WWW.IJRPR.COM

International Journal of Research Publication and Reviews

Journal homepage: www.ijrpr.com ISSN 2582-7421

Prompt Firewall: A Server-Side Enforcement Layer for Securing LLM-Based Chat Applications Against Prompt Injection

Sikkandhar A[1], Saran S[2], Sujith Kumar G[3], Sanjay Prasath M*[4],[Mr. Nishmitha KR][5]

Assistant Professor, Department of Information Technology, [Sri Shakthi Institute of Engineering And Technology].

A B S T R A C T :

Large language model (LLM) applications are exposed to prompt injection, in which crafted input tries to override the instructions or safeguards of the application. Checks that run only in the browser cannot stop such input, and many prototype tools can label text as risky without controlling what is actually sent to the AI provider. This paper presents Prompt Firewall, a security-enforcing chatbot application in which the server, not the client, decides whether a prompt may reach an AI provider. The system combines a React chat and administration interface with an Express backend that applies request validation, rate limiting, layered detection, and an ALLOW, REVIEW, or BLOCK decision. Blocked prompts are never forwarded, prompts under review wait for an authorised reviewer, and detector failures are designed to fail safely instead of defaulting to ALLOW. SQLite stores a minimal audit trail, a review queue, and operational statistics, with message-content retention off by default. The work is organised as a phased implementation plan covering repository hygiene, detector contracts, enforcement, authentication, privacy, frontend behaviour, and deployment readiness, together with a test and verification matrix. At the starting point, both React frontends built successfully, the standalone Prompt Wall Detector passed its four API test cases, and all 45 Prompt Firewall backend source and test files passed a Node.js syntax check; the backend test suite could not yet be run because a native SQLite dependency could not be built in the available Windows environment. Detection accuracy and enforcement tests are therefore reported as pending, and the system should be regarded as one layer of defence in depth rather than a guarantee against prompt injection.

Keywords: Prompt Injection, LLM Security, Prompt Firewall, Server-Side Enforcement, Rule-Based Detection, Human Review, Audit Logging, OWASP LLM Top 10.

Introduction

Large language models (LLMs) are now embedded in chat assistants, customer-support tools, and internal productivity applications. These systems accept free-form text and pass it to a model that cannot reliably distinguish the instructions of the developer from the words of the user. This weakness is exploited by prompt injection, in which crafted input tries to override instructions, reveal hidden prompts, or make the application behave in ways its designers did not intend [2], [3]. Prompt injection is listed first among the risks in the OWASP Top 10 for LLM applications [1], and systematic studies have attempted to formalize and benchmark both the attacks and the defences against them [4].

Detecting such input is difficult because the same sentence may be harmless in one context and hostile in another, and because the safety behaviour learned by a model can be bypassed by rephrasing or by mismatched framing [5]. A practical application therefore needs more than a classifier: it needs a place in the architecture where a decision is actually enforced, a way to handle uncertain cases, and records that allow decisions to be reviewed without exposing private conversations.

1.1. Existing challenges in prompt-injection defence

Existing approaches include browser-side input checks, phrase-matching filters, provider-side safety settings, standalone detection services, and manual moderation. Each provides useful signals, but each has limitations when used alone. A check that runs in the browser can be bypassed by calling the API directly, so it can improve user feedback but cannot be a security boundary. A detector that only returns a score leaves the calling application free to forward the prompt regardless of that score, so the protection depends on every caller handling the result correctly.

Detection quality is a second challenge. Phrase matching and small test sets can look accurate on the examples they were written for while missing obfuscated, multilingual, or indirectly phrased attacks, and the same rules may flag harmless text. Both kinds of error have a cost: false negatives admit risky input, while false positives block legitimate users. A third challenge is operational. Audit logs that store raw prompts may expose personal or confidential data, administrator and review endpoints must be protected, secrets must stay out of source code and client code, and an application that fails open when a detector is unavailable defeats its own purpose.

