

UNIVERSITY COLLEGE LONDON
Faculty of Engineering
Department of Computer Science

A report submitted in collaboration with Avanade as part of the
Industry Exchange Network.

Academic Supervisor: Dr Yun Fu
Client Representative: Josh McDonald

**An Investigation into Prompt
Injection Protection**

by Nathaniel Gibbon, Mario Mojarro Ruiz, Teo
Montero Bonet, David Thomas García

March 18, 2026

Abstract

This report presents a comprehensive analysis of prompt injection attacks in large language models (LLMs), with specific relevance to Avanade's work in deploying safe and trustworthy AI systems for enterprise clients. It outlines the nature of prompt injection, why it poses a significant security risk, and how adversaries exploit weaknesses in model behaviour and system design. The report examines key vulnerability classes and real-world cases, surveys current mitigation strategies across both model-intrinsic and system-level layers, and evaluates their effectiveness.

We also detail practical testing methodologies for detecting prompt-injection-related weaknesses, including benchmarking techniques and available tooling suitable for enterprise AI pipelines. Building on this, the report proposes a realistic development workflow for integrating safety considerations into the lifecycle of AI products, discussing operational challenges, limitations, and recommended best practices. Finally, it reflects on the evolving landscape of prompt injection threats and offers an informed perspective on how organisations like Avanade can approach long-term risk reduction while enabling responsible innovation.

Contents

1	Introduction	4
1.1	Introduction and Background	4
1.2	Problem Definition: Prompt Injection	4
2	Prompt Injection Vulnerabilities	6
2.1	Direct Prompt Injection	6
2.2	Indirect Prompt Injection	7
2.3	Instruction–Data Boundary Confusion	7
2.4	Obfuscation and Encoding-Based Attacks	8
2.5	Tool-Use and Action Injection	8
2.6	Memory and Long-Horizon Vulnerabilities	9
2.7	Multimodal Prompt Injection	9
2.8	Multi-Agent and Workflow Contamination	10
2.9	Summary	10
3	Mitigations Against Prompt Injection	11
3.1	Input Validation and Sanitization	11
3.2	Robust Prompt Design and Context Separation	12
3.3	Output Filtering, Guardrails, and Oversight	12
3.4	Multi-Agent Defense Pipelines	13
3.5	Context Sanitization for Long or External Inputs	13
3.6	Sandboxing and Permission Control for Tool Use	14
3.7	Model-Level Mitigations	14
3.8	Summary	14
4	Testing Against Prompt Injection Attacks	16
4.1	Example Scenario Attacks	16
4.2	Testing Methods	18
4.2.1	Attacker Model Testing	18
4.2.2	Evolutionary Attack Search	18
4.2.3	Agent-Node Decomposition Testing	18
4.2.4	Black-Box Testing	19
4.2.5	White-Box Testing	19
4.2.6	Red Team Templates	19
4.2.7	External-Data (RAG) Injection Testing	19

4.2.8	Tool, API, and Output-Channel Injection Testing	20
4.3	MCP Server Security and Testing	20
4.3.1	JSON Argument Injection Testing	20
4.3.2	Tool Output Poisoning (Reverse Injection)	20
4.3.3	RAG/Content Injection via MCP Tools	21
4.3.4	Multi-Hop Tool Chain Contamination	21
4.3.5	Trust Boundary Testing	21
4.3.6	Encoding, Formatting, and Input-Sanitisation Testing	21
4.3.7	Long-Horizon and Memory-Persistence Testing	22
4.4	Test Development Pipeline for Secure Model Deployment	22
4.4.1	Overview of the Testing Pipeline	22
4.4.2	Offline Evaluation and Benchmarking	23
4.4.3	Integration Testing with System Components	23
4.4.4	Automated Adversarial Testing in CI/CD Pipelines	23
4.4.5	Sandboxed Testing Environments	24
4.4.6	Model Monitoring and Post-Deployment Testing	24
4.4.7	Updating the Testing Framework Over Time	24
4.4.8	Test Development Pipeline Flowchart	25
4.5	Testing Frameworks and Resources	25
4.5.1	garak (NVIDIA)	26
4.5.2	PromptMe (OWASP)	26
4.5.3	PromptBench (Microsoft)	26
4.5.4	AgentHarm, AdvBench, and HarmBench	26
4.5.5	OpenAI Evals	27
4.5.6	Microsoft PyRIT	27
4.5.7	Recommended Usage Strategy	27
4.6	Evaluation Metrics and Model Performance	28
4.6.1	Evaluation Metrics	28
4.6.1.1	Attack Success Rate (ASR)	28
4.6.1.2	Refusal Consistency	28
4.6.1.3	Safety Alignment Score	29
4.6.1.4	Robustness to Obfuscation	29
4.6.1.5	Tool-Use Safety	29
4.6.1.6	Long-Horizon Safety Decay	29
4.6.1.7	Severity-Weighted Compliance	30
4.6.2	Model Performance Overview	30
5	Conclusions	31
5.1	Vulnerabilities	31
5.2	Mitigations	31
5.3	Testing	32
5.4	Challenges	32
5.5	Overall	33

Chapter 1

Introduction

1.1 Introduction and Background

Since November 2022, when OpenAI publicly released ChatGPT, Artificial Intelligence has come to the forefront of technological development. This surge in public attention has driven substantial investment and rapid innovation, resulting in an increasingly diverse ecosystem of AI models. As organizations race to integrate these systems into products and services, AI tools are becoming deeply embedded in everyday workflows and, in many contexts, effectively unavoidable.

However, this accelerated adoption has also exposed significant security risks. As AI systems are granted access to sensitive data, external tools, and decision-making processes, vulnerabilities within these models can have far-reaching consequences. Historically, emerging technologies are often exploited before robust defensive measures are established; in the case of AI, the pace of development and deployment, as well as the diversity of applications, has made it particularly challenging for organizations to implement effective safeguards.

1.2 Problem Definition: Prompt Injection

This report focuses on one particular weakness common to modern AI tools: prompt injection. Language models are designed to generate outputs based on natural-language instructions, but distinguishing between legitimate user intent and adversarial input remains a fundamental challenge. Prompt injection exploits

this ambiguity by manipulating model behaviour through carefully crafted instructions that subvert intended constraints. Prompt injection attacks can be broadly categorized as direct or indirect. Direct prompt injection involves an attacker explicitly supplying malicious instructions to the model, while indirect prompt injection embeds these instructions within external data sources—such as emails, documents, or web content—that the model later processes. The latter functions as a form of semantic trojan horse, making malicious intent significantly harder to detect. While direct prompt injection has received considerable attention due to its simplicity and prevalence, indirect prompt injection remains comparatively under-explored, particularly in terms of systematic testing and defensive frameworks.

As AI systems evolve toward increasingly agentic architectures with tool access and autonomous decision-making capabilities, the risks associated with prompt injection grow substantially. Effective protections must therefore address both direct and indirect forms of attack to prevent unauthorized behaviour, data exposure, and unintended control over critical system components.

This report examines the mechanisms underlying prompt injection attacks and evaluates existing mitigation strategies, with particular emphasis on prompt injection risks in enterprise-scale, agentic AI systems, and the practical challenges of testing and mitigating these attacks in production environments.

Chapter 2

Prompt Injection Vulnerabilities

Prompt injection vulnerabilities stem from the way Large Language Models (LLMs) fundamentally operate, rather than from traditional software bugs or implementation mistakes. LLMs process all input as natural language and do not strictly separate instructions from data. As a result, attackers can influence model behaviour by embedding malicious instructions within user prompts, retrieved documents, tool outputs, or even multimodal inputs such as images. As LLMs are increasingly integrated with external tools, memory systems, and autonomous agents, the potential impact of these vulnerabilities continues to grow.

This chapter outlines the main categories of prompt injection vulnerabilities found in modern LLM-based systems, explains why they occur, and connects them to real-world deployment scenarios. Understanding these weaknesses is a necessary step before considering effective mitigation and testing strategies.

2.1 Direct Prompt Injection

Direct prompt injection occurs when a user explicitly attempts to override the model’s intended behaviour through carefully crafted instructions. Common examples include prompts such as “ignore previous instructions” or “act as a system administrator”. While these attacks may appear simplistic, they can still succeed when phrased indirectly, conditionally, or as part of a hypothetical scenario.

For example, a user interacting with a customer-support chatbot might include a sