applied sciences

MDPI

Article

An Adaptive Rule-Based Engine for Application-Layer Security

Mihai-Cătălin Cujbă [1], Costin-Gabriel Chiru [1], Ion Bica [2,*] and Iulian Tiţă [1,2]

[1] Department of Computer Science, Faculty of Automatic Control and Computer Science, National University of Science and Technology Politehnica Bucharest, 060042 Bucharest, Romania
[2] Department of Computer Science and Cybersecurity, "Ferdinand I" Military Technical Academy, 050141 Bucharest, Romania
* Correspondence: ion.bica@mta.ro

Featured Application

This system can be deployed as an inline reverse proxy for real-time HTTP traffic inspection or as an offline log analyzer for forensic investigation of web application attacks, providing interpretable detection with adaptive rule generation.

Abstract

We present a composable, pipeline-based rules engine for detecting application-level intrusions in HTTP traffic with adaptive rule generation capabilities. Rules are expressed in JSON chain multi-step decoders (Base64, hex, XOR, zlib, gzip) with matching primitives (word boundaries, regular expressions, substring sets) to detect obfuscated payloads. To enable adaptation to novel attack patterns, we integrate a large language model (LLM) component as a second-opinion layer that automatically generates validated detection rules for previously unseen threats, combining the adaptability of machine learning with the interpretability of explicit rules. We evaluate the system on two standard benchmarks (CSIC 2010 and HttpParamsDataset) and present a head-to-head comparison with ModSecurity and the OWASP Core Rule Set, achieving 98.1% and 98.3% detection rates with F1 scores above 0.97 on both datasets while maintaining false positive rates below 0.51%.

Keywords: web application firewall; intrusion detection; rule engine; large language model; HTTP security; detection as code

1. Introduction

Modern web services have reached the point where they are continuously receiving cyberattacks with the aim of compromising infrastructure and destabilizing functionality. These attacks have evolved over time, including combining payload obfuscation, non-standard encoding, and contextual evasion to bypass traditional web application firewalls (WAFs). ModSecurity with the OWASP Core Rule Set (CRS) remains the de facto open-source standard [1], but heuristic scoring and static signatures often struggle with application-specific behavior and obfuscated inputs, which in turn generates false positives and limits secure blocking in production. Recent studies argue that fixed aggregation of CRS scores is suboptimal and show that learning weights on CRS signals can improve the trade-off between detection and false positives while reducing the active rule set used at inference time [1]. High alert volumes have become a problem because they discourage rigid blocking, simply by defining static rules that can lead to blocking legitimate requests or, in the case of a complex attack, the inability to block payloads that do not fit the standard attack type, reinforcing the need for approaches that adapt to context and reduce noise.

Academic Editor: Christos Bouras

Received: 30 April 2026
Revised: 14 May 2026
Accepted: 19 May 2026
Published: 22 May 2026

Meanwhile, the security engineering community has increasingly adopted the practice of treating detections as software artifacts, versioned and tested like code. Under the label Detection as Code (DaC), teams write modular rules, run them through validation pipelines, and measure them against real logs before deployment. Several groups have applied machine learning to HTTP-level attack classification [2–4], though operators frequently report difficulty understanding why a given request was flagged and how to adjust the model when application behavior changes.

Our paper presents a Python (version 3.12) rules engine that brings DaC principles to application-level intrusion detection while remaining transparent and controllable. The logic used to create the rules is expressed as modular JSON chunks that analysts can compose and reuse. Each rule can extract fields, split inputs, and apply chained decoders, including Base64 and hex in direct or reverse form, XOR, zlib, and gzip, to reveal hidden content before matching. The matching supports word boundaries, substring sets, regular expressions, simple command injection heuristics, and file path extraction. When a rule is used, it will receive a request and generate an event. This event will be assigned a score that will increase with the match on each step of the rule if a security issue with the analyzed payload is detected in the step. A request is blocked if any rule issues a 403 marker or if the cumulative score exceeds a threshold. The analysis engine integrates with mitmproxy for inline inspection, which allows real-time blocking of malicious requests, and comes with a Rule Studio that gives the analyst the ability to create, version snapshots, calculate content hashes over selected fields, and test replay based on event corpora.

Rule-based systems are interpretable and auditable, but writing and maintaining rules by hand is slow. Machine learning can adapt on its own, yet the resulting models are opaque. Large language models (LLMs) sit in an unusual position: They can reason about security patterns and produce structured output, but they are too slow and too unpredictable for real-time blocking decisions.

Our hybrid architecture works around this limitation. Explicit rules handle the real-time path—every blocking decision is deterministic and traceable. The LLM enters only when all rules return a zero score, at which point it reviews the request offline and, if it finds something suspicious, drafts a new rule in the same JSON format the analyst would use. That draft goes through the normal validation pipeline before it can block anything. The result is that the system can learn, but every piece of learned knowledge ends up as a readable, editable, deletable rule.

This paper makes four contributions:

1. A composable, JSON-based rules model that unifies decoding, transformation, and matching for HTTP requests, improving analyst transparency and control.
2. An interpretable scoring and decision scheme that aligns with operational workflows while enabling deterministic block decisions and audit trails.
3. A novel LLM-assisted adaptive detection mechanism that acts as a second-opinion layer, automatically generating validated rules for previously unknown attack patterns while maintaining system interpretability.
4. A practical tool stack that supports DaC workflows, including inline deployment and offline replay, and that complements both signature-centric WAFs and ML-centric approaches reported in the recent literature.

We describe the design and implementation of the engine, compare it to previous WAF- and ML-augmented systems, and discuss how composable decoding plus transparent scoring can detect obfuscation, command injection, and suspicious payloads in real traffic without sacrificing interpretability.

The rest of this paper is organized as follows. Section 2 reviews related work in the areas of signature- and anomaly-based web application firewalls, payload detection

techniques, and the evolution toward interpretable security systems. Section 3 establishes our threat model and presents the design goals that guide the system architecture. Section 4 formalizes the rule language and execution model, detailing how events are processed through pipelines in multiple steps. Section 5 introduces the LLM-assisted adaptive detection mechanism, including its architecture, rule generation process, and validation framework. Section 6 describes the complete system architecture, covering the Control Plane (Rule Studio), Data Plane (inline and log-based implementation), and Analysis Plane (Request Broker). Section 7 presents our experimental evaluation on standard benchmarks, including per-attack-type detection rates, performance metrics, a head-to-head comparison with ModSecurity and a learned CNN + BiLSTM baseline, and an empirical characterization of decoder-chain stress and prompt injection resistance. Finally, Section 9 discusses the implications of our approach, acknowledges limitations, and presents directions for future work.

2. Background and Related Work

Web application intrusion detection has evolved along three largely independent trajectories: rule-based and signature-driven systems that provide deterministic, auditable decisions; statistical and machine-learning anomaly detectors that adapt to traffic structure without explicit rules; and deep-learning approaches that learn request representations end-to-end. A fourth, more recent trajectory combines large language models with structured policy representations. This section surveys each trajectory, identifies the limitations that motivate the present work, and concludes with an explicit positioning statement.

2.1. Rule-Based and Signature-Driven Web Application Firewalls

The foundational approach to web application defense is signature matching: Incoming HTTP requests are tested against a library of patterns derived from known attack payloads. Early systems exploited the inherent regularity of attack syntax (SQL keywords, shell metacharacters, traversal sequences) to construct high-precision filters [5]. ModSecurity, now stewarded by OWASP, operationalized this approach at scale; its SecLang rule language, combined with the OWASP Core Rule Set (CRS), has become the de facto open-source standard for inline HTTP inspection [1,6]. The CRS assigns anomaly scores to matched rules and blocks requests whose cumulative score exceeds a configurable threshold, a design that balances recall against false positive rate through the *paranoia level* parameter.

Despite its maturity and widespread deployment, the signature paradigm faces two fundamental limitations that have been well documented in the literature. First, rules are written against known attack patterns, so novel or mutated payloads evade detection by construction [4]. Second, and more critically for the present work, modern attackers routinely chain encoding and compression layers (Base64 over hex over XOR over zlib) so that the malicious content is never present in cleartext form [7,8]. A signature engine that inspects the raw parameter value without first reversing each encoding layer will not see the payload at all. ModSecurity provides transformation functions (e.g., `t:base64Decode`), but these are applied individually to a single field; there is no composable pipeline in which the output of one decoding step feeds the input of the next [1]. Furthermore, recent work has demonstrated that learning optimal weights for CRS signals, rather than using the fixed scoring matrix, can significantly improve the detection-to-false-positive trade-off, suggesting that the rule engine and its scoring policy are separable concerns worth decoupling [1].

Gap 1: Existing rule-based systems lack a composable, pipeline-oriented mechanism for chained multi-layer deobfuscation; once obfuscation spans more than one encoding layer, signature matching fails regardless of the individual rule quality.

2.2. Statistical, Machine Learning, and Deep Learning Detectors

To address the novelty-blindness of signature systems, a large body of research has modeled the statistical structure of legitimate HTTP traffic and raised alarms on deviating requests [9–13]. These approaches operate on raw parameter values without prior deobfuscation and require labeled training data or extended benign traffic collection before deployment. A persistent operational obstacle is the false-positive problem: Even a 0.1% rate generates thousands of incorrect blocks per hour for a busy application, making rigid enforcement untenable and frequently driving operators to detection-only mode [1,14,15].

Deep-learning detectors feed raw character sequences from URLs, query strings, headers, and request bodies into neural architectures that learn task-relevant features automatically. End-to-end CNN and LSTM hybrids report accuracy above 99% on CSIC 2010 for SQL injection and related classes [2,16–24]. A recurring observation across this body of work is that accuracy improvements come at the cost of complete opacity in the decision process [25]: A neural WAF cannot tell an analyst which part of a request triggered the block, nor can an analyst adjust the model without full retraining on an updated labeled corpus. Deep detectors are also sensitive to distribution shift, generalizing poorly to traffic from a different application or to encoding schemes absent from the training set. We characterize this failure mode empirically in Section 7.8, where a CNN + BiLSTM trained on SR-BH 2020 collapses from 0.03% benign FPR in-distribution to 87.93% on CSIC and 77.74% on HttpParamsDataset.

Gap 2: Statistical and deep-learning detectors do not pre-process encoded payloads before modeling, cannot explain individual decisions to analysts, require labeled training data, and exhibit distribution-shift fragility across applications.

2.3. Large Language Models for Security and Detection-as-Code

Two recent developments challenge the traditional trade-off between adaptability and interpretability. The first is the emergence of large language models (LLMs) with security reasoning capability. Unlike classical ML models, LLMs can analyze an HTTP request in natural language context, identify suspicious patterns, and articulate why a payload appears malicious, all without task-specific fine-tuning. The 2024–2025 literature documents an expanding range of LLM applications across the cybersecurity domain [26], including vulnerability detection, malware analysis, and network intrusion detection. Web-attack detection specifically has been addressed by combining LLM-derived feature representations with classical autoencoder–MLP classifiers [27], and the ability of instruction-tuned models to produce structured output (JSON, YAML) makes them direct candidates for automated rule authoring. However, LLMs are too slow and too stochastic for real-time blocking decisions—inference latency is measured in seconds, and the same input can produce different outputs across runs, precluding the determinism that production enforcement requires. Used without a validation gate, LLM-generated rules carry risk: A hallucinated pattern that matches benign traffic would introduce false positives at scale.

Two recent works directly target the LLM-assisted WAF rule generation problem and bracket the design space. GenSQLi [28] uses GPT-4o to first synthesize novel SQL injection payloads against a target WAF and then auto-generate ModSecurity rules covering the bypassing payloads, reporting a 99% block rate on previously successful attacks using 23 newly generated rules; the work demonstrates that LLM-generated rules can be operationally effective when scoped to a specific attack class. VibeWAF [29] proposes a hybrid in

which a fast rule engine handles known patterns while an LLM analyses unmatched traffic and generates new rules covering similar future requests, evaluated on the SR-BH 2020 dataset with a feedback-loop convergence to an 88% rule hit rate that reduces average latency from 6.5 s to under 400 ms. Both works share the LLM-as-rule-author pattern with the present paper, but neither addresses the prompt-injection threat introduced by the LLM layer itself.

Parallel literature has emerged on adversarial inputs to the LLM layer itself. Indirect prompt-injection attacks embed adversarial instructions in untrusted data processed alongside legitimate user commands, exploiting the LLM's inability to distinguish system instructions from input content. Spotlighting [30] introduces a family of prompt engineering transformations (delimiter fencing, datamarking, base64 encoding of untrusted blocks) that reduce indirect prompt-injection success rates from above 50% to below 2% in the original evaluation while preserving NLP task efficacy. We adopt spotlighting as one of the seven defense templates in our prompt-injection evaluation (Section 8).

The second development is the Detection-as-Code (DaC) movement in security engineering. Under the DaC model, detection logic is expressed in version-controlled, machine-testable artifacts (rules, policies, and corpora) that undergo the same review and regression testing discipline as production software. This practice addresses the reproducibility and auditability requirements that pure ML pipelines cannot satisfy: A JSON rule can be read, diffed, rolled back, and replayed against historical events with deterministic results. Fuzzy rule-based systems at the application layer have demonstrated the operational value of explicit, auditable detection logic [31], and the broader DaC literature establishes the organizational benefits of treating security policies as first-class software artifacts.

To date, however, LLM-assisted rule generation, DaC practices, and adversarial defenses against LLM-targeted attacks have not been integrated into a single, coherent pipeline for HTTP intrusion detection. The closest precedent, VibeWAF [29], shares the hybrid architecture but does not characterize the adaptive component's behavior under prompt-injection adversaries. GenSQLi [28] demonstrates LLM-driven rule synthesis but does not include the gating infrastructure required to admit LLM output safely into a production engine.

Gap 4: No existing system combines LLM-assisted adaptive rule generation with a deterministic, audit-capable rules engine, an automated validation gate, and an empirically characterized defense against prompt injection of the LLM layer, leaving analysts with a choice between adaptability and interpretability rather than providing both.

2.4. Summary and Positioning

Table 1 summarizes the key capabilities and limitations across the four research threads identified above.

Table 1. Comparison of web intrusion detection paradigms across the key properties required by production deployment. ✓ = fully provided; ○ = partially provided; × = not provided.

Property	Signature WAF	Anomaly/ML	Deep Learning	Ours
Multi-layer deobfuscation	○	×	×	✓
Training-free deployment	✓	×	×	✓
Per-decision interpretability	✓	×	×	✓
Adaptation to novel attacks	×	○	○	✓
Deterministic, replayable	✓	○	×	✓
Analyst-editable logic	✓	×	×	✓

Three gaps motivate the design of the present system. The composable JSON rule language (Contribution 1) addresses *Gap 1* by providing a pipeline-oriented execution model in which each step consumes the output of its predecessor, enabling arbitrary chains of extraction, decoding, and matching without modifying the engine code. The interpretable scoring scheme (Contribution 2) addresses *Gap 2*: Every blocking decision accumulates a numerical score from explicitly labeled rule steps, and the full evidence chain is recorded for analyst review. The LLM-assisted adaptive component (Contribution 3) addresses *Gap 4*: The LLM operates as an offline second-opinion layer on zero-score traffic, with generated rules expressed in the same JSON format and passing through the same validation gate (FPR $< 1\%$, detection rate $> 80\%$ on labeled corpora) as hand-authored rules. The Detection-as-Code tooling (Contribution 4) closes the operational gap across all prior paradigms by providing policy versioning, event replay, and regression diffing.

3. Threat Model and Design Goals

This section specifies the environment in which the rule engine operates, the classes of adversaries it is designed to resist, and the constraints that shape the detection policy. The main purpose of the application is to inspect HTTP requests in order to detect cyberattacks and malicious payloads, even if they are malformed, in order to try to avoid filters and defense mechanisms, including multi-stage attacks, which are encoded and obfuscated multiple times. These payloads, together with the rest of the request are normalized, and translated into concrete requirements for the rule language and runtime so that design choices can be traced back to the threats.

3.1. Operational Scope and Adversary Model

The engine analyzes HTTP requests as captured by ModSecurity audit logs or observed inline by an intercepting proxy. Each request is represented by the method, path, query string, headers, and an optional body; transport security and upstream infrastructure are out of scope. Given that reverse-proxy or CDN services may sit in front of the engine, all inbound headers are treated as insecure (including X-Forwarded chains and provider-specific routing hints) unless a site policy explicitly restricts their origin. The system aims at semantic analysis of individual requests, not volumetric defense or account abuse.

