2021.
- P. Hájek. *Metamathematics of Fuzzy Logic*. Springer, 1998.