1.2. Role of server-side enforcement and layered detection

Two long-standing design principles apply directly to this problem. Complete mediation requires that every access be checked against the authority that decides it, and fail-safe defaults require that the absence of an explicit permission results in denial [6]. In an LLM application, the access being mediated is the forwarding of a prompt to an AI provider. Applying these principles means that the server must evaluate every prompt before any provider call, and that an error in the detector must not be turned into permission.

Layered detection supports this design. Deterministic rules can identify clear patterns at low cost, second-stage checks can examine cases the first layer cannot settle, and an optional external detector can be attached through an adapter. A three-valued outcome (ALLOW, REVIEW, BLOCK) allows uncertain cases to be routed to an authorised human reviewer rather than being forced into an automatic yes or no. Detection scores are treated as risk signals and not as a definitive measure of user intent, and results should be presented together with the reasons that produced them.

1.3. Problem statement and objectives

Many prototype tools demonstrate that a prompt can be labelled as risky but do not show that the label is enforced, how uncertain cases are handled, or how the system behaves when a component fails. There is therefore a need for an application in which the enforcement decision is made on the server, can be tested, and is accompanied by access control, privacy safeguards, and operational procedures.

The main objective of this work is to describe and plan Prompt Firewall, a security-enforcing chatbot application, together with the verification activities needed to show that its enforcement behaves as intended. The specific objectives are:

i. To make the Prompt Firewall backend the authoritative enforcement boundary, so that a prompt reaches an AI provider only after a server-side ALLOW decision.
ii. To ensure that blocked prompts are never forwarded and that prompts under review are not forwarded until an authorised reviewer approves them.
iii. To define a stable, documented detector result contract containing the decision, score, matched categories or reasons, and a request identifier.
iv. To make detection measurable against a documented, versioned evaluation corpus, reporting false positives, false negatives, precision, recall, and decision latency.
v. To protect administrator and review operations through authentication, authorization, validation, rate limiting, and auditability.
vi. To preserve privacy by avoiding unnecessary prompt storage and by redacting sensitive data from logs.
vii. To provide a mock-provider development path, health checks, and safe configuration defaults.
viii. To keep the companion applications clearly bounded and honest about their scope, and to evaluate the system through unit, integration, and usability testing.

Materials and Methods

1.4. Development approach

The project follows a phased implementation plan for a repository that currently contains four separate applications. Prompt Firewall is the authoritative security-enforcement application. Sentinel remains an educational, frontend-only demonstration, Prompt Wall Detector remains a standalone detector API and evaluation prototype, and JARVIS remains a separate personal AI assistant. The applications stay independently runnable unless a later product decision explicitly combines them.

The plan is divided into nine phases. Phase 0 establishes repository and credential hygiene. Phase 1 defines project boundaries and documentation. Phase 2 covers installation, configuration, and local development. Phase 3 defines detector contracts and the evaluation corpus. Phase 4 verifies enforcement and provider reliability. Phase 5 hardens authentication, authorization, privacy, and operations. Phase 6 aligns the frontend behaviour with the backend. Phase 7 checks the correctness and separation of JARVIS. Phase 8 adds continuous integration, release, and deployment readiness. Each phase has a stated purpose, a checklist of tasks, and acceptance criteria that must be met before the phase is considered complete.

Delivery follows a priority order. Baseline safety and the truthfulness of documentation come first (P0), followed by reproducible setup, proof of core enforcement, and authorization and privacy (P1). Detection quality, the user interface, and operations are addressed next (P2), and automated checks with staged deployment complete the work (P3). Implementation proceeds one phase at a time, with small changes and tests placed at the enforcement boundary.

1.5. System architecture

The repository is organised so that each application keeps its own folder, configuration, and run commands. Table 1 summarises the four applications and their present implementation. The repository root is not treated as a hidden fifth application that starts every service.

Table 1 - Applications in the repository and their present implementation.

