International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 10 Issue: 10 | Oct - 2026 SJIF Rating: 8.659 ISSN: 2582-3930

A Context-Aware Framework for Detecting Prompt Injection in Retrieval-Augmented AI Systems

Prof.S. AbikayilAarthi
Assistant Professor,
Department of Computer Science and Engineering,
Kings College of Engineering,
Punalkulam, TN, India,
aarthi.cse@kingsengg.edu.in

Narmadha G
Department of Computer Science and Engineering,
Kings College of Engineering,
Punalkulam, TN, India,
narmadhagovindharaj07@gmail.com

Lakshmi Priya K
Department of Computer Science and Engineering,
Kings College of Engineering,
Punalkulam, TN, India,
laxupree@gmail.com

Lavanya K
Department of Computer Science and Engineering,
Kings College of Engineering,
Punalkulam, TN, India,
dharan2002k@gmail.com

AtchayaSri D
Department of Computer Science and Engineering,
Kings College of Engineering,
Punalkulam, TN, India,
atchayasri7596@gmail.com

Merudula S
Department of Computer Science and Engineering,
Kings College of Engineering,
Punalkulam, TN, India,
merudula196@gmail.com

***ABSTRACT*—Retrieval-augmented generation (RAG) connects large language models (LLMs) to external documents, websites, repositories, and organizational knowledge bases. Although retrieval improves factual grounding, it also exposes the model to untrusted text that may contain instructions designed to override the application's intended behaviour. This threat, commonly described as prompt injection, may appear directly in a webpage or indirectly inside a file, email, code comment, or hidden markup. This paper proposes PromptShield, a context-aware screening framework placed between retrieval and generation. It combines provenance tagging, structural parsing, instruction-intent classification, risk scoring, and policy-aware sanitization. Instead of blocking every imperative sentence, it examines whether a passage targets the assistant, conflicts with the user's task, requests a protected action, or attempts to change the instruction hierarchy. Low-risk evidence is retained, suspicious spans may be sanitized, and high-risk passages are quarantined. The paper defines the threat model, system architecture, dataset plan, candidate detection methods, evaluation metrics, and deployment safeguards. As no implementation experiment is claimed, the results section specifies a reproducible evaluation plan rather than reporting invented measurements. PromptShield aims to provide a practical foundation for secure RAG applications while balancing attack detection, evidence retention, latency, and usability.**

***Keywords*—Prompt Injection, Retrieval-Augmented Generation, Large Language Models, AI Security, Trust Boundaries, Adversarial NLP, Content Sanitization.**

I. INTRODUCTION

A. Background

Large language models [4] are increasingly used in chatbots, coding assistants, document search, customer-support systems, and personal productivity tools. A standalone model responds using learned parameters and conversation context. A RAG system adds a retrieval step: it searches a document collection, selects relevant passages, and inserts those passages into the model context before generation. This approach allows the system to use current or private information without retraining the model whenever documents change.

However, the retrieved text becomes part of the same input stream that the model uses to interpret instructions. A malicious passage may tell the model to ignore previous rules, reveal hidden prompts, disclose confidential information, or call an external tool. If the model treats the passage as authority rather than evidence, the application may produce unsafe or unintended behaviour.

B. Problem Statement

Many applications validate the user's prompt but do not consistently inspect retrieved content. This creates a gap because a user can ask a harmless question while a webpage or file returned by the retriever contains an adversarial instruction. Simple keyword filters are inexpensive but brittle: benign security documentation may discuss words such as "ignore," "secret," or "system," while an attack can use paraphrases or hide instructions in markup. The challenge is to identify instructions that seek to control the model without removing useful factual evidence or adding excessive latency. The problem is therefore both a classification task and a system-design task.

C. Motivation

The key boundary in a RAG application is the boundary between trusted application instructions and untrusted retrieved evidence. A passage can be highly relevant to a question and still be unsafe to obey as an instruction. For example, a technical page may correctly explain how to configure an API token while also containing a malicious paragraph asking the assistant to print credentials. Relevance ranking alone cannot distinguish these roles. A secure system must evaluate the content's meaning, its source, the action it requests, and the consequences of following it. This is especially important when the LLM can use tools, edit files, query databases, or send messages [5].

D. Objectives and Contributions

This paper has six objectives: define a threat model for indirect prompt injection; design a multi-stage detector for retrieved passages; combine semantic intent with provenance and structural signals; specify a dataset and annotation process; define measurable evaluation criteria; and describe deployment controls that remain effective if the detector fails. The main contribution is a modular PromptShield architecture with four policy outcomes—allow, sanitize, quarantine, and reject source. The paper also proposes a reproducible comparison of rules, classical machine learning, transformer classification, and an optional second-stage verifier. All proposed metrics and expected trade-offs are clearly distinguished from experimental results.

II. LITERATURE REVIEW

A. Retrieval-Augmented Generation

Retrieval-augmented generation [1] combines parametric language-model knowledge with non-parametric external memory. The retriever may use keyword search, dense vector embeddings, hybrid retrieval, or reranking to identify relevant passages. The generator then uses the selected evidence to formulate an answer. RAG can improve factual coverage and domain adaptation, but it introduces dependencies on source quality, indexing, chunking, and access control. A malicious instruction may be isolated from its surrounding explanation during chunking, and an external source may change after it has been indexed. These properties mean that RAG security must include retrieval-time controls and source provenance rather than relying only on the model's final response.

B. Direct and Indirect Prompt Injection

Direct prompt injection arrives through a user's own message, whereas indirect prompt injection [2] is embedded in content that the application retrieves or processes. Examples include webpages, emails, documents, issue descriptions, code comments, calendar invitations, and shared files. The user may not know that the content contains adversarial text. Attackers may ask the model to reveal hidden instructions, treat a webpage as a higher authority, disclose private data, or invoke a tool. Some attacks are explicit, while others are framed as administrative notices, debugging steps, or helpful instructions. The same words may also appear harmlessly in an educational article, making context essential.

C. Existing Mitigation Families

Common mitigation families [3] include instruction hierarchy [6], delimiters around untrusted text, source allowlists, input and output filters, prompt hardening, tool permission checks, and model-based classifiers. Delimiters clarify that retrieved passages are data, but their effectiveness depends on the model respecting the boundary. Keyword filters are fast and transparent but struggle with paraphrases. Learned classifiers can identify semantic intent, although they may be biased by their training data. Tool-level authorization is critical because a text detector should never be the only mechanism preventing high-impact actions. Defence in depth is therefore more realistic than relying on one universal filter.

D. Research Gap

A practical RAG system needs a control point that screens retrieved content before generation, preserves useful evidence when possible, and explains its decisions to developers. It must consider source provenance, mixed-content passages, and latency. A binary "safe or unsafe" label may be insufficient because a passage can contain both legitimate information and one suspicious instruction. PromptShield addresses this gap by combining multiple signals and applying different actions according to risk. The framework also treats independent tool authorization and privacy-conscious monitoring as required parts of the overall design.

III. THREAT MODEL AND REQUIREMENTS

A. Assets to Protect

The assets include system and developer instructions, confidential documents, user information, tool credentials, database permissions, and the integrity of generated answers. Tool-enabled applications may also expose capabilities to send emails, modify records, execute code, or retrieve private files. The system should preserve the user's legitimate task while preventing untrusted text from changing policy or accessing protected resources. Integrity is important as well as confidentiality: a malicious passage may cause the assistant to provide false claims, suppress relevant facts, or perform an unauthorized operation.

B. Adversary Model

The adversary may publish a webpage, edit a shared file, contribute a repository document, or submit a crafted email that is later indexed. The adversary may know that the application uses RAG but is assumed not to control the model weights, system prompt, or authentication infrastructure. Attack methods may include natural-language instructions, role impersonation, Unicode confusables, hidden HTML, code comments, repeated text, or instructions split across chunks. The threat model does not assume that every external source is malicious; it assumes that source trust is uncertain and that content must be screened before it is treated as evidence.

C. Security Requirements

PromptShield should detect attempts to override policy, request secrets, redirect the user's task, or trigger unauthorized tool use. It should distinguish quoted attack examples from operative instructions, use provenance without treating it as a guarantee, minimize sensitive logging, and preserve benign evidence where possible. It should produce reproducible decisions and have a defined fallback mode if the detector is unavailable. The system must also ensure that detection does not grant permissions: tool calls must be checked independently against identity, authorization, argument schema, and confirmation requirements.

