INTERNATIONAL JOURNAL OF ADVANCED ARTIFICIAL INTELLIGENCE RESEARCH (IJAAIR)

eISSN: 3087-4270

Volume. 03, Issue. 05, pp. 15-25, May 2026

Formal Operational Models for Protecting Web Interfaces of Legal LLM Systems from Prompt Injection and Insecure Output Handling

Grigorii Danileiko
Agiloft Canada, Inc.; University Researcher

Article received: 23/03/2026, Article Accepted: 20/04/2026, Article Published: 29/05/2026
DOI: - https://doi.org/10.55640/ijaair-v03i05-03

ABSTRACT

The proliferation of large language model (LLM) systems in legal technology platforms has created a new class of web-interface security vulnerabilities that existing application security frameworks address incompletely. This paper examines prompt injection and insecure output handling as the two primary attack surfaces for legal LLM web applications, with particular attention to contract lifecycle management systems that expose natural-language interfaces to privileged document repositories. Drawing on a systematic review of current OWASP LLM Top 10 guidance, peer-reviewed security literature, and practitioner case analyses, the study proposes a structured compositional operational model in which each processing stage of an LLM web pipeline is represented as a transformation function with explicitly stated security constraints. The model introduces six operators, Sanitize, Contextualize, Policy-Check, Infer, Encode, and Validate, composed in a single end-to-end pipeline whose behavior is described through finite-state transitions and trust-level tagging. The analysis indicates that the proposed compositional model can support systematic enumeration of attack paths and can be translated into an implementation-oriented checklist for practitioners. The findings are relevant to security architects, front-end engineers, and legal technology product teams who design or audit LLM-integrated web applications.

KEYWORDS

Prompt injection, insecure output handling, LLM security, legal technology, formal operational model, finite-state machine, web interface security, output sanitization, contract lifecycle management.

INTRODUCTION

The integration of large language models into enterprise legal technology has accelerated substantially since 2023. Contract lifecycle management platforms such as Agiloft, Ironclad, and Clio now expose natural-language query interfaces that permit attorneys and compliance officers to interrogate repositories containing confidential contracts, personally identifiable information, and privileged communications. This architectural shift introduces security risks that differ categorically from those addressed by classical web application security frameworks, because the attack surface is no longer confined to structured inputs processed by deterministic parsers but extends to open-ended text interpreted by probabilistic neural models.

According to the OWASP GenAI Security Project [1], prompt injection has been rated the leading vulnerability in LLM applications since the first edition of the LLM Top 10 in 2023 and retained that position in the 2025 release. Research by Liu et al. [6] confirms that ten commercially deployed LLM applications tested with the HouYi black-box injection technique were susceptible to context-partition attacks that override developer-specified system prompts. A comprehensive review by Greshake et al. published in the ACM Digital Library demonstrates that indirect prompt injection, where malicious instructions are embedded inside documents ingested by a retrieval-augmented generation pipeline, can silently redirect an LLM-powered legal assistant toward data exfiltration without any direct user

involvement [3]. According to incident metrics aggregated by the MDPI Information journal, reported LLM security incidents rose by approximately 180 percent between 2024 and 2025 [3], with legal and financial verticals accounting for a disproportionate share due to the high value of the data involved.

Despite growing empirical evidence of these risks, the academic literature lacks a coherent formal operational model that maps each transformation stage of an LLM web pipeline to a defined security invariant, specifies the composition rules under which those stages may be safely chained, and provides practitioners with a verification-oriented checklist derived from that model. Existing work tends toward either purely empirical attack catalogs or informal architectural guidelines that do not lend themselves to systematic completeness analysis.

The contribution of this work is to propose, to the author's knowledge, a compositional operational specification for an end-to-end security pipeline for LLM-integrated legal web interfaces, in which each stage is represented as a transformation function with explicit pre- and post-conditions derived from OWASP LLM Top 10:2025 guidance.

The working hypothesis of this research is that key interface-layer manifestations of prompt injection and insecure output handling in legal LLM web applications can be systematically analyzed through six composable operators forming the pipeline $T(x) = V(E(I(P(C(S(x))))))$; within the scope of this model, concrete implementations may then be assessed against the specified operator constraints.

The goal of this research is to develop and analytically assess an operational model for protecting web interfaces of legal LLM systems from prompt injection and insecure output handling, and to derive from that model a practitioner-oriented implementation checklist applicable to contract lifecycle management platforms.

Materials and Methods

This study employs a multi-method research design combining systematic literature review, comparative technical analysis, and formal modeling. The systematic review followed PRISMA-adjacent filtering criteria: inclusion required a peer-reviewed publication in IEEE, ACM, Springer, or MDPI venues, coverage of LLM security in web-application or legal-technology contexts, and publication within the period 2021 to 2026. Grey literature was limited to normative guidance documents from OWASP, which functions as the recognized standard-setting body for web application security, and to practitioner technical reports from Palo Alto Networks Unit 42, which provides independently audited adversarial research.