Application	Role	Present implementation
Prompt Firewall (prompt-firewall/)	Security-enforcing chatbot	React frontend and Express backend; layered detection; ALLOW/BLOCK/REVIEW enforcement; AI provider adapters; SQLite; audit and review routes; admin UI
Sentinel (eee-project/)	Educational demo	React/Vite UI with overview, scanner, threat patterns, history, and about pages; local sample phrase checks; no backend connection
Prompt Wall Detector (prompt-wall-detector/)	Standalone detector prototype	Express API on port 3001; rule-based detection; test and evaluation scripts
JARVIS (repository root)	Personal AI assistant	Express server, static browser UI, SQLite-related code, chat and assistant features

Within Prompt Firewall, the system is organised into a user interface layer, an API layer, a detection layer, a provider layer, and a persistence layer (Fig. 1). The browser is treated as an untrusted client. Each request passes through the Express API, where a request identifier is assigned, rate limits and validation are applied, and authentication is checked where the route requires it. The detection coordinator then evaluates the prompt using the rule engine, second-stage checks, and optionally a detector adapter. The resulting decision determines what happens next.

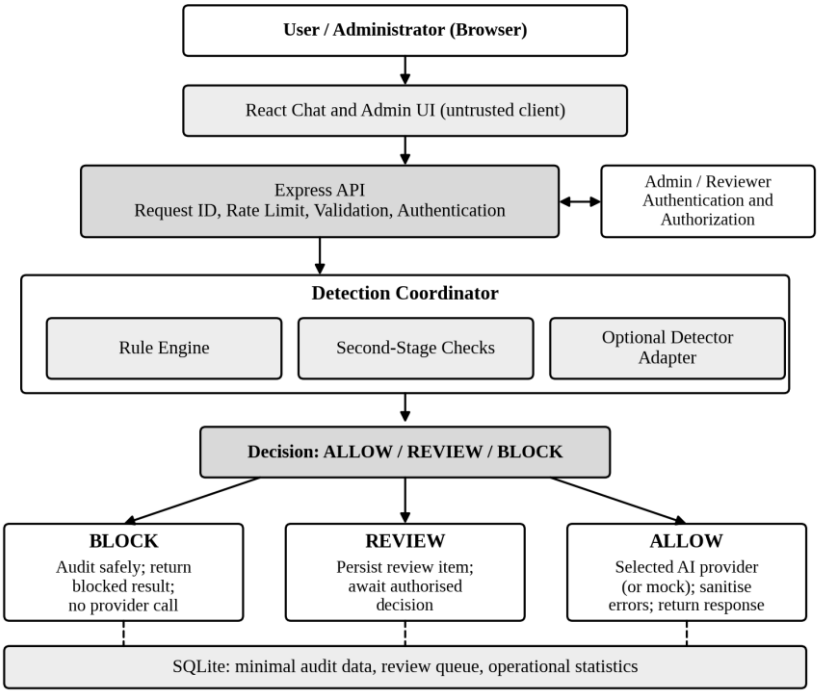


Fig. 1 - System architecture of Prompt Firewall.

The architecture is governed by a small set of rules. The backend owns all enforcement decisions, and a client-side check is never an enforcement boundary. A BLOCK result must not invoke an AI provider, and a REVIEW result must not invoke one until an authorised reviewer approves it. The detector returns a stable result containing the decision, score, matched categories or reasons, and a request identifier. Provider keys and administrative secrets remain on the server. Detector or model errors fail safely and are never silently converted into ALLOW. If Prompt Wall Detector is used by Prompt Firewall in future, it is reached only through an adapter with an explicit timeout, error policy, authentication, and observability.

1.6. Technology stack and development environment

Prompt Firewall uses React for the chat and administration interface and Express on Node.js for the backend API [8], [9], [12]. SQLite, accessed through the better-sqlite3 library, stores the audit trail, review queue, and operational statistics [10], [11]. Because better-sqlite3 is a native module, installation on Windows may require C++ build tools that match the chosen Node.js version, and this requirement is documented as part of the setup. Sentinel is built with React and Vite, and both React frontends are verified with their production build commands.

Provider access is implemented through adapters. The provider name and model are selected from an allowlist, and the client cannot supply an arbitrary provider endpoint. A mock provider supports a complete local chat flow without external credentials, so that development and automated tests do not depend on a hosted AI service. Live-provider tests are reserved for opt-in runs that use developer-supplied credentials. Security middleware is to be verified in each of the following areas: security headers (for example through Helmet [13]), explicit CORS origin allowlists, request-size limits, JSON validation, rate limiting, and error-handling middleware. Configuration is validated at startup, including the provider name, ports, CORS origins, database

path, thresholds, timeouts, and production authentication secrets. Dependencies are installed from lockfiles with npm ci, and the supported Node.js LTS version is to be selected and documented after checking native dependency compatibility.