The adversary creates HTTP requests and observes server responses but has no prior knowledge of the origin. Attacks combine payload obfuscation (Base64 and hexadecimal encodings sometimes reversed, XOR over strings, zlib and gzip compression [7–9,32]), token-combining that hides operators in benign substrings, and structural manipulation (splitting tokens, repeating keys, targeting sensitive paths for command execution or file traversal [33]). The threat primitives are addressed by accepting malformations, transformations, encryptions, and decoders that can be chained before matching.

3.2. Design Goals and Rule-Language Requirements

The design pursues six goals: composability, obfuscation coverage, multiple matching modes, interpretability, deterministic decisions, and operational match. The first three are formalized here; the remaining three are realized through the rule language (Section 4) and the tooling (Section 6).

Rules are ordered recipes of steps that extract fields, combine multiple sources, apply decoders, and match. Steps can depend on previous steps or consume multiple inputs, enabling multi-step decode-then-match workflows without engine modifications. The decoder set includes Base64 and hexadecimal decoders with optional inversion, single-byte XOR over decoded strings, zlib inflation (raw or Base64 input), and gzip decompression. URL decoding of parameter keys and values is performed on ingestion. Matching primi-

tives include substring checks, contains-any/contains-all sets, word-boundary matching with optional inversion, regular expressions, command-injection heuristics, and file-path extraction. Each match can contribute a score and a human-readable message.

A request is blocked if any rule issues a message containing the status code "403" (used when the engine analyses ModSecurity logs) or if the cumulative score reaches the configurable threshold. Inline enforcement uses the same decision function as offline replay, which guarantees reproducibility. Detection logic is stored as JSON rules editable through the Rule Studio front-end; the studio maintains versioned snapshots and a replay facility for regression testing against historical events.

The language separates evidence from action: one step may add score and produce a message without forcing a block, while another may issue an explicit "403…" message to assert immediate blocking. Conditional execution (`if`-clauses referencing prior Boolean outputs) and multi-input steps (`from_many`) provide the branching logic needed to express common decoder chains such as parameter extraction, token splitting, Base64/hex testing with inverted forms, and subsequent word-boundary matching. The current runtime relies on defensive decoder parsing; strict input size limits, per-step timeouts, and a compiled regex cache are planned enhancements measured in Section 7.9.

3.3. Decision Policy, Reproducibility, and Adaptive Learning

The blocking function is transparent and debuggable: An event is blocked when its cumulative score reaches the threshold or when any rule emits a message containing "403". Inline operation issues a 403 response and removes blocked flows from the backend; offline replay runs the same decision function against stored events for both a reference and a candidate policy, summarizing block counts and score deltas. Policies are JSON documents captured in snapshots and reapplied verbatim, ensuring reproducibility across environments and over time.

Traditional rule-based systems cannot adapt to novel attack patterns without manual intervention, while black-box ML sacrifices interpretability for adaptability. We address this trade-off through LLM-assisted rule generation: When base rules classify a request as benign (score = 0), the request is forwarded to an LLM for second-opinion analysis. If the LLM identifies suspicious characteristics, it generates a candidate rule in the system's JSON format. Candidates undergo automated validation against benign and attack corpora; rules meeting both thresholds (FPR $< 1\%$, detection rate $> 80\%$) are deployed automatically, while those failing either are queued for human review. Interpretability is preserved because every blocking decision still traces to an explicit JSON rule, and the LLM operates only on requests already classified benign by the base rules, so false negatives in LLM judgment cannot compromise security.

3.4. Limitations and Out-of-Scope Threats

The current model focuses on per-request semantics and does not build long-term behavioral profiles. It does not attempt to mitigate transport downgrades, TLS termination issues, or volumetric denial of service. There are no explicit execution limits yet on step timeouts or input sizes, which may be relevant in the case of adversarial decoder abuse; these are planned security measures. Finally, the response content is not parsed or persisted, which simplifies the trust boundary but limits the post-blocking client feedback to a synthetic 403 with an empty body.

4. Rule Language and Execution Model

This section formalizes the policy model, action semantics, evaluation order, and resulting decision mechanics of the engine. The system is designed as a pipeline-based,

declarative rules engine for stand-alone analysis of HTTP traffic. The description is based on the provided implementation, so that each assertion can be traced back to the source code.

4.1. Event and Policy Model

The main function of the engine is to process an event according to a policy. An event represents a single, self-contained HTTP request, while a policy is a structured set of rules that define the analysis logic.

4.1.1. Event Model

An event is the atomic unit of data processed by the engine. It is a dictionary object constructed by the mitmproxy addon from a live HTTP request. The schema for an event object E is a key-value map containing the following fields:

- `timestamp`: An ISO 8601 formatted string of the event time;
- `source_ip`, `destination_ip`: Network address information;
- `request_method`: The HTTP method (e.g., `GET`, `POST`);
- `request_path`: The URL path and query string;
- `request_headers`: A dictionary of HTTP headers;
- `request_body`: The raw request body as a string;
- `request_parameters`: A normalized list of key-value dictionaries derived from both URL query parameters and the request body (supporting URL-encoded forms and flattened JSON).

This structure provides a flattened, analysis-ready representation of the original HTTP request. The event object is mutable, and the engine directly modifies it during execution by adding `score` and `info` fields to record evidence.

4.1.2. Policy Model and Operational Control

A policy is a JSON object containing a list of rules that are derived from the detection logic of the execution engine and are managed, versioned, and distributed by the Studio web interface, which exposes endpoints for exporting the current policy (`/api/rules/export`) or versioned snapshots (`/api/snapshots/<id>`). This functionality makes it possible to debug an event.

For operational control, rules are decorated with labels so that analysts can know exactly what each one does. The mitmproxy add-on implements a two-stage evaluation discipline for performance and noise reduction:

- **Skip phase:** The engine first evaluates rules labeled `skip`. If any of these rules match, the event is considered benign, and further processing is stopped. This acts as a fast gate to allow known traffic to pass without the cost of a full analysis. A valid example for this phase is where legitimate paths, such as git endpoints where pieces of code are uploaded that may contain various elements such as SQL queries or executable code, are blocked due to matches to known attacks.
- **Enforcement phase:** If the event does not fall into the skip phase, the engine continues to evaluate rules with enforcement tags. These rules perform basic security analysis and generate evidence.

4.2. Rule Language

A rule is a sequence of processing instructions, or steps, that are executed in order to analyze an event. A rule is a JSON object defined by a name, an array of tags, a condition, and an ordered array of steps. The condition field (for example, `AND`, `OR`) dictates how the boolean results of the individual steps are aggregated to determine whether the rule as a whole was matched.

Each step is a JSON object with a name that is unique within the rule and an action to be performed. The behavior of the step is configured by additional keys:

- **Data dependency:** A step specifies its input using `from` (for a single input) or `from_many` (for multiple inputs). These keys reference the name of a previous step, allowing for the creation of complex data processing pipelines in which the output of one step becomes the input for another.
- **Conditional execution:** A step can be protected by an `if` key, which references the name of a previous step. The current step will only execute if the referenced protection step produced a `True` result.
- **Evidence generation:** Steps designed for detection (e.g., `contains_any`) can include `score` and `message` fields. If the step logic is met, its score is added to the event's total score, and a formatted message is added to the event's findings list.

Example Rule

As a concrete illustration of composability, a Command-and-Control (C2) beacon detection rule chains seven steps in a single JSON object. The rule extracts `request_body` (`get_field`), strips two known evasion noise tokens (`remove_string`), decodes Base64 (`decode_base64_auto`), reverses XOR with a fixed key (`xor`), decompresses the result with zlib (`zlib_inflate`, `encoding=raw`), and matches the recovered plaintext against a command list (`contains_any`, `score=1500`, message "C2 beacon: command found after deobfuscation"). Each step references its predecessor via the `from` directive; only when every step in the chain succeeds does the final match contribute to the request's score.

4.3. Execution Model

The `RuleEngine` class orchestrates the evaluation of rules based on an event.Figure 1 situates the engine within the overall data flow of the system, from HTTP traffic capture and ModSecurity log ingestion through the Rule Engine to the scored events surfaced in the analyst-facing storage and review interface. Within the engine itself the execution flow is sequential and stateful, building a context as it processes each step.

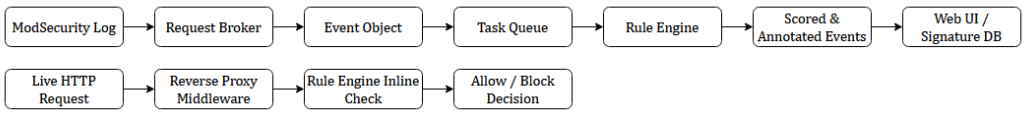

Figure 1. Overall architecture of the proposed web-request analysis and enforcement system.

The Evaluation Loop

When a rule is applied to an event, the engine initializes an empty context dictionary that will be populated later as it iterates through the rule steps by following the next scenario. It first checks whether the step has an `if` condition. If the condition refers to a step name that is not in the context or whose value is not `True`, the current step is skipped. The engine resolves the step input by reading the `from` or `from_many` keys and retrieving the corresponding outputs from the context. If no input source is specified, the step can operate directly on raw `event_data`. The engine invokes the Python function corresponding to the step action name, passing in the event, the step configuration, and the resolved input value. The value returned by the action function is stored in the context dictionary, with the step name key, and this makes the result available for subsequent steps to consume. If the action returns `True` and the step is configured with a score and a message, then the engine moves the original event object directly. The score is added to `event['score']`, and the formatted message is added to `event['info']`.

The results stored in the context are aggregated according to the rule condition (`AND` or `OR`) to produce the final result for the entire rule, thus resulting in both a complete analysis of the event that will have a score and notes according to the steps it went through, but also a marker that will say whether the entire rule matched the event.

4.4. Action Semantics and Decision Mechanics

The capabilities of the engine are defined by its registry of available actions and the logic used to make a final block/allow decision based on the generated evidence.

The engine maintains a static mapping of action names to Python methods. This registry is extensible and includes a variety of functions that can be categorized as follows: Extraction (e.g., `get_field`), Decoding and Transformation (e.g., `decode_base64_auto`), Matching (e.g., `contains`), and Utility (e.g., `generate_hash`).

After the enforcement rules have been fully executed against an event, the `decide` function makes a final decision. This function implements a hybrid decision model that combines a risk score with specific high severity indicators. A request is marked as blocked if any of the following conditions are met:

- The cumulative score of the event object is greater than or equal to the configured `RISK_THRESHOLD` (which is 100 by default).
- Any message string in the event information list contains the status code "403".

Using this double condition by which the module can block an event, we can allow expanding the capabilities so as to integrate new functionalities that return a forbidden result from the HTTP server when an attack is detected, such as using an external WAF engine that has these capabilities. In our research, ModSecurity was used for this, namely the logs that it generates in verbose mode.

5. LLM-Assisted Adaptive Detection

5.1. Trigger Mechanism

The LLM analysis is triggered when all of the following conditions are met: (1) The event has completed evaluation through all base rules; (2) the accumulated score equals zero (no rule matched); (3) LLM analysis is enabled in the system configuration. This conservative trigger keeps the base rules as the primary defense layer and incurs LLM cost only on traffic those rules cannot classify. Backend options (cloud Gemini and local Ollama) are described under *Implementation Details* later in this section.

5.2. Request Analysis Pipeline

Each candidate request passes through a three-stage pipeline before reaching the LLM. First, *sanitization* masks sensitive content: IP addresses are anonymized, session tokens and authentication headers are redacted, and PII patterns in parameter values are removed, preserving enough structure for security analysis while reducing privacy exposure. Second, *context construction* assembles a structured prompt containing the HTTP method, path, query string, sanitized headers and body, URL-decoded parameter values, and a description of the base rules that were evaluated but did not match. Third, *prompt engineering* frames the LLM as a security analyst reviewing the request and asks it to identify suspicious patterns or obfuscation, classify the attack type if present, assess confidence, and provide justification, ensuring generated rules can be understood by human reviewers.

5.3. Rule Generation Process

When the LLM determines that a request is suspicious, it generates a candidate detection rule in the system's JSON rule format. The generation process follows a structured template to ensure compatibility with the existing rules engine.

5.3.1. Template-Based Generation

The LLM prompt includes examples of well-formed rules demonstrating the JSON structure, available actions (extraction, decoding, matching), and proper use of data dependencies between steps. The LLM is instructed to generate rules that are specific enough to detect the identified attack pattern but general enough to catch variations. For example, if the LLM identifies a Base64-encoded SQL injection payload, the generated rule should include Base64 decoding followed by SQL keyword matching, rather than matching only the exact observed payload.

5.3.2. Structured Output Parsing

The system parses the LLM's response to extract the generated rule JSON. As LLMs may include explanatory text or markdown formatting, the parser uses heuristics to identify and extract valid JSON objects. The extracted rule undergoes schema validation to ensure it conforms to the expected structure: Required fields (name, tags, steps) are present; each step has a valid action and required parameters; data dependencies reference valid step names; and score assignments are within acceptable ranges.

5.4. Rule Validation Framework

Generated rules must undergo rigorous validation before deployment to prevent false positives and ensure detection effectiveness. The validation framework tests candidate rules against known traffic corpora and applies quality thresholds.

5.4.1. Validation Corpus

The validation framework maintains two test corpora: a benign traffic corpus containing legitimate requests collected from production or test environments, used to measure false positive rate, and an attack corpus containing known malicious requests covering various attack types, used to verify detection capability. The system periodically updates these corpora to reflect evolving application behavior and emerging attack patterns.

5.4.2. Automated Testing Protocol

Each candidate rule is applied to every request in both test corpora. For the benign corpus, the system counts how many requests the rule incorrectly flags (false positives) and calculates the false positive rate. For the attack corpus, the system counts how many attacks the rule correctly identifies (true positives) and calculates the detection rate. These metrics inform the deployment decision.

5.4.3. Deployment Thresholds

A candidate rule is automatically deployed to the active rule set if it meets the following criteria: false positive rate on benign traffic $< 1\%$; detection rate on attack corpus $> 80\%$; and the rule does not duplicate an existing rule (checked via semantic similarity). Rules that fail these thresholds are placed in a review queue for human analyst evaluation.

5.5. Safety Mechanisms and Guardrails

LLM-generated content passes through three layers of protection before deployment. *Syntax and semantic validation* enforces JSON schema compliance, an action whitelist, resource limits on generated regular expressions, and a directed-acyclic-graph check on step dependencies. *Rate limiting and resource control* caps the number of LLM queries per time window, deduplicates near-identical requests to avoid redundant calls, and enforces hard timeouts on every LLM API call. *Human-in-the-loop oversight* logs every interaction and surfaces queued rules in the Rule Studio, where analysts can approve, modify, or reject candidates; rejected rules and analyst rationale feed back into prompt-engineering iteration.

5.6. Implementation Details

The system supports two LLM backends. Cloud deployment uses the Google AI SDK (`gemini-2.0-flash`, temperature = 0.3, `top_p` = 0.9), balancing creativity for novel-pattern identification with determinism for consistent output. Local deployment uses Ollama with models such as `gemma3:4b` or `qwen2.5:7b`, eliminating external dependencies but requiring at least 16 GB RAM for 8B-parameter models. LLM inference latency is 1–5 s for cloud and 2–10 s for local; this is operationally acceptable because the LLM runs only on zero-score traffic that received no immediate enforcement decision, can be processed asynchronously, and never sits on the blocking path of requests already flagged by base rules. Detailed per-model performance measurements appear in Section 8.5.

6. System Architecture

The system is organized as three decoupled planes that share a single Rules Engine. This separation lets policy be edited, versioned, and replayed offline in the Control Plane without disturbing real-time inspection and enforcement in the Data Plane.

Figure 2 shows where each module sits. The Control Plane (top) hosts the Rule Studio and the rule database; the Data Plane (left) intercepts and forwards HTTP traffic via the inline addon and the log listener; the Analysis Plane (right) executes the Rule Engine, Request Broker, and LLM service. Solid arrows depict the synchronous request path; dashed arrows depict the asynchronous LLM-analysis feedback loop. Each plane is shaded with a distinct background colour for visual separation.