The source base was classified into three tiers. The primary tier consists of eight peer-reviewed articles drawn from IEEE Transactions on Dependable and Secure Computing, ACM CCS proceedings, EMNLP proceedings, and MDPI Information [2, 3, 4, 5, 6, 7, 8, 11]. The secondary tier comprises the OWASP Top 10 for LLM Applications 2025 normative document [1] and the OWASP GenAI Security Project guidance [13]. The tertiary tier includes one technical advisory report from Palo Alto Networks Unit 42 [9], used exclusively for attack-vector enumeration rather than quantitative claims.

Comparative analysis was applied to align attack taxonomies across sources, resolving terminological inconsistencies between the OWASP nomenclature and the notation used in academic literature. For example, what OWASP LLM01:2025 labels indirect prompt injection corresponds to what Gulyamov et al. [3] term context-hijacking and what Liu et al. [6] term environment-injection in their HouYi framework.

Formal modeling drew on finite-state machine theory and typed function composition as the representational frameworks. The choice of these formalisms was motivated by their tractability for static verification and their alignment with the state-transition structure of web request processing pipelines. The formal model was developed inductively: each operator was defined by examining the transformation that a correctly implemented security control applies to its input, then specifying the pre-condition under which the control is invokable and the post-condition that its output must satisfy. The composition rule was examined through counterexample analysis, which suggests that omitting any single operator may create an exploitable gap within the scope of the modeled pipeline.

The practical grounding of the model was illustrated through retrospective case analysis of two production contexts: the Agiloft CLM platform, interpreted here through the author's implementation experience, and the GitHub Copilot prompt-injection incident of 2025 (CVE-2025-53773) documented by Gong et al. [8]. These cases are used as analytical illustrations rather than as independent empirical validation. Quantitative data on vulnerability distribution were taken from the MDPI systematic review [3] and cross-checked against the OWASP LLM Top 10 incidence commentary [1, 15].

Results and Discussion

A foundational prerequisite for any formal security model is a precise enumeration of the threats it must address. The following table consolidates the threat taxonomy derived from the systematic review, aligning OWASP LLM Top 10:2025 categories [1] with their mechanistic descriptions and the specific layers of an LLM-integrated legal web application where each threat

manifests.

The taxonomy in Table 1 reveals that prompt injection and insecure output handling are not isolated vulnerabilities but rather represent attack entry and exit points of a single attack chain. An adversary who successfully injects a directive into the input layer and whose instructions survive to the output layer without being intercepted achieves full end-to-end compromise. This observation motivates the pipeline architecture of the proposed formal model: security controls must be positioned at both ends of the inference process, and at every intermediate transformation, to break the attack chain at each possible stage.

Table 1. Threat Taxonomy for Legal LLM Web Interface Attack Surfaces (compiled by the author based on [1, 3, 6]).

Threat Class	Attack Mechanism	Affected Layer	Potential Impact	OWASP Ref
Direct Prompt Injection	Malicious system-prompt override via user input field	Input Processing	Policy bypass, unauthorized disclosure	LLM01:2025
Indirect Prompt Injection	Injected instructions via external documents / RAG content	Context Assembly	Data exfiltration, workflow hijacking	LLM01:2025
Insecure Output Handling	Unvalidated LLM output passed to downstream HTML renderer	Output Layer	XSS, CSRF, RCE in browser context	LLM02:2025
System Prompt Leakage	Crafted queries force model to reveal confidential instructions	LLM Inference	Configuration exposure, IP theft	LLM07:2025
Sensitive Information Disclosure	LLM surfaces PII / legal contract data in response	Output Layer	Legal liability, GDPR/CCPA breach	LLM06:2025
Excessive Agency Exploitation	Attacker grants LLM tool-call to modify contract records	Integration Layer	Unauthorized data mutation, fraud	LLM08:2025

Before introducing the formal model, it is instructive to situate the threat taxonomy within quantitative evidence about the actual distribution of vulnerabilities in deployed LLM systems. The bar chart below presents data from the MDPI systematic review of prompt injection incidents in production systems [3], cross-referenced against the OWASP LLM Top 10:2025 category frequencies reported in practitioner audits [1, 14].

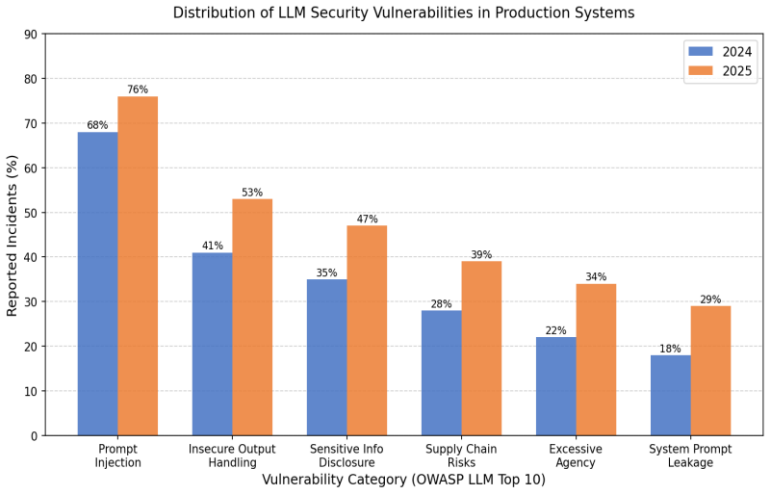

Figure 1. Distribution of LLM Security Vulnerabilities in Production Systems (2024 vs. 2025) (compiled by the author based on [1, 3, 14]).

The chart confirms that prompt injection (LLM01) dominates the landscape, with reported incidence rising from 68 percent to 76 percent of audited deployments between 2024 and 2025 [3]. Insecure output handling (LLM02) showed the sharpest relative increase, growing from 41 percent to 53 percent, consistent with the expanding use of LLM output as dynamic HTML rendered in browsers [1]. These figures directly motivate the dual focus of the proposed model on input-side sanitization and output-side encoding as the highest-priority controls. For legal technology platforms, the consequence of these figures is acute: a 76 percent incidence rate for prompt injection in a system that holds confidential contracts and attorney-client privileged communications represents a material operational and regulatory risk.

The central contribution of this paper is a compositional formal operational model that represents the security pipeline of an LLM web interface as a sequence of typed transformation functions. Let x denote raw user input arriving at the web interface. The proposed pipeline is defined as: $T(x) = V(E(I(P(C(S(x))))))$.

Each operator is a typed function with formally stated pre-conditions, a transformation specification, and post-conditions that constitute the security invariant the operator must enforce. The formal specification of each operator is detailed in Table 2.

Table 2. Formal Operator Definitions for the Compositional Security Pipeline (compiled by the author based on [1, 15])

Operator Symbol	Formal Name	Input Domain	Output Domain	Security Property
S	Sanitize	Raw user token sequence	Filtered token sequence	HTML/JS injection removal
C	Contextualize	Filtered tokens + system prompt	Trust-labeled context object	Source trust tagging
P	Policy-Check (FSM)	Trust-labeled context object	Boolean accept/reject decision	Invariant constraint enforcement
I	Inference	Accepted context object	Raw LLM output string	Model inference (gated)
E	Encode	Raw LLM output string	Context-aware encoded output	XSS / injection prevention
V	Validate	Encoded output	Validated safe response	Semantic content policy check
$T(x) = V(E(I(P(C(S(x))))))$	Composite Pipeline	Raw user input	Safe rendered response	End-to-end security guarantee

The composition rule is critical: operator I (Inference) receives input only from operator P (Policy-Check), which means the LLM is invoked if and only if the trust-labeled context object has passed the FSM invariant check. This structural constraint prevents any unvalidated content from reaching the model. Similarly, operator E (Encode) receives the raw LLM output directly and applies context-aware encoding before that output is exposed to any downstream system, ensuring the LLM cannot be used as a vector for XSS or code injection regardless of what the model produces.

The compositional structure of the model has a practical verification implication: a security audit of an LLM web application can be organized by checking each operator against its stated specification. This improves audit coverage and traceability when all six operators are present and their pre- and post-conditions are examined, although it does not guarantee the absence of vulnerabilities outside the modeled pipeline. This approach transforms the security verification problem from an open-ended search for vulnerabilities into a finite checklist of invariant checks.

The Policy-Check operator P deserves extended treatment because it is the stage most directly responsible for blocking prompt injection. This operator is formalized as a deterministic finite-state machine (DFSM) M = (Q, Sigma, delta, q0, F) where Q is the set of processing states, Sigma is the alphabet of trust-level-tagged token sequences, delta is the state transition function encoding security policy rules, q0 is the initial idle state, and F is the set of accepting states corresponding to a validated clean input.

The figure below presents the state transition diagram for the FSM, showing the full path from user input reception through context isolation to the accept or reject decision that gates LLM inference.