1.7. Data collection and processing

Prompt Firewall processes the information needed to enforce its decisions: submitted chat prompts, detector results (decision, score, matched categories, request identifier), audit events, review items, operational statistics, and administrator credentials or tokens. Prompts are treated purely as text data to be inspected; they are never executed.

Personal information should be minimised. Audit events store only the data required for operations and evaluation, and message-content retention is off by default. If retention is enabled for an approved use case, the purpose, access policy, and retention or deletion period must be documented. Credentials, personal information, and provider error content are sanitised or redacted before they appear in logs or responses.

Input is validated before detection. The constraints cover the JSON object shape, the prompt type, the maximum length, encoding behaviour, and empty input. Evaluation uses a versioned corpus containing benign, suspicious, adversarial, obfuscated, multilingual, and boundary-length examples. Every evaluation report must identify the corpus version and the rules or model version used, and a result on a small corpus is not reported as a general security guarantee. Practical guidance on authentication, rate limiting, and input handling is drawn from OWASP guidance [14].

1.8. Database design

A suitable data model contains records for audit events, review items, and operational statistics. Audit records associate the request identifier, time, decision, score, and matched categories with an event and, by default, do not contain the raw prompt. Review records hold the items that are waiting for an authorised decision together with the outcome of that decision. Statistics summarise the number of allowed, blocked, and reviewed requests and related operational values.

Audit queries are written with parameterised SQL, pagination, bounded result sizes, and indexes that support the common lookups. The plan includes verification of schema initialisation, migrations, concurrent writes, write-ahead logging (WAL) behaviour, and database file permissions. SQLite databases, backups, and logs are excluded from version control. Backup and restore are exercised with a disposable database so that production data is never overwritten, and the procedure is documented for operators.

1.9. Detection and enforcement

Detection is performed in layers. The rule engine must give deterministic results, use safe regular expressions, and perform bounded work. Second-stage checks examine cases that the first layer cannot settle. An optional detector adapter is subject to a timeout, validation of the returned structure, authentication, controlled retries, and a fail-closed or fail-to-review policy when the detector is unavailable or returns malformed output. Detection and enforcement remain separate: the detector reports a decision, and the enforcement code applies it.

The detector result contract specifies the decision (ALLOW, REVIEW, or BLOCK), a score within a documented range, the thresholds that map scores to decisions, and the matched categories or reasons. The meaning of each decision for the rest of the system is summarised in Table 2.

Table 2 - Enforcement decisions and their effect on provider dispatch.

Decision	Condition	Provider call	Persistence
ALLOW	Result is within the range configured for normal use	Forwarded to the selected provider; provider errors mapped to explicit, non-sensitive errors	Operational statistics and minimal audit data
REVIEW	Result is within the range configured for human review	Not called until an authorised reviewer approves	Review item stored; awaits authorised decision
BLOCK	Result meets the blocking condition	Never called	Audited safely; blocked result returned
Detector error	Detector unavailable or output malformed	Never silently treated as ALLOW	Fail-closed or fail-to-review as configured

Rule-based detection has inherent limits. Attacks that are obfuscated, written in other languages, split across several messages, or embedded in external content may not match any rule, and benign text may occasionally match. For this reason the detector is treated as one layer of defence in depth. Thresholds are tuned using measured results, false positives and false negatives are tracked, and every fixed detector defect receives a regression test. Demonstration patterns used in Sentinel are kept visibly separate from the production detector rules.

1.10. Request workflow

A chat request begins when the user submits a prompt through the chat interface. The API assigns a request identifier, applies rate limiting, validates the input, and checks authentication where required. The detection coordinator returns a decision, and the server acts on it. A BLOCK decision is audited and a blocked result is returned to the user without any call to an AI provider. A REVIEW decision is stored in the review queue, and the user is informed that the prompt is pending. An ALLOW decision is sent to the selected provider, and the response is sanitised before it is returned (Fig. 2).