D. Attack Categories

The threat taxonomy includes policy override, secret extraction, tool misuse, role impersonation, retrieval poisoning, hidden-text injection, cross-chunk instruction assembly, and task redirection. Each example should be labelled by attack category, target asset, severity, and expected mitigation. A

security article that explains prompt injection should not be labelled the same as a webpage attempting to execute one. Mixed-content passages should indicate whether safe factual evidence can be retained after suspicious spans are removed.

IV. PROPOSED PROMPTSHIELD ARCHITECTURE

A. Architecture Overview

PromptShield is positioned between the retriever and the generator. The retriever returns candidate passages with source identifiers, timestamps, and chunk metadata. A normalizer extracts text and records structural anomalies. A fast rule stage checks known suspicious patterns, and a semantic classifier estimates whether a passage contains an instruction intended to control the assistant. The risk engine combines the classifier output with provenance, action impact, and obfuscation signals. A policy gate then allows, sanitizes, quarantines, or rejects the passage. Approved content is passed to the LLM with explicit markers stating that retrieved text is evidence, not governing instruction.

B. Processing Pipeline

The processing pipeline contains eight stages: retrieve candidate passages; attach source metadata; normalize text while preserving anomaly flags; identify instruction-like spans; run rule-based and semantic detectors; compute risk; apply the policy decision; and record an audit event. Low-risk content passes with source labels. Moderate-risk passages may have a narrow instruction span removed or quoted as untrusted text. High-risk content is excluded from the model context and may be sent for authorized review. The output validator checks the generated response, while a separate authorization layer checks any proposed tool call.

C. Provenance and Trust Labels

Each chunk is represented by a record containing source type, source owner where known, retrieval timestamp, content hash, trust tier, detector scores, and policy action. Possible tiers include curated internal source, authenticated user document, public webpage, unknown external content, and untrusted attachment. These labels influence risk but do not replace content analysis: an internal source can be compromised and a public source can be accurate. Provenance should be supplied by authenticated retrieval infrastructure rather than accepted from the document itself, since an attacker could claim to be an administrator or trusted policy source.

D. Risk Scoring

A proposed normalized risk score is

$$R = w_1 I + w_2 P + w_3 A + w_4 S + w_5 O \quad (1)$$

where I is instruction-intent probability, P is policy-override likelihood, A is action impact, S is source-risk level, and O is evidence of obfuscation or suspicious structure. Weights and thresholds must be selected using validation data; they are not claimed as experimentally optimized in this paper. A minimum high-impact rule may quarantine a passage that requests credentials or unauthorized tool use even if its overall score is close to a threshold. Risk scoring supports consistent decisions but cannot prove that a passage is safe.

E. Policy Outcomes

Allow: the passage provides relevant evidence without attempting to control the assistant. Sanitize: a narrow suspicious span is removed or quoted as untrusted data. Quarantine: the passage is excluded from the context. Reject source: repeated malicious patterns or invalid provenance may suppress the source. Each outcome records a reason code, confidence band, and content hash. User-facing explanations should be concise and should not reveal exact internal thresholds that could help an attacker tune future payloads.

F. Independent Tool Guarding

Pre-generation screening is only one layer. The generated response should be checked for protected instruction disclosure, accidental reproduction of secrets, and unsupported claims that an action has completed. Tool requests must be validated independently for the user's authorization, permitted operation, argument schema, and confirmation requirements. Sending messages, deleting files, changing account settings, or executing code should not happen merely because the LLM was influenced by retrieved text. For high-impact actions, explicit user confirmation should remain mandatory.

V. DATASET DESIGN AND PREPROCESSING

A. Dataset Sources

A dataset should combine public prompt-injection examples, benign technical documentation, educational discussions, synthetic adversarial variants, and application-specific retrieved passages collected with permission. Domains may include web search, code repositories, customer support, email, and enterprise documents. Synthetic data can improve coverage, but should not dominate the test set because generated attacks may be easier to recognize than real examples. Data collection must respect licensing, privacy, and institutional requirements.

B. Label Schema

Each passage should receive labels for malicious instruction presence, attack category, target asset, severity, and recommended action. A separate label should indicate whether useful benign evidence is present and should be preserved. Annotators must distinguish quoted attacks from operative instructions and mark ambiguous cases for adjudication. An annotation guide should include benign security articles, malicious mixed-content documents, ordinary procedural text, and examples where a suspicious phrase is only discussed rather than followed.

C. Data Cleaning and Structure

Preprocessing should normalize Unicode, parse HTML, extract visible and hidden text, and remove exact duplicates. The original representation should be preserved securely for reproducibility, while the detector receives normalized text and structural indicators. HTML comments, metadata, alt text, code blocks, and unusual formatting should be checked because instructions can be placed outside ordinary paragraphs. Normalization must not erase evidence of obfuscation; therefore, transformation flags should remain available to the classifier.

D. Train, Validation, and Test Splits

Random sentence-level splitting can leak near-duplicate attacks across subsets and inflate measured performance. The dataset should be split by source, attack template, and collection period. Training data supports parameter fitting, validation data selects thresholds, and the test set is reserved for final evaluation. A separate out-of-domain set should include unseen source types and attack families. Source-separated testing is particularly important when the same malicious phrase appears in several documents.

E. Class Imbalance and Quality

Benign passages will often outnumber malicious passages in real retrieval systems. Accuracy alone can therefore appear high even when the detector misses attacks. Training may use class weights or sampling, but evaluation should reflect realistic prevalence or clearly state how prevalence affects precision. Annotation agreement, duplicate rate, domain coverage, synthetic-data proportion, and missing metadata should be reported. Label quality is a major factor: unclear definitions can make a good detector appear weak or a poor detector appear strong.

VI. DETECTION METHODS

A. Rule-Based Baseline

The simplest baseline uses patterns for explicit attempts to ignore prior instructions, reveal hidden prompts, expose credentials, or invoke tools without authorization. It may also flag hidden markup and suspicious encodings. Rules are fast and explainable, making them useful as an early screening stage. However, they can be bypassed through paraphrasing and may over-block educational content. Rule matches should therefore be treated as features rather than the only decision criterion.

B. Classical Machine Learning

A logistic-regression or linear support-vector classifier can use word and character n-grams, sentence embeddings, punctuation patterns, source features, and instruction verbs. These models are comparatively inexpensive and can be inspected through feature weights. They are suitable baselines for a student implementation. Their limitation is context: the same phrase can be malicious in one setting and harmless in another, while long-range relationships across a document may not be represented well by bag-of-words features.

C. Transformer Classification

A compact transformer [7], [8] fine-tuned for binary or multi-label classification can estimate instruction intent using contextual representations. Inputs should include a candidate passage and selected provenance fields within a strict token budget. Long documents may be segmented, but neighboring context should be included when possible because chunk boundaries can hide intent. Confidence calibration is necessary because a model may be overconfident on unfamiliar domains. Evaluation should measure both attack detection and retention of benign evidence.

D. LLM-Assisted Verification

An LLM may serve as a second-stage verifier for uncertain passages by answering a constrained classification question. The candidate must be presented as quoted untrusted data, and the verifier must not execute its instructions. This method can produce flexible semantic judgments but may share vulnerabilities with the target model. It also increases cost and latency. It should therefore be reserved for a narrow uncertainty band or high-impact cases rather than applied to every passage.

E. Hybrid Cascade

The proposed cascade runs low-cost checks first, then a compact classifier, and finally an optional verifier for borderline cases. Early exits keep average latency manageable. Mandatory quarantine rules handle critical patterns, and every model or rule update is evaluated against a fixed regression suite. Rollback should be possible if attack recall drops or false positives increase. The cascade should be benchmarked against each component individually to verify that added complexity produces a measurable benefit.

VII. IMPLEMENTATION PLAN

A. Prototype Components

A practical prototype can be implemented in Python. A retrieval layer searches a local document collection; a parser extracts text and provenance; a rule engine handles explicit patterns; and a classifier returns probabilities and categories. The policy gate should be a separate module with unit tests for each action. The generator is called only after the gate approves the context. The initial system can operate on local PDF, HTML, and text files before any external tools are connected.

B. Suggested Technology Stack

Python supports data preparation and experimentation. Scikit-learn can implement classical baselines, while a transformer library [8] supports fine-tuning. A lightweight API framework can expose the detector as a service, and SQLite or structured JSON can store non-sensitive audit metadata. The project should pin package versions, record model and rule versions, and provide a reproducible configuration. Tool execution should be disabled during detector evaluation to prevent unintended side effects.

C. Pseudocode

Input: user query q and retrieved chunks C. For each chunk, attach authenticated provenance; normalize text while recording transformations; extract candidate instruction spans; compute rule features and classifier probabilities; combine intent, source risk, obfuscation, and impact into a risk score; quarantine critical cases; sanitize moderate-risk spans; and allow low-risk content with untrusted-evidence markers. Build the approved context, pass it to the generator, validate the response and proposed tool calls, and store decision metadata without unnecessary sensitive content.

D. Logging and Monitoring

Operational logs should capture detector version, source hash, risk band, action, latency, and error status. They should avoid storing full prompts, personal documents, or secrets unless a documented requirement and appropriate authorization

permit it. Monitoring should track false-positive reports, user overrides, source-distribution changes, confidence drift, and processing failures. A sudden decline in detected attacks may indicate a new attack style or broken integration rather than an improvement.

E. Failure Handling

If the detector is unavailable, the application must follow an explicit fallback policy. A low-risk informational system may use a stricter rule-only mode, while a sensitive tool-using system should fail closed and prevent untrusted content from triggering actions. Fallback must never bypass authorization. Timeouts, malformed documents, unsupported encodings, and oversized passages should be treated as operational events and tested deliberately.

VIII. EXPERIMENTAL SETUP

A. Research Questions

The evaluation asks: RQ1, does context-aware detection improve attack recall over keyword rules? RQ2, how much benign evidence is lost during sanitization? RQ3, what latency overhead does the cascade add? RQ4, does provenance help when text is ambiguous? RQ5, how well does the detector generalize to unseen domains and attack templates? These questions connect model quality with system usefulness rather than measuring classification accuracy alone.

B. Experimental Conditions

All candidate methods should be evaluated on the same source-separated test set. Thresholds are selected using validation data, not the final test set. Results should be reported for direct instructions, indirect document attacks, obfuscated payloads, benign quoted examples, and mixed-content passages. Behavioural testing in the style of CheckList [12] can structure these categories. An ablation study removes provenance, structural features, semantic classification, and second-stage verification one at a time. Repeated runs with fixed seeds help identify variation caused by training randomness.

C. Baselines and Reproducibility

The comparison should include keyword-only rules, a classical classifier, a compact transformer, and the full cascade. Training parameters, model versions, split identifiers, thresholds, and preprocessing steps must be recorded. If a commercial LLM is used as a verifier, its model identifier and evaluation date should be documented because behaviour may change. Report confidence intervals where appropriate and clearly describe the limitations of the test corpus.

D. Evaluation Metrics

Attack precision is the fraction of flagged passages that are genuinely malicious; attack recall is the fraction of malicious passages detected. F1 balances the two. False-positive rate measures benign passages incorrectly blocked. Evidence retention estimates how much useful content remains after sanitization. End-to-end latency includes parsing, classification, policy decisions, and any verifier call. Memory use and throughput matter for modest hardware. Metrics should be reported by attack category as well as overall.

E. Utility-Aware Evaluation

A detector that blocks every passage could achieve perfect attack recall but make RAG useless. Evaluation must measure the trade-off between security and utility. A utility score can combine answer-support retention, retrieval coverage, and the fraction of benign passages passed through. Human review of sanitized outputs should determine whether retained text still supports the original query. The operating threshold should be justified by application risk, not solely by the highest F1 score.

IX. RESULTS AND ANALYSIS PLAN

A. Expected Analysis

No dataset was collected and no model was trained for this paper; therefore, this section does not claim measured accuracy. The first analysis will compare rule-based filtering with learned classifiers on attack recall and false-positive rate. The second will test whether the cascade improves recall at acceptable latency. The third will measure evidence retention after sanitization. The fourth will evaluate unseen domains and obfuscation variants, where generalization may be more difficult.

B. Error Analysis

False negatives should be grouped by paraphrase, hidden formatting, split instructions, role impersonation, and cross-chunk dependencies. False positives should be examined for quoted attacks, software documentation, security research, fictional dialogue, and ordinary task instructions. Each error should be assigned a root cause such as missing features, annotation ambiguity, metadata failure, threshold selection, or model limitation. This process supports targeted improvement rather than indiscriminate increases in model size.

C. Ablation and Robustness

The ablation study should remove provenance, structural parsing, semantic classification, and verification in turn. Robustness tests should include Unicode confusables, whitespace changes, paraphrases, multilingual text where supported, and attacks split across adjacent chunks. Tests should also measure whether harmless documents are flagged merely because they discuss prompt injection. Source-level resampling should be considered when passages from the same document are not independent.

D. Results Table Policy

The final report should contain real computed values for precision, recall, false-positive rate, evidence retention, and latency. Until experiments are executed, table cells should remain marked as unavailable rather than filled with invented values. Results should include per-category performance and uncertainty estimates. A small benchmark can reveal implementation defects but cannot establish production safety. Real-world validation requires different retrievers, document types, model families, and user tasks.

X. SECURITY, PRIVACY, AND ETHICS

A. Defence in Depth

PromptShield is a screening layer, not a complete security boundary. Applications must isolate secrets, apply least-

privilege permissions, validate structured arguments, and require confirmation for consequential actions. Retrieved documents should not be able to rewrite system policy. The generator should treat sources as evidence and ignore embedded requests unrelated to the user's task. Output validation can reduce leakage but must not be the only safeguard.

B. Privacy-Preserving Telemetry

Security telemetry can expose confidential content. By default, logs should record hashes, decision codes, aggregate counts, and latency rather than entire documents or conversations. Access to diagnostic samples should be role-restricted, time-limited, and governed by retention policies. If user data is used for a research dataset, the project should minimize collection, remove identifiers, document consent or other lawful basis, and follow institutional requirements. Redaction must be validated because indirect identifiers may remain.

C. Fairness and Language Coverage

A detector trained mainly on English attacks may perform poorly on other languages, code-switching, and domain-specific terminology. It may also over-flag assertive writing styles. Evaluation should report performance by language and source type when enough examples exist. Multilingual coverage should be expanded through reviewed examples and separate calibration, rather than assuming translation always preserves attack semantics.

D. Responsible Disclosure

A research prototype should use controlled test documents and synthetic secrets. It should not test against third-party systems without authorization or publish operational details that expose real credentials. When a vulnerability is discovered in a deployed product, responsible disclosure should follow the vendor's reporting process. The focus is defensive detection and safe system design.

XI. DISCUSSION AND APPLICATIONS

A. Design Trade-Offs

The main challenge is improving security without destroying retrieval utility. Aggressive filtering reduces the chance that malicious instructions reach the model, but can remove useful procedural information. A permissive policy retains more evidence but leaves more room for attacks. The staged policy makes this trade-off explicit and allows application-specific thresholds. A personal knowledge assistant may tolerate added delay for stronger checks, while a high-volume support system may need predictable overhead.

B. Why Context Matters

The same text can play different roles. A security article may quote an attack phrase as an example, while a poisoned webpage may use it to redirect the assistant. A keyword detector sees the same phrase in both cases. Context-aware classification considers whether the text is quoted, whether it addresses the assistant, whether it conflicts with the user's task, and whether it requests a protected action. Provenance adds another signal but should not automatically mark every external source malicious.

C. Real-World Applications

Potential applications include enterprise document assistants that search internal policies; coding assistants that inspect repository files and comments; browser research tools that read webpages; customer-support automation that processes emails and attachments; educational assistants that discuss AI security; legal search systems that summarize documents; and personal productivity agents that process calendar invitations or shared notes. In each setting, retrieved content should remain evidence rather than authority, and tool permissions should be checked independently.

D. Limitations

The framework is conceptual and has not been validated on a collected benchmark. Risk weights and thresholds require calibration. Attack categories evolve, and a detector may fail on novel multi-step or cross-modal attacks. Text-only screening does not address images or audio unless extracted representations are also screened. Provenance may be missing or forged, and an LLM verifier may share vulnerabilities with the target generator. These limitations require cautious claims and ongoing testing.

XII. FUTURE WORK

A. Benchmark Development

The first priority is a benchmark with source-separated evaluation, consistent labels, and realistic mixed-content passages. A public benchmark should document licensing, privacy review, synthetic-data proportion, annotation procedures, and known limitations. Periodic updates are needed because new attack styles and model behaviours can make a fixed benchmark stale.

B. Adaptive Thresholding

Future versions may select thresholds according to action impact, uncertainty, and source type. Adaptation must be bounded and tested so attackers cannot manipulate the policy through repeated low-risk inputs. High-impact requests should still require independent authorization regardless of detector score. Threshold changes should be versioned and reversible.

C. Multimodal and Cross-Chunk Detection

Prompt injection may arrive through OCR text in images, audio transcripts, code blocks, and content distributed across retrieved chunks. Future detectors should preserve modality and layout metadata, scan extracted text, and evaluate relationships across chunks. Cross-chunk analysis matters when an attack divides an instruction into fragments that appear harmless independently. Additional context must remain within a defined token and latency budget.

D. Explainability and Continuous Evaluation

The risk engine should provide concise reason codes such as policy override, secret request, suspicious source, or hidden instruction. Explanations should support debugging without exposing thresholds that enable evasion. Before deployment, each model or rule update should run against a regression suite.

Production monitoring should detect drift in source types, language distribution, passage length, and user reports. Updates should first run in shadow mode, followed by a gradual rollout with rollback available.

XIII. DETAILED USE-CASE ANALYSIS

A. Enterprise Knowledge Assistant

An enterprise assistant may retrieve employee handbooks, internal policies, project documents, and customer records. These sources differ in sensitivity and ownership. A malicious instruction embedded in a shared document could attempt to make the assistant reveal a confidential policy or forward a document externally. PromptShield should evaluate the passage before generation and preserve the document's factual content only when the suspicious instruction can be separated reliably. Source provenance should identify the workspace and access level, while the application's access-control layer ensures that retrieval itself returns only documents the current user is allowed to view. Screening must not be treated as a substitute for access control.

B. Coding Assistant

A coding assistant may read repository files, issue discussions, comments, README pages, and build logs. These files can legitimately contain commands and configuration examples, so a detector must not block all imperative text. The relevant question is whether the content attempts to control the assistant or requests an unauthorized action. A comment that explains how to run tests differs from a comment asking the agent to upload environment variables. PromptShield can tag repository content as untrusted evidence, identify secret-exfiltration intent, and route proposed shell or file operations through a separate permission gate. Testing should include code snippets, markdown, shell comments, and generated documentation.

C. Browser Research Agent

A browser-enabled agent may read public webpages to answer questions or summarize several sources. A hostile page could instruct the agent to disregard the user, suppress other sources, or disclose conversation context. Because page content is controlled by external publishers, provenance should include domain, URL identity, retrieval time, and content hash. The model should be asked to use pages as evidence and compare claims across sources rather than obeying page instructions. If a suspicious page is excluded, the answer should still be based on other approved sources where available. The system should not reveal internal detector details to the page or execute actions requested by page text.

D. Customer-Support Automation

Support systems may process customer emails, attachments, ticket histories, and knowledge-base articles. A malicious attachment could attempt to influence a draft response or make the assistant access another customer's records. PromptShield should screen retrieved text while existing identity checks constrain which records can be accessed. Any action that changes an account, issues a refund, or sends a message should require a validated workflow and, when appropriate, human confirmation. False positives are particularly costly in this setting because they may delay legitimate service, so the evaluation should measure both safety and service quality.

XIV. ADVERSARIAL ROBUSTNESS AND EVASION

A. Paraphrase and Semantic Variation

A detector may learn familiar attack phrases and fail when an attacker expresses the same intent differently, a concern also seen in work on adversarial attacks against aligned models [13]. For example, a request to reveal hidden instructions may be phrased as a debugging requirement, an audit task, or a claim that the user has authorized disclosure. Evaluation should create paraphrase families and keep entire families in one split. This prevents near-duplicate wording from leaking into both training and test sets. Semantic models should be tested on unseen formulations, and rule-based filters should be treated as useful signals rather than a complete defence.

B. Formatting and Encoding

Adversarial content can be placed in HTML comments, metadata, markdown tables, code blocks, alt text, or unusual Unicode sequences. A parser should identify these locations and preserve a flag describing the original structure. Blindly deleting all unusual formatting is not reliable because legitimate documentation may use code or tables. The detector should evaluate the normalized text together with structural features and record when decoding or normalization occurred. Test cases should include whitespace changes, punctuation insertion, mixed casing, confusable characters, and instructions split by markup.

C. Multi-Chunk Attacks

Retrieval systems often divide documents into chunks to fit embedding and model limits. An attacker may distribute an instruction across neighboring chunks so that each chunk looks harmless by itself. A chunk-level detector can miss the combined intent. PromptShield should therefore support selective neighbor-context analysis when a chunk contains suspicious references, incomplete directives, or repeated patterns. This analysis must be bounded to avoid unnecessary token usage and should not merge unrelated documents. The test set should include single-chunk and multi-chunk examples, with source-level splits to avoid leakage.

D. Adaptive Attackers

A determined attacker may probe an application and adjust content based on observable outcomes. The application should not expose exact thresholds, detailed classifier probabilities, or internal system instructions. User-facing messages can explain that content was excluded because it contained unrelated or unsafe instructions without disclosing the detector's full decision process. Regression tests should be refreshed with newly observed patterns. However, secrecy of the detector is not a replacement for robust access control and tool authorization; the system must remain safe even when an attacker understands its broad architecture.

XV. HUMAN REVIEW AND GOVERNANCE

A. Review Workflow

Human review is useful for ambiguous cases, but should not become a bottleneck for every retrieved passage. The policy engine can reserve review for high-impact actions, repeated suspicious content, or low-confidence decisions that cannot be resolved automatically. Reviewers should see the source, relevant context, reason code, and proposed mitigation, while unnecessary personal data remains masked. The interface should support decisions such as confirm malicious, mark benign, preserve safe excerpt, or request further analysis. Each reviewer action should be recorded with role and timestamp.

B. Feedback and Model Updates

Feedback can improve the detector, but unreviewed user labels may be noisy or adversarial. The training pipeline should require curation, duplicate detection, label validation, and a held-out regression test before new examples are incorporated. Model updates should be versioned, and thresholds should be recalibrated after changes in the model or source distribution. A rollback plan should restore the previous stable version if false positives increase or attack recall falls. The team should document which feedback sources contributed to each release.

C. Governance and Accountability

Organizations deploying a RAG security layer should define who owns the model, who approves policy thresholds, who investigates incidents, and who may access diagnostic samples. Documentation should state known limitations and supported languages. High-impact systems may require formal risk assessment, audit logging, and periodic security reviews, consistent with information security management practice [14]. The project should identify how a source can be disabled, how exposed credentials are rotated, and how affected users are informed. Clear ownership prevents a detector from becoming an unmaintained component that silently fails.

D. Accessibility and User Communication

Security warnings should be understandable to non-technical users. A concise explanation can say that a retrieved passage was excluded because it attempted to change the assistant's instructions or request a restricted action. The user may be offered a safe summary of the source when appropriate. Excessive warnings can cause users to ignore important alerts, so messages should be proportional to the risk. Administrators can receive more detailed reason codes through an authenticated interface, while general users should not see sensitive internal prompts or thresholds.

XVI. PERFORMANCE AND RESOURCE MANAGEMENT

A. Latency Budget

Every screening stage adds processing time. The implementation should measure parsing, rule checks, classifier inference, policy evaluation, and optional verification separately. A cascade can reduce average cost by allowing low-risk passages to exit early. The verifier should be reserved for uncertain or high-impact cases. Reporting only the detector's model inference time is insufficient; end-to-end latency should include network calls and serialization if those are part of deployment. Experiments should use a fixed hardware environment and report the number of passages processed.

B. Throughput and Batching

High-volume systems may screen several retrieved chunks for one user query. Batch inference can improve throughput for transformer classifiers, but large batches can increase latency for small requests. The implementation should compare single-item and batched processing under realistic workloads. Cache keys must include the content hash and detector version so that a previous safe decision is not reused after content or policy changes. Caching should not allow one user's permissions or context to be reused for another user.

C. Memory and Deployment

A student prototype may run on a standard laptop, while a larger deployment may use a separate inference service. Resource measurements should include model memory, peak memory during document parsing, and the effect of long passages. The system should set maximum document and chunk sizes, handle malformed input safely, and avoid unbounded processing. A compact classical classifier may be appropriate where resources are limited; transformer inference can be enabled for higher-risk contexts if latency permits.

D. Cost-Aware Policy

The risk of an action should influence how much verification is performed. A passage used to summarize a public article may need only the fast screening stage, whereas a passage that could influence an agent with access to customer records should receive stronger checks. The policy must remain explicit and testable rather than dynamically granting privileges based on classifier confidence. Costs should be measured against security and utility outcomes, and any use of external verification services should consider data minimization and privacy.

XVII. TESTING STRATEGY AND QUALITY ASSURANCE

A. Unit Tests

Unit tests should verify text normalization, extraction of structural features, risk-score calculation, threshold behaviour, and every policy outcome. Tests should include empty passages, malformed HTML, very long inputs, Unicode characters, and missing provenance. Each function should have a documented contract and predictable error handling. Unit tests do not establish security, but they prevent basic implementation defects from invalidating the evaluation.

B. Integration Tests

Integration tests should trace a passage from retrieval through screening, generation, and output validation. A malicious passage must not bypass the policy gate due to a serialization error or an unexpected data type. A blocked passage should not remain in the context through a duplicate chunk. Tool requests must be rejected when authorization fails even if the generated explanation claims the action is safe. Logs should record the outcome without copying sensitive content unnecessarily.

C. Regression Tests

A fixed regression suite should contain confirmed false negatives, false positives, benign quoted examples, and previously observed obfuscation variants. Every detector update should run against this suite before release. Performance changes should be compared with a documented baseline. If the test set is modified, its version and reason for change should be recorded so that comparisons across releases remain meaningful.

D. Acceptance Criteria

A project may define acceptance criteria before implementation, such as a target recall on a specified test set, a maximum false-positive rate, a latency budget, and a minimum benign-evidence retention rate. These thresholds must be selected according to the intended application and reported honestly. They should not be retroactively chosen only to make results look favourable. If a criterion is not met, the report should explain the cause and propose further work rather than hide the result.

XVIII. LIMITATIONS AND THREATS TO VALIDITY

A. Dataset Validity

A benchmark may not reflect real-world traffic. Synthetic examples may be formulaic, while public attack examples may overrepresent popular model behaviours. A source-separated split reduces leakage but does not guarantee generalization to every domain. Researchers should describe how data were collected, what sources are absent, how ambiguous cases were labelled, and whether the prevalence of malicious passages resembles deployment conditions. External replication is needed before broad conclusions can be made.

B. Model and Environment Dependence

Detector performance may vary across generator models, retrieval strategies, chunk sizes, and system prompts. A classifier that performs well with one model may fail when the generator changes its instruction-following behaviour. The evaluation should therefore record the generator and retriever configuration and, where feasible, test more than one setting. Changes to external model services may require re-evaluation even when the detector itself has not changed.

C. Measurement Limitations

Precision and recall do not capture every user-facing effect. Evidence retention may not fully represent answer quality, and latency averages can hide rare slow requests. Human assessment can be subjective unless reviewers use a common rubric. Statistical intervals should account for document-level clustering where feasible. A small final-year prototype can demonstrate a method and expose implementation challenges, but it cannot prove that all prompt injection attacks will be detected.

D. Generalization and Residual Risk

Some attacks may be indistinguishable from legitimate instructions without application-specific context. A malicious request can be hidden in a larger workflow or distributed across multiple sources. The detector may also be manipulated through adversarial examples that exploit its own learned features. Residual risk should be documented and managed through access controls, least privilege, user confirmation, source review, and incident response. No single classifier can guarantee secure behaviour.

XIX. FINAL PROJECT IMPLEMENTATION ROADMAP

A. Phase One: Requirements and Threat Modelling

The project begins by identifying the target application, assets, source types, and actions that must be protected. The team should draw a data-flow diagram and mark trust boundaries between user input, retrieval, model context, output, and tools. The threat taxonomy and expected policy outcomes should be written before data collection. This creates a clear scope and prevents the prototype from becoming only a keyword filter.

B. Phase Two: Baseline and Dataset

The next phase builds a local RAG prototype and collects a small, licensed dataset of benign and adversarial passages. The annotation guide should be tested on a pilot sample and refined when annotators disagree. A rule-based baseline provides a transparent reference point. The team should freeze source-separated training, validation, and test splits before comparing more advanced models.

C. Phase Three: Classifier and Policy Gate

A classical classifier can be trained first because it is relatively easy to implement and explain. If resources allow, a compact transformer can be evaluated on the same split. The policy gate combines detector outputs with provenance and action impact, and tests verify that allow, sanitize, quarantine, and reject-source actions behave as intended. The generator receives only approved context, with retrieved passages clearly labelled as untrusted evidence.

D. Phase Four: Evaluation and Demonstration

The final phase reports precision, recall, false-positive rate, evidence retention, latency, and resource use. Error analysis should include both missed attacks and benign passages incorrectly blocked. A demonstration interface can show the source, risk band, reason code, and action taken. The final report should separate measured outcomes from proposed features, include limitations, and avoid unsupported claims about complete security.

XX. SUMMARY OF DESIGN DECISIONS

Table I summarizes the core design choices proposed for PromptShield. These choices should be verified during implementation and may be adjusted based on measured results.

TABLE I. SUMMARY OF PROPOSED DESIGN DECISIONS

Component	Design choice	Reason
Provenance	Authenticated metadata and content hash	Traceability and source-aware risk
Fast screening	Rules plus structural indicators	Low-cost detection of obvious patterns
Semantic stage	Classical or transformer classifier	Context-sensitive instruction detection
Policy gate	Allow, sanitize, quarantine, reject	Different actions for different risks
Tool security	Independent authorization checks	Prevents detector bypass from granting access
Monitoring	Minimal metadata and regression tests	Auditing while reducing data exposure

XXI. DATA FLOW AND TRUST BOUNDARIES

A. Data-Flow Stages

The end-to-end data flow begins when the user submits a query. The retrieval service searches only sources available to that user and returns candidate chunks with source identifiers. A normalization layer converts supported formats into text while retaining source and layout metadata. The screening engine evaluates each chunk and emits a decision record. The context builder then assembles approved evidence, and the generator produces a response. An output validator examines the response and any proposed tool request before returning the answer. Each stage should have a defined input and output contract so that a component cannot silently bypass screening.

B. Boundary Enforcement

A trust boundary is useful only if it is enforced by code rather than merely described in a prompt. The context builder should label retrieved passages as untrusted evidence and should not merge them with developer instructions. The application should prevent a retrieved document from changing tool permissions or system configuration. Where the model supports structured message roles, source text should remain in a data-bearing message rather than being inserted into a privileged instruction field. Unit tests should inspect the final assembled context to ensure that source labels and boundaries survive formatting.

C. Source Identity

The same text may appear in several places, such as a mirrored webpage and an internal copy. Source identity should therefore include a canonical source identifier and a content hash. When a source is updated, the system should re-screen its content rather than reusing a previous decision without validation. If an administrator revokes trust in a source, cached content from that source should be invalidated. These controls make decisions traceable and reduce the risk that stale screening results remain active after a document changes.

D. Safe Context Assembly

After screening, the context builder should prioritize relevant approved evidence and enforce a token budget. It should not reintroduce quarantined passages through summaries or duplicate retrieval results. If a sanitized excerpt no longer contains enough information to answer the question, the generator should state that evidence is insufficient or request a safer source. This behaviour is preferable to silently restoring the removed instruction or inventing facts. Context assembly should be tested separately from classification because many failures occur in integration logic.

XXII. RISK REGISTER AND MITIGATION

A. Risk Register Purpose

A risk register, aligned with established AI risk-management guidance [11], links each threat to a control, an owner, and a verification method. For prompt injection, important risks include missed malicious instructions, excessive blocking of benign content, provenance spoofing, sensitive logging, detector outage, and unsafe tool execution. The register should classify likelihood and impact using a documented scale. Ratings should be reviewed after experiments and incidents rather than treated as permanent values. This helps the team prioritize controls according to the application's real exposure.

B. Missed Detection

A false negative may allow an adversarial passage into the model context. Mitigations include source-aware screening, semantic classification, regression tests, least-privilege tools, and explicit confirmation for consequential actions. The team should analyse missed cases by attack type and add representative examples to the regression suite. No individual control eliminates the risk, so the architecture should assume that some malicious text may reach the model and limit the damage it can cause.

C. False Positives

A false positive may remove legitimate instructions or useful evidence, lowering answer quality. The system should evaluate benign-evidence retention and maintain a safe path for quoted security examples. Sanitization is preferable to rejecting an entire document when the unsafe span can be isolated with confidence. Human review can help resolve persistent ambiguous cases, but review outcomes should be curated before they are used as training data. Thresholds should be chosen based on measured security and utility trade-offs.

D. Operational Failure

A service outage, parsing failure, or incompatible model update can cause the screening layer to behave unpredictably. The application should define fail-closed behaviour for sensitive operations, timeouts, alerting, and rollback. A detector failure must not automatically send all retrieved text to the generator. Monitoring should detect changes in volume, latency, error rate, and risk distribution. Recovery procedures should be rehearsed so that the team can disable a problematic source or restore a stable detector version.

XXIII. COMPARATIVE EVALUATION FRAMEWORK

A. Fair Comparison

Candidate detectors should use the same training and test examples, the same label definitions, and comparable preprocessing. Each method should be tuned on validation data and evaluated once on the held-out test set. If a transformer uses

extra contextual metadata, an ablation should measure whether that metadata explains the difference. Runtime comparisons should use the same machine and input sizes. A fair comparison makes it possible to determine whether additional complexity provides real benefit.

B. Category-Level Performance

Overall scores can conceal weaknesses. A detector may perform well on direct instructions but poorly on hidden markup or cross-chunk attacks. Results should therefore be broken down by category, source type, language, and benign context. Small categories should be reported with caution and should not support broad claims. Error examples should be selected systematically rather than only showing cases that make the proposed system look successful.

C. Calibration

A classifier's confidence should be compared with observed correctness. Poorly calibrated scores can make risk thresholds unreliable, especially when the test distribution differs from training. Calibration can be assessed with reliability diagrams or a suitable scalar metric, using validation data to select any calibration method. Threshold selection should account for the cost of false negatives and false positives. The paper should record whether reported confidence values are calibrated or raw model outputs.

D. Human Utility Assessment

Automatic metrics do not fully measure whether a sanitized passage remains useful. A small human study can ask reviewers whether the retained excerpt supports the original query, whether the warning is understandable, and whether the final answer omits important facts. Reviewers should use a fixed rubric and should not know which detector produced the output when practical. The study should report the number of examples, review procedure, disagreement, and limitations.

XXIV. IMPLEMENTATION DETAILS FOR REPRODUCIBILITY

A. Configuration Management

All model settings should be stored in a versioned configuration file, including tokenizer version, maximum sequence length, classification threshold, risk weights, and policy actions. The code should log the configuration identifier used for each evaluation run. This avoids a situation where results cannot be reproduced because parameters were changed informally during development. Configuration changes should be reviewed and linked to test outcomes.

B. Data Versioning

Each dataset version should have a manifest that records source identifiers, label schema, preprocessing version, and split assignment, in the spirit of datasheets for datasets [10]. Sensitive raw documents should be stored separately with restricted access. A content hash can identify duplicates without exposing the full text in ordinary logs. If a label is corrected, the change should be documented so that the effect on reported results is traceable.

C. Model Versioning

The detector should identify the model and rule-set version in every audit event, and model documentation can follow the model-card practice [9]. A new model should not replace the deployed model until regression tests and performance checks pass. Where multiple versions are compared, the evaluation should use identical test inputs. Model versioning is particularly important when using remote inference services whose behaviour may change without the application code changing.

D. Reproducible Reporting

The final report should include the hardware environment, software versions, dataset split policy, hyperparameters, threshold-selection method, and metric definitions. Tables should state whether values are measured, estimated, or placeholders. A concise experiment log should record failed runs as well as successful ones. Reproducibility does not guarantee correctness, but it enables reviewers to inspect how conclusions were reached and repeat the evaluation.

XXV. MULTILINGUAL AND DOMAIN-SPECIFIC CONTENT

A. Language Variation

Prompt-injection examples may use languages other than English, transliteration, mixed-language sentences, or regional phrasing. A detector trained only on standard English may miss the intended meaning or over-flag unfamiliar expressions. The dataset should document language coverage and include reviewed examples for supported languages. Translation can help expand training data, but translated text should be checked because idioms, politeness, and imperative forms do not always map directly across languages. The system should be honest about unsupported languages and apply stricter handling where the detector has low confidence.

B. Domain Terminology

Technical domains use words that may resemble suspicious instructions. Security manuals discuss secrets and attacks; operating-system documentation includes commands; customer support uses phrases such as reset, delete, and account. A context-aware model should learn the role of these terms in the passage rather than rely on isolated keywords. Domain-specific evaluation sets can reveal whether a detector is overfitting to generic examples. When a system is deployed in a new domain, the team should run a targeted evaluation before enabling automated actions.

C. Code and Structured Data

Code repositories may contain executable commands, configuration fields, comments, and serialized messages. A parser should preserve the difference between code and natural-language explanation. The detector should identify instructions that target the assistant while allowing legitimate code examples to remain available as evidence. JSON, YAML, XML, and markdown can encode text in nested structures, so test cases should cover escaped strings, multiline values, and comments where supported. Parsing should be deterministic and should not execute source content.

D. Coverage Reporting

Language and domain coverage should be reported as part of the system's scope. If there are too few examples in a category, the paper should mark the result as exploratory rather than definitive. Performance differences may reflect dataset size as well as model quality. Future dataset releases should invite reviewed contributions from different domains while maintaining consistent labels and privacy safeguards.

XXVI. SOURCE QUALITY AND RETRIEVAL GOVERNANCE

A. Source Admission

A retrieval collection should have a documented process for adding sources. Curated documents may be reviewed before indexing, while public sources may require stronger screening and more frequent revalidation. Admission does not guarantee that a source will remain safe: content can change after review or an account can be compromised. Source governance should therefore combine initial review, provenance tracking, content hashing, and re-screening after updates.

B. Retrieval Ranking and Risk

Relevance ranking and security risk should be separate signals. A relevant source should not receive authority simply because it ranks highly, and a source with lower trust should not automatically be discarded if it contains valuable evidence. The policy layer can use risk to decide whether to retain, sanitize, or quarantine a passage after retrieval. Future work may study whether risk-aware reranking improves overall utility, but such reranking must be evaluated to ensure it does not systematically suppress legitimate external information.

C. Source Revocation

If a source is identified as compromised, the application should be able to disable it, invalidate cached chunks, and record the time of revocation. A source-level block should not be the only remedy when the same content has been copied elsewhere. Content hashes and similarity checks can help locate duplicates, although similarity matching may also flag benign copies. The incident process should include reindexing, regression testing, and review of any outputs produced while the source was active.

D. Accountability

Source governance needs an accountable owner. A team should know who approves curated sources, who reviews reported attacks, and who can override source restrictions. Changes should be auditable and reversible. For student projects, a simple allowlist of local files and an explicit source registry can demonstrate these principles without requiring a complex enterprise platform.

XXVII. OUTPUT VALIDATION AND ANSWER QUALITY

A. Answer Grounding

After generation, the application should check whether key claims are supported by approved evidence. This is distinct from prompt-injection detection: a passage may be safe but factually incorrect, and a model may generate an unsupported statement without any malicious input. A grounding check can compare answer claims with the approved context or ask for citations to source identifiers. Unsupported claims should be revised, qualified, or removed rather than presented as established facts.

B. Secret and Policy Leakage

An output validator can scan for known secret patterns and inappropriate disclosure of protected instructions. It should avoid storing the sensitive value when logging a detection; a hash or redacted indicator may be enough. The system should also consider whether the answer reproduces a malicious instruction in a way that could be mistaken for an endorsed recommendation. However, educational explanations may need to quote short examples, so output checks should use context rather than block every suspicious phrase.

C. Tool-Call Validation

A tool request should be checked against a schema and authorization policy before execution. The policy should validate the action, arguments, resource scope, user permission, and whether confirmation is required. If the model requests an action outside its allowed capability, the tool layer should reject it and return a safe error. A detector's low risk score must never grant new privileges, and a high score should not be bypassed merely because the model offers a persuasive explanation.

D. Answer Recovery

When suspicious evidence is excluded, the assistant may still answer using other approved sources. If the remaining evidence is insufficient, it should state the limitation or ask the user for a trusted source. It should not silently restore quarantined text or invent details to compensate for missing context. Recovery behaviour should be tested because a detector can be technically correct while the overall application still produces an unhelpful answer.

XXVIII. DEPLOYMENT SCENARIOS AND CONFIGURATION

A. Low-Risk Read-Only Scenario

A read-only assistant that summarizes public articles may use a fast detector, source labels, and output grounding checks. The main risks include inaccurate answers, task redirection, and disclosure of conversation context. The system can sanitize moderate-risk spans and exclude high-risk passages while retrieving alternative evidence. Evaluation should focus on false positives, answer quality, and latency as well as attack recall.

B. Sensitive Document Scenario

An assistant that searches employee, financial, or customer records needs strict retrieval permissions and privacy-preserving logs. PromptShield should operate after access-controlled retrieval, not as a replacement for it. Suspicious passages should be quarantined, and generated answers should be checked for unauthorized disclosure. Tool calls that export or modify sensitive records should require stronger authorization and often human confirmation. Retention policies should define how long audit metadata and review samples remain available.

C. Tool-Using Agent Scenario

An agent that can run commands, send emails, or update records has a higher impact profile than a read-only summarizer. In this setting, risk thresholds may be more conservative, but the most important control is independent tool authorization. The model should receive only the capabilities required for the task, and sensitive operations should use allowlisted actions and typed arguments. The evaluation should include attempts to trigger tool calls through retrieved text and verify that the action layer rejects unauthorized requests.

D. Configuration Profiles

A deployment profile can specify source categories, allowed tools, risk thresholds, logging rules, and fallback behaviour. Profiles should be reviewed and versioned. A change from read-only use to tool-enabled operation requires a new threat assessment and evaluation, because the impact of an identical attack may change substantially. Configuration should not be silently inferred from user wording; it should be controlled by the application.

XXIX. ETHICAL AND ACADEMIC REPORTING

A. Research Integrity

The final paper should state clearly which parts of PromptShield were implemented and which remain proposed. Any reported metric must come from an executed experiment with documented data and procedures. If the work is a design study, the title and abstract should avoid implying that the system has already demonstrated superior performance. A transparent account of limitations is more valuable than unsupported claims of high accuracy.

B. Citation Verification

References should be checked against the original publication or an authoritative bibliographic record. The final author team should confirm the exact title, venue, year, volume, and page range where applicable. A citation should support the nearby claim rather than merely be topically related. If a source cannot be verified, it should be corrected or removed before submission. The present reference list is a starting point and should be reviewed before academic use.

C. Dataset and Privacy

Dataset construction should minimize collection of personal information and avoid using confidential documents without permission. The project should document licences, source restrictions, and any consent or institutional review requirements. Public availability does not automatically mean unrestricted reuse. If examples contain credentials or personal data, they should be replaced with synthetic values or redacted carefully before use in a classroom demonstration.

D. Limitations as a Contribution

A well-written limitation section helps readers understand where a method can and cannot be applied. PromptShield may fail on novel attack phrasing, non-English content, multimodal inputs, or malicious instructions spread across sources. It may also block benign material. Recording these limitations provides a useful roadmap for further work and prevents a prototype from being mistaken for a production-grade security guarantee.

XXX. PROJECT DEMONSTRATION PLAN

A. Demonstration Setup

The demonstration can use a local collection containing benign technical documentation, a benign educational article about prompt injection, a mixed-content document, and several controlled adversarial examples. A user enters a query, the retriever selects passages, and the interface displays source metadata and the detector's decision. The model then generates an answer using only approved context. No real credentials or external tool actions should be used during the demonstration.

B. Visualizing Decisions

The interface may display a source name, risk band, short reason code, and action. It should not expose the exact system prompt or sensitive detector thresholds. A comparison panel can show how a keyword-only baseline and the hybrid system treat the same examples. This makes the research question visible to reviewers and helps explain why context matters. Any sample score should be labelled as a model output rather than a universal measure of safety.

C. Demonstrating False Positives

A useful demonstration includes benign material that discusses suspicious phrases without attempting to execute them. If the detector blocks such material, the team should explain the failure and show how context or a revised threshold may improve behaviour. Demonstrating limitations makes the project more credible and reveals whether the system is preserving legitimate evidence rather than simply blocking everything.

D. Demonstrating Tool Safety

A mock tool can represent a protected action such as sending a message or reading a private file. A retrieved attack may cause the model to request the action, but the independent authorization layer should reject it unless the user and policy permit it. The demo should log the attempted request and the reason for rejection without performing an external side effect. This illustrates that detection and authorization are complementary controls.

XXXI. CONCLUDING OBSERVATIONS

A. Central Principle

The central principle of PromptShield is that retrieved content should be treated as evidence, not authority. A relevant document can inform an answer without being allowed to rewrite the application's rules. This separation should be expressed in the architecture, data representation, context assembly, and tool authorization layer. It should not depend exclusively on a carefully worded prompt.

B. Practical Value

The framework provides a structured path from a simple rule-based prototype to a more context-aware system. A final-year team can implement the retrieval boundary, create a labelled dataset, compare classifiers, add a risk policy, and evaluate utility retention. The modular design makes it possible

to test each component independently and to explain decisions during a project demonstration. Its practical value must ultimately be established through implementation and measured evaluation.

C. Research Responsibility

A security paper should distinguish design claims from empirical evidence. The current manuscript describes an architecture and proposed evaluation, not a completed experiment. The next step is to implement the prototype, verify all references, collect or select a suitable dataset, and report the actual results. Any weaknesses should be stated directly so that the work remains credible and useful to future developers.

D. Final Note

Secure retrieval-augmented generation requires cooperation between data engineering, model evaluation, privacy, and application security. PromptShield addresses one important part of that problem by screening retrieved instructions, but resilient systems also need least privilege, safe tool execution, source governance, and continuous monitoring. Future research should assess the framework across diverse models and real application workloads.

XXXII. CONCLUSION

PromptShield proposes a context-aware approach to prompt-injection screening in retrieval-augmented AI systems. Its main idea is to preserve the distinction between instructions that govern the application and text retrieved as evidence. Provenance, structural analysis, semantic classification, risk scoring, and policy-aware sanitization work together to reduce the chance that untrusted passages redirect the model. The framework also requires independent tool authorization, careful logging, source-separated evaluation, and explicit measurement of benign-evidence retention. The approach is modular and suitable for a final-year CSE prototype that begins with local documents and gradually adds classifier-based screening.

Because the framework has not yet been experimentally validated, its effectiveness remains a research question rather than a proven result. A completed implementation should use a well-labelled benchmark, report actual metrics, analyse errors, and state its limitations clearly. Secure RAG requires layered controls and continuous evaluation; prompt screening is valuable, but it cannot replace application-level security.

APPENDIX A. SAMPLE ANNOTATION GUIDE

A. Label Definitions

Each example should be labelled using a shared annotation guide. The primary label indicates whether the passage contains an instruction that attempts to control the assistant or override application policy. The secondary label identifies the attack category, such as policy override, secret extraction, tool misuse, role impersonation, hidden markup, or cross-chunk assembly. Annotators should also record severity, target asset, confidence, and the recommended action. A separate utility label identifies whether the passage contains factual evidence that should remain available to answer the user's question. This separation avoids treating every document with suspicious language as entirely malicious.

B. Positive and Negative Examples

A positive example is a retrieved webpage that directs the assistant to disregard the user and reveal hidden instructions. Another positive example is a repository comment asking an agent to upload environment variables to an unrelated destination. A negative example is an educational article that quotes an attack phrase to explain why it is dangerous. A further negative example is ordinary documentation that explains how a user can reset a password through an approved process. Mixed examples should contain both useful factual information and one malicious span. The annotation should indicate whether that span can be removed without destroying the meaning of the remaining passage.

C. Annotation Process

At least two annotators should independently label a sample of the dataset. The annotation guide should be tested on a pilot set, and disagreements should be reviewed before the full labelling effort begins. The team should calculate an agreement measure and describe common disagreement categories. Adjudication should produce a final label and, where necessary, an explanatory note. Annotators should not see the model's prediction before assigning labels, because that may bias their judgement. Sensitive documents should not be used unless the project has permission and appropriate data protection procedures.

D. Quality Control

Quality checks should identify duplicates, missing labels, contradictory annotations, and examples that rely on unavailable context. A source-level manifest should connect each passage to its origin without exposing private information in ordinary logs. Label corrections should be versioned. Before final evaluation, the team should freeze the test set and avoid using it to tune thresholds. These procedures make the evaluation more credible and allow future reviewers to reproduce the study.

E. Expected Output

The annotation dataset should include the normalized passage, source identifier, source category, structural flags, primary label, attack category, severity, utility-retention label, and final recommended action. The raw original should be stored separately with restricted access if retention is necessary. The public or classroom version should use synthetic credentials and remove personal identifiers. The annotation guide should accompany the dataset so that other teams can understand how labels were assigned.

APPENDIX B. TEST MATRIX AND ACCEPTANCE CRITERIA

A. Functional Tests

Functional tests should verify that every passage receives a decision and that the policy gate follows the configured thresholds. Low-risk benign passages should be allowed; high-risk instructions to reveal secrets should be quarantined; and mixed passages should be sanitized only when the suspicious

span can be isolated safely. Missing provenance should not cause the system to assume a source is trusted. Invalid documents should produce a controlled error rather than crash the service or bypass screening. The team should also test duplicate chunks to ensure that blocked text cannot re-enter the context through another retrieval result.

B. Security Tests

Security tests should cover explicit policy override, secret requests, unauthorized tool instructions, role impersonation, hidden HTML, Unicode variation, and instructions distributed across neighboring chunks. The tool mock should reject unauthorized actions even if the generator requests them. The system should not log raw secrets when a detector flags a passage. When the screening service times out, the configured fallback should be followed consistently. A sensitive deployment should fail closed for actions that could expose data or change records.

C. Quality Criteria

The project should set target values before final testing for attack recall, false-positive rate, benign-evidence retention, and processing latency. The targets should reflect the prototype's purpose and available resources. If the detector misses a target, the report should show the actual value and explain the likely causes. Acceptance criteria must not be changed after seeing the test result simply to claim success. Where the dataset is small, the report should state that the estimate is preliminary and should avoid broad generalization.

D. Test Record

Each test record should include an identifier, scenario, source type, expected decision, observed decision, processing time, and pass/fail status. Failures should be classified as parser errors, detector errors, policy errors, context-assembly errors, or authorization errors. This classification is important because the same final failure can originate in different components. The team should rerun failed tests after fixes and preserve the original result in the project log to maintain traceability.

E. Demonstration Safety

All adversarial examples should use a controlled local dataset. The demonstration should not contain actual credentials, private user records, or live third-party tool access. A mock action can demonstrate how an attempted operation is rejected by authorization. This provides evidence of layered security without creating external side effects. The demonstration should also include at least one benign educational passage to show that the detector is not merely matching keywords.

APPENDIX C. IMPLEMENTATION MODULES

A. Retriever and Parser

The retriever module accepts a user query and returns a ranked list of chunks. Each result should include the source identifier, chunk identifier, retrieval score, and provenance metadata. The parser converts supported files into normalized text and records the original format and any structural anomalies. It should not execute scripts or macros embedded in source documents. Parsing failures should be reported to the policy layer, which applies the configured fallback instead of silently passing unprocessed content to the generator.

B. Detector Interface

The detector interface should accept a passage and its metadata and return a structured object containing intent probability, attack categories, risk band, and reason codes. The interface should be independent of the specific model so that a rule-based baseline can be replaced by a learned classifier without rewriting the policy gate. Input length limits and error behaviour should be explicit. The module should be tested with empty text, malformed metadata, and unexpected encodings. Confidence values should be labelled as model estimates rather than proof of malicious intent.

C. Policy Gate

The policy gate maps detector output to one of four actions: allow, sanitize, quarantine, or reject source. The mapping should be deterministic for a given configuration. Sanitization should preserve a record of the removed span's location without unnecessarily storing sensitive text. Quarantine should prevent the passage from reaching the model context, including through duplicate retrieval results. Reject-source decisions should be reversible and should include a reason code and the source identifier.

D. Context Builder and Output Validator

The context builder assembles approved passages in a consistent format and marks them as untrusted evidence. It should enforce token budgets and prevent quarantined content from being reintroduced. The output validator checks for unsupported claims, protected information, and malformed tool requests. Tool authorization should be implemented separately and should validate each action against the current user's permissions. A clear module boundary makes it easier to test whether a failure originates in retrieval, screening, generation, or authorization.

E. Audit Service

The audit service records the detector version, source hash, action, risk band, latency, and error state. By default, it should not store full documents or conversation prompts. Access to any diagnostic samples should be restricted and time-limited. The audit service should support aggregate reports showing action counts, false-positive feedback, latency distributions, and source types. It should also support export of a sanitized experiment log for academic evaluation.

APPENDIX D. RESEARCH REPORTING TEMPLATE

A. Abstract Reporting

The abstract should identify the problem, proposed method, evaluation approach, and main measured finding if experiments have been completed. If the work remains a design proposal, the abstract should say so and avoid claims of demonstrated superiority. It should not state an accuracy value unless that value was calculated from a documented test set. The keywords should describe the central concepts without overloading the abstract with implementation details.

B. Method Reporting

The methods section should describe the retrieval setup, data sources, annotation procedure, preprocessing, train-validation-test split, model configuration, threshold selection, and policy rules. Readers should be able to understand how the experiment was conducted without guessing which settings were used. If any data source cannot be shared, the paper should explain the restriction and provide enough detail to assess possible bias. All material borrowed from earlier work should be cited and paraphrased responsibly.

C. Results Reporting

Results should include precision, recall, false-positive rate, evidence retention, and latency, with category-level breakdowns where possible. Tables should include units and define abbreviations. A baseline should be reported alongside the proposed approach. If a model fails on a category, the discussion should describe the failure rather than hide it. Any confidence interval or statistical test should be explained, and small samples should be treated cautiously. Expected outcomes and hypotheses must not be presented as measured results.

D. Discussion and Limitations

The discussion should explain what the results imply for the research questions and whether the proposed design improved the balance between security and utility. It should compare observed outcomes with related research without overstating differences. The limitations should address dataset coverage, source leakage, model dependence, latency, and residual risk. Future work should follow directly from observed limitations rather than offer unrelated features. A clear limitations section strengthens the credibility of the work.

ACKNOWLEDGMENT

The authors acknowledge the guidance and support of the faculty guide and the Department of Computer Science and Engineering, Kings College of Engineering, Punalkulam, Tamil Nadu, India. The academic environment and available learning resources supported the preparation of this proposed research framework.

REFERENCES

[1] P. Lewis et al., “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks,” Advances in Neural Information Processing Systems, vol. 33, pp. 9459–9474, 2020.

[2] K. Greshake et al., “Not What You’ve Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection,” in Proc. AISec Workshop, 2023.

[3] OWASP Foundation, “LLM01: Prompt Injection,” OWASP Top 10 for Large Language Model Applications, 2025.

[4] T. Brown et al., “Language Models are Few-Shot Learners,” Advances in Neural Information Processing Systems, vol. 33, pp. 1877–1901, 2020.

[5] S. Yao et al., “ReAct: Synergizing Reasoning and Acting in Language Models,” in Proc. International Conference on Learning Representations, 2023.

[6] L. Ouyang et al., “Training Language Models to Follow Instructions with Human Feedback,” Advances in Neural Information Processing Systems, vol. 35, 2022.

[7] A. Vaswani et al., “Attention Is All You Need,” Advances in Neural Information Processing Systems, vol. 30, 2017.

[8] T. Wolf et al., “Transformers: State-of-the-Art Natural Language Processing,” in EMNLP: System Demonstrations, pp. 38–45, 2020.

[9] M. Mitchell et al., “Model Cards for Model Reporting,” in Proc. Conference on Fairness, Accountability, and Transparency, pp. 220–229, 2019.

[10] T. Gebru et al., “Datasheets for Datasets,” Communications of the ACM, vol. 64, no. 12, pp. 86–92, 2021.

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

[12] M. Ribeiro et al., “Beyond Accuracy: Behavioral Testing of NLP Models with CheckList,” in Proc. ACL, pp. 4902–4912, 2020.

[13] A. Zou et al., “Universal and Transferable Adversarial Attacks on Aligned Language Models,” arXiv:2307.15043, 2023.

[14] ISO/IEC, “Information Security, Cybersecurity and Privacy Protection—Information Security Management Systems—Requirements,” ISO/IEC 27001:2022, 2022.