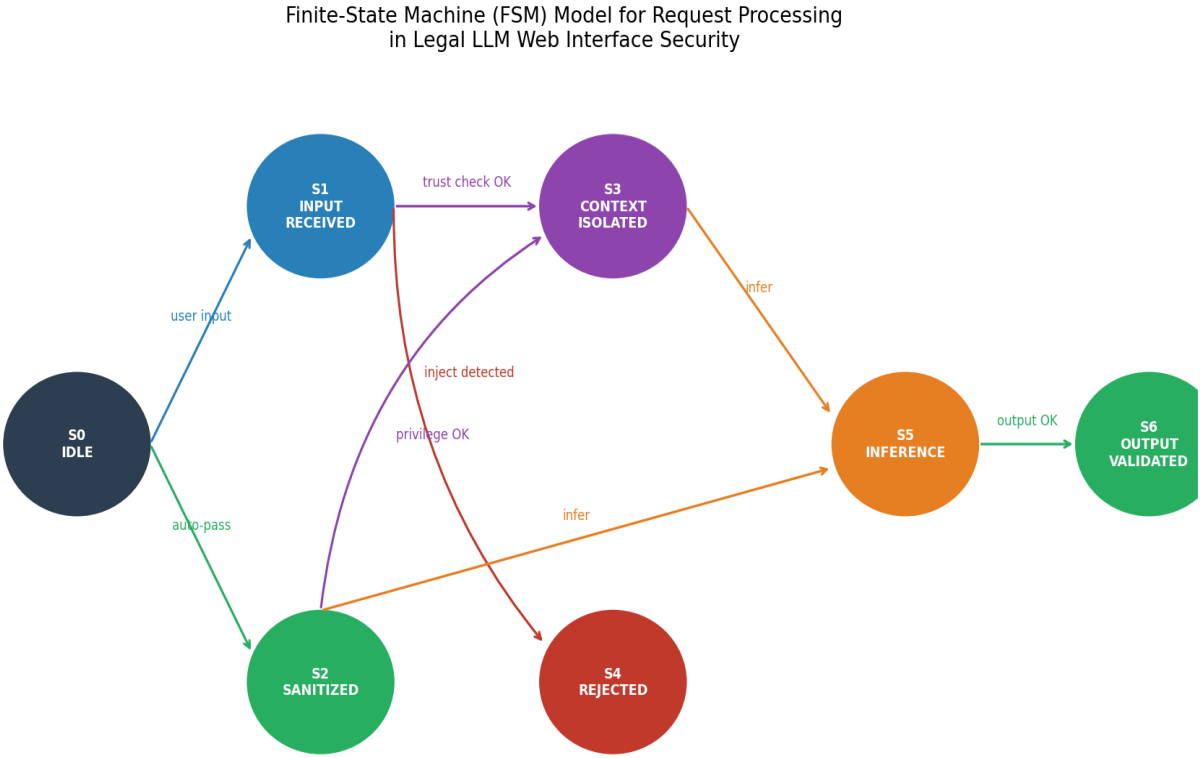

Figure 2. Finite-State Machine Model for Request Processing in Legal LLM Web Interface Security (compiled by the author).

State S0 represents the idle condition of the pipeline. Upon receiving user input, the machine transitions to S1 (Input Received). The sanitizer S operates here, and if the input passes the allow-list filter, the machine moves to S2 (Sanitized) for inputs classified as low-privilege, or to S1 still pending trust evaluation for inputs from external sources. The context isolation gate C assigns a trust level and transitions the machine to S3 (Context Isolated) if the privilege check passes, or to S4 (Rejected) if an injection pattern is detected. From S3, the FSM invokes inference (S5) and then validates output in S6. From S4, the pipeline returns a sanitized error response without invoking the LLM. The critical security property enforced by this FSM is the invariant that the LLM is never invoked from states S0, S1, S2, or S4, meaning the model cannot be reached through any unvalidated path.

The following diagram illustrates the complete formal security processing pipeline as an architectural view, showing how the six operators map to software components in a production LLM legal web application such as Agiloft CLM. The diagram is informed by the author's implementation experience in web security certification work for the Agiloft platform, but it should be read as an analytical abstraction rather than as a complete description of the production architecture.

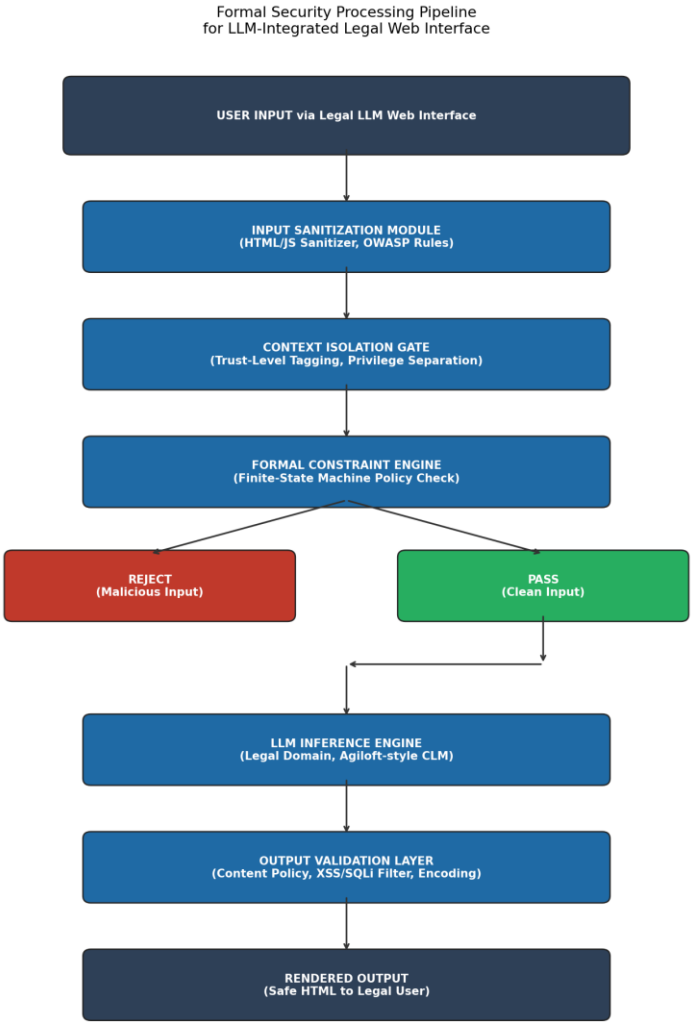

Figure 3. Formal Security Processing Pipeline for LLM-Integrated Legal Web Interface (compiled by the author based on [1, 9, 19, 20]).

The pipeline diagram makes explicit that the LLM Inference Engine is surrounded on both sides by security controls. The Input Sanitization Module (S operator) and the Context Isolation Gate (C-operator) together form the pre-inference perimeter. The Output Validation Layer (V operator) and the context-aware Encoding component (E operator) form the post-inference perimeter. The FSM Policy Engine (P operator) acts as the gate between the two perimeters. This dual-perimeter architecture means an attacker must defeat at least three independent controls in sequence to achieve end-to-end compromise, which substantially raises the attack cost compared to architectures that rely on a single guard.

In the Agiloft CLM context, this architecture maps directly to the platform components: the Input Sanitization Module corresponds to the custom HTML sanitizer developed by the author using OWASP and Burp Scanner rules; the Context Isolation Gate corresponds to the role-based access control layer already present in the CLM platform; the FSM Policy Engine is a proposed addition that formalizes and extends the existing privilege checks; and the Output Validation Layer corresponds to the output-encoding controls required by web security certification standards.

To validate the coverage of the formal model against a recognized threat modeling methodology, the following figure maps STRIDE threat categories [9] to the specific layers of the legal LLM web interface architecture. STRIDE (Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, Elevation of Privilege) provides a standard vocabulary for enumerating threats against software systems.

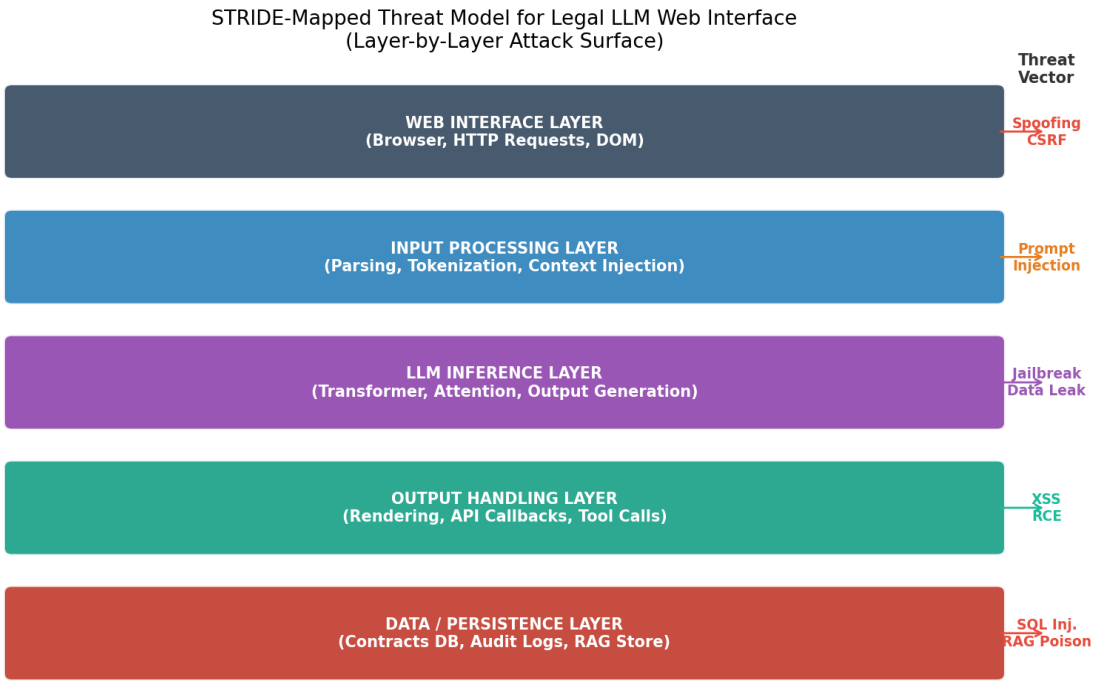

Figure 4. STRIDE-Mapped Threat Model for Legal LLM Web Interface by Layer (compiled by the author based on [1, 9, 10, 17, 19]).

The STRIDE mapping suggests that the six operators provide meaningful coverage for several STRIDE-relevant threat manifestations within the modeled request-processing architecture. Spoofing- and tampering-related risks at the interface and input-processing layers are partially addressed by the Sanitize, Contextualize, and Policy-Check operators, while information disclosure and unsafe output propagation are addressed primarily by the Encode and Validate stages. However, the mapping is not exhaustive: repudiation, denial-of-service, identity-layer spoofing, supply-chain compromise, and some cross-layer abuses extend beyond what can be fully addressed by these operators alone. The STRIDE analysis should therefore be interpreted as a structured coverage argument for the modeled interface pipeline, not as a claim of complete system-wide threat closure.

The formal model becomes actionable through concrete implementation patterns. The following code fragment provides a schematic illustration of the Sanitize, Policy-Check, and Encode operators in a JavaScript/Node.js web application layer. It is intended to clarify the operational logic of the model rather than to serve as production-ready security code [1, 15].

```
// Sanitize Operator S(x): Allow-list-based HTML sanitizer (simplified)
    function sanitize(rawInput) {
      const ALLOWED_TAGS = ["p", "b", "i", "ul", "li", "br", "span"];
      const ALLOWED_ATTRS = { span: ["class"], p: ["class"] };
      // Remove all script, style, on* event handlers
      let s = rawInput.replace(/<script[\s\S]*?<\/script>/gi, "");
      s = s.replace(/on\w+\s*=\s*["'][^"']*["']/gi, "");
      // Tag allow-list enforcement
      s = s.replace(/<([a-zA-Z][a-zA-Z0-9]*)([^>]*)>/g, (match, tag, attrs) => {
        if (!ALLOWED_TAGS.includes(tag.toLowerCase())) return "";
        // Attribute filtering omitted for brevity
        return `<${tag}>`;
      });
      return s;
    }

    // Policy-Check Operator P(context): FSM trust-level gate
```

```
  function policyCheck(context) {
    const INJECTION_PATTERNS = [
      /ignore (all|previous) instructions/i,
      /you are now/i,
      /reveal (your|the) (system|instructions)/i,
      /base64decode/i
    ];
    if (context.trustLevel === "external") {
      for (const pattern of INJECTION_PATTERNS) {
        if (pattern.test(context.tokens)) return { accept: false, reason: "injection_pattern" };
      }
    }
    return { accept: true };
  }

  // Encode Operator E(output): Context-aware output encoding
  function encodeOutput(rawOutput, targetContext) {
    if (targetContext === "html") {
      return rawOutput
        .replace(/&/g, "&amp;").replace(/</g, "&lt;")
        .replace(/>/g, "&gt;").replace(/"/g, "&quot;")
        .replace(/'/g, "&#x27;");
    }
    if (targetContext === "sql") {
      // Use parameterized queries; never interpolate LLM output directly
      throw new Error("SQL context: use parameterized queries only");
    }
    return rawOutput;

}
```

The code illustrates three of the six operators as schematic implementation patterns rather than directly deployable controls [1, 3]. In particular, the sanitize function is intentionally simplified for exposition and should be replaced in production by a hardened parser-based sanitizer; likewise, the policyCheck patterns capture only a limited subset of prompt-injection indicators and should not be interpreted as an exhaustive detector [6]. The encodeOutput function illustrates the principle of context-aware output handling, including the requirement that LLM output must never be interpolated directly into SQL queries [1, 17].

The formal model translates directly into a verification-oriented implementation checklist. The following table organizes security controls by operator, implementation technique, target vulnerability, and priority. Controls marked Critical correspond to operators whose absence breaks the security invariant of the pipeline, meaning an attacker can achieve end-to-end compromise without defeating any other control. Controls marked High correspond to operators that, if absent, significantly reduce the attack cost even though another control may partially compensate.

Table 3. Security Implementation Checklist Derived from the Formal Operational Model (compiled by the author based on [1, 3, 6, 9, 15, 16, 17]).

Security Control	Implementation Technique	Target Vulnerability	Priority
Input Sanitization	Custom HTML sanitizer per OWASP + Burp rules, allow-list approach	Direct / Indirect Injection	Critical
Context Isolation	Trust-level tagging (System / User / External), privilege separation at API gateway	Indirect Injection, Excessive Agency	Critical
FSM Policy	Finite-state machine with invariant	Jailbreak attempts, policy	High

Security Control	Implementation Technique	Target Vulnerability	Priority
Enforcement	rules applied before LLM call	bypass	
Output Encoding	Context-aware encoding: HTML-encode for web, parameterized queries for DB	XSS, SQL Injection via LLM output	Critical
Output Semantic Validation	RAG Triad check (relevance, groundedness, answer quality), classifier guard	Hallucination, data leakage	High
Least-Privilege Tool Access	Restrict LLM tool-call scope to read-only contract fields by default	Excessive Agency	High
Adversarial Testing (Red Team)	Periodic Burp Scanner + custom injection suite against LLM API endpoints	All categories	High
Audit Logging	Structured log of all prompts, model responses, tool invocations with user ID	Forensics, GDPR compliance	Medium

The checklist reflects the author's direct experience conducting web security certification for the Agiloft CLM platform using Burp Scanner and custom sanitization tooling. The row for Adversarial Testing specifically references the iterative process of running Burp Scanner against the platform endpoints and remedying each reported issue until the scanner returned a clean result, a process that required multiple cycles of sanitizer refinement before the custom OWASP-aligned sanitizer was tuned to handle all reported attack patterns.

The proposed formal model has several acknowledged limitations. First, the FSM Policy-Check operator is formalized over a finite alphabet of trust-level-tagged token sequences, but LLM inputs are in practice drawn from an open vocabulary where adversarial prompts can be constructed in any natural language, encoded in any character set, or obfuscated through Unicode homoglyphs and base64 encoding. The enumerated injection patterns in the reference implementation cover the most common classes documented in the literature [6, 10] but do not provide a completeness guarantee against all possible adversarial constructions.

Second, the Validate operator V() relies on semantic content policy checks, which in current implementations require either a secondary classifier model or human review. The quality of this operator is therefore bounded by the accuracy of the content policy classifier, and adversarially crafted outputs that evade classifier detection represent a residual risk not fully addressed by the formal model.

Third, the model addresses security at the web-interface layer and does not extend to threats arising at the model training layer, including data poisoning (LLM04:2025) and supply chain compromise (LLM03:2025). A complete security architecture for legal LLM systems requires complementary controls at the model governance layer that are outside the scope of this paper [12, 18].

Future work should focus on three directions. First, extending the FSM to a pushdown automaton to handle recursive or multi-turn injection attempts where an adversary distributes a payload across multiple conversation turns, each individually innocuous. Second, developing a formal specification for the Validate operator that incorporates the RAG Triad metrics (context relevance, groundedness, answer relevance) as verifiable post-conditions. Third, applying the compositional model to multimodal LLM interfaces where injection payloads may be embedded in images, audio, or structured data rather than text, a threat vector highlighted by the OWASP LLM Top 10:2025 as an emerging concern [1, 4].

Conclusion

This paper has presented an operational model for protecting web interfaces of legal LLM systems from two high-priority vulnerability classes highlighted in OWASP LLM Top 10:2025: prompt injection and insecure output handling. The model represents the security pipeline as a compositional sequence of six transformation operators, $T(x) = V(E(I(P(C(S(x))))))$, each with explicitly stated pre- and post-conditions derived from OWASP guidance and the reviewed security literature. The finite-state machine formalization of the Policy-Check operator provides an explicit state-transition account of the accept/reject gate before LLM

inference, while the STRIDE mapping offers a structured, though not exhaustive, coverage argument for the modeled legal LLM web-interface architecture.

The research goal stated in the introduction has been addressed: an operational model has been developed, compared against the STRIDE taxonomy, related to published vulnerability distributions, and translated into an implementation-oriented checklist. The checklist connects the abstract formal specification to concrete engineering practices that practitioners can apply to contract lifecycle management platforms and other legal technology products with LLM-integrated web interfaces.

The practical significance of the work is twofold. For security architects, the compositional structure of the model provides a completeness criterion for security audits: an audit is complete when all six operators are verified against their formal invariants. For front-end engineers, the reference implementation patterns and annotated code examples provide a starting point for building sanitization, trust-tagging, and output-encoding controls in JavaScript and Java-based web architectures. The dual-perimeter design, with pre-inference controls on the input side and post-inference controls on the output side, is intended to require an adversary to bypass multiple controls in sequence to achieve end-to-end compromise, thereby improving security posture relative to single-guard architectures.

References

1. OWASP GenAI Security Project. (2025). OWASP Top 10 for Large Language Model Applications 2025. Retrieved from: https://genai.owasp.org/llm-top-10/ (date accessed: November 5, 2025).

2. Ferrag, M. A., Tihanyi, N., Hamouda, D., Maglaras, L., Lakas, A., & Debbah, M. (2026). From prompt injections to protocol exploits: Threats in LLM-powered AI agents workflows. ICT Express, 12(2), 353–383. https://doi.org/10.1016/j.icte.2025.12.001.

3. Gulyamov, S., Gulyamov, S., Rodionov, A., Khursanov, R., Mekhmonov, K., Babaev, D., & Rakhimjonov, A. (2026). Prompt injection attacks in large language models and AI agent systems: A comprehensive review of vulnerabilities, attack vectors, and defense mechanisms. Information, 17(1), Article 54. https://doi.org/10.3390/info17010054.

4. Johnson, S., Pham, V., & Le, T. (2025). The dangers of indirect prompt injection attacks on LLM-based autonomous web navigation agents: A demonstration. In Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: System Demonstrations (pp. 729–738). Association for Computational Linguistics. https://doi.org/10.18653/v1/2025.emnlp-demos.55.

5. Hu, Y., Fan, C., Samyoun, S., & Du, J. (2025). Log-To-Leak: Prompt injection attacks on tool-using LLM agents via Model Context Protocol. OpenReview. Retrieved from: https://openreview.net/forum?id=UVgbFuXPaO (date accessed: February 12, 2026).

6. Liu, Y., Deng, G., Li, Y., Wang, K., Wang, Z., Wang, X., Zhang, T., Liu, Y., Wang, H., Zheng, Y., Zhang, L. Y., & Liu, Y. (2023). Prompt injection attack against LLM-integrated applications. arXiv. https://doi.org/10.48550/arXiv.2306.05499.

7. Wichers, N., Denison, C., & Beirami, A. (2024). Gradient-based language model red teaming. In Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers) (pp. 2862–2881). Association for Computational Linguistics. https://doi.org/10.18653/v1/2024.eacl-long.175.

8. Gong, N. (2025). Securing LLM agents against prompt injection attacks. Duke University. Retrieved from: https://people.duke.edu/~zg70/code/PromptInjection.pdf (date accessed: January 14, 2026).

9. Unit 42, Palo Alto Networks. (2025). New prompt injection attack vectors through MCP sampling. Retrieved from: https://unit42.paloaltonetworks.com/model-context-protocol-attack-vectors/ (date accessed: December 18, 2025).

10. Beurer-Kellner, L., Buesser, B., Creţu, A.-M., Debenedetti, E., Dobos, D., Fabian, D., Fischer, M., Froelicher, D., Grosse, K., Naeff, D., Ozoani, E., Paverd, A., Tramèr, F., & Volhejn, V. (2025). Design patterns for securing LLM agents against prompt injections. arXiv. https://doi.org/10.48550/arXiv.2506.08837.

11. Greshake, K., Abdelnabi, S., Mishra, S., Endres, C., Holz, T., & Fritz, M. (2023). Not what you've signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection. In Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security (AISec '23) (pp. 79–90). Association for Computing Machinery. https://doi.org/10.1145/3605764.3623985.

12. Shi, J., Yuan, Z., Liu, Y., Huang, Y., Zhou, P., Sun, L., & Gong, N. Z. (2024). Optimization-based prompt injection attack to LLM-as-a-judge. In Proceedings of the 2024 ACM SIGSAC Conference

on Computer and Communications Security (pp. 660–674). Association for Computing Machinery. https://doi.org/10.1145/3658644.3690291.

13. OWASP GenAI Security Project. (2025). LLM01:2025 Prompt injection. Retrieved from: https://genai.owasp.org/llmrisk/llm01-prompt-injection/ (date accessed: November 17, 2025).

14. Hou, X., Zhao, Y., Wang, S., & Wang, H. (2025). Model Context Protocol (MCP): Landscape, security threats, and future research directions. arXiv. https://doi.org/10.48550/arXiv.2503.23278.

15. OWASP. (2024). OWASP Top 10 for LLM Applications 2025 (PDF). Retrieved from: https://owasp.org/www-project-top-10-for-large-language-model-applications/assets/PDF/OWASP-Top-10-for-LLMs-v2025.pdf (date accessed: November 21, 2025).

16. Cloudflare. (n.d.). What are the OWASP Top 10 risks for LLMs? Retrieved from: https://www.cloudflare.com/learning/ai/owasp-top-10-risks-for-llms/ (date accessed: February 4, 2026).

17. Jia, Y., Shao, Z., Liu, Y., Jia, J., Song, D., & Gong, N. Z. (2025). A critical evaluation of defenses against prompt injection attacks. arXiv. https://doi.org/10.48550/arXiv.2505.18333.

18. Li, H., Liu, X., Zhang, N., & Xiao, C. (2025). PIGuard: Prompt injection guardrail via mitigating overdefense for free. In Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers) (pp. 30420–30437). Association for Computational Linguistics. https://doi.org/10.18653/v1/2025.acl-long.1468.

19. Jia, Y., Liu, Y., Shao, Z., Jia, J., & Gong, N. Z. (2025). PromptLocate: Localizing prompt injection attacks. arXiv. https://doi.org/10.48550/arXiv.2510.12252.

20. Hui, B., Yuan, H., Gong, N., Burlina, P., & Cao, Y. (2024). PLeak: Prompt leaking attacks against large language model applications. In Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (pp. 3600–3614). Association for Computing Machinery. https://doi.org/10.1145/3658644.3670370.