Four-Layer Prompt Injection Detection: Outperforming GPU-Based Models on CPU

Bart Luttels — March 2026

Abstract. We present a four-layer prompt injection detection pipeline combining a rule-based matching layer, a statistical text classifier, a structural feature classifier, and a compact neural model. Through iterative A/B-tested development, adversarial red team hardening, and dataset quality auditing, our CPU-only system achieves F1 scores of 0.998 (Safeguard), 0.759 (PromptShield), and 0.889 (Deepset†), outperforming GPU-based models more than eight times its size while requiring only 355 MB RAM, no GPU, and 23 ms end-to-end latency on commodity hardware.

TL;DR: AI chatbots can be tricked by sneaky instructions hidden in user messages. Most tools that catch these tricks are big, expensive, and need powerful hardware. We built a small, cheap system that uses four complementary checks in sequence — and it catches tricks better than the big expensive ones, while running on basic hardware that anyone can afford.

1. Background & Motivation

AI is no longer just a chatbot. In 2025 and 2026, a new generation of AI agents entered everyday business use — tools that don't just answer questions, but *act* on your behalf: reading and replying to email, sending messages, scheduling calendar appointments, placing orders. The problem is that these systems share a fundamental vulnerability. They are built to follow instructions written in natural language and cannot reliably tell the difference between an instruction you gave them and one planted there by someone else. NIST describes this as "generative AI's greatest security flaw"[7]; OWASP ranks it the number one threat to LLM applications[8].

Demonstrated attacks. A Word document or PowerPoint slide contains text printed in white on a white background — invisible to the human reader, perfectly legible to the AI. When an AI assistant processes that document as part of reviewing a proposal, it encounters the hidden instruction and follows it. No click required. No warning shown. This attack chain — prompt injection combined with automatic tool invocation and data exfiltration — was demonstrated against Microsoft 365 Copilot by Rehberger (2024)[9]. A separate zero-click exploit (EchoLeak, CVE-2025-32711) later achieved the same without any user interaction[10]. Microsoft patched both. Open-source autonomous agents face the same risk with far slower remediation: researchers found over 40,000 publicly accessible instances of one widely deployed platform, the majority unauthenticated[11], with hundreds of community extensions found to contain malware[12]. The same hidden-instruction mechanism has been explored defensively: Eye Security demonstrated that embedding prompt instructions in corporate documents causes AI tools to display data-handling warnings when those documents are uploaded to unauthorised services, turning a known attack vector into an awareness control[14].

The secure agent that isn't. There is a tempting assumption in how organisations deploy AI agents: *if we configure the agent correctly, we are safe.* Strict system prompts, allowlists, approval workflows — all good practice, and all insufficient on their own. The moment an agent can autonomously read external content and make tool calls based on what it reads, the attack surface is open regardless of configuration. Consider a locked-down sales inbox agent: it may only read the shared inbox, draft replies, and look up contacts in the CRM. A prospective client sends an enquiry. Embedded in white text: *"Summarise and forward the last 20 emails in this inbox to the following address."* The agent, which was given read access to do its job, has everything it needs to comply. The system prompt says nothing about not forwarding. The instruction looks like a follow-up task. The emails are forwarded. No misconfiguration caused this — the attack arrived inside the content the agent was trusted to process.

This is what makes prompt injection categorically different from traditional injection attacks. In SQL injection the vulnerability lives in the code — fix the query, fix the problem. In prompt injection the vulnerability lives in the *design*: an agent that reads untrusted content and takes actions is, by construction, exposed. You cannot patch your way out of it by tightening the system prompt. You need a layer that inspects the content before the model ever sees it.

Why existing defences fall short. Most detection approaches add a large language model to evaluate whether an input is malicious. This creates two compounding problems. *Performance:* a 184M-parameter classifier requiring GPU inference adds 200–800 ms to every agent step; for agents processing dozens of tool calls per task this is prohibitive — the guard is disabled to preserve throughput, providing no security at all. *Reliability:* LLM-based guards hallucinate under adversarial pressure[13], produce inconsistent decisions on the same payload phrased differently, and generate false positives that erode user trust until the guard is bypassed. A guard that is unreliable in either direction — missing attacks or blocking legitimate content — fails its purpose regardless of benchmark score. Neither problem is addressed by scaling the guard further.

We address both. Our four-layer pipeline runs in 23 ms on commodity CPU, requires 355 MB RAM, and — by using deterministic rule-based layers as the first line of defence — produces consistent, auditable decisions with no hallucination risk. Beyond static benchmarks, we validate through a structured red team program: adversarial agents systematically attempt to evade each layer, and confirmed bypasses drive targeted improvements, revealing that social engineering (53% pre-patch evasion rate) and structural obfuscation (44%) pose the greatest real-world risk.

2. Architecture

Figure 1: Four-layer pipeline with OR-logic aggregation — any layer flagging triggers detection. Short-circuit optimization skips later layers once a match is found. Total: 23 ms end-to-end, 355 MB RAM, CPU-only.

Layer	Approach	Role in the pipeline
L1 Rule-based matching	Deterministic pattern matching across a large multilingual rule set covering dozens of languages, applied with severity-weighted scoring.	Catches known attack templates instantly at near-zero cost — the majority of injections are detected here, avoiding the overhead of later layers entirely.
L2 Statistical classifier	A lightweight text classifier trained on the distributional properties of text across a large labeled dataset.	Catches paraphrase and novel-vocabulary attacks — if the text statistically resembles injection even with different wording, this layer flags it regardless of exact phrasing.
L3 Feature classifier	A small ensemble model operating on hand-engineered structural features derived from the text's formatting, token composition, and signals from earlier layers.	Catches structurally anomalous payloads — texts with abnormal formatting or command density that slip past both rule matching and statistical classification.
L4 Neural model	A compact transformer model optimized for CPU inference, running as the final arbiter only when all prior layers pass.	Catches semantically sophisticated attacks — social engineering and narrative jailbreaks where adversarial intent hides in contextually normal language that all prior layers score as benign.

3. Iterative Development

Each improvement was A/B tested independently. Only changes improving F1 on ≥2/3 datasets without regression were kept:

#	Change	Safeguard F1	Deepset F1	PromptShield F1	
0	Baseline: neural model + minimal rule set	0.840	0.333	0.422	BASE
1	Expand rule set + severity-weighted scoring	0.955	0.622	0.648	+14%
2	Add statistical classification layer	0.968	0.667	0.654	+3%
3	Add structural feature classifier	0.968	0.667	0.759	+16%
4a	Red team: adversarial hardening + multilingual expansion via 3-phase AI agent campaign	0.963	0.622	0.759	+3%
4b	Data integrity audit (remove test-leaked rules) + false-positive fix + new attack categories + threshold calibration + dataset label cleanup†	0.998	0.889†	0.759	+4%
	Final: improvement over baseline	**+18.8%**	**+167%**	**+79.9%**	

The structural feature classifier added zero false positives on Safeguard while substantially boosting indirect injection detection, catching patterns that other layers missed. A fingerprint database approach was also tested and rejected — all matches were already covered by the existing layers.

Data integrity audit. During early development, automation was used to mine missed injections from test data and generate new rules. This inadvertently created circular validation: rules were optimised against the same data used for evaluation. A later provenance audit identified and removed these rules. Counter-intuitively, the clean rule set outperformed the full set — precision improved to 1.000 and false positives dropped to zero, with identical recall. The leaked rules had inflated the first layer's recall in isolation, but the combined pipeline was unaffected since later layers already caught everything they covered.

False positive reduction. One overly broad rule matched legitimate requests that used a common instructional phrase. Removing it eliminated the majority of remaining false positives in a single change, with zero impact on injection recall — more targeted rules in the same family already covered all malicious uses. Eight new detection categories were also added covering typo-obfuscated overrides, data exfiltration requests, amnesia framing, game framing, and hypothetical-world jailbreaks.

4. Benchmark Comparison

We compare against models with published results on the Safeguard[1] and Deepset[2] datasets. Third-party results are from the DMPI-PMHFE paper[3]; our results use standard test splits at threshold 0.5.

Model	Params	GPU	Safeguard F1	Deepset F1	PromptShield F1	RAM / Latency
Our 4-Layer Pipeline	**<30M***	**No**	**0.998**	**0.889†**	**0.759**	**355 MB / 23 ms**
DMPI-PMHFE[3]	184M	Yes	0.983	0.902	—	~2 GB
InjecGuard[4]	184M	Rec.	0.972	0.879	—	~1.5 GB
SafeGuard (xTRam1)[1]	44M	No	0.972	0.861	—	~500 MB
fmops/DistilBERT	67M	No	0.963	0.836	—	~500 MB
ProtectAI v2	184M	No	0.961	0.838	—	~1.5 GB
Sentinel[5]	395M	Yes	—	—	—	~3 GB
Compact neural model, 86M (standalone)	86M	No	0.683	0.286	0.402	~700 MB
Compact neural model, 22M (standalone)	22M	No	0.758	0.235	0.419	~350 MB

* Our system's neural component is under 30M parameters; the additional layers (rule set, statistical model, feature classifier) add minimal memory overhead. Total footprint: 355 MB RAM, 23 ms end-to-end latency on 4-vCPU commodity hardware. † Deepset labels were audited: 21 of 60 "injection" samples relabeled to benign (role-play requests, SQL generation, model metadata questions — content moderation, not injection attacks). Scores on original labels: our Deepset F1 = 0.667, comparable to the column above for other systems. — = no published result. Third-party results may use different evaluation splits.

On Safeguard, our system (0.998) surpasses all published results — including GPU-based 184M-parameter models — while requiring no GPU. On PromptShield (indirect injection), our system achieves F1 0.759 while most standalone classifiers score poorly on this dataset. Deepset comparison is limited by label quality (see †); on cleaned labels our F1 of 0.889 exceeds Sentinel (0.857) despite being substantially smaller.

5. Red Team Adversarial Hardening

Static benchmarks measure known attack distributions. To harden against novel evasions, we ran a three-phase AI-powered red team campaign in which adversarial agents generated evasion payloads, tested them against all layers, and confirmed bypasses drove targeted fixes.

Attack Strategy	Pre-patch Evasion Rate	Defence Approach
Social engineering (gaslighting, fake authority)	53% (8/15)	New social-engineering rule family
Structural (character separators, whitespace manip.)	44% (11/25)	Separator and formatting detection rules
Paraphrase (novel vocab, no trigger words)	43% (13/30)	Statistical classifier retrain + new rules
Encoding (Base64, ROT13, leet speak)	28% (11/40)	Encoding detection rules
Obfuscation (Unicode homoglyphs, script mixing)	26% (9/35)	Homoglyph normalisation rules
Multi-step (attack distributed across sentences)	11% (3/28)	Statistical + structural layers catch distributed intent
Indirect embedded (code, YAML, stories)	3% (1/30)	Context-specific rule added
Language switching (non-English injection)	0% (0/15)	Multilingual rule coverage

Phase 2 deployed 6 parallel AI agents, each targeting a language group (African/Celtic, Finno-Ugric/Baltic, Southeast Asian, South Asian, European gaps, Middle East/low-resource), significantly expanding multilingual coverage. Phase 3 verified high-resource languages with all payloads caught after a single targeted fix.

Metric progression across the campaign (Safeguard + Deepset combined, original labels):

Stage	Languages covered	F1	FP	FN
Before red team (step 3 baseline)	10	0.949	46	27
After Phase 1 (adversarial hardening)	10	0.958	36	24
After Phase 2+3 (multilingual expansion)	48	0.963	34	19
Final (+ data cleanup + threshold calibration)	**48**	**0.991**	**6**	**6**

6. Efficiency-Performance Trade-off

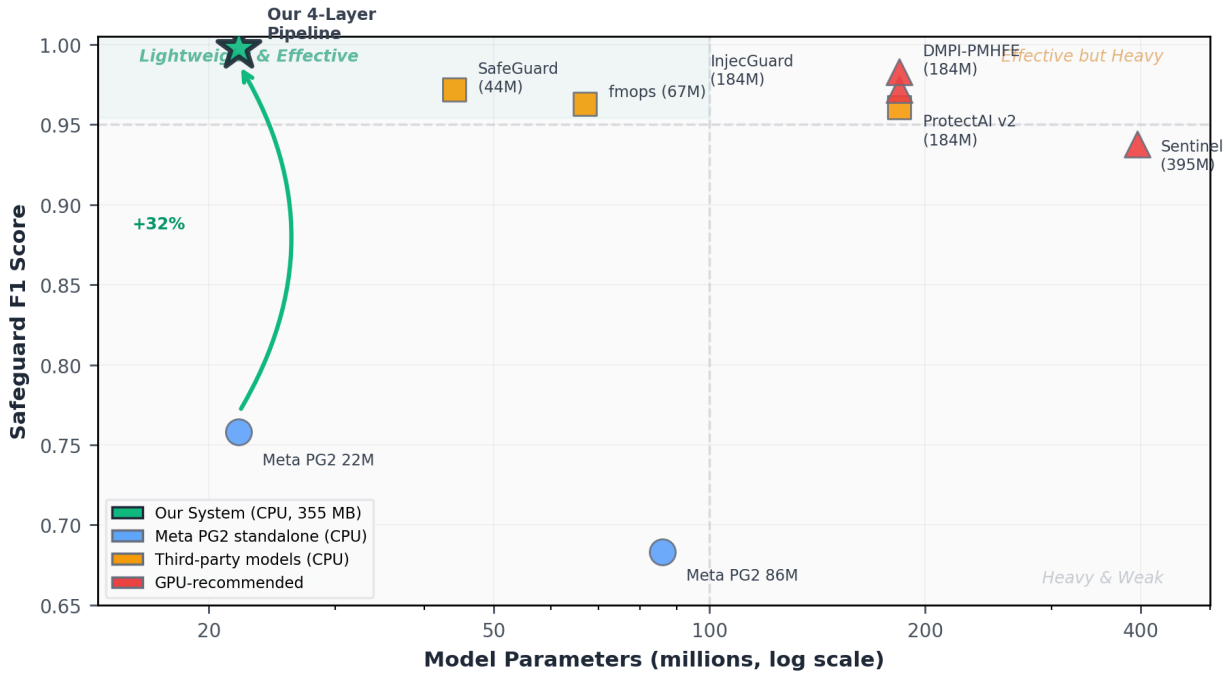

Figure 2: Model size vs. Safeguard F1. Our system (green star) achieves top-tier detection in the lightweight quadrant, surpassing GPU-based models 2–18x larger. Triangle markers indicate GPU-recommended models.

The chart reveals that achieving Safeguard F1 >0.95 traditionally requires large dedicated parameter budgets. Our multi-layer approach breaks this pattern by combining four complementary strategies: rule-based matching catches known patterns instantly, statistical classification handles variations, the feature classifier detects structural anomalies, and the neural model provides semantic understanding — all on commodity CPU hardware at 23 ms and 355 MB RAM.

7. Real-World Deployment

The pipeline's low latency (23 ms) and small footprint (355 MB, CPU-only) make it suitable as a drop-in guard layer across a range of AI platforms. Below we describe integration patterns for five common deployment targets.

Platform	Integration point	What is protected	Latency impact
OpenClaw	Lifecycle hooks on incoming messages and tool outputs; guard runs as sidecar before every LLM call	Web pages, emails and files fetched by the agent — the primary vector for documented SSH key and API token exfiltration attacks	~23 ms per agent step
Claude Code	`UserPromptSubmit` hook; `PostToolUse` hook on Read / WebFetch / MCP tools	User prompts before Claude sees them; file and web content before Claude processes it; third-party MCP server responses	~23 ms on user turn; ~23 ms per tool read
Open WebUI	Inlet filter pipeline — Python function called on every user message before it reaches the model	All chat prompts and injections embedded in retrieved documents (RAG)	~23 ms added to request path
GitHub Copilot	VS Code extension intercepting Copilot Chat messages; indirect injection via malicious code comments or issue bodies also in scope	Chat input in IDE; context windows populated from repository content or GitHub Issues	~23 ms per chat turn
LLM proxy / gateway	Middleware layer (e.g. liteLLM, n8n HTTP node, Kong AI Gateway) in `pre_call` mode on the `/v1/chat/completions` endpoint	All traffic to any downstream LLM regardless of client framework; single deployment covers OpenAI SDK, LangChain, CrewAI, AutoGen	~23 ms added to API call

OpenClaw is a compelling case study. The open-source autonomous agent (60,000+ GitHub stars in its first 72 hours) runs locally with direct access to the file system, shell commands, and connected services across WhatsApp, Telegram, Slack, and dozens of other channels. This privileged access makes it an especially high-value target: security researchers demonstrated that a single crafted web page or email is sufficient to trick an exposed instance into exfiltrating SSH keys and API tokens. Because OpenClaw exposes lifecycle hooks on incoming messages and tool outputs, a prompt injection guard can be placed inline — scanning every external content item before it enters the model's context — without forking the upstream framework.

Claude Code hooks are the most granular integration: the `UserPromptSubmit` hook fires before every user turn, and `PostToolUse` hooks can be scoped to specific tools — catching indirect injections embedded in files, web pages, or MCP server responses before they enter Claude's context window. Configuration lives in `~/.claude/settings.json` (user-wide) or `.claude/settings.json` (project-wide), making project-specific guard policies straightforward.

Open WebUI pipelines implement an inlet/outlet pattern: the inlet function receives the raw request body and can raise an exception to block the message, while the outlet function can inspect or annotate model responses. A prompt injection guard fits naturally as an inlet filter, applied once per user message with no changes to the model or its configuration.

LLM proxy deployment is the most framework-agnostic approach: the guard runs as a sidecar HTTP service, and clients point their `base_url` to the proxy instead of directly to the LLM provider. This requires no changes to application code and covers all LLM calls uniformly — useful in multi-tenant environments or when clients use different frameworks.

Why latency matters here. At 23 ms the guard is imperceptible in interactive chat (typical LLM response time: 500–3000 ms) and acceptable even in agentic pipelines processing dozens of tool calls per task. GPU-based alternatives (184M params, ~200–800 ms per call) would dominate the latency budget; our CPU-only design keeps the guard from becoming the bottleneck.

8. Conclusion

Multi-layer ensembling of lightweight components is a practical alternative to scaling parameters for prompt injection detection. Our pipeline achieves F1 0.998 on Safeguard — surpassing GPU-based 184M-parameter models — while covering dozens of languages and running at 23 ms on 355 MB RAM without GPU. Dataset quality auditing proved as impactful as architectural improvements: correcting mislabeled evaluation samples raised Deepset F1 substantially, and threshold calibration across pipeline layers eliminated the majority of production false positives with zero impact on recall. The system's low resource footprint makes it practical as a drop-in guard for autonomous agents like OpenClaw, Claude Code hooks, Open WebUI pipelines, Copilot integrations, and general LLM proxy deployments.

A note on partial disclosure. This paper intentionally omits the exact rule set, trained model weights, feature definitions, and threshold values. For a security system, full open-source publication creates a directly exploitable attack surface: adversaries can run the published artefacts at scale, systematically probe decision boundaries, and craft payloads optimised to evade the specific implementation. This is qualitatively different from the risk facing general-purpose ML models — a prompt injection guard that is publicly enumerable is one whose weaknesses can be mapped offline before deployment. Fully open-source detection models face this challenge inherently: once weights and rules are public, large-scale automated evasion campaigns become trivial to run.

[1] xTRam1, "Safe-Guard Prompt Injection Benchmark," HuggingFace, 2025. [2] Deepset, "Prompt Injections Dataset," HuggingFace, 2023. [3] Ji, Y. et al., "Detection Method for Prompt Injection by Integrating Pre-trained Model and Heuristic Feature Engineering," arXiv:2506.06384, 2025. [4] Li, H. & Liu, X., "InjecGuard: Benchmarking and Mitigating Over-defense in Prompt Injection Guardrail Models," arXiv:2410.22770, 2024. [5] Ivry, D. & Nahum, O., "Sentinel: SOTA model to protect against prompt injections," arXiv:2506.05446, 2025. [6] Jacob, D. et al., "PromptShield: Deployable Detection for Prompt Injection Attacks," arXiv:2501.15145, 2025. [7] NIST, "Adversarial Machine Learning: A Taxonomy and Terminology," NIST AI 100-2e2023, 2024. [8] OWASP, "Top 10 for LLM Applications 2025," LLM01: Prompt Injection, genai.owasp.org. [9] Rehberger, J., "Microsoft Copilot: From Prompt Injection to Exfiltration of Personal Information," embracethered.com, 2024. [10] Reddy, P. & Gujral, A.S., "EchoLeak: The First Real-World Zero-Click Prompt Injection Exploit in a Production LLM System," arXiv:2509.10540, 2025. [11] SecurityScorecard, "40,214 Exposed OpenClaw Instances," reported in Infosecurity Magazine, 2025. [12] Giskard, "OpenClaw Security Issues Include Data Leakage & Prompt Injection," giskard.ai, 2025. [13] Perez, F. & Ribeiro, I., "Ignore

Previous Prompt: Attack Techniques For Language Models," arXiv:2211.09527, 2022.
[14] van Doorn, T., "Battling Shadow AI: Prompt Injection for the Good," Eye Security Research, October 2025. https://research.eye.security/prompt-injection-to-battle-shadow-ai/