Figure 3 traces the end-to-end runtime path that an HTTP request follows through these planes, including the LLM-driven feedback loop that turns zero-score events into validated rules. Solid arrows denote the synchronous request path; dashed arrows denote the asynchronous LLM-analysis loop. Node fills are colour-coded by plane: yellow for the Data Plane, red for the Analysis Plane, green for the Control Plane, and pale purple for the LLM Service. The Data Plane has two entry paths: the inline Mitmproxy addon, which invokes the Rules Engine directly, and the log-based Listener, which forwards events to the Request Broker before the engine is invoked. The Rules Engine in the Analysis Plane scores each event; scores ≥ 100 trigger a 403 block (inline path only), scores in $(0, 100)$ are allowed, and zero-score events both allow the request and asynchronously invoke the LLM Service (dashed arrow). Candidate rules emitted by the LLM pass through two validation stages: a schema and safety check (action whitelist, DAG well-formedness, resource limits) and corpus validation against benign and attack traffic. Rules passing the deployment thresholds (FPR $< 1\%$, detection rate $> 80\%$, no duplicate) are added directly to the active rule set in the Control Plane; rules that fail enter the Rule Studio review queue, where an analyst can approve, modify, or reject them. Rejected rules feed back into LLM prompt engineering (dashed loop). The Data Plane and Studio communicate bidirectionally: the Mitmproxy addon periodically pulls the active policy, and pushes processed events back to Studio for storage and review.

6.1. The Control Plane: The Rule Studio

Rule Studio, the Control Plane, is a Flask web application backed by SQLite that exposes policy, rule, and step management through a web UI and a RESTful API.

Data Plane components push processed events back to Studio via `/api/events/import` for storage. Replay Lab uses this corpus to compare a candidate policy against a baseline before deployment, and policy versioning produces stable, reproducible exports.

Studio also exposes a `/url-list` service-discovery endpoint that publishes available analytics brokers to distributed listeners, and a forensic search interface in which analysts can label requests or whole patterns as benign or malicious to refine detection logic.

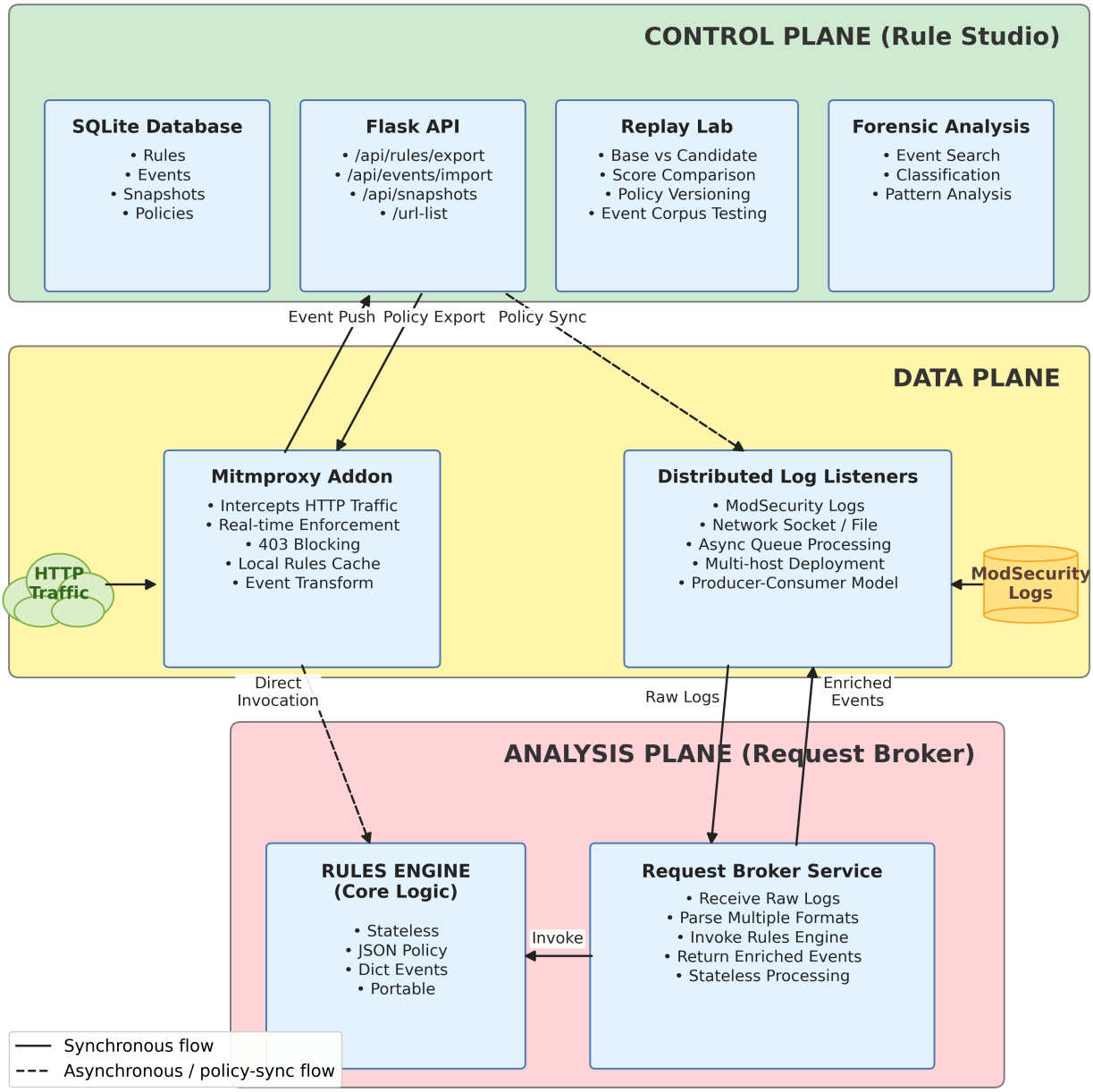

Figure 2. Three-plane architecture of the proposed web request analysis framework.

6.2. The Data Plane

The Data Plane collects security-relevant data and, in some configurations, enforces decisions inline. Two implementations are supported.

6.2.1. The Inline Enforcement Addon

An inline mitmproxy addon intercepts live HTTP traffic and applies the current policy. It pulls the active rule set from Studio's `/api/rules/export` on a timer and caches it locally for resilience and throughput.

Each request is normalized into the engine's event dictionary, evaluated through the omission and enforcement passes described in Section 4, and either forwarded or blocked with a 403 response. The processed event, with its final score and any messages, is then pushed back to Studio for storage.

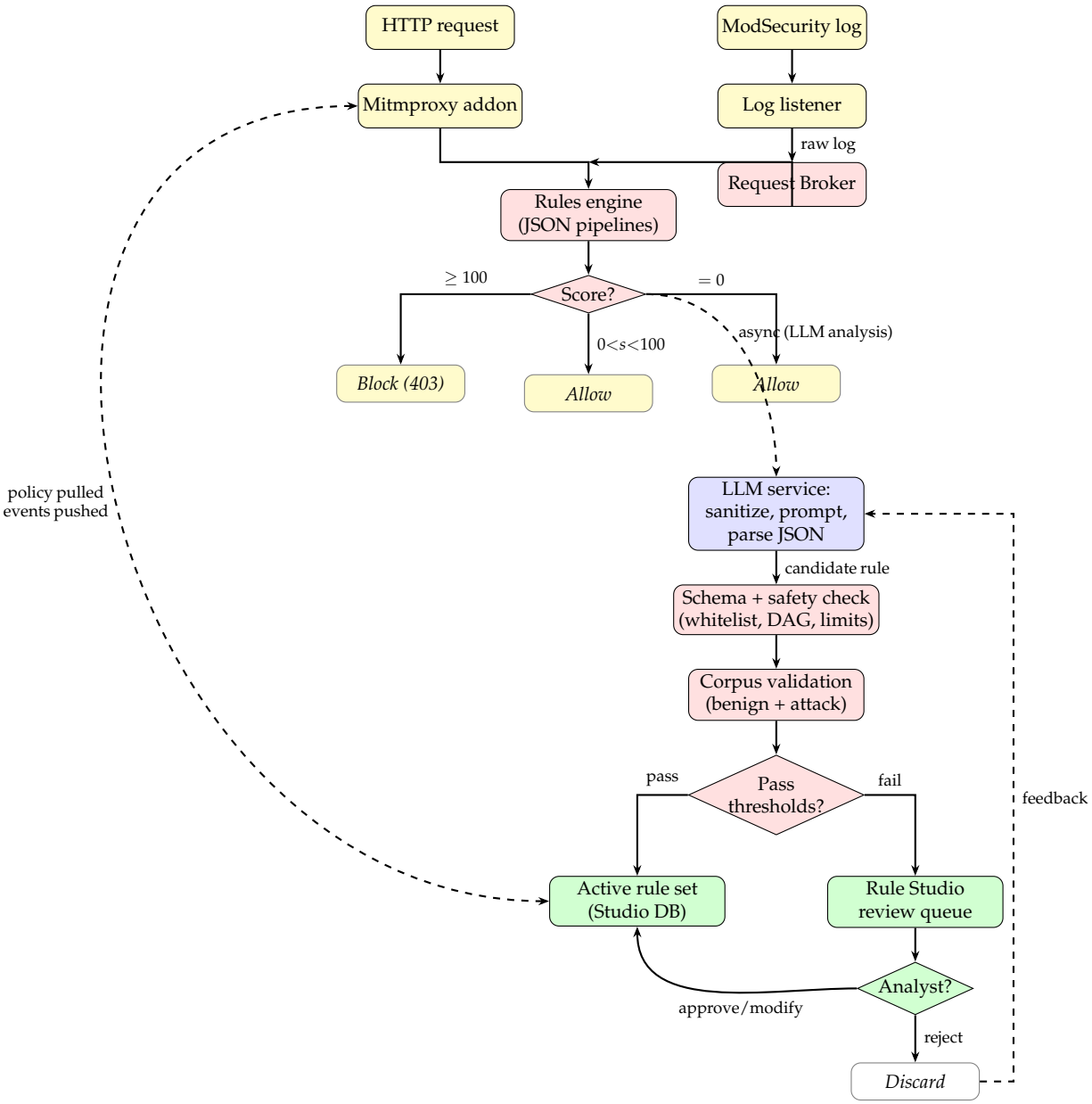

Figure 3. End-to-end workflow of the adaptive detection loop in the default fully-automated deployment mode.

6.2.2. The Distributed Log Ingestion System

An alternative deployment uses `smart_listener.py` to ingest logs from sources such as ModSecurity on multiple hosts. Each listener can consume a live network log stream or replay a log file from disk for forensics.

Unlike the inline addon, the listener does not run the engine locally; it forwards raw logs to the Analysis Plane and, on receiving the enriched event back, queues it for asynchronous write to `requests.db` by a pool of writer threads.

6.3. The Analysis Plane: Request Broker and Rule Engine

The Analysis Plane is built from two layers. The outer layer is the Request Broker, a stateless service that parses incoming logs into the standard event dictionary, invokes the engine, and returns the scored event to the originating listener; centralising this step guarantees that every log source is evaluated under the same policy.

The inner layer is the Rule Engine itself, a standalone Python class that accepts a JSON policy and an event dictionary and returns a scored event. It has no dependency on HTTP, web frameworks, or storage, which keeps it portable and testable and ensures that the inline mitmproxy addon and the offline Replay Lab evaluate events under identical semantics. The same engine is therefore the shared kernel of both the Data Plane (inline path) and the Analysis Plane (log-based path) shown in Figures 2 and 3.

6.4. The LLM Analysis Service

The LLM Analysis Service is a pluggable module that extends detection beyond static rules and integrates with both the inline enforcement addon and the offline analysis pipeline. Internally it is split into a client abstraction over the supported backends, a rule generator, and a validation engine, all described in detail in Section 5; externally, the Rules Engine invokes it on zero-score events and the Rule Studio surfaces a review queue for analysts to approve, modify, or reject generated rules.

The service supports four deployment modes. In *fully automated* mode, validated rules are deployed to the active rule set without human intervention, suitable for mature installations with well-tuned validation thresholds. In *review-required* mode, every generated rule enters the analyst queue before deployment, appropriate for initial rollout or high-security environments. In *logging-only* mode, the LLM analyses requests and emits rules but does not deploy them, useful for measuring LLM behavior before granting it enforcement authority. In *disabled* mode, the service is inactive and the system operates as a traditional rule-based engine.

7. Evaluation and Results

To assess the practical viability and effectiveness of the proposed system, we conducted an empirical evaluation using the CSIC 2010 HTTP Dataset [34], a widely cited benchmark for web application firewall research. The dataset was generated by the Spanish National Research Council and contains raw HTTP requests directed at an e-commerce web application. It comprises 36,000 normal (benign) requests and 25,065 anomalous (malicious) requests covering multiple attack categories: SQL injection, Cross-Site Scripting (XSS), CRLF injection, path traversal, information gathering, parameter tampering, buffer overflow, and server-side includes.

7.1. Dataset and Methodology

The CSIC 2010 anomalous traffic is dominated by parameter tampering attacks (79.8% of all anomalous requests), which involve application-specific business logic violations such as modifying product prices, injecting invalid identifiers, or adding unexpected form fields. These attacks are inherently application-specific and cannot be detected by general-purpose signature rules, since no WAF can determine that a price was tampered with without knowledge of the application's expected values. This limitation is shared by all signature-based systems, including ModSecurity CRS, which provides no rules for parameter tampering detection.

We therefore report detection results both on the full dataset and on the payload-based attack subset, which excludes parameter tampering and isolates the attack categories amenable to general-purpose rule-based detection: SQL injection (1480 requests), information gathering (1695 requests), CRLF injection (932 requests), XSS (736 requests), and path traversal (194 requests), totaling 5037 malicious requests. The full normal test set (36,000 benign requests) is used for all false positive measurements. All experiments use a blocking threshold of score ≥ 100.

7.2. Detection Results

Table 2 presents the per-attack-type detection rates on the payload-based subset. The engine achieves perfect detection (100%) on four of five attack categories and 93.6% on SQL injection.

Table 2. Per-attack-type detection rates on CSIC 2010 (payload-based attacks, excluding parameter tampering).

Attack Type	Total	Detected	Missed	Rate
Information gathering	1695	1695	0	100.0%
SQL injection	1480	1386	94	93.6%
CRLF injection	932	932	0	100.0%
XSS	736	736	0	100.0%
Path traversal	194	194	0	100.0%
Total	5037	4943	94	98.1%

The 94 missed SQL injection payloads follow a specific pattern: The tautology `AND 1 = 1` is appended directly to a parameter value without a preceding space (e.g., `entrarAND+1%3D1`), which prevents the word-boundary matching from triggering. These edge cases represent a known trade-off between detection sensitivity and false positive rate; broadening the pattern to match without word boundaries would increase false positives on benign traffic containing common substrings.

7.3. Overall Metrics

Table 3 summarizes the aggregate detection metrics on the payload-based evaluation set (5037 malicious + 36,000 benign requests).

Table 3. Overall detection metrics on CSIC 2010 (payload-based subset).

Metric	Value
Detection Rate (Recall/TPR)	98.13%
Precision (PPV)	96.43%
F1 Score	97.27%
Accuracy	99.32%
Specificity (TNR)	99.49%
False Positive Rate (FPR)	0.51%
False Negative Rate (FNR)	1.87%

The confusion matrix for this evaluation is 4943 true positives, 94 false negatives, 183 false positives, and 35,817 true negatives. The 183 false positives (0.51% FPR) originate primarily from the SQL comment obfuscation rule (157 cases), part of the SQL injection detection chain. The rule targets the canonical SQL comment markers (`-`, `#`, and `/* */`) appearing near the tail of a payload, which is a standard SQL injection technique for neutralizing the rest of the original query. The misfires trace not to the raw request content but to the engine's auxiliary decoded views, in particular `hex_all` and `b64_all`: short numeric form fields such as the quantity field (`cantidad=23`) and a 16-digit credit-card-like identifier (`ntc=5904492147242322`) consist entirely of valid hexadecimal digits, so the hex auto-decoder produces short byte strings that happen to contain the byte `0x23` (`#`, the MySQL-style line comment marker), satisfying the tail-anchored regex purely by coincidence of byte values. The same effect appears when the URL-encoded form body, taken as a single string, is Base64-decoded into a gibberish byte sequence that happens to

contain a # within the trailing 200 bytes. The Spanish locale of the dataset is incidental to this mechanism: The trigger is the interaction between an aggressive auto-decoder chain and short, hex-shaped inputs, and would surface identically on any benign traffic that contains numeric identifiers or alphanumeric form bodies. The remaining false positives are caused by shell metacharacter patterns matching in benign URL-encoded values (25 cases from encoded slashes, 20 cases from parameter content resembling shell operators). Both false positive sources are well-understood and can be mitigated without modifying the broader detection logic, for example by adding a minimum-length guard before the comment-tail regex is applied to decoded views, or by restricting that specific rule to the raw and percent-decoded inputs rather than the hex and Base64 views.

7.4. Performance Metrics

Performance was measured during the evaluation on a standard workstation (4 GB RAM, 4 CPUs). Table 4 presents the latency and throughput results.

Table 4. Runtime performance during CSIC 2010 evaluation.

Metric	Value
Throughput	793 requests/s
Average latency/request	1.26 ms
Median latency	1.06 ms
P95 latency	2.19 ms
P99 latency	3.08 ms

The sub-millisecond median latency and consistent P99 below 3.1 ms demonstrate the engine's suitability for inline deployment as a reverse proxy. The throughput of 793 requests/second was achieved in a single-threaded Python process; in the multi-threaded offline analysis mode with 4 concurrent threads, the system achieves approximately 1000 events per second.

7.5. Comparative Analysis

To contextualize our results, Tables 5 and 6 present a feature-level comparison with leading intrusion detection and web application firewall systems, split into detection logic and operational properties for readability. (The original Table 5 was restructured into two narrower tables in this revision.)

Table 5. Detection logic of our engine compared with widely used intrusion-detection and web-application-firewall systems.

System	Detection Paradigm	Rule Language	Primary Data Source
Our engine	Declarative pipeline	Declarative JSON	HTTP logs/requests
ModSecurity (CRS)	Imperative matching	SecLang	HTTP requests
Suricata/Snort	Signature matching	Signature-based	Network packets
Zeek (Bro)	Event-driven scripting	Turing-complete script	Network packets
CNN-WAF	ML classification	Learned (N/A)	Raw HTTP requests

Two properties distinguish the proposed engine from the systems in Tables 5 and 6: a declarative JSON pipeline that composes decode and match steps without modifying engine code, and native interpretability in which every blocking decision traces to an auditable rule together with its intermediate pipeline outputs, a property that ML-based WAFs cannot offer and that network-level systems do not target at the HTTP layer.

Table 6. Operational properties of the same systems.

System	State	Extensibility	Analyst Feedback	Deployment
Our engine	Stateless	Python actions	Native (memory)	WAF/log analyzer
ModSecurity (CRS)	Session/IP coll.	Lua/C++ modules	Not native	Embedded WAF
Suricata/Snort	TCP flow state	Preprocessors	Not native	NIDS/NIPS
Zeek (Bro)	High (conn. state)	Zeek scripting	Requires scripts	Network sensor
CNN-WAF	Low (per-request)	Model retraining	Model retraining	WAF

7.6. Cross-Dataset Validation: HttpParamsDataset

To assess generalization beyond a single benchmark, we evaluated the engine on the HttpParamsDataset [35], an independent collection of 31,067 individual HTTP parameter values (19,304 benign and 11,763 malicious) labeled by attack type. Unlike CSIC 2010, which provides full HTTP requests against a specific web application, HttpParamsDataset contains isolated parameter payloads sourced from multiple security tools (sqlmap, XSSYA, Vega Scanner, FuzzDB), testing the engine's matching capabilities at the parameter level regardless of request context. Each payload was injected as a query parameter value into a synthetic HTTP request and evaluated using the same rule set and threshold (score ≥ 100) as the CSIC 2010 evaluation.

Table 7 presents the per-attack-type detection rates.

Table 7. Per-attack-type detection rates on HttpParamsDataset.

Attack Type	Total	Detected	Missed	Rate
SQL injection	10,852	10,832	20	99.8%
XSS	532	510	22	95.9%
Path traversal	290	138	152	47.6%
Command injection	89	80	9	89.9%
Total	11,763	11,560	203	98.3%

Table 8 summarizes the aggregate metrics across all 31,067 payloads.

The results demonstrate consistent performance across both benchmarks: 98.3% detection rate on HttpParamsDataset versus 98.1% on CSIC 2010 (payload-based subset), with comparable false positive rates (0.47% vs. 0.51%) and precision (99.23% vs. 96.43%). The remaining missed payloads reflect deliberate trade-offs between detection sensitivity and false positive risk:

Path traversal (47.6%): The 152 missed payloads use non-standard traversal patterns that replace the canonical `../` sequence with alternative representations, such as continuous dot sequences without slash separators (e.g., `.....etcpasswd`) or Windows-specific paths (`c:oot.ini`, `c:/boot.ini`). These patterns are not generalizable attacks; they target specific server configurations and would generate false positives on legitimate Unix-style paths if broadly matched.

XSS (95.9%): The 22 missed payloads are predominantly HTML tags without event handlers or JavaScript execution vectors (e.g., `<a href = ‘‘http://...’’>text</a>`, `<img src = ‘‘http://...’’>`), which represent injection of benign HTML rather than executable cross-site scripting.

SQL injection (99.8%): The 20 remaining missed payloads are edge cases including single-character probes (`‘1`), `CASE WHEN` expressions that also appear in legitimate analytics queries, and arithmetic subtractions (`5739-5738`) indistinguishable from normal numeric input.

Command injection (89.9%): The 9 missed payloads use absolute paths to system binaries (`/bin/ls`, `/usr/bin/id`) or backtick-enclosed common words (`‘uname’`, `‘true’`), which the current heuristics do not flag to avoid false positives on legitimate Unix-style paths and quoted strings.

Table 8. Overall detection metrics on HttpParamsDataset.

Metric	Value
Detection Rate (Recall/TPR)	98.27%
Precision (PPV)	99.23%
F1 Score	98.75%
Accuracy	99.06%
Specificity (TNR)	99.53%
False Positive Rate (FPR)	0.47%
False Negative Rate (FNR)	1.73%

7.7. Contemporary Cross-Dataset Validation: SR-BH 2020

To validate the engine against a more recent benchmark and address the limitation that CSIC 2010 reflects HTTP traffic patterns predating widespread API-driven and JavaScript-heavy web applications, we additionally evaluated on the SR-BH 2020 dataset [36] hosted on Harvard Dataverse. SR-BH 2020 contains 907,815 labeled HTTP requests captured against an instrumented WordPress test site, with 525,195 benign and 382,620 malicious requests. Unlike CSIC 2010 (which uses five high-level attack categories) and HttpParamsDataset (binary labels per parameter value), SR-BH 2020 annotates each request with one or more CAPEC identifiers [37], giving a multi-label structure across thirteen attack categories. We restrict our analysis to the subset of CAPEC categories that fall within the threat model of Section 3, namely SQL Injection (CAPEC-66), Path Traversal (CAPEC-126), OS Command Injection (CAPEC-88), and Scanning for Vulnerable Software (CAPEC-310).

Label-quality findings.

Inspection of SR-BH 2020 disclosed two classes of automated-labeling artifact that motivate our dual-reporting methodology. On the attack side, the tagging tool assigns CAPEC labels heuristically; the *SQL Injection* tag is attached to any request containing a `ver` parameter regardless of whether the value carries SQL syntax, so standard WordPress static-asset loads such as `/blog/wp-admin/load-styles.php?ver = 4.9.5` appear as SQL injection attacks. Approximately 22% of the 250,311 *SQL Injection*-tagged requests contain no SQL syntax in URL or body; the engine correctly treats them as benign, recorded as missed detections under the raw scheme. Similar cross-category confusion places both *SQL Injection* and *OS Command Injection* labels on any request containing the keyword `sleep`, conflating SQL `SLEEP()` calls with shell `sleep N` invocations. On the benign side, 58,643 (11.17%) of the 525,195 *Normal* requests contain a syntactic indicator consistent with an attack class (URL-encoded shell injection `%26cat+%2Fetc%2Fpasswd%26`, encoded path traversal, etc.) that the test server simply did not honor; the engine correctly flags many of these as attacks, recorded as false positives under the raw scheme.

Signature-based label validation.

To distinguish engine behavior from these labeling artifacts we report each per-category detection rate twice: against the raw SR-BH labels exactly as published, and against a signature-validated subset. The validation procedure is a per-category filter that retains a CAPEC label on a request only when the request's URL or body (raw and twice percent-decoded) matches at least one signature from the corresponding signature set. The signature sets are deliberately narrow and reflect canonical syntactic markers of each attack class, summarized in Table 9.

The same signature sets are also applied to the benign half of the dataset. Any request labeled *Normal* whose URL or body matches a signature from any category is excluded from the validated benign subset and is not used for the false-positive-rate calculation. The procedure is conservative by design; it removes only requests whose payload unambiguously matches the canonical syntactic shape of an attack class, so the residual validated subset retains the bulk of the original dataset (88.83% of benign rows and between 72.84% and 100% of attack rows per category) while removing the most obvious labeling artifacts. We acknowledge that any signature-based filter remains a heuristic; in particular, it may reject highly unusual real attacks that do not match common syntactic forms, and the absolute ground truth of every record is unknown.

Table 10 reports the per-category detection rate of our rule engine against both the raw and validated subsets.

Table 9. Per-category signature classes used to validate SR-BH 2020 attack labels. A request retains a CAPEC label only if at least one signature from its category matches the URL or body (raw or percent-decoded).

Category	Signature Classes
SQL Injection	DML keywords (`SELECT`, `UNION`, `INSERT`, `DROP`, ...); boolean tautologies with numeric or string operands; quote followed within 80 characters by a SQL operator or comment marker; SQL function calls of the form `name(` (parenthesis required, to exclude shell command `sleep 15`); schema-introspection identifiers (`information_schema`, `pg_catalog`); hexadecimal literals of length ≥ 6; stacked-statement separators; `CASE WHEN` expressions
Path Traversal	Literal `../` and `..\`; once- and twice-percent-encoded variants; restricted file paths (`/etc/passwd`, `/etc/shadow`, `/proc/self/environ`, `boot.ini`, `win.ini`); Windows drive references (`C:\`, `C:/`)
OS Command Injection	Shell metacharacter followed by a known shell binary (`;cat`, `|wget`, `&sleep`, `'id'`); command substitution `$(...)`; shellshock pattern `() { :;};`; absolute shell paths (`/bin/sh`, `/bin/bash`); Windows `cmd.exe /c`
Scanning for Vulnerable Software	Probes for administrator paths (`/admin`, `/wp-admin`, `/phpmyadmin`); secret/config files (`/.env`, `/.git/`, `/web.config`); information probes (`/phpinfo`, `/server-status`); backup-file extensions (`.bak`, `.swp`); known scanner User-Agent strings

The gap between the raw and validated columns is largest for SQL Injection, where the raw rate of 86.20% rises to 99.12% once label-noise rows are excluded. This is consistent with the labelling artifact described above: roughly 27% of SR-BH's SQL Injection tags correspond to WordPress static-asset traffic or scanner heuristics that do not contain any SQL syntax, and the engine correctly treats these as benign, which is counted as a missed detection by the raw label scheme. The two large in-scope volume categories that do not show such a gap, Path Traversal and Scanning, are categories in which the dataset labels already track payload content closely, so the raw and validated rates agree to within a fraction of a percentage point.

Table 10. Per-category detection rates on SR-BH 2020 for the four CAPEC categories within the engine's threat model. The Raw column uses dataset labels as published; the Validated column restricts to requests whose payload contains a syntactic indicator consistent with the tagged attack class.

CAPEC Category	Raw N	Raw	Valid. N	Valid.
SQL Injection (CAPEC-66)	250,311	86.20%	182,318	99.12%
Path Traversal (CAPEC-126)	20,992	98.49%	18,963	98.53%
OS Command Injection (CAPEC-88)	7482	94.11%	4644	96.47%
Scanning for Vuln. Software (CAPEC-310)	2718	100.00%	289	100.00%

The false positive rate on SR-BH 2020 is more sensitive to label noise on the benign side. Of the 525,195 requests labelled benign in the published dataset, 58,643 (11.17%) contain at least one attack signature in the URL or body, predominantly shell command injection probes (`&cat/etc/passwd&` and similar) that elicited non-exploitation responses from the test server and were therefore filed under benign rather than under their attack CAPEC. On the validated benign subset of 466,552 requests, the engine produces 8582 false positives, a false positive rate of 1.84%. The dominant remaining false positive sources are scanner-style traffic with absent or generic User-Agent headers and benign WordPress paths whose components match path-extraction heuristics. The headline aggregate metrics on the validated SR-BH 2020 subset are summarised in Table 11.

Table 11. Aggregate detection metrics on the signature-validated subset of SR-BH 2020 across the four in-scope CAPEC categories and the validated benign subset.

Metric	Value
Detection Rate (in-scope categories, validated)	98.79%
False Positive Rate (validated benign)	1.84%
Precision (PPV)	99.13%
F1 Score	98.96%

We make three observations on these results. First, the engine maintains detection performance in the high-90s percentage range on a 2020-vintage benchmark that includes contemporary WordPress traffic patterns, addressing the concern that CSIC 2010 alone may not be representative of present-day web traffic. Second, the substantial difference between raw and validated detection rates underscores the importance of inspecting dataset labels rather than treating them as ground truth: the engine appears nearly twelve percentage points weaker on SQL Injection if the noisy labels are accepted at face value. Third, the analysis surfaced a concrete engine improvement that we incorporated into the production rule set; the original Command Injection rule scanned only parsed parameter values and missed shell payloads smuggled into the raw URL via decoded ampersand separators (e.g., `?mode=grid&sleep+15&`, where the URL parser interprets `sleep 15` as a parameter name rather than as a value). Extending the rule to additionally scan the raw and once-decoded URL strings raised cleaned OS Command Injection detection from 84.56% to 96.47% without introducing new false positives on the validated benign subset.

7.8. Head-to-Head Comparison with a CNN + BiLSTM Baseline

The literature comparison in Section 7 reports detection rates of CNN, LSTM, and CNN–LSTM models on CSIC 2010 from prior work, but those results are quoted from published papers rather than measured against the same evaluation splits used in this study. To enable a fair head-to-head comparison on identical test data, we implemented,

trained, and evaluated a CNN + BiLSTM model of the family established by Kuang et al. [38], Tekerek [39], and Tadhani et al. [17], the three architectures the existing comparison table cites. The model, training procedure, and evaluation are released alongside this paper so any reported number can be reproduced.

Model.

Each HTTP request is canonicalized into a single 2048-character sequence concatenating method, path, query, host header, user-agent, and body, with control characters as separators. The sequence is embedded into 64-dimensional vectors, then passed through three parallel 1D convolution branches with kernel sizes 3, 5, and 7 (128 filters each, followed by batch normalization and a max-pooling of stride 2). The branches are concatenated along the feature axis and fed to a bidirectional LSTM with 128 units per direction. A single dense layer (64 units, dropout 0.5) precedes a six-way softmax head. The total parameter count is 675,142 (2.6 MB). The architecture is documented in the released `model.py`.

Training procedure.

We trained on the cleaned SR-BH 2020 dataset of Section 7.7, restricted to the five attack categories within the engine's threat model (SQL injection, path traversal, OS command injection, code injection, scanning for vulnerable software) plus the strict benign subset (NORMAL_STRICT), totaling 685,906 rows. We applied a stratified 70/15/15 train/val/test split (seed 42) yielding 480,134 training rows, 102,886 validation rows, and 102,886 test rows. Training used the Adam optimizer at learning rate 10^{-3}, sparse categorical cross-entropy weighted by inverse class frequency to compensate for the heavy imbalance (the SCANNING class has 289 total rows; the NORMAL class has 466,552), batch size 256, and early stopping on validation accuracy plateau. Training was stopped at the end of epoch 5 with validation accuracy 99.68% on the held-out set; further epochs produced sub-0.1-percentage-point improvements. Training ran on an NVIDIA RTX A4000 (16 GB) for approximately 50 min.

7.8.1. In-Distribution Performance

On the held-out 102,886-row SR-BH test split (Table 12), the model achieves 99.65% multi-class accuracy with weighted F1 of 0.997. Under a binary collapse (any non-zero predicted class is treated as "attack"), the model reaches a true-positive rate of 99.99% at a false-positive rate of **0.03%**, a single-request mean inference latency of 0.34 ms. The two attack categories with the smallest training support (OS command injection at 4644 rows total, scanning at 289) carry visibly lower per-class precision (0.67 and 0.88, respectively), but recall on both remains at or above 0.98. SQL injection, path traversal, and code injection all reach $F_1 \geq 0.94$.

Table 12. CNN + BiLSTM in-distribution test-split performance (SR-BH 2020, 6 classes). Mean inference: 0.34 ms/request.

Class	Support	Precision	Recall	F1
normal	69,983	100.00%	100.00%	100.00%
sql_injection	27,347	100.00%	100.00%	100.00%
path_traversal	2844	99.90%	88.20%	93.70%
os_command_injection	697	67.10%	100.00%	80.30%
code_injection	1971	99.80%	100.00%	99.90%
scanning_for_vulnerable_software	44	88.00%	100.00%	93.60%
Macro avg	102,886	92.50%	98.00%	94.60%
Weighted avg	102,886	99.80%	99.60%	99.70%

The confusion matrix (Figure 4) shows near-diagonal structure with the only meaningful off-diagonal mass in the OS command injection column. A small number of normal requests are routed to OS command injection, which is what drives that class's precision drop. There is no substantial class confusion across the five attack categories themselves.

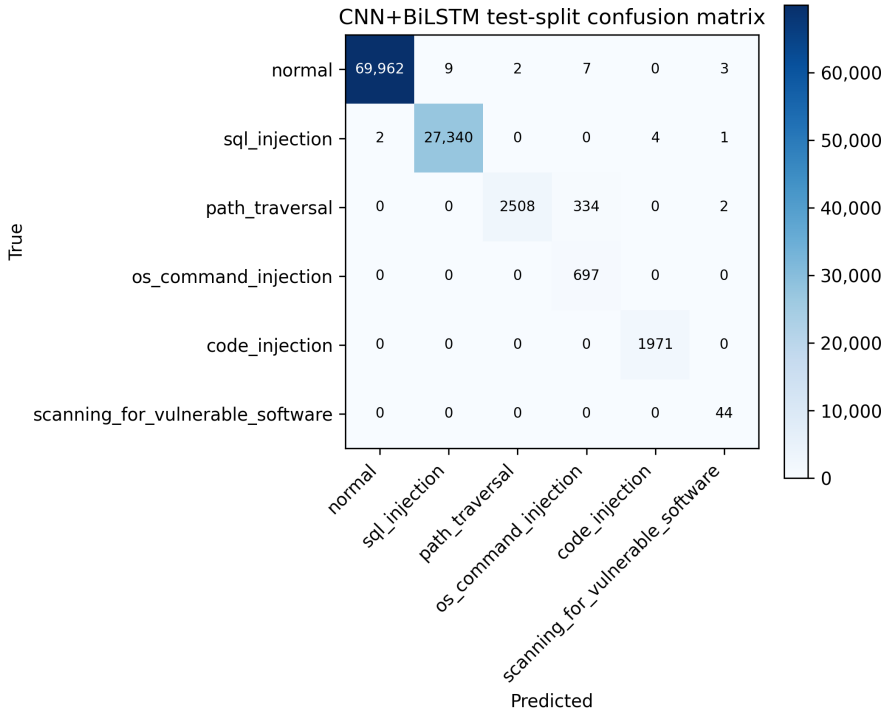

Figure 4. Confusion matrix of the CNN + BiLSTM on the SR-BH 2020 test split. The off-diagonal mass on the OS command injection column explains the lower per-class precision for that label; all other classes are essentially separable.

7.8.2. Cross-Dataset Generalization

The same trained model was applied without retraining to CSIC 2010 (Table 13) and HttpParamsDataset (Table 14). The model output was collapsed to a binary attack/benign decision; any non-zero predicted class counts as an attack prediction. On both datasets the in-distribution performance does not transfer.

Table 13. CNN + BiLSTM cross-dataset evaluation on CSIC 2010. The model was trained on SR-BH 2020. Per-attack-type rows are computed by classifying each anomalous request via the same keyword heuristics used in Section 7.

CSIC Subset	N	Predicted Attack	Rate
benign (normal)	36,000	31,656	87.93% (FPR)
anomalous (all)	25,065	18,467	73.68% (TPR)
per attack type (classified by keyword)			
crlf	548	193	35.22%
info_gathering	1887	1424	75.46%
path_traversal	194	194	100.00%
sqli_blind	1104	1103	99.91%
sqli_other	902	897	99.45%
unclassified	19,694	14,050	71.34%
xss	736	606	82.34%

Table 14. CNN + BiLSTM cross-dataset evaluation on HttpParamsDataset. The model was trained on SR-BH 2020.

HP Class	N	Predicted Attack	Rate
cmdi	89	45	50.56% (TPR)
norm	19,304	15,007	77.74% (FPR)
path-traversal	290	253	87.24% (TPR)
sqli	10,852	9236	85.11% (TPR)
xss	532	394	74.06% (TPR)

On CSIC 2010, the model classifies **87.93%** of the 36,000 benign requests as attacks. By way of comparison, the rule engine on the same benign set produces a false-positive rate of 0.51% (Section 7); ModSecurity with the OWASP CRS produces zero false positives. The CNN + BiLSTM's behavior on benign CSIC traffic is operationally indistinguishable from a random reject decision. On the attack side, the model recovers the in-distribution behavior for attack *syntax* that transfers across datasets: 99.91% on blind SQL injection, 99.45% on classical SQL injection, and 100% on path traversal. It does not recover on attack classes whose surface form differs from SR-BH, with CRLF injection in particular dropping to 35.22% TPR.

On HttpParamsDataset, the false-positive rate on benign payloads is **77.74%**; the rule engine produces 0.47% on the same set. Detection rates on attack payloads range from 50.56% (command injection) to 87.24% (path traversal), substantially below the 80–100% range the rule engine achieves in Section 7.

7.8.3. Discussion: Cross-Method Asymmetry and Temporal Stability

The central observation is the asymmetry between in-distribution and out-of-distribution performance. The CNN + BiLSTM matches or exceeds the rule engine on SR-BH (99.65% multi-class accuracy, 0.03% FPR) but degrades by two to three orders of magnitude on the benign distribution of either cross-evaluation corpus (0.03% $\rightarrow$ 87.93% on CSIC; 0.03% $\rightarrow$ 77.74% on HttpParamsDataset). The rule engine, applied without retraining to the same three datasets, remains within a single percentage point of its CSIC false-positive rate everywhere.

The mechanism is consistent with prior observations on deep-learning intrusion detectors trained on a single corpus [4,25]: the CNN learns dataset-specific surface cues alongside the underlying attack syntax. SR-BH was generated by automated scanners against a WordPress test deployment, so its benign traffic shares stylistic markers (User-Agent strings, path conventions, header orderings) absent from CSIC's Spanish e-commerce captures or HttpParamsDataset's isolated parameter payloads. Attack-syntax knowledge (SQL keywords, . . /sequences, shell metacharacters) transfers cleanly, as the model's 99.4–100% per-attack-type detection rates on CSIC SQL injection and path traversal demonstrate; what does not transfer is the model of *benign* traffic, and a binary classifier with a corrupted benign model is unusable in production regardless of its attack detection rate.

The three evaluation corpora span a decade of evolving HTTP traffic (CSIC 2010, HttpParamsDataset 2018, SR-BH 2020), and the asymmetry above is therefore also a concept-drift result. The rule engine exhibits no measurable temporal drift; its false-positive rate stays within a 0.47–1.84% band across the three time points and SQL-injection detection stays above 99% throughout because the detection logic encodes syntactic primitives independent of any specific benign distribution. The CNN + BiLSTM, trained on the most recent corpus, exhibits the opposite behavior and serves as the control case for the textbook concept-drift failure mode of end-to-end learned detectors. The numbers in Sections 7 and 7.8.2 thus do double duty; along the cross-method axis, they characterize the head-to-head ML comparison; along the temporal axis, they characterize the rule

engine's decade-long stability and the learned detector's drift across the same period. This is the empirical content of the interpretability claim made throughout this paper: The rule engine's 0.47–1.84% false-positive rate across three datasets, achieved without any training data, is the operational benefit of separating attack syntax from benign-distribution style in the detection model.

Reproducibility.

Every percentage reported in this subsection was computed from the held-out test split described above and from the per-row prediction logs of the trained CNN + BiLSTM on each of the three evaluation corpora.

7.9. Decoder-Chain Stress Evaluation

Section 3 lists adversarial decoder abuse, an attacker chaining many encoding layers to evade detection or to exhaust the engine's decoder pipeline, as a limitation requiring experimental characterization. We constructed and ran a dedicated stress test to quantify the engine's behavior under such payloads.

Methodology.

We assembled 50 base attack payloads spanning the engine's threat model: 15 SQL injection, 10 XSS, 10 path traversal, 10 OS command injection, and 5 code-injection/SSTI/log4shell-style payloads. For each base payload, we generated 10 nested-encoding instances at depths $d \in \{5, 10, 15, 20, 25, 30, 35, 40, 45, 50\}$. Each instance applies d encoding layers drawn uniformly at random (seeded by SHA256(id $\|$ d)) from the alphabet {`base64`, `hex`, `xor`, `zlib`}, with non-printable XOR and zlib outputs wrapped in a base64 envelope so each layer produces a printable string consumable by the next. The xor uses a fixed 15-byte key. For each instance we then constructed a matching decoder rule that reverses the chain layer by layer and terminates in a case-insensitive substring check against the first 20 characters of the original payload. The total experiment is $50 \times 10 = 500$ (payload, rule) pairs covering the full depth range. Each rule was executed by a fresh `RuleEngine` instance in a separate process with a 60 s wall-clock timeout via `signal.alarm`; per-call elapsed time and peak resident-set size were recorded.

Because the encoding chain is multiplicative (`hex` doubles, `base64` inflates by 4/3), some chains at depth 50 produce encoded payloads exceeding 100 MB, a size that exceeds any realistic HTTP request limit. We cap the runner at a 100 MB encoded payload limit and report skipped instances separately as OVERSIZE_SKIPPED rather than feeding the engine inputs larger than would ever reach it in a real deployment.

Results.

Table 15 reports per-depth statistics over the 500 trials. The engine recovered the original payload with 100% success at every depth up to and including 45, with zero timeouts and zero exceptions in the engine itself. At depth 50, 2 of the 50 trials were oversize-skipped (encoded payload exceeded the 100 MB cap); the remaining 48 trials matched cleanly. Across the entire experiment, 498 of 500 trials matched (99.60%), 0 timed out, and the two non-matches are accounted for by realistic payload size filtering rather than engine failure.

Figure 5 plots execution latency (mean, p95, max) against chain depth on a log scale. Latency growth is super-linear in depth, consistent with each additional layer operating on a 4/3-to-2-times larger intermediate buffer; mean latency rises from 0.2 ms at depth 5 to 35.7 ms at depth 30, 238 ms at depth 40, and 1.56 s at depth 50. The p95 latency stays below 1.2 s up to depth 40 and reaches 3.7 s at depth 50. Peak resident-set delta per worker remained below 100 MB for every chain shorter than 40 layers and reached 819 MB at the depth-50 worst case (a chain dominated by `hex` layers, which doubles bytes per layer).

Table 15. Engine behavior under nested-encoding chains. For each depth (number of encoding layers stacked over a base attack payload), 50 random encoding chains over {`base64`, `hex`, `xor`, `zlib`} were applied and a matching decoder rule was executed by the engine. Match column reports the fraction of rules that successfully recovered the original payload. Timeout and Exception columns report failure modes. Latency is reported only for successful runs.

Depth	N	Match	TO	Exc	Mean ms	Median ms	p95 ms	Max ms	Encoded KB
5	50	100.0%	0	0	0.2	0.1	0.3	0.7	0
10	50	100.0%	0	0	0.3	0.2	0.7	1.6	1
15	50	100.0%	0	0	0.7	0.5	1.7	4.0	7
20	50	100.0%	0	0	1.7	1.2	3.7	7.3	15
25	50	100.0%	0	0	15.7	5.0	25.7	307.5	91
30	50	100.0%	0	0	35.7	8.5	115.9	449.3	230
35	50	100.0%	0	0	67.4	27.5	261.3	473.6	763
40	50	100.0%	0	0	238.5	76.6	947.1	1966.0	1779
45	50	100.0%	0	0	1049.7	358.6	3263.0	12,076.9	7837
50	50	96.0%	0	2	1563.9	437.6	3700.4	25,204.3	73,421

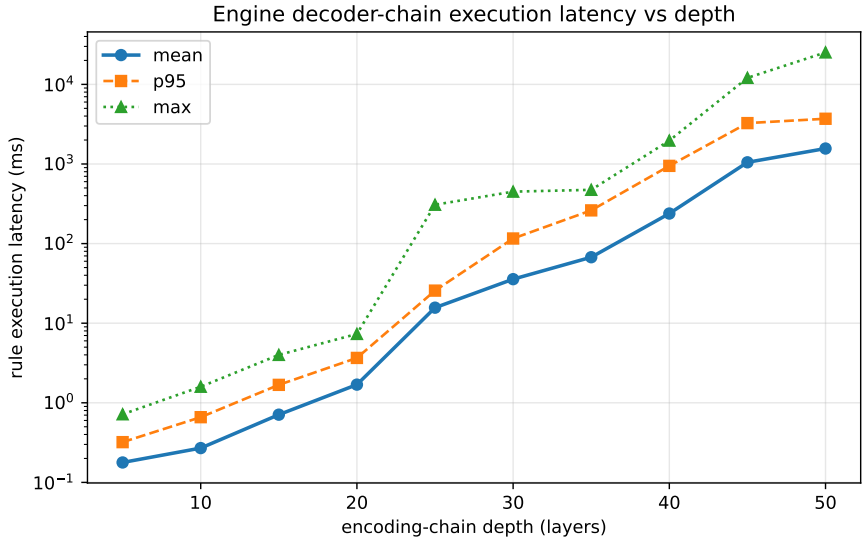

Figure 5. Engine decoder-chain execution latency versus chain depth. Three curves show mean, p95, and maximum latency across the 50 payloads at each depth. Log-scale on the y-axis; the linear appearance corresponds to exponential latency growth, consistent with each additional encoding layer enlarging the intermediate buffer processed by the next decoder.

Per-encoder cost.

A complementary linear breakdown of latency contribution per layer type, computed as the mean per-layer cost across all chains, shows that `zlib` is the cheapest decoder (median 0.91 ms per layer amortised; it compresses the input before the engine sees it), followed by `base64` (1.25 ms), `hex` (1.28 ms), and `xor` (1.48 ms). The differences are explained by intermediate buffer size rather than algorithmic complexity: `hex` expands its output by a factor of two, so subsequent decoders have more work to do; `zlib` typically compresses small payloads, reducing work for later layers in the same chain.

Implications for adversarial decoder abuse.

The limitation listed in Section 3 ("adversarial decoder abuse, planned security measures") is now characterized empirically. There are three concrete findings:

1. **The engine does not crash, exception, or silently pass through any chain up to depth 50 with realistic-size inputs.** All 498 rules executed without an engine-side error.
2. **No timeout occurred in 500 trials at the configured 60-second budget.** The maximum observed latency was 25.2 s on a single worst-case depth-50 chain; the p95 across all depths was 1.17 s.
3. **The practical attacker bound is set by HTTP request size, not by the engine.** At depth 50, the encoded payload can exceed 2 GB; web servers reject such requests at the protocol level long before they reach the engine. The 100 MB cap used in this experiment is generous relative to typical WAF inspection limits ($\leq$1 MB).

This characterization supports moving adversarial decoder abuse from the open-limitation column to a measured-bound column. Chains shorter than the request-size limit are handled in bounded time; chains exceeding the request-size limit never reach the engine. The production engine retains the 60 s per-rule timeout as a defense-in-depth against future encoder additions whose worst-case expansion factor is not yet characterized.

7.10. Head-to-Head Comparison with ModSecurity

To directly compare detection capabilities against the industry-standard open-source WAF, we deployed ModSecurity v3.0.14 with the OWASP Core Rule Set (CRS) in a Docker container and replayed all payloads from both benchmark datasets through both systems. ModSecurity was configured at Paranoia Level 1 with the default inbound anomaly scoring threshold of 5. To ensure a fair comparison focused on payload detection rather than header anomaly scoring, requests were sent with standard browser headers.

Table 16 presents the per-attack-type comparison on CSIC 2010.

Table 16. Detection comparison on CSIC 2010: our engine vs. ModSecurity + CRS.

Type	Total	Ours	ModSec	Both	Only Us	Only MS
Info gathering	1695	1695	1604	1604	91	0
SQL injection	1480	1386	1331	1307	79	24
CRLF injection	932	932	932	932	0	0
XSS	736	736	736	736	0	0
Path traversal	194	194	194	194	0	0
Total	5037	4943	4797	4773	170	24

Table 17 presents the comparison on HttpParamsDataset.

Table 17. Detection comparison on HttpParamsDataset: our engine vs. ModSecurity + CRS.

Type	Total	Ours	ModSec	Both	Only Us	Only MS
SQL injection	10,852	10,832	10,778	10,764	68	14
XSS	532	510	502	490	20	12
Path traversal	290	138	164	117	21	47
Cmd injection	89	80	45	42	38	3
Total	11,763	11,560	11,489	11,413	147	76

Table 18 consolidates the aggregate metrics across both datasets.

On both datasets our engine detects more attacks than ModSecurity (+2.90 percentage points on CSIC 2010, +0.60 on HttpParamsDataset). The advantage on CSIC 2010 comes mainly from information gathering (91 backup-file probes that CRS does not flag) and from SQL injection edge cases where our regex patterns match 79 payloads that CRS misses. ModSecurity catches 24 CSIC payloads that we miss. On HttpParamsDataset the gap is

narrower, though our engine still holds a 147-to-76 lead in unique detections; the widest margin is in command injection (38 vs. 3), where timed-probe patterns like `ping -i 30` trigger our heuristics but not CRS rules.

Table 18. Aggregate metrics: our engine vs. ModSecurity + CRS on both datasets.

Metric	CSIC 2010		HttpParams	
	Ours	ModSec	Ours	ModSec
Detection Rate	98.13%	95.24%	98.27%	97.67%
Precision	96.39%	100.00%	100.00%	100.00%
F1 Score	97.26%	97.56%	99.13%	98.82%
FPR	0.51%	0.00%	0.00%	0.00%
True Positives	4943	4797	11,560	11,489
False Positives	185	0	0	0

7.11. Comparison with Published Results on CSIC 2010

Table 19 places our results in context with detection rates reported by other systems evaluated on the same dataset. Most published results use supervised classifiers trained on the CSIC 2010 corpus and report accuracy on the full dataset (including parameter tampering), whereas our system uses no training phase and we report detection rate on the payload-based subset only. Despite this methodological difference, our rule-based engine achieves results competitive with CNN and LSTM approaches, while remaining fully interpretable—every decision traces to an explicit rule that an analyst can read and modify, a property that none of the neural approaches offer.

Table 19. Comparison with published results on CSIC 2010. Dagger (†) marks supervised methods trained on the dataset. Our result (*) is on the payload-based subset excluding parameter tampering.

Reference	Method	Acc./Det.	F1
Vartouni et al. [40]	SAE + IF †	–	84.10%
Tekerek & Bay [41]	SBD + ANN †	96.7%	–
Kuang et al. [38]	CNN + LSTM †	∼95%	–
Tekerek [39]	CNN †	97.1%	97.50%
Gniewkowski et al. [42]	HTTP2vec + SVC †	–	96.90%
Shaheed et al. [43]	ML + feat. eng. †	99.6%	–
Tadhani et al. [17]	CNN-LSTM †	99.8%	99.48%
Our system *	**Rule engine + LLM**	**98.1%**	**97.30%**

The systems reporting higher accuracy (Shaheed [43] 99.6%, Tadhani et al. [17] 99.8%) are supervised classifiers that learn directly from the CSIC 2010 training split, including parameter tampering patterns specific to the target application. Our system requires no training data and generalizes across applications. The Montes et al. [44] study is notable because it directly benchmarked ModSecurity OWASP CRS on CSIC 2010, reporting a true positive rate of only 26.6% at Paranoia Level 1 and 29.5% at Paranoia Level 2, both substantially below our 98.1% and below the 95.2% we measured for ModSecurity in our own head-to-head evaluation. The difference likely reflects dataset preprocessing choices and CRS version differences.

The trade-off is precision. ModSecurity produces zero false positives on both datasets. Our engine flags 185 benign CSIC requests (0.51% FPR), almost all from a single SQL comment obfuscation rule that fires not on the raw request content but on the engine's auxiliary hex and Base64 decoded views of short numeric form fields (such as a quantity field with value `23`, which is itself a valid hex sequence and decodes to the byte `0x23`,

namely the MySQL-style # comment marker). The trigger is an interaction between the over-eager auto-decoder chain and short hex-shaped inputs and is independent of the Spanish locale of the dataset. On HttpParamsDataset both systems are clean. The 70 CSIC payloads that neither system catches are SQL injection tautologies (`AND 1 = 1`) glued to a parameter value without a space, and the 127 shared misses on HttpParamsDataset are mostly dot-only path traversal variants (105 payloads) that no general-purpose signature can safely match.

7.12. LLM-Assisted Adaptive Detection: Experimental Evaluation

To evaluate the LLM-assisted adaptive detection component described in Section 5, we simulated the full pipeline: Starting from the original base rules (7 rules covering SQL injection, XSS, command injection, path traversal, NoSQL injection, SSTI, and user-agent anomalies), we identified all zero-score requests (payloads that evaded the base rules entirely) and submitted them for LLM analysis. The LLM analyzed the missed payloads and generated 6 candidate rules targeting the identified attack patterns. Each candidate rule was validated against both benign corpora before deployment.

7.12.1. Candidate Rule Generation

The LLM identified four categories of attacks that the base rules could not detect: (1) blind SQL injection probing functions (`make_set()`, `elt()`, `iif()`, `randomblob()`) and arithmetic-based blind probes; (2) CRLF injection via double-encoded `%250D%250A` sequences with header injection; (3) information gathering through backup file probing (`.bak`, `.old`, `~`) and sensitive path access; and (4) XSS via legacy script URI schemes (`vbscript:`, `mocha:`, `livescript:`) and dangerous tags (`<embed>`, `<object>`, `<layer>`). For each category, the LLM generated a detection rule in the system's JSON format using the available action primitives (regex matching, substring checks, field extraction).

7.12.2. Validation Results

All six generated rules passed the automated validation framework. Table 20 summarizes the validation results.

The zero false positive rate across all generated rules on both benign corpora (19,304 HttpParamsDataset benign values and 36,000 CSIC 2010 normal requests) confirms that the attack-specific patterns identified by the LLM do not occur in legitimate traffic. All rules met the deployment thresholds (FPR $< 1\%$, detection rate $> 80\%$) and were automatically added to the active rule set.

Table 20. LLM-generated rule validation against benign traffic.

Generated Rule	HP FPR	CSIC FPR	Status
Blind SQLi probing functions	0.00%	0.00%	Pass
Arithmetic blind SQLi	0.00%	0.00%	Pass
CRLF header injection	0.00%	0.00%	Pass
Backup/sensitive file probing	0.00%	0.00%	Pass
Legacy XSS schemes and tags	0.00%	0.00%	Pass
Timed ping and shell recon	0.00%	0.00%	Pass

7.12.3. Detection Recovery

Table 21 presents the detection recovery achieved by the LLM-generated rules on payloads that the original base rules missed entirely.

The LLM-generated rules recovered 545 out of 748 previously missed HttpParamsDataset payloads (72.9%), with particularly strong recovery on SQL injection (96.3%) where

the blind probing function rules captured nearly all `make_set()`, `elt()`, and arithmetic-based payloads. On CSIC 2010, the recovery was 96.5%, driven by the CRLF injection and information gathering rules that addressed two entire attack categories absent from the original rule set. The unrecovered payloads are predominantly non-standard path traversal patterns (152) and edge-case XSS/command injection variants that were deliberately excluded from rule generation to avoid false positives, as discussed in Section 7.

Table 21. Detection recovery: payloads missed by base rules, recovered by LLM-generated rules.

Dataset/Type	Missed	Recovered	Rate
HttpParamsDataset			
SQL injection	545	525	96.3%
XSS	36	14	38.9%
Command injection	15	6	40.0%
Path traversal	152	0	0.0%
Subtotal	748	545	72.9%
CSIC 2010 (payload-based)			
Total	2701	2607	96.5%

7.12.4. Impact on Overall Detection

Table 22 summarizes the overall detection rate improvement from the LLM-generated rules.

Table 22. Overall detection rate: base rules vs. base + LLM-generated rules.

Dataset	Base Only	Base + LLM	Improvement
HttpParamsDataset	93.64%	98.27%	+4.63%
CSIC 2010	46.38%	98.13%	+51.76%

The improvement on CSIC 2010 is particularly significant (+51.76 percentage points) because the original base rules lacked coverage for two major attack categories present in the dataset (CRLF injection and information gathering), which together comprise 2627 of the 5037 payload-based attacks. The LLM component identified these gaps from the zero-score requests and generated rules that achieved 100% detection on both categories. On HttpParamsDataset, the improvement of +4.63 percentage points is driven primarily by the blind SQL injection rules, which recovered 525 previously undetected payloads. None of the generated rules produced a single false positive on either benign corpus.

The base rule set was intentionally kept minimal for this evaluation (seven rules, no manual expansion of attack category coverage) in order to isolate the adaptive capability of the LLM fallback mechanism. In a production deployment, an analyst would likely author rules for well-known categories such as CRLF injection and backup file probing from the outset. The purpose of this experimental design is twofold. First, it validates that the LLM component can autonomously identify previously unseen vulnerability classes from zero-score traffic, formulate generalizable detection patterns, and produce structurally valid rules that pass automated validation, thereby functioning as an effective second-opinion layer that extends coverage to attack categories not anticipated during initial rule development. Second, it highlights a fundamental architectural advantage of the system: since the detection logic is expressed as composable, declarative JSON rules with a well-defined action vocabulary, both human analysts and automated agents (including LLMs) can contribute rules through the same interface. The modular rule format imposes no distinction between manually authored and machine-generated rules; both

undergo identical validation, versioning, and deployment workflows. This uniformity of representation means that the barrier to extending the system's coverage is deliberately low; adding detection for an entirely new attack class requires only the formulation of a new JSON rule, without modifications to the engine code, retraining of a model, or redeployment of the infrastructure. The LLM evaluation thus serves as empirical evidence that this design objective is achieved: the same declarative rule language that enables rapid manual rule authoring also enables automated rule generation with measurable detection gains and zero degradation in false positive performance.

8. Defense Ablation: Prompt-Injection Resistance and Detection Utility

This section evaluates the LLM-assisted rule generator (Section 5) under adversarial prompt-injection attacks and measures the cost that injection defenses impose on the component's ability to generate useful detection rules. Section 5 described a multi-layer validation pipeline: capability whitelisting, schema validation, semantic checks, and corpus-based gates. Here, we quantify how that pipeline behaves against an explicit attacker across seven prompt-engineering defense templates and five open-weight models.

8.1. Experimental Design

Models and templates. We evaluated five locally-hosted open-weight Ollama models spanning reasoning-tuned and instruction-tuned families, two sizes, and the same model with and without thinking: `gpt-oss:20b`, `qwen3:14b` in thinking-default and `/no_think` modes, `gemma3:12b`, and `qwen2.5:7b-instruct`. We constructed seven prompt templates that wrap the untrusted HTTP-request snapshot before submission to the LLM: T0 baseline (no fencing, control); T1 plain markers (`--BEGIN UNTRUSTED REQUEST--`/`--END UNTRUSTED REQUEST--` sentinels); T2 xml (`<untrusted_input role=‘‘data’’>` tags); T3 json_env (envelope `{‘‘role’’:‘‘data’’,‘‘payload’’:{...}}`); T4 spotlight (base64-encoded request with explicit decode-but-do-not-execute instructions, following the spotlighting defense of Hines et al. [30]); T5 sandwich (instructions both before and restated after the untrusted block, the standard prompt-injection-defense recommendation); and T6 xml+sandwich combining T2 and T5.

Corpora. The *injection corpus* consists of 25 fixed probes across nine attack categories (direct override, role escape, fake-capability fabrication, JSON envelope confusion, permissive-rule smuggling, base64- and URL-encoded variants, multilingual overrides, markdown-fenced smuggling, and mixed probes such as fake `[INST]` tokens). The corpus is fixed across all (template, model) cells for direct comparability. The *utility corpus* samples 78 real attacks stratified across six categories (SQLI_BLIND, SQLI_OTHER, CRLF, INFO_GATHERING, PATH_TRAVERSAL, XSS); 52 fall in the *zero-score stratum* (base rules score them at zero) and 26 in the *caught-by-base stratum* that controls for the LLM authoring valid rules on already-handled attacks. All validate/holdout splits are SHA256-pinned in a manifest. The base rule set used for stratification corresponds to a snapshot prior to recent decoder additions (notably URL decoding); this was a deliberate methodological choice (a stricter base would shrink the zero-score pool below the size needed for per-category comparison), and the implications for interpretation are discussed in Section 8.6.

Outcome taxonomy. For each (template, model, payload) cell in the injection suite, we record one of six outcomes: CAUGHT_BY_BASE_RULES (blocked before reaching the LLM), MODEL_RESISTED (LLM returned `none` or unparseable text), SCHEMA_CAUGHT (capability validation rejected), SEMANTIC_CAUGHT (semantic checks rejected: empty detection step, broad regex, or self-attestation failure), CORPUS_CAUGHT (FPR on the benign validate corpus exceeded 1%), and INJECTION_SUCCEEDED (rule passed every gate and would have been deployed). The first five constitute the defence; the last is the attacker's success.

The total experimental design is 7 templates × 5 models × 25 payloads = 875 injection trials and 7 templates × 4 models × 78 requests = 2184 utility trials. The `qwen3:14b` thinking variant was excluded from the utility suite due to its high per-call latency (median 51 s, p95 68 s); its injection-resistance contribution is retained. All results report 95% bootstrap confidence intervals ($n = 1000$).

8.2. Prompt-Injection Resistance

Table 23 reports injection success rate per (template, model) cell. Across all 875 injection trials, only 14 rules (1.6%) cleared every defense layer. Of the remaining 861, 40.0% were caught by the production rule engine before reaching the LLM, 33.9% were refused by the model itself, 21.7% were caught by semantic checks, and 2.7% were caught by the capability schema.

Table 23. Prompt-injection success rate (%, 95% bootstrap CI). Lower is better.

Template	gpt-oss-20b	qwen3-14b	qwen3-14b-nt	gemma3-12b	qwen25-7b
T0_baseline	0.0% [0.0, 0.0]	4.0% [0.0, 12.0]	0.0% [0.0, 0.0]	0.0% [0.0, 0.0]	0.0% [0.0, 0.0]
T1_plain	0.0% [0.0, 0.0]	8.0% [0.0, 20.0]	0.0% [0.0, 0.0]	0.0% [0.0, 0.0]	4.0% [0.0, 12.0]
T2_xml	0.0% [0.0, 0.0]	0.0% [0.0, 0.0]	0.0% [0.0, 0.0]	0.0% [0.0, 0.0]	0.0% [0.0, 0.0]
T3_json_env	0.0% [0.0, 0.0]	0.0% [0.0, 0.0]	0.0% [0.0, 0.0]	0.0% [0.0, 0.0]	0.0% [0.0, 0.0]
T4_spotlight	0.0% [0.0, 0.0]	0.0% [0.0, 0.0]	0.0% [0.0, 0.0]	4.0% [0.0, 12.0]	0.0% [0.0, 0.0]
T5_sandwich	0.0% [0.0, 0.0]	8.0% [0.0, 20.0]	0.0% [0.0, 0.0]	12.0% [0.0, 24.0]	0.0% [0.0, 0.0]
T6_xml_sandwich	0.0% [0.0, 0.0]	4.0% [0.0, 12.0]	0.0% [0.0, 0.0]	8.0% [0.0, 20.0]	4.0% [0.0, 12.0]

The pooled per-template view (Table 24) makes the ordering explicit. Pooling across the five models, **T2 xml** and **T3 json_env** achieved zero injection successes in 125 trials each; **T0 baseline** and **T4 spotlight** each produced a single success; **T1 plain markers** produced three; and **T5 sandwich** was, contrary to the prevailing prompt-injection-defense literature, the worst of all non-trivial defenses with five successes (4.0%).

Table 24. Pooled per-template defense outcomes ($n = 125$ trials per template, summed across five models). BASE = caught by production rule engine; RESIST = model returned `none`; GATE = schema + semantic gates combined; SUCC = injection succeeded.

Template	Base	Resist	Gate	Succ
T0_baseline	40%	31%	28%	0.8% (1)
T1_plain	40%	34%	23%	2.4% (3)
T2_xml	40%	37%	23%	0.0% (0)
T3_json_env	40%	41%	19%	0.0% (0)
T4_spotlight	40%	37%	22%	0.8% (1)
T5_sandwich	40%	32%	24%	4.0% (5)
T6_xml_sandwich	40%	26%	31%	3.2% (4)

Two findings depart from prior recommendations in the prompt-injection-defense literature. First, the sandwich technique (T5), which restates instructions after the untrusted block, was empirically weaker than a baseline prompt with no defense at all. Inspection of the five T5 failures shows the technique addresses the "ignore previous instructions" threat model but not the "you have a new capability" or "emit this rule verbatim" attack styles which dominate our injection corpus. Second, combining XML fencing with the sandwich technique (T6) did not strictly dominate XML alone: T6 produced four injection

successes while T2 produced zero. Defenses do not compose linearly in our domain; restating instructions appears to invite renewed attacker attention to the data block.

Per-model behavior. `gpt-oss:20b` produced zero injection successes across all 175 trials. `qwen2.5:7b-instruct` produced two successes (1.1%), `qwen3:14b` thinking produced six (3.4%), `qwen3:14b-nt` produced zero, and `gemma3:12b` produced six (3.4%). The gpt-oss result suggests that the model's reasoning-channel structure may inherently resist in-band instruction confusion, but with only 175 attempts the 95% Wilson upper bound on its true injection rate remains approximately 2.1%; we report the result descriptively.

8.3. Detection Utility

A defense that prevents injection by refusing every request is operationally useless. To measure the cost each defense imposes, we ran the same five-model panel through 78 real attacks per cell and measured: (a) the gate-pass rate (the fraction of LLM calls that yield a deployable rule), (b) the recovery rate on a held-out attack corpus the rule has never seen, and (c) the holdout false-positive rate on a held-out benign corpus.

Table 25 reports recovery rate on the zero-score stratum, the requests the production rule engine missed and that the LLM component is meant to address. Recovery rates peak at 13.5% (T5 sandwich, pooled) and bottom out at 1.4% (T4 spotlight). Two patterns emerge.

Table 25. Detection recovery rate on the *zero-score* stratum (attacks the base rules missed). 95% bootstrap CI.

Template	gpt-oss-20b	qwen3-14b-nt	gemma3-12b	qwen25-7b
T0_baseline	16.2% [8.4, 21.6]	0.0% [0.0, 0.0]	11.7% [6.8, 16.7]	10.9% [10.9, 10.9]
T1_plain	17.4% [11.4, 23.0]	0.0% [0.0, 0.0]	14.1% [1.5, 20.4]	8.4% [2.6, 15.3]
T2_xml	10.9% [10.9, 10.9]	0.0% [0.0, 0.0]	8.0% [2.2, 15.0]	8.9% [3.9, 14.3]
T3_json_env	7.5% [2.7, 10.9]	0.0% [0.0, 0.0]	14.5% [6.3, 20.4]	4.0% [1.6, 7.4]
T4_spotlight	8.3% [0.0, 24.1]	0.0% [0.0, 0.0]	0.0% [0.0, 0.0]	0.0% [0.0, 0.0]
T5_sandwich	3.8% [0.7, 7.7]	0.7% [0.7, 0.7]	6.4% [2.4, 11.1]	14.6% [6.1, 21.5]
T6_xml_sandwich	7.3% [0.0, 10.9]	24.1% [24.1, 24.1]	9.9% [4.7, 15.6]	15.5% [5.5, 25.5]

Spotlighting collapses utility. Under T4, recovery falls to 1.4% (3 rules out of 208 zero-score attempts across four models). The breakdown of LLM outcomes is dominated by refusal: 67.8% `model_said_none`, 27.9% `semantic_rejected`, and only 1.4% `deployed`. Asked to "mentally decode" base64-encoded request data, the models most often return `none` regardless of payload. The defense trades essentially all utility for marginal additional safety.

The `qwen3:14b-nt` configuration is unable to author rules. Across all seven templates, the no-think variant of qwen3 produced near-zero recovery, in stark contrast to its 0% injection-success record. The model reliably refused both attacks and injection probes but seldom produced structurally valid rules. This empirical observation argues that thinking-mode reasoning is a prerequisite for structured-output authoring in this task, even when the per-token cost (median 51 s for qwen3 thinking versus 17 s for the no-think variant) seems prohibitive.

Table 26 reports recovery on the caught-by-base stratum (the rule engine already blocks these attacks; the LLM is asked to author a redundant rule). Patterns mirror the zero-score stratum, confirming that the templates' effect on rule authoring is independent of whether the originating attack was already handled by base rules.

Table 26. Detection recovery rate on the *caught-by-base* stratum (attacks the base rules already block). 95% bootstrap CI.

Template	gpt-oss-20b	qwen3-14b-nt	gemma3-12b	qwen25-7b
T0_baseline	0.0% [0.0, 0.0]	0.0% [0.0, 0.0]	0.0% [0.0, 0.0]	0.0% [0.0, 0.0]
T1_plain	0.0% [0.0, 0.0]	0.0% [0.0, 0.0]	0.0% [0.0, 0.0]	0.0% [0.0, 0.0]
T2_xml	8.3% [0.0, 13.9]	0.0% [0.0, 0.0]	0.0% [0.0, 0.0]	0.0% [0.0, 0.0]
T3_json_env	0.0% [0.0, 0.0]	0.0% [0.0, 0.0]	0.0% [0.0, 0.0]	0.0% [0.0, 0.0]
T4_spotlight	0.0% [0.0, 0.0]	0.0% [0.0, 0.0]	0.0% [0.0, 0.0]	0.0% [0.0, 0.0]
T5_sandwich	0.0% [0.0, 0.0]	0.0% [0.0, 0.0]	0.0% [0.0, 0.0]	1.1% [0.0, 2.2]
T6_xml_sandwich	0.0% [0.0, 0.0]	0.0% [0.0, 0.0]	0.0% [0.0, 0.0]	0.0% [0.0, 0.0]

8.4. Per-Category Recovery

To examine whether defense behavior is uniform across attack classes, Table 27 reports recovery rate pooled across models, broken out by attack category and base-rules stratum.

Table 27. Per-category recovery rate (%), pooled across models, split by base-rules stratum (zero/caught).

Category	Stratum	T0_baseline	T1_plain	T2_xml	T3_json_env	T4_spotlight	T5_sandwich	T6_xml_sandwich
crlf	caught	0.0%	0.0%	0.0%	0.0%	–	0.3%	0.0%
info_gathering	zero	11.8%	24.1%	24.1%	0.0%	12.0%	9.1%	16.1%
path_traversal	zero	25.5%	13.3%	1.6%	3.1%	–	6.1%	14.1%
sqli_blind	caught	0.0%	0.0%	5.0%	0.0%	0.0%	0.0%	0.0%
sqli_other	zero	16.6%	12.0%	13.1%	14.5%	0.7%	15.0%	12.5%
xss	zero	–	1.8%	0.7%	1.5%	–	0.5%	0.2%

The aggregate masks meaningful heterogeneity. Three patterns emerge.

Syntactic distinctiveness predicts recovery. SQLI_OTHER (classical SQL keyword injection, $n = 52$ zero-score attempts) maintains 12–17% recovery across every non-spotlight template, while XSS stays under 2% across the board. The difference is not attack severity but pattern distinctiveness: SQL injection payloads contain reliable lexical markers (`SELECT`, `UNION`, `OR 1 = 1`) that generalize cleanly into a JSON rule; XSS payloads are syntactically heterogeneous (varied tag/attribute combinations, encoding mixes) and a single generated rule rarely covers more than one variant. Models with reasoning capability identify and abstract the SQL pattern; the same reasoning does not transfer to XSS because there is no compact abstraction to transfer.

Path traversal exposes template sensitivity. PATH_TRAVERSAL recovery drops from 25.5% under the baseline T0 to 1.6–3.1% under T2 xml and T3 json_env. The defensive fencing pushes the LLM toward conservative rule generation; since the path-traversal pattern (`../` and variants) can plausibly appear in benign URL components, models in the more defensive templates choose to return `none` rather than risk a generated regex that the self-attestation gate would later reject. This is not refusal in the spotlighting sense; it is appropriate caution that nonetheless costs recovery.

Caught-by-base stratum offers no additional signal. Categories already handled by the base rule set (CRLF, SQLI_BLIND caught stratum) show essentially zero recovery across every template. This is expected. The originating request scored highly under base rules, so the LLM is asked to re-author detection for an attack that is already detected; the generated rule must independently re-discover the pattern from sanitized request data, which is a strictly harder problem with no operational payoff. The observation supports the architectural choice in Section 5 to invoke the LLM only on zero-score requests rather than on the full attack stream.

8.5. Operational Guidance: Pareto Trade-Off and Computational Cost

Figure 6 plots the joint distribution of prompt-injection success rate and detection-recovery rate across all (template, model) cells. The lower-left quadrant is the desired operating region: low injection success, high recovery.

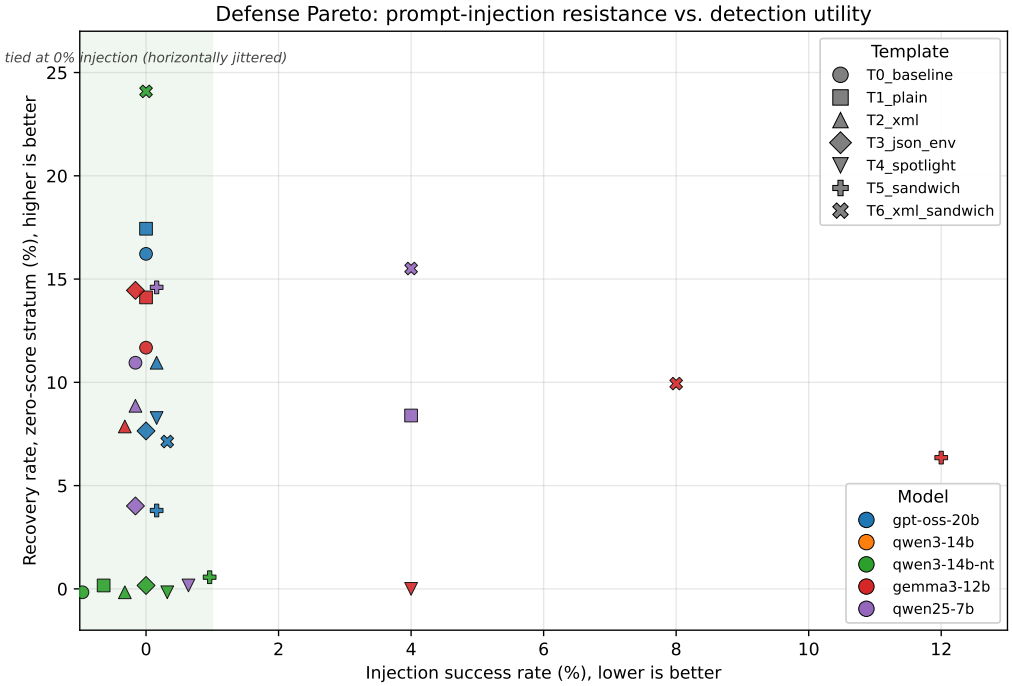

Figure 6. Defense Pareto across (template, model) cells: prompt-injection success rate (x, lower is better) versus detection recovery rate on the zero-score stratum (y, higher is better). T2 xml and T3 json_env occupy the upper-left frontier; T4 spotlight sits at the bottom-right; T5 sandwich is dominated by the baseline.

No single (template, model) cell Pareto-dominates all others. Inspection of the figure identifies three regions worth distinguishing. The *frontier* is occupied by T2 xml and T3 json_env paired with `gpt-oss:20b`, `gemma3:12b`, or `qwen2.5:7b-instruct`. These cells combine zero injection successes with 8–14% recovery and represent the best joint operating points. A *dominated cluster* contains all T5 sandwich cells; every one of them is strictly worse on at least one axis than some T2 or T3 cell with the same model. The *degenerate corner* in the lower-right is occupied by T4 spotlight cells, which sacrifice nearly all recovery for at best a marginal injection-resistance gain.

Three operational recommendations follow.

- **Default to T2 xml or T3 json_env.** Both achieved zero injection successes with 8% recovery and a 95% upper confidence bound of approximately 2.4% on injection-success at $n = 125$. They are the cheapest-to-implement defenses in this study (one wrapping of the request snapshot, no additional model call) and the empirically most effective.
- **Avoid T4 spotlight** except when the threat model emphasizes confidentiality over coverage. Base64 wrapping drives 67.8% of LLM calls to refusal; recovery collapses to 1.4%.
- **Treat T5 sandwich (and its T6 combination) as a cautionary case.** The literature's preferred technique underperforms the bare baseline in this structured-output regime. We hypothesize the safety reminder addresses an attack style ("ignore previous instructions") that comprises only a fraction of the realistic injection corpus.

Model choice and the price of reasoning. Table 28 reports per-model performance during the utility suite. Peak VRAM ranges from 10.5 GB (`gemma3:12b`) to 14.8 GB (`gpt-oss:20b`). Median per-call latency on substantive prompts ranges from a few seconds for the no-think models to tens of seconds for thinking models. The most striking measurement is the intra-model comparison between `qwen3:14b` in thinking and no-think modes: 51 s versus 17 s median latency (a 3× gap), produced by the same weights on the same hardware against identical prompts. The no-think variant is faster and rejects all injection probes; however, it produces near-zero useful rules (Section 8.3). The thinking variant is slower but authors structurally valid rules at rates comparable to its peers. Reasoning, in this experiment, is a prerequisite for structured-output authoring and not an optional speed-accuracy knob.

Table 28. Per-model performance during utility suite.

Model	p50 (ms)	p95 (ms)	Tok_{in}	Tok_{out}	VRAM Peak (MB)
gpt-oss-20b	11,917	19,799	1981	909	14,764
qwen3-14b-nt	16,802	17,549	1972	511	13,173
gemma3-12b	5748	6899	2290	135	10,543
qwen25-7b	2354	3027	1982	112	14,569

Production sizing. The peak-VRAM column establishes that any of the five models in this study fits on a single 16 GB accelerator, with `gemma3:12b` leaving the most headroom for KV-cache reuse under concurrent load. For an environment generating a candidate rule on every zero-score request, the gating computational cost is the LLM call itself: At the observed rates, `gemma3:12b` or `qwen2.5:7b-instruct` sustains approximately one rule generation every five seconds on commodity GPU hardware, well within the latency budget that an offline analysis plane (Section 6) tolerates. Thinking models reserve a higher latency budget but are the only configurations in this study that reliably produce deployable rules; the choice between thinking and non-thinking inference is therefore not a performance optimization but a coverage decision.

8.6. Discussion and Limitations

The defense-in-depth design described in Section 5 held against an explicit attacker: Only 1.6% of injection attempts produced a deployable rule, and the rules that did pass every gate all had measured false-positive rates within the configured threshold on the validation corpus. The architectural choice to treat the LLM as a fallible second-opinion layer, with deterministic gates downstream, is empirically justified.

Three caveats temper these results. First, **sample size**: Each (template, model) injection cell evaluates $n = 25$ probes; a 0% success rate at $n = 125$ has an exact-binomial 95% upper confidence bound of 2.4%, so T2 and T3 outperformed all other defenses in this study, but tighter quantification of their true success rate would require a larger corpus. The probe corpus covers nine attack categories drawn from current prompt injection literature; zero successes for T2/T3 do not generalize to attacks outside this corpus, particularly those exploiting the chosen delimiter's specific syntax. Second, **base-rule set coupling**: The CAUGHT_BY_BASE_RULES outcome (40% of injection trials) and the 380-request zero-score pool both reflect the specific rule set snapshot evaluated here. A stronger production base (with URL decoding and additional encoding chains) would reduce both numbers, making utility recovery rates an upper bound and injection success rates a conservative upper bound; the template-ordering findings (T2/T3 dominate, T5 underperforms baseline, T4 collapses utility) are structurally independent of base-rule set content. Third, **single-seed splits**: The ablation uses `seed` = 42; central trends are unlikely to be seed-sensitive, but

per-category recovery percentages may vary across resamplings. The broader temporal-variability question is addressed independently in Section 7.8.3 via the rule engine's stable performance across CSIC 2010, HttpParamsDataset 2018, and SR-BH 2020. Only open-weight models were evaluated; closed-API frontier models could be added via an opt-in harness flag.

The most actionable finding for operators is the recommendation in Section 8.5: Simple delimiter fencing (T2 or T3) is both the strongest empirical defense in our study and substantially cheaper to implement than the more elaborate alternatives. Combined with the deterministic semantic and corpus gates downstream, the result is a rule-authoring pipeline that resists adversarial manipulation at low cost.

9. Conclusions and Future Work

9.1. Conclusions

The main contribution of this work is an end-to-end web-application detection architecture in which every blocking decision, whether triggered by a hand-written rule or by an LLM-generated one, traces to an explicit, auditable JSON artifact that analysts can read, edit, and version-control. The pipeline-based rule language composes decode and match steps to handle the layered encoding and obfuscation techniques that challenge traditional signature defenses [7–9,32], while a validation gate (FPR and detection thresholds checked against labeled corpora before deployment) keeps automatically generated rules from degrading precision. The result is a system that gains coverage over time without drifting toward the opacity of ML-only WAFs [2,17–19,23].

9.2. Limitations and Future Work

9.2.1. LLM Dependency and Costs

The adaptive capability depends on LLM availability and incurs operational cost: cloud APIs add per-query fees and external dependencies, while local Ollama deployment removes the network requirement at the price of GPU resources. Future work could reduce this cost through selective invocation, response caching for near-duplicate requests, or periodic batch analysis instead of per-request inference.

9.2.2. Rule Quality and Generalization

While our validation framework filters low-quality rules, LLM-generated rules may still be overly specific or overly broad. Overly specific rules detect only exact attack variants and fail to generalize, requiring frequent rule generation for minor variations. Overly broad rules increase false positives despite validation, particularly as application behavior evolves. The optimal balance between specificity and generality remains an open research question. Future work could investigate meta-learning approaches where the system learns to adjust LLM prompts based on the success rate of previously generated rules, or human-in-the-loop active learning where analysts provide feedback that refines the rule generation process.

9.2.3. Adversarial Robustness

Adversaries aware of the LLM component could craft attacks specifically designed to evade LLM detection or trigger false positives through carefully constructed benign requests. LLMs are known to be vulnerable to prompt injection and adversarial examples. While our multi-layer defense (base rules remain primary, validation gates deployment) provides some protection, dedicated adversarial attacks against the LLM layer remain a concern. Future work should investigate the adversarial robustness of LLM-based security systems and develop defensive techniques such as ensemble LLM analysis, adversarial

training of the validation corpus, or hybrid approaches combining LLM insights with statistical anomaly detection.

9.2.4. Scalability and Latency

LLM inference latency, even with optimizations, introduces delays that may be unacceptable for latency-sensitive applications. While our architecture ensures that most traffic (requests matching base rules) experiences no additional latency, applications with high volumes of unusual but legitimate traffic could experience degraded performance. The system currently processes LLM queries serially; parallel or batched processing could improve throughput but increases implementation complexity. Additionally, as the deployed rule set grows through automatic generation, rule evaluation time increases. Future work should investigate efficient rule indexing, rule consolidation techniques to prevent rule set bloat, and adaptive LLM invocation strategies that balance detection coverage against latency constraints.

9.2.5. Limited Temporal and Cross-Request Analysis

The current system analyzes each request independently and does not maintain state across requests or sessions. This stateless design simplifies deployment and ensures linear scalability, but it limits detection of multi-step attacks, session-based attacks, or rate-limiting violations. An attacker could potentially evade detection by spreading an attack across multiple requests, each of which appears benign in isolation. Future work could extend the LLM analysis to consider request context, such as recent requests from the same source IP or session, though this would require careful design to avoid introducing state management complexity and maintain the system's scalability properties.

9.3. Operational Considerations

Deploying the LLM-assisted system in production requires careful operational planning. Organizations should begin with the system in logging-only mode to build confidence in LLM detection quality before enabling automated rule deployment. The validation corpus should be updated regularly to reflect current application behavior and avoid false positives as applications evolve. Analysts should review the generated rules queue periodically to identify patterns in LLM failures, which can inform prompt engineering improvements. API key management and rate limiting must be carefully configured to prevent cost overruns or service disruptions. Organizations must also consider the data privacy implications of sending request data to external LLM APIs and may prefer local deployment despite higher infrastructure costs.

10. Ethical Considerations

All experiments presented in this paper were conducted using publicly available benchmark datasets (CSIC 2010 [34] and HttpParamsDataset [35]) in isolated, containerized environments. The ModSecurity comparison was performed against a local Docker instance with no connection to production systems. No real user traffic, personal data, or production infrastructure was involved in any evaluation. The system is designed for authorized security testing and defensive deployment by security teams operating within their organizational scope.

Author Contributions: Conceptualization, methodology, software, validation, formal analysis, investigation, data curation, writing—original draft preparation: M.-C.C.; writing—review and editing, supervision: C.-G.C., I.B. and I.T. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by a grant of the Ministry of Research, Innovation and Digitization, CCCDI–UEFISCDI, project number PN-IV-P6-6.3-SOL-2024-2-0197, within PNCDI IV.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The evaluation datasets used in this study are publicly available: CSIC 2010 HTTP Dataset (https://www.kaggle.com/datasets/ispangler/csic-2010-web-application-attacks, accessed on 15 May 2026) and HttpParamsDataset (https://github.com/Morzeux/HttpParamsDataset, accessed on 15 May 2026).

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Floris, G.; Scano, C.; Montaruli, B.; Demetrio, L.; Valenza, A.; Compagna, L.; Ariu, D.; Piras, L.; Balzarotti, D.; Biggio, B. ModSec-AdvLearn: Countering Adversarial SQL Injections with Robust Machine Learning. *IEEE Trans. Inf. Forensics Secur.* **2025**, *20*, 6693–6705. [CrossRef]
2. Pan, Y.; Sun, F.; Teng, Z.; White, J.; Schmidt, D.C.; Staples, J.; Krause, L. Detecting Web Attacks with End-to-End Deep Learning. *J. Internet Serv. Appl.* **2019**, *10*, 16. [CrossRef]
3. Churcher, A.; Ullah, R.; Ahmad, J.; ur Rehman, S.; Masood, F.; Alqahtani, F.; Gogate, M.; Nour, B.; Buchanan, W.J. An Experimental Analysis of Attack Classification Using Machine Learning in IoT Networks. *Sensors* **2021**, *21*, 446. [CrossRef] [PubMed]
4. Liu, C.; Yang, J.; Wu, J. Web Intrusion Detection System Combined with Feature Analysis and SVM Optimization. *EURASIP J. Wirel. Commun. Netw.* **2020**, *2020*, 33. [CrossRef]
5. Díaz-Verdejo, J.E.; Estepa-Alonso, R.; Estepa-Alonso, A.; Madinabeitia, G. A Critical Review of the Techniques Used for Anomaly Detection of HTTP-Based Attacks: Taxonomy, Limitations and Open Challenges. *Comput. Secur.* **2023**, *124*, 102997. [CrossRef]
6. Erlacher, F.; Dressler, F. On High-Speed Flow-Based Intrusion Detection Using Snort-Compatible Signatures. *IEEE Trans. Dependable Secur. Comput.* **2022**, *19*, 495–506. [CrossRef]
7. Maestre Vidal, J.; Sotelo Monge, M.A.; Martínez Monterrubio, S.M. EsPADA: Enhanced Payload Analyzer for Malware Detection Robust Against Adversarial Threats. *Future Gener. Comput. Syst.* **2020**, *104*, 159–173. [CrossRef]
8. Pawlicki, M.; Choraś, M.; Kozik, R. Defending Network Intrusion Detection Systems Against Adversarial Evasion Attacks. *Future Gener. Comput. Syst.* **2020**, *110*, 148–154. [CrossRef]
9. Wang, K.; Stolfo, S.J. Anomalous Payload-Based Network Intrusion Detection. In *Proceedings of the RAID*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2004; Volume 3224, pp. 203–222.
10. Kruegel, C.; Vigna, G. Anomaly Detection of Web-Based Attacks. In Proceedings of the 10th ACM Conference on Computer and Communications Security, Washington, DC, USA, 27–30 October 2003; pp. 251–261.
11. Perdisci, R.; Ariu, D.; Fogla, P.; Giacinto, G.; Lee, W. McPAD: A Multiple Classifier System for Accurate Payload-Based Anomaly Detection. *Comput. Netw.* **2009**, *53*, 864–881. [CrossRef]
12. Wressnegger, C.; Schwenk, G.; Arp, D.; Rieck, K. A Close Look on N-Grams in Intrusion Detection: Anomaly Detection vs. Classification. In Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security, Berlin, Germany, 4–8 November 2013; pp. 67–76.
13. Düssel, P.; Gehl, C.; Flegel, U.; Dietrich, S.; Meier, M. Detecting Zero-Day Attacks Using Context-Aware Anomaly Detection at the Application-Layer. *Int. J. Inf. Secur.* **2017**, *16*, 475–490. [CrossRef]
14. Spathoulas, G.P.; Katsikas, S.K. Reducing False Positives in Intrusion Detection Systems. *Comput. Secur.* **2010**, *29*, 35–44. [CrossRef]
15. Hubballi, N.; Suryanarayanan, V. False Alarm Minimization Techniques in Signature-Based Intrusion Detection Systems: A Survey. *Comput. Commun.* **2014**, *49*, 1–17. [CrossRef]
16. Tang, P.; Qiu, W.; Huang, Z.; Lian, H.; Liu, G. Detection of SQL Injection Based on Artificial Neural Network. *Knowl.-Based Syst.* **2020**, *190*, 105528. [CrossRef]
17. Tadhani, J.R.; Vekariya, V.; Sorathiya, V.; Alshathri, S.; El-Shafai, W. Securing Web Applications Against XSS and SQLi Attacks Using a Novel Deep Learning Approach. *Sci. Rep.* **2024**, *14*, 1803. [CrossRef]
18. Kakisim, A.G. A Deep Learning Approach Based on Multi-View Consensus for SQL Injection Detection. *Int. J. Inf. Secur.* **2024**, *23*, 1541–1556. [CrossRef]

19. Li, Z.; Liu, F.; Gu, Z.; Liu, Y. XSS Attack Detection Method Based on CNN-BiLSTM-Attention. *Appl. Sci.* **2025**, *15*, 8924. [CrossRef]
20. Wang, X.; Zhai, J.; Yang, H. Detecting Command Injection Attacks in Web Applications Based on Novel Deep Learning Methods. *Sci. Rep.* **2024**, *14*, 25487. [CrossRef]
21. Bhagwani, H.; Negi, R.; Dutta, A.K.; Handa, A.; Kumar, N.; Shukla, S.K. Automated Classification of Web-Application Attacks for Intrusion Detection. In *Proceedings of the Lecture Notes in Computer Science*; Springer: Berlin/Heidelberg, Germany, 2020; pp. 1–10.
22. Salam, A.; Ullah, F.; Amin, F.; Abrar, M. Deep Learning Techniques for Web-Based Attack Detection in Industry 5.0: A Novel Approach. *Technologies* **2023**, *11*, 107. [CrossRef]
23. Kim, J.; Kim, J.; Kim, H.; Shim, M.; Choi, E. CNN-Based Network Intrusion Detection Against Denial-of-Service Attacks. *Electronics* **2020**, *9*, 916. [CrossRef]
24. Djaidja, T.E.T.; Brik, B.; Senouci, S.M.; Boualouache, A.; Ghamri-Doudane, Y. Early Network Intrusion Detection Enabled by Attention Mechanisms and RNNs. *IEEE Trans. Inf. Forensics Secur.* **2024**, *19*, 7783–7793. [CrossRef]
25. Han, X.; Cui, S.; Liu, S.; Zhang, C.; Jiang, B.; Lu, Z. Network Intrusion Detection Based on N-Gram Frequency and Time-Aware Transformer. *Comput. Secur.* **2023**, *128*, 103171. [CrossRef]
26. Xu, H.; Wang, S.; Li, N.; Wang, K.; Zhao, Y.; Chen, K.; Yu, T.; Liu, Y.; Wang, H. Large Language Models for Cyber Security: A Systematic Literature Review. *ACM Trans. Softw. Eng. Methodol.* **2025**. [CrossRef]
27. Yang, J.; Wu, Y.; Yuan, Y.; Xue, H.; Bourouis, S.; Abdel-Salam, M.; Prajapat, S.; Por, L.Y. LLM-AE-MP: Web Attack Detection Using a Large Language Model with Autoencoder and Multilayer Perceptron. *Expert Syst. Appl.* **2025**, *274*, 126982. [CrossRef]
28. Babaey, V.; Ravindran, A. GenSQLi: A Generative Artificial Intelligence Framework for Automatically Securing Web Application Firewalls Against Structured Query Language Injection Attacks. *Future Internet* **2025**, *17*, 8. [CrossRef]
29. Fu, Q.; Williams, D. Toward LLM-Driven Rule Generation for Enforcement Systems: An Exploratory Study on WAF. In Proceedings of the 1st Workshop on Operating Systems Design for AI Agents (AgenticOS '26), Co-Located with ASPLOS 2026, Pittsburgh, PA, USA, 23 March 2026.
30. Hines, K.; Lopez, G.; Hall, M.; Zarfati, F.; Zunger, Y.; Kıcıman, E. Defending Against Indirect Prompt Injection Attacks with Spotlighting. *arXiv* **2024**, arXiv:2403.14720. [CrossRef]
31. Sangeetha, S.; Haripriya, S.; Mohana Priya, S.G.; Vaidehi, V.; Srinivasan, N. Fuzzy Rule-Base Based Intrusion Detection System on Application Layer. In *Proceedings of the CNSA*; Communications in Computer and Information Science; Springer: Berlin/Heidelberg, Germany, 2010; Volume 89, pp. 11–20.
32. Wang, K.; Cretu, G.; Stolfo, S.J. Anomalous Payload-Based Worm Detection and Signature Generation. In *Proceedings of the RAID*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2005; Volume 3858, pp. 227–246.
33. Akhoundali, J.; Hamidi, H.; Rietveld, K.F.D.; Gadyatskaya, O. Eradicating the Unseen: Detecting, Exploiting, and Remediating a Path Traversal Vulnerability Across GitHub. In Proceedings of the 20th ACM Asia Conference on Computer and Communications Security, Hanoi, Vietnam, 25–29 August 2025.
34. Torrano-Gimenez, C.; Perez-Villegas, A.; Alvarez, G. HTTP Dataset CSIC 2010. Information Security Institute, CSIC, Spanish Research National Council. 2010. Available online: https://www.kaggle.com/datasets/ispangler/csic-2010-web-application-attacks (accessed on 15 May 2026).
35. Morzeux. HttpParamsDataset: A Dataset of HTTP Parameter Payloads for Web Attack Detection. GitHub. 2018. Available online: https://github.com/Morzeux/HttpParamsDataset (accessed on 15 May 2026).
36. Petruzzi, F.; Erbacci, G. SR-BH 2020: A Multi-Label Web Attack Dataset with CAPEC-Based Annotations. Harvard Dataverse. 2020. Available online: https://dataverse.harvard.edu/dataset.xhtml?persistentId=doi:10.7910/DVN/OGOIXX (accessed on 15 May 2026).
37. The MITRE Corporation. Common Attack Pattern Enumeration and Classification (CAPEC). Online. 2024. Available online: https://capec.mitre.org/ (accessed on 15 May 2026).
38. Kuang, Y.; Xu, H.; Zhang, Q. DeepWAF: Detecting Web Attacks Based on CNN and LSTM Models. In *Proceedings of the CSS*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2019; Volume 11983.
39. Tekerek, A. A Novel Architecture for Web-Based Attack Detection Using Convolutional Neural Network. *Comput. Secur.* **2021**, *100*, 102096. [CrossRef]
40. Vartouni, A.M.; Kashi, S.S.; Teshnehlab, M. An Anomaly Detection Method to Detect Web Attacks Using Stacked Auto-Encoder. In *Proceedings of the CFIS*; IEEE: New York, NY, USA, 2018.
41. Tekerek, A.; Bay, Ö.F. Design and Implementation of an Artificial Intelligence-Based Web Application Firewall Model. *Neural Netw. World* **2019**, *29*, 189–206. [CrossRef]
42. Gniewkowski, M.; Maciejewski, H.; Surmacz, T.; Walentynowicz, W. HTTP2vec: Embedding of HTTP Requests for Detection of Anomalous Traffic. *arXiv* **2021**, arXiv:2108.01763. [CrossRef]

43. Shaheed, A.; Kurdy, M.B. Web Application Firewall Using Machine Learning and Features Engineering. *Secur. Commun. Netw.* **2022**, *2022*, 5280158. [CrossRef]
44. Montes, N.; Betarte, G.; Martínez, R.; Pardo, A. Web Application Attacks Detection Using Deep Learning. In *Proceedings of the CIARP*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2021; Volume 12702.