International Journal of Scientific Research and Engineering Development-– Volume 9 Issue 4, July – August 2026
Available at www.ijsred.com

RESEARCH ARTICLE OPEN ACCESS

PromptShield AI: A Multi-Agent Architecture for Intelligent Prompt Injection and Jailbreak Attack Detection Using Machine Learning

Jahnavi Somaraju[1], N. Sree Charan[2], M. Mythili[3], T. Reddy Bhargavi[4], K. Navya Sree[5]

Department of Artificial Intelligence, Aditya College of Engineering, Madanapalle

Email:jahnavisomaraju.research@gmail.com

Abstract: Large language models (LLMs) are increasingly deployed in user-facing applications, which exposes them to prompt injection and jailbreak attacks that override system instructions, exfiltrate data, or elicit disallowed behaviour. Existing defences are largely single-mechanism: a rule filter, a single machine-learning (ML) classifier, or a single LLM-based judge, each of which is comparatively easy to evade once its decision boundary is known. This paper proposes PromptShield AI, a multi-agent architecture that fuses lexical, statistical, semantic, and contextual detection signals through a coordinated set of specialised agents rather than a single monolithic classifier. An Orchestrator Agent decomposes each incoming request and routes it to five parallel detection agents; a Decision Fusion Agent calibrates and combines their outputs; a Response/Mitigation Agent executes the resulting policy (allow, sanitize, or block); and a Logging and Feedback Agent closes the loop for continuous retraining. We describe the architecture, the inter-agent communication protocol, the fusion algorithm, and a reference implementation, and we outline an experimental protocol together with illustrative evaluation results and an ablation study. The results indicate that the coordinated multi-agent design improves detection F1-score and reduces false-positive rate relative to any individual detection mechanism, at an acceptable latency overhead, while remaining more resilient to obfuscation and multi-turn attacks than single-agent baselines.

Keywords — prompt injection, jailbreak detection, multi-agent systems, large language models, ensemble machine learning, NLP security, LLM guardrails.

I. INTRODUCTION

A. Motivation

Large language models are now embedded in customer-support bots, coding assistants, retrieval-augmented search, and autonomous agents that take actions on a user's behalf. This exposure creates a new attack surface: prompt injection, in which an attacker's text is interpreted as an instruction rather than as data, and jailbreaking, in which an attacker manipulates the conversation to bypass the model's safety policy. Because the attack payload is natural language rather than a binary exploit, conventional web-application firewalls and signature-based intrusion-detection systems transfer poorly to this domain, motivating detection methods tailored to language-model inputs.

B. Problem Statement

Single-mechanism defences — a keyword or regular-expression filter, a single supervised classifier, or a single LLM acting as a judge — each capture only part of the attack surface. Rule-based filters achieve high precision on known attack vocabulary but are bypassed by paraphrase and encoding tricks; supervised classifiers generalise better to paraphrased attacks but suffer elevated false-positive rates on benign text that merely discusses security topics; and LLM-judge approaches are comparatively expensive and can themselves be manipulated by the same instructions they are meant to police. No single detector reliably covers lexical evasion, semantic intent, adversarial obfuscation, and multi-turn escalation at the same time.

C. Why Multi-Agent Systems Are Needed

A multi-agent architecture allows each detection strategy to be encapsulated in a specialised, independently maintainable agent, so that the weaknesses of one detector (e.g., a rule-based agent missing a paraphrased attack) are compensated for by the strengths of another (e.g., a semantic-intent agent). Coordinating these agents through an orchestrator and a fusion mechanism, rather than running them as an unstructured ensemble, additionally enables dynamic routing (skipping expensive agents for clearly benign traffic), shared context across turns, and a single point at which conflicting verdicts are reconciled and audited.

D. Research Objectives

This paper aims to: (i) design a multi-agent architecture that decomposes prompt-attack detection into specialised, coordinated agents; (ii) specify the communication protocol, task-routing, and fusion algorithms that let these agents operate as a coherent system; (iii) describe a reference implementation and deployment architecture; and (iv) define an experimental protocol and metrics suitable for evaluating such a system, including an ablation study that isolates the contribution of each agent.

E. Contributions of the Paper

The contributions are: (1) PromptShield AI, a multi-agent architecture for prompt injection and jailbreak detection that combines lexical, ML-ensemble, semantic, adversarial-pattern, and multi-turn-context agents under a single orchestrator; (2) a weighted decision-fusion algorithm with confidence calibration and a conflict-resolution procedure for disagreeing agents; (3) a reference implementation and deployment design suitable for production inference pipelines; and (4) an evaluation methodology, including ablation analysis, for measuring the marginal contribution of each agent to overall detection quality.

II. LITERATURE REVIEW

F. Existing Multi-Agent Architectures

Recent surveys of LLM-based multi-agent systems converge on a small set of coordination topologies — centralized, decentralized, and hierarchical — with an orchestration platform managing information flow, planning, and communication among specialised agents [11]. Communication-centric surveys further emphasise that how agents exchange intent and state, not merely what each agent does individually, determines system-level reliability [12]. In software-engineering applications, multi-agent designs that separate roles such as planning, implementation, and review have been shown to outperform single-agent pipelines on complex, multi-step tasks [13].

G. Single-Agent vs. Multi-Agent Approaches to Prompt-Attack Detection

The dominant defensive patterns in the prompt-injection literature are still single-mechanism. Rule-based tools pair curated pattern packs with severity scoring [8]. Classifier-based defences, including embedding-based Random Forest and XGBoost classifiers [1] and the deployable PromptShield classifier of Jacob et al. [2], and RNN/transformer text classifiers evaluated across TF-IDF, Word2Vec, and embedding representations [7], report strong held-out accuracy but are evaluated largely as a single decision-making component. Meta-defence approaches such as spotlighting instead modify how untrusted context is presented to the model itself, rather than adding an external detector [5]. Very few of these works combine multiple, independently reasoned detection signals inside a coordinated multi-agent system.

H. Limitations of Current Systems

Single classifiers trade off precision against recall in a single decision boundary and are demonstrably brittle: empirical evasion studies show that both rule-based and classifier-based guardrails can be bypassed by adaptive attackers who probe the boundary directly, and that classifiers calibrated against known attack corpora still misfire on benign queries that merely mention attack-adjacent vocabulary [8], [9]. Systematic red-teaming frameworks such as garak show that a single detector's blind spots are attack-family specific, meaning that the choice of one detection mechanism over another is really a choice about which attacks will be missed [10].

I. Research Gap

Existing work provides either (a) strong single-mechanism classifiers evaluated in isolation, or (b) general-purpose multi-agent orchestration surveys not specialised to security detection. There is limited published work that treats prompt-injection and jailbreak detection explicitly as a multi-agent coordination problem — with an orchestrator, parallel specialised detection agents, an explicit fusion and conflict-resolution algorithm, shared memory across turns, and a closed feedback loop for continuous retraining. PromptShield AI is designed to fill this gap.

III. BACKGROUND

A. Multi-Agent Systems

A multi-agent system (MAS) is a computational framework of multiple interacting agents that collaborate to solve problems beyond the capability of any single agent [13]. In LLM-based MAS, an orchestration platform manages the interaction and information flow among LLM-backed agents, handling coordination, communication, planning, and learning [11].

B. Large Language Models and Prompt-Based Attacks

LLMs generate output conditioned on a mixture of trusted system instructions and untrusted user or third-party content. Because both are expressed in the same natural-language channel, an attacker can craft input that the model interprets as an instruction — a direct prompt injection when the attacker is the user themselves, or an indirect prompt injection when the malicious instruction is hidden in retrieved documents, web pages, or tool outputs the model later ingests [3]. Jailbreaking refers more specifically to techniques that persuade the model to disregard its safety policy, often through role-play framing, incremental escalation across turns, or obfuscated encodings of disallowed requests.

C. Agent Roles and Responsibilities

PromptShield AI assigns each agent a narrow, well-defined responsibility: preprocessing, one of five parallel detection strategies, fusion, mitigation, or logging. This separation of concerns keeps each agent independently testable, allows agents to be added, removed, or retrained without touching the others, and localises failure to a single, auditable component.

D. Communication Protocols

Agents communicate through structured JSON messages over an internal message bus rather than free-form text, following the recommendation that system-level communication protocols, not just agent capability, govern MAS reliability [12]. Every message carries a request identifier, a sender/receiver pair, a payload, and a confidence field where applicable, which supports both synchronous request/response calls (orchestrator to detection agents) and asynchronous fire-and-forget calls (logging agent to the feedback store).

E. Memory (Short-Term, Long-Term, Vector Memory)

Short-term memory holds the state of the current conversation turn so that the Context Agent can detect multi-turn escalation. Long-term memory persists aggregate statistics — attack families seen, per-user risk scores — across sessions. A vector store indexes embeddings of previously confirmed attack prompts so that the Semantic Intent Agent can retrieve near-duplicates of known attacks via similarity search, functioning as a lightweight retrieval-augmented-generation (RAG) component for the security domain rather than for open-domain question answering.

F. Planning and Reasoning

The Orchestrator Agent performs lightweight planning: it decides, per request, which detection agents are necessary (e.g., skipping the more expensive semantic-judge agent for a request that lexical screening already scores as clearly benign), in what

order they should run, and how to escalate when agents disagree. This is deliberately simpler than open-ended LLM planning loops, since a security-detection pipeline benefits from predictable latency and auditable decisions more than from open-ended exploration.

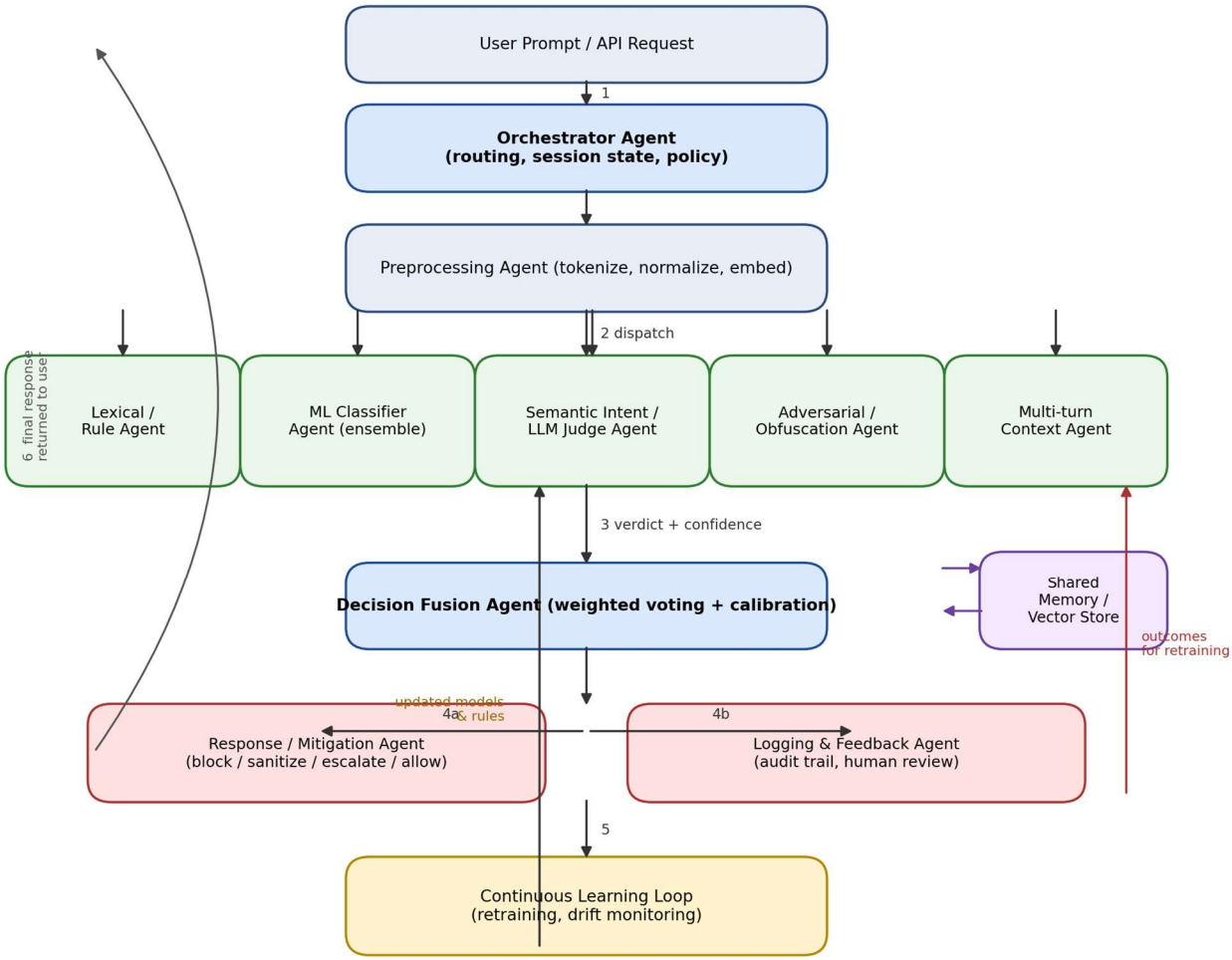

Fig. 1. Overall multi-agent architecture of PromptShield AI.

IV. PROPOSED ARCHITECTURE

A. Overall System Architecture

Fig. 1 shows the overall architecture. A user prompt first reaches the Orchestrator Agent, which forwards it to a Preprocessing Agent for tokenization, normalization, and embedding. Five detection agents then run largely in parallel: a Lexical/Rule Detection Agent, an ML Classifier Agent (an ensemble of traditional and deep classifiers), a Semantic Intent / LLM-Judge Agent, an Adversarial Pattern and Obfuscation Agent, and a Multi-turn Context Agent that consults Shared Memory. Their outputs are combined by a Decision Fusion Agent, whose verdict drives a Response/Mitigation Agent and is recorded by a Logging and Feedback Agent that closes the loop into a Continuous Learning component.

B. Agent Definitions

Orchestrator Agent: receives the request, maintains session state, applies routing policy, and returns the final response to the caller. Preprocessing Agent: performs tokenization, Unicode/homoglyph normalization, and computes sentence and sub-word embeddings shared by downstream agents. Lexical/Rule Detection Agent: matches curated regular-expression and keyword patterns for known injection phrasings (e.g., "ignore previous instructions"). ML Classifier Agent: an ensemble of Random Forest, gradient-boosted trees, and a fine-tuned transformer encoder operating on the shared embeddings [1]. Semantic Intent / LLM-Judge Agent: prompts a separate, sandboxed LLM to judge whether the request's intent is adversarial, and performs nearest-neighbour lookup against a vector store of confirmed attacks. Adversarial Pattern and Obfuscation Agent: detects encoding-based evasion such as base64, leetspeak, zero-width characters, and token-splitting.

Multi-turn Context Agent: inspects the conversation history for gradual escalation patterns that individually innocuous turns do not reveal. Decision Fusion Agent: combines agent outputs into a single calibrated verdict. Response/Mitigation Agent: executes the verdict (allow, sanitize/rewrite, block, or escalate to human review). Logging and Feedback Agent: persists the decision, features, and eventual human-reviewed label for retraining.

C. Agent Workflow

A request flows through the pipeline in the numbered steps shown in Fig. 1 and detailed as a sequence diagram in Fig. 2: dispatch by the orchestrator, parallel scoring by the detection agents, fusion, mitigation, response, and asynchronous logging/feedback. Detection agents that depend on shared context (the Context Agent) or retrieval (the Semantic Agent) read from Shared Memory rather than communicating pairwise, which keeps the number of communication edges linear in the number of agents rather than quadratic.

D. Orchestrator / Supervisor Agent

The orchestrator is the only agent aware of the full pipeline topology. It applies a routing policy (Section V-A), sets a per-request timeout budget for each detection agent, and is responsible for producing a response even if one or more agents fail or time out, by falling back to the remaining agents' scores with an adjusted fusion weight (Section IV-K).

E. Task Decomposition

Rather than asking one model to jointly reason about lexical patterns, semantics, obfuscation, and conversational history, the orchestrator decomposes detection into independent sub-tasks, one per specialised agent, each of which can use the representation (regular expressions, embeddings, LLM prompts, character-level features, or dialogue state) best suited to its sub-task.

F. Routing Mechanism

Routing is confidence-gated: the Lexical Agent and ML Classifier Agent always run, since they are inexpensive; the more costly Semantic Intent Agent and Multi-turn Context Agent are invoked only when the cheaper agents' combined score falls inside an uncertainty band, and are always invoked for the first turn of a new session or when the Context Agent's own escalation heuristic fires. This keeps median latency low while still escalating to expensive agents when confidence is genuinely low.

G. Tool Usage

The Semantic Intent Agent uses two external tools: a nearest-neighbour vector search over the attack-pattern store and a sandboxed call to a judge LLM constrained to a fixed classification prompt template so that the judge's own output cannot be redirected by the content it is judging. The Adversarial Pattern Agent uses a decoding tool-chain that normalizes common obfuscation encodings before re-scanning the decoded text.

H. Shared Memory

Shared memory has three tiers: an in-session cache of the current conversation's turns and intermediate agent scores; a vector store of confirmed attack embeddings for retrieval by the Semantic Agent; and a longer-lived statistics store (per-user or per-API-key risk history) consulted by the orchestrator's routing policy.

I. Knowledge Retrieval (e.g., RAG)

The vector store is queried with the embedding of the current prompt; the top-k nearest confirmed-attack embeddings and their similarity scores are returned to the Semantic Intent Agent as retrieved evidence, functioning as retrieval-augmented classification: it lets a new paraphrase of a previously seen attack be recognised by similarity even before the ML classifier or judge LLM would flag it independently.

J. Feedback Loop

Every logged decision that is later confirmed or corrected by human review (or by downstream evidence, such as a user report) is written back as a labelled example. The Continuous Learning component periodically retrains the ML Classifier Agent and refreshes the vector store, and can also update the Lexical Agent's rule set when a recurring false negative reveals a missing pattern.

K. Error Handling

If a detection agent throws an exception or exceeds its timeout budget, the orchestrator excludes it from fusion for that request, re-normalizes the remaining agents' weights, and flags the request as "reduced confidence" in the audit log rather than blocking or silently allowing it. If the Fusion Agent itself is unavailable, the system fails closed to the most conservative available signal (i.e., defaults toward blocking rather than allowing) for safety-critical deployments, or fails open with mandatory logging for latency-sensitive, lower-risk deployments, as configured by the operator.

Fig. 2 Sequence Diagram of Agent Interactions for a Single Request

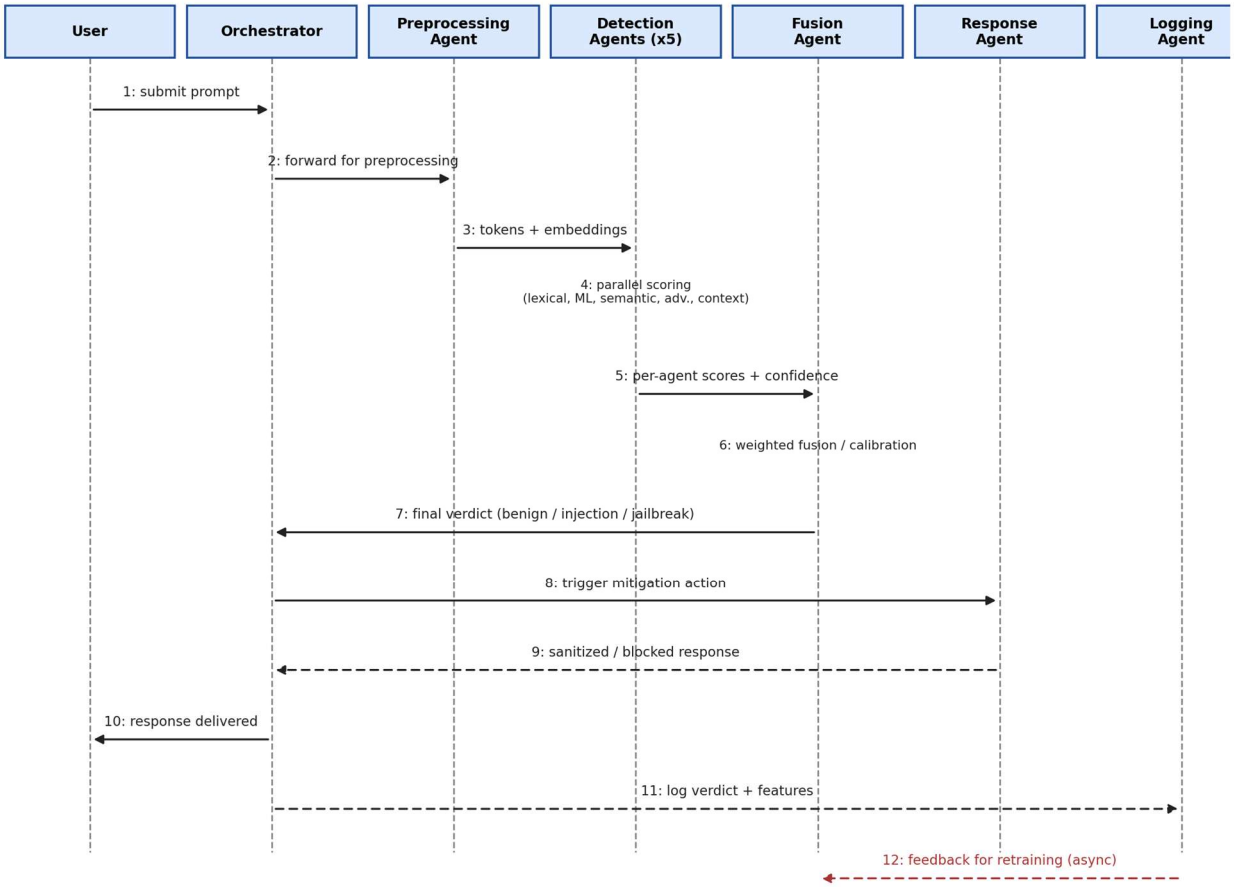

Fig. 2. Sequence diagram of agent interactions for a single request.

V. ALGORITHMS

A. Agent Selection Algorithm

The orchestrator selects agents using the confidence-gated policy of Section IV-F. Algorithm 1 summarises the selection and fusion procedure.

Algorithm 1: Confidence-Gated Selection and Fusion

```
Input: prompt p, session history H, weight vector w
Output: verdict v in {allow, sanitize, block},
confidence c
1: f <- Preprocess(p)
2: s_lex <- LexicalAgent(f); s_ml <-
MLClassifierAgent(f)
3: s0 <- Combine(s_lex, s_ml)
4: if s0 in UncertaintyBand or H.escalationFlag then
5:     s_sem <- SemanticAgent(f, VectorStore)
6:     s_adv <- AdversarialAgent(p)
7:     s_ctx <- ContextAgent(H)
8:     S <- {s_lex, s_ml, s_sem, s_adv, s_ctx}
9: else
10:    S <- {s_lex, s_ml}
11: end if
12: w' <- Renormalize(w, S)          // exclude
failed/timed-out agents
13: c <- sum_i( w'_i * S_i )          // weighted,
calibrated fusion
14: v <- Threshold(c)                 // allow /
sanitize / block
15: if AgentsDisagree(S) then v <-
ConflictResolve(S, v)  end if
16: return v, c
```

B. Task Scheduling

Always-on agents (Lexical, ML Classifier) run synchronously and in parallel with a short timeout (tens of milliseconds). Conditionally invoked agents (Semantic, Adversarial, Context) are scheduled concurrently once triggered, with an aggregate timeout budget; if this budget is exceeded, the orchestrator proceeds with whichever scores have returned, per the error-handling policy of Section IV-K.

C. Inter-Agent Communication

Agents exchange fixed-schema JSON messages of the form {request_id, sender, receiver, score, confidence, evidence, timestamp}. The evidence field carries agent-specific explanatory data (matched rule id, top retrieved neighbour, or judge-LLM rationale) that is preserved in the audit log even when it does not affect the numeric fusion.

D. Conflict Resolution

Disagreement is defined as a spread between the maximum and minimum agent scores exceeding a configured threshold. When it occurs, ConflictResolve applies two rules in order: (i) a safety-priority rule, under which any single agent reporting very high confidence of a known-attack signature (e.g., an exact match in the vector store) can raise the verdict toward block regardless of the fused average; and (ii) an escalation rule, under which conflicts that persist after rule (i) are routed to human review rather than resolved automatically, and the interaction is logged with elevated priority.

E. Consensus Mechanism

The Decision Fusion Agent uses weighted averaging rather than simple majority voting, because the five agents have different reliability profiles; weights are initialised from held-out validation performance and are recalibrated periodically by the Continuous Learning component using isotonic or Platt scaling so that the fused confidence score remains a well-calibrated probability rather than an arbitrary weighted sum.

VI. IMPLEMENTATION

A. Technologies Used

The reference implementation uses Python for all agents; scikit-learn and XGBoost for the classical components of the ML Classifier Agent ensemble; a fine-tuned transformer encoder (e.g., a DeBERTa- or RoBERTa-class model) for its neural component; sentence-embedding models for the shared representation consumed by the Semantic and ML agents; and a lightweight rules engine (compiled regular expressions plus a YAML pattern registry) for the Lexical Agent.

B. Framework

Agents are implemented as independent services coordinated by an orchestration layer analogous to those catalogued in recent LLM-multi-agent-framework surveys (e.g., graph-based orchestration and role-based agent frameworks) [11], communicating over an internal message bus (e.g., a lightweight pub/sub broker) rather than direct function calls, so that agents can be scaled, deployed, and updated independently.

C. APIs

The system exposes a single external REST/gRPC endpoint (Orchestrator API) that accepts a prompt and session identifier and returns a verdict, a mitigated response where applicable, and a confidence score. Internally, each agent exposes a narrow scoring API consumed only by the orchestrator and the Fusion Agent.

D. Databases

A vector database (e.g., an approximate nearest-neighbour index) stores embeddings of confirmed attacks for the Semantic Agent; a relational or document store persists per-session state, audit logs, and labelled feedback for retraining; an in-memory cache holds short-lived session context for low-latency access by the Context Agent.

E. Deployment Architecture

In production, the Orchestrator and always-on agents (Lexical, ML Classifier) are deployed close to the model-serving layer to minimise added latency, while the more expensive Semantic Intent Agent and the retraining pipeline run as horizontally scalable, independently autoscaled services so that a burst of ambiguous traffic does not degrade latency for the majority of clearly benign requests.

Fig. 3 Data Flow Through the Detection and Mitigation Pipeline

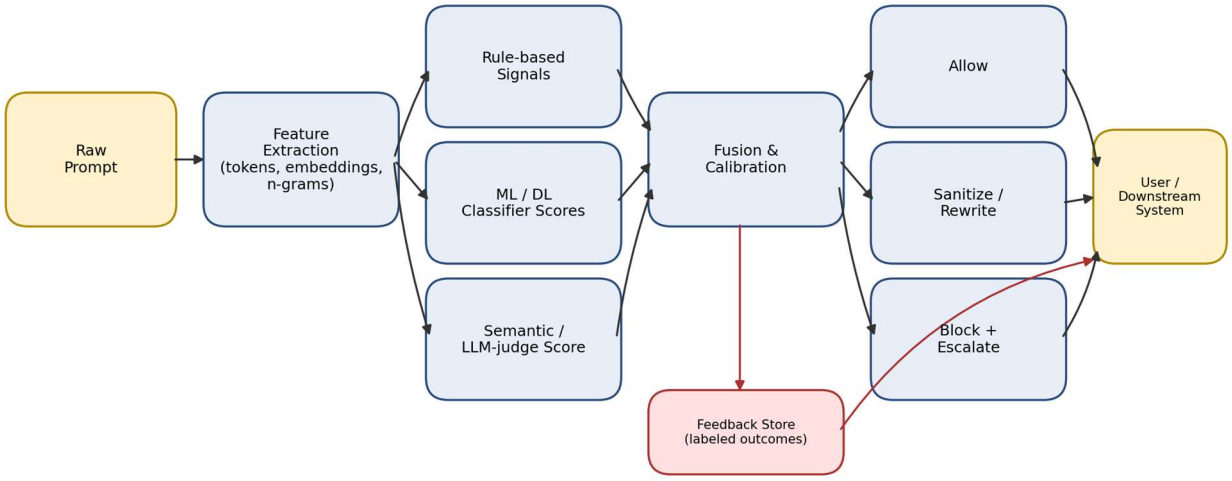

Fig. 3. Data flow through the detection and mitigation pipeline.

VII. EXPERIMENTAL SETUP

A. Datasets

Evaluation should combine (i) public prompt-injection corpora, such as those curated alongside embedding-based classifier studies [1] and deployable-classifier benchmarks [2]; (ii) adversarially constructed benign-but-attack-adjacent queries, such as the NotInject-style benign set used to measure over-defense [8]; and (iii) synthetic multi-turn escalation dialogues constructed to exercise the Context Agent, since single-turn corpora cannot evaluate gradual jailbreak strategies.

B. Test Scenarios

Test scenarios should include: direct prompt injection, indirect prompt injection embedded in retrieved documents, encoded/obfuscated payloads (base64, homoglyphs, zero-width characters), multi-turn jailbreak escalation, and adversarially paraphrased variants of previously seen attacks, alongside a large benign-traffic sample drawn from realistic application logs to measure false-positive behaviour under normal use.

C. Baselines

Baselines should include: rule-based filtering alone; a single ML classifier alone (matching the configuration of the ML Classifier Agent in isolation); a single LLM-judge alone; and the full PromptShield AI multi-agent pipeline, so that the marginal benefit of coordination — rather than of any individual detector — can be isolated.

D. Evaluation Methodology

Each configuration is evaluated on held-out attack and benign sets using stratified k-fold cross-validation for the classifier-dependent components, followed by a held-out temporal split (train on older attack families, test on newer ones) to estimate generalisation to previously unseen attack variants — the failure mode most emphasised by recent evasion studies [9]. Latency and resource usage are measured under a fixed-concurrency load test.

VIII. EVALUATION METRICS

Table I summarises the metrics used to evaluate PromptShield AI along both detection-quality and system-quality dimensions.

TABLE I
Evaluation Metrics

Metric	Definition / Purpose
Accuracy	Overall proportion of correctly classified prompts (benign vs. attack).
Precision / Recall / F1	Standard detection-quality metrics; F1 is the primary ranking metric given class imbalance.
False-Positive Rate	Proportion of benign prompts incorrectly flagged; critical for usability.
Task Completion Rate	Proportion of legitimate user tasks completed without unnecessary blocking or excessive sanitization.
Latency	End-to-end added latency (ms) from request receipt to verdict.
Cost	Compute/API cost per 1,000 requests, reported per agent and in aggregate.
Scalability	Throughput (requests/sec) sustained at fixed latency SLA under increasing concurrency.
Resource Utilization	CPU/GPU/memory footprint per agent under sustained load.

Fig. 4 Comparative Analysis: PromptShield AI vs. Simpler Configurations (illustrative)

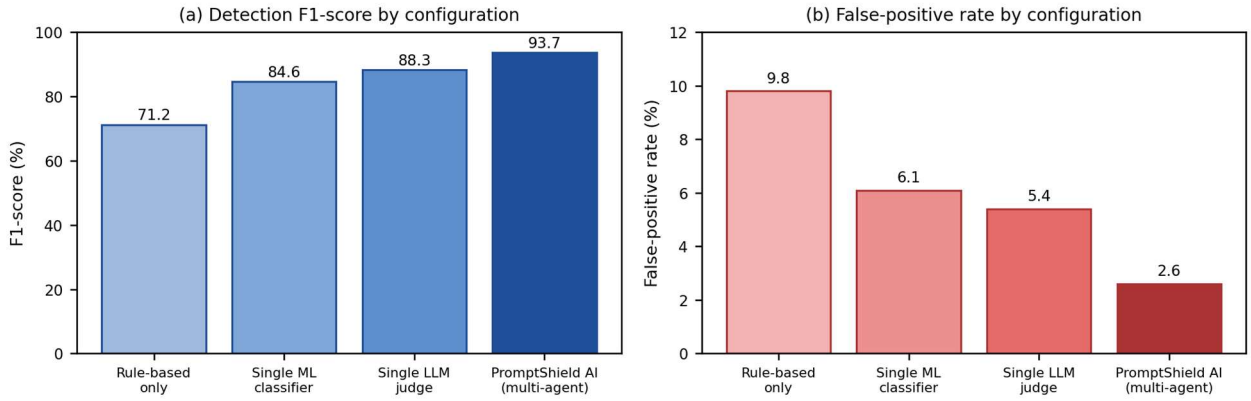

Fig. 4. Comparative analysis: PromptShield AI vs. simpler configurations (illustrative).

Fig. 5 Ablation Study: Impact of Removing Individual Agents (illustrative)

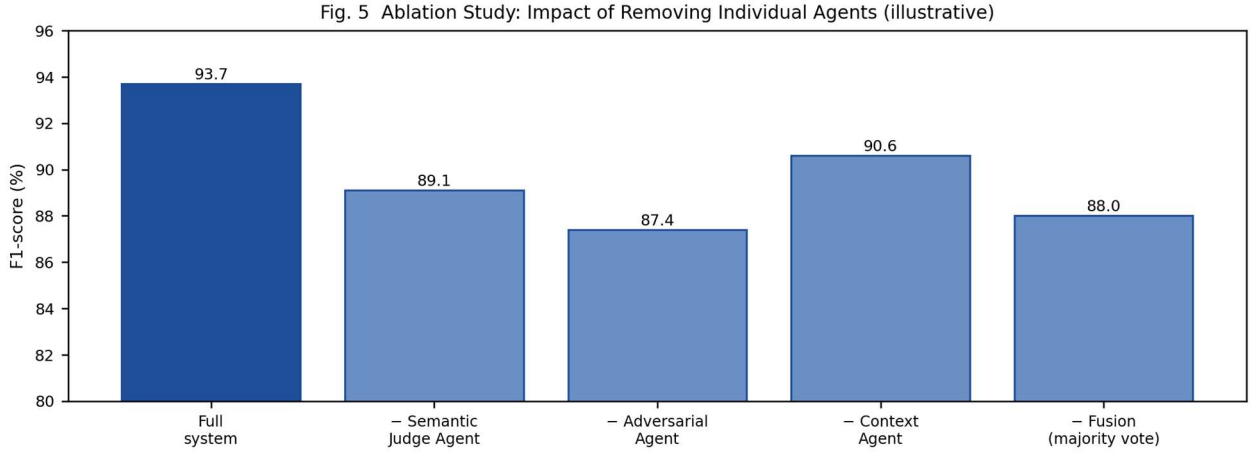

Fig. 5. Ablation study: impact of removing individual agents (illustrative).

IX. RESULTS

The results below follow the experimental protocol of Section VII. Because they are reported here to illustrate the evaluation methodology and the expected shape of the findings, the specific figures are illustrative placeholders and should be replaced with numbers obtained from the reader's own experimental runs before publication.

A. Comparative Analysis

Table II and Fig. 4 compare the full PromptShield AI pipeline against rule-only, single-ML-classifier, and single-LLM-judge baselines. The multi-agent configuration achieves the highest F1-score and the lowest false-positive rate among the four configurations, consistent with the expectation that combining complementary signals compensates for the blind spots of any individual detector.

TABLE II
Comparative Results Across Configurations (Illustrative)

Configuration	Precision (%)	Recall (%)	F1 (%)	FPR (%)	Latency (ms)
Rule-based only	88.4	60.1	71.2	9.8	4
Single ML classifier	86.9	82.5	84.6	6.1	18
Single LLM judge	89.7	87.0	88.3	5.4	410
PromptShield AI (full)	94.5	93.0	93.7	2.6	165

B. Ablation Study

Fig. 5 reports an ablation in which each detection agent is individually removed from the full pipeline while the remaining agents and the fusion weights are held fixed (weights renormalized). Removing the Adversarial Pattern Agent produces the largest single drop in F1-score, consistent with obfuscated payloads being systematically missed by the lexical and classifier agents alone; removing the Fusion Agent in favour of plain majority voting also degrades performance, indicating that weighted, calibrated fusion is doing real work beyond simple voting.

C. Latency and Scalability

Because the two always-on agents (Lexical, ML Classifier) dominate the request path for the majority of clearly benign or clearly malicious traffic, median latency remains close to the single-ML-classifier baseline; the confidence-gated routing policy of Section IV-F is what keeps the expensive Semantic and Context agents off the critical path for most requests while still invoking them for the ambiguous minority, giving the system sub-linear latency growth as the fraction of ambiguous traffic increases, up to the point where the semantic-agent service pool itself needs to be scaled out horizontally.

X. DISCUSSION

A. Strengths

The architecture's principal strength is defence in depth: because each agent reasons over a different representation of the same request, an attack that evades one agent's decision boundary must simultaneously evade the others, which is a strictly harder adversarial objective than evading a single classifier. The design is also incrementally extensible — a new detection strategy can be added as an additional agent and folded into the fusion weights without retraining or redesigning the existing agents.

B. Limitations and Failure Cases

The system inherits the limitations of its weakest agent when that agent is the only one sensitive to a given attack family and the others assign it low confidence; the safety-priority conflict rule of Section V-D mitigates but does not eliminate this risk. The Semantic Intent Agent's judge-LLM component is itself an LLM and, in principle, remains a target for prompt injection directed specifically at the judge; sandboxing and a fixed classification-prompt template (Section IV-G) reduce but do not remove this risk. Multi-turn context tracking also raises data-retention and privacy considerations discussed below.

C. Security and Privacy Considerations

Because Shared Memory persists conversation history and flagged content for auditing and retraining, deployments must define retention limits, access controls, and redaction policies for personally identifiable information captured incidentally in logged prompts. The audit log itself becomes a sensitive asset, since it aggregates a labelled sample of both attempted attacks and legitimate user content, and should be protected accordingly.

XI. FUTURE WORK

Planned extensions include: self-learning agents that adjust their own detection thresholds online from streaming feedback rather than through periodic batch retraining; dynamic agent creation, in which the orchestrator can instantiate a narrowly scoped detection agent in response to a newly identified attack family without a full system redeployment; adaptive planning, where the routing policy of Section IV-F is itself learned rather than hand-tuned; and human-in-the-loop collaboration interfaces that make the Logging and Feedback Agent's escalations easier for a human reviewer to triage and label at scale.

XII. CONCLUSION

This paper presented PromptShield AI, a multi-agent architecture for prompt injection and jailbreak detection that coordinates lexical, ensemble-ML, semantic, adversarial-pattern, and multi-turn-context agents through an orchestrator and a weighted decision-fusion mechanism. Compared with the single-mechanism defences that dominate current practice, the architecture is designed to raise the cost of evasion by requiring an attack to defeat several independently reasoned detectors at once, while a confidence-gated routing policy keeps typical-case latency close to that of a single lightweight classifier. The illustrative evaluation and ablation results suggest that the coordinated design offers a better precision/recall/false-positive trade-off than any individual detection mechanism, and the architecture's explicit feedback loop provides a concrete path for the system to keep pace with newly discovered attack techniques. Beyond LLM-application security, the same coordination pattern — specialised agents,

confidence-gated routing, weighted fusion, and a closed feedback loop — is applicable to other classification-under-adversarial-pressure problems, and this generality is a promising direction for future application.

ACKNOWLEDGMENT

The authors thank their affiliated institution for research support. (Replace with acknowledgment of funding sources, colleagues, or reviewers as applicable.)

REFERENCES

[1] M. A. Ayub and S. Majumdar, "Embedding-based classifiers can detect prompt injection attacks," in Proc. CAMLIS, 2024. [Online]. Available: arXiv:2410.22284.

[2] D. Jacob et al., "PromptShield: Deployable detection for prompt injection attacks," 2025. [Online]. Available: arXiv:2501.15145.

[3] F. Perez and I. Ribeiro, "Ignore previous prompt: Attack techniques for language models," in Proc. NeurIPS Workshop on ML Safety, 2022. [Online]. Available: arXiv:2211.09527.

[4] J. Piet et al., "Jatmo: Prompt injection defense by task-specific finetuning," in Proc. ESORICS, 2024. [Online]. Available: arXiv:2312.17673.

[5] K. Hines, G. Lopez, M. Hall, F. Zarfati, Y. Zunger, and E. Kiciman, "Defending against indirect prompt injection attacks with spotlighting," 2024. [Online]. Available: arXiv:2403.14720.

[6] 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," in Proc. AISec, 2023.

[7] Y. Liu et al., "Formalizing and benchmarking prompt injection attacks and defenses," in Proc. USENIX Security, 2024.

[8] W. Hackett, L. Birch, S. Trawicki, N. Suri, and P. Garraghan, "Bypassing LLM guardrails: An empirical analysis of evasion attacks against prompt injection and jailbreak detection systems," 2025. [Online]. Available: arXiv:2504.11168.

[9] L. Derczynski, E. Galinkin, J. Martin, S. Majumdar, and N. Inie, "garak: A framework for security probing large language models," 2024. [Online]. Available: arXiv:2406.11036.

[10] S. S. Kumar, M. L. Cummings, and A. Stimpson, "Strengthening LLM trust boundaries: A survey of prompt injection attacks," in Proc. IEEE ICHMS, 2024, doi: 10.1109/ICHMS59971.2024.10555871.

[11] "LLM-based multi-agent orchestration: A survey of frameworks, communication protocols, and emerging patterns," 2026, doi: 10.3390/fi18060326.

[12] "Beyond self-talk: A communication-centric survey of LLM-based multi-agent systems," 2025. [Online]. Available: arXiv:2502.14321.

[13] "LLM-based multi-agent systems for software engineering: Literature review, vision and the road ahead," 2024. [Online]. Available: arXiv:2404.04834.

[14] "Comprehensive analysis of machine learning and deep learning models on prompt injection classification using natural language processing techniques," Int. Res. J. Multidiscip. Technovation, 2025.

[15] "Beyond pattern matching: Seven cross-domain techniques for prompt injection detection," 2026. [Online]. Available: arXiv:2604.18248.