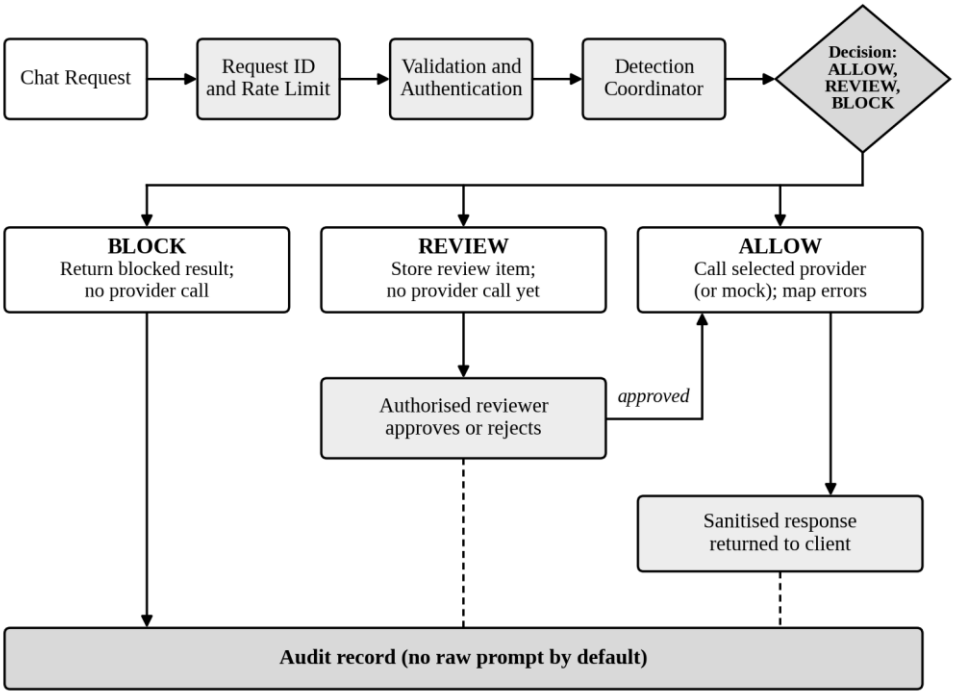


Fig. 2 - Request workflow of Prompt Firewall.

An approved review follows a documented authorisation and dispatch flow in which only an authorised reviewer can approve or reject the item. Provider failures, timeouts, malformed responses, and rate limits produce explicit and non-sensitive errors, and never a success-shaped fallback response. Request identifiers appear in responses, safe logs, and audit records so that a single request can be traced across them. A health endpoint reports process and dependency health without exposing secrets or internal configuration.

1.11. Testing and evaluation

Testing is organised by application. Unit tests cover the rule engine, validation, and utility functions. API tests start the server explicitly and clean it up afterwards. Enforcement tests use a mock provider to count provider calls and must show a call count of zero for BLOCK and for a pending REVIEW. Authorization tests cover every privileged endpoint with unauthenticated and insufficiently authorised callers. Privacy tests check that the default audit trail does not persist raw prompt content. Backup and restore tests use a disposable database.

Frontend verification includes a clean install, linting, a production build, and checks of login, logout, expired credentials, review queue loading, approve and reject actions, pagination, empty states, and API failure states, together with keyboard navigation, labels, contrast, and mobile behaviour. At repository level the checks include a secret scan, git diff --check, a clean-checkout setup, and consistency between documentation and scripts. Continuous integration installs from lockfiles, uses test-only databases and mock providers, and never uses production credentials or data.

Any quantitative claim about accuracy, latency, or reliability is reported only when supported by actual measurements. Corpus evaluation reports false positives, false negatives, precision, recall, and decision latency, and, where the corpus allows, does so per category rather than only as a single overall figure.

Results and Discussion

1.12. Proposed system functionality

Prompt Firewall brings prompt screening, enforcement, human review, provider access, and auditing into one application. Its planned functions are a chat interface that shows ALLOW, BLOCK, and REVIEW outcomes accurately; server-side detection and enforcement; a review queue with approve and reject actions for authorised reviewers; an administrator interface with audit access; a mock provider for local use; and health checks for operations. The three companion applications remain separate: Sentinel as a local-only educational demonstration, Prompt Wall Detector as a standalone detector prototype, and JARVIS as an independent assistant. The actual behaviour of each function must be established through the tests in Section 2.8 and is reported below only where it has been observed.

1.13. Verified starting point and functional evaluation

Table 3 lists the checks that had been completed at the starting point of the plan. These results describe the existing repository and do not yet demonstrate the enforcement behaviour described in this paper.

Table 3 - Verified starting point of the repository.

Check	Observed result	Status
Production build of both React frontends	Built successfully	Verified
Prompt Wall Detector API test (server running)	All four test cases passed	Verified
Syntax check (node --check) of 45 Prompt Firewall backend source and test files	All files passed	Verified (syntax only)
Prompt Firewall backend tests	Could not be run: better-sqlite3 requires native build support that is not available in the current Windows environment	Blocked
Enforcement, authorization, privacy, and corpus evaluation tests	Not yet executed	Pending

A syntax check shows that files can be parsed, but it does not show that the code behaves correctly, so the blocked backend test run remains the most important open item. Resolving the native build requirement by documenting a supported Node.js version and the Windows build tools is therefore scheduled early in the delivery order. The review also identified documentation that does not match the implementation: the JARVIS README describes Gemini usage, while the inspected server contains Groq and NVIDIA provider configuration, and the root server is to be examined for embedded credential-like values that must be removed and, if genuine, revoked.

Representative functional test scenarios are summarised in Table 4. They define the expected behaviour for each module. Results should be recorded from observed behaviour, and scenarios that have not been run are identified as pending.

Table 4 - Representative functional test scenarios for Prompt Firewall.

ID	Module	Test scenario	Expected behaviour	Status
T1	Authorization	Call review and audit endpoints without credentials and with insufficient privileges	Request rejected; no review item or log is returned	Pending
T2	Validation	Submit empty, non-string, and over-length prompts	Clear validation error; detector and provider not called	Pending
T3	Enforcement	Submit a prompt that the detector marks BLOCK	Blocked result returned; provider call count is zero	Pending
T4	Enforcement	Submit a prompt marked REVIEW	Item queued; provider call count is zero until approval	Pending
T5	Enforcement	Submit a benign prompt using the mock provider	Prompt allowed; mock provider called once; response returned	Pending
T6	Detector failure	Make the detector adapter time out or return malformed output	Request is not silently allowed; fail-closed or fail-to-review behaviour applies	Pending
T7	Privacy	Inspect the default audit record after a request	No raw prompt content stored; sensitive values redacted	Pending
T8	Configuration	Start in production mode with wildcard CORS or a default administrator credential	Startup refused with an explicit error	Pending

Any failed tests should be investigated, corrected with a regression test, and repeated. Evaluation of the detector should use the versioned corpus and report the corpus and rule versions used.

1.14. Comparison with conventional approaches

Conventional approaches include browser-side filtering, phrase-matching demonstrations, detector services that return a score, and provider-side safety settings. Prompt Firewall is intended to differ in where the decision is applied: the server evaluates every prompt before dispatch, routes uncertain cases to a reviewer, and fails safely when a component is unavailable. Its design also keeps audit data minimal and protects administrative functions with server-side authorization.

These advantages are architectural. They do not mean that Prompt Firewall detects more attacks than other tools, and no such claim should be made without comparative testing on the same corpus. Provider-side safety settings and Prompt Firewall are complementary, since the former act inside the model service while the latter acts before the request leaves the application.

1.15. Discussion

The plan provides a framework for turning a collection of prototypes into a testable security application. Separating detection from enforcement allows the detector to be improved or replaced without changing the rule that blocked and pending prompts never reach a provider, and keeping the applications independent prevents the demonstration code in Sentinel from being mistaken for production protection.

The system also faces practical challenges. Rule-based detection can be evaded, and a conservative threshold increases the false positives that frustrate legitimate users. The REVIEW path adds human effort and delay, so the review queue must be monitored and its volume managed. Strict privacy defaults limit the data available for later analysis, which must be balanced against the need to evaluate and tune the detector. SQLite is convenient for a single-node deployment, and larger deployments would need to reconsider storage, backup, and concurrency. Finally, a corpus created by the development team may not represent the attacks seen in practice.

A risk-based approach, as promoted by the NIST AI Risk Management Framework [7], suggests that such controls be measured and revisited as the application and its threats change. Further evaluation should therefore measure precision and recall, decision latency, review-queue behaviour, and usability, and report them together with the corpus version. Without these measurements, the effectiveness of the system cannot be claimed.

Conclusion

Prompt Firewall presents a server-side approach to securing LLM chat applications in which every prompt is evaluated before it can reach an AI provider. The design combines layered detection, an ALLOW, REVIEW, and BLOCK decision model, safe failure behaviour, authorised review, a minimal audit trail, and a mock-provider path for development and testing.

The modular plan provides a path for completing and verifying these functions. However, its practical effectiveness depends on the successful execution of the enforcement, authorization, and privacy tests, the quality and versioning of the evaluation corpus, and the thresholds chosen for each decision. At the starting point the frontends build and the standalone detector API test passes, but the Prompt Firewall backend tests are blocked by a native dependency and have not yet been run.

Future development can focus on resolving the native build requirement and completing the verification matrix, expanding the corpus with obfuscated and multilingual examples, adding continuous integration and staged deployment, evaluating an external detector only after its security, privacy, cost, and operational requirements are approved, and integrating Prompt Wall Detector through an explicit adapter if required. Further testing and operational review are required before the system is deployed.

In conclusion, Prompt Firewall offers a foundation for a prompt-security application whose protective claims can be tested rather than assumed. With appropriate implementation, validation, and privacy safeguards, it can contribute to safer use of LLM applications as one layer within a broader defence.

Acknowledgments

The authors would like to thank [College Name] for providing the academic environment and facilities required for this project. The authors also express their sincere gratitude to the Head of the Department of Information Technology, the project guide, and the faculty members for their guidance, encouragement, and support throughout the development of the project.

REFERENCES

[1] OWASP Foundation. OWASP Top 10 for Large Language Model Applications. Available at: https://owasp.org/www-project-top-10-for-large-language-model-applications/

[2] F. Perez and I. Ribeiro, “Ignore Previous Prompt: Attack Techniques for Language Models,” in *NeurIPS ML Safety Workshop*, 2022.

[3] K. Greshake, S. Abdelnabi, S. Mishra, C. Endres, T. Holz, and M. Fritz, “Not What You’ve Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection,” in *Proc. 16th ACM Workshop on Artificial Intelligence and Security (AISec)*, 2023.

[4] Y. Liu, Y. Jia, R. Geng, J. Jia, and N. Z. Gong, “Formalizing and Benchmarking Prompt Injection Attacks and Defenses,” in *Proc. 33rd USENIX Security Symposium*, 2024.

[5] A. Wei, N. Haghtalab, and J. Steinhardt, “Jailbroken: How Does LLM Safety Training Fail?” in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[6] J. H. Saltzer and M. D. Schroeder, “The Protection of Information in Computer Systems,” *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.

[7] National Institute of Standards and Technology, *Artificial Intelligence Risk Management Framework (AI RMF 1.0)*, NIST AI 100-1, 2023.

[8] React. React Documentation. Available at: https://react.dev/

[9] Express. Express Documentation. Available at: https://expressjs.com/

[10] SQLite. SQLite Documentation. Available at: https://www.sqlite.org/docs.html

[11] better-sqlite3. better-sqlite3 Repository. Available at: https://github.com/WiseLibs/better-sqlite3

[12] OpenJS Foundation. Node.js Documentation. Available at: https://nodejs.org/docs/latest/api/

[13] Helmet. Helmet Documentation. Available at: https://helmetjs.github.io/

[14] OWASP Foundation. OWASP Cheat Sheet Series. Available at: https://cheatsheetseries.owasp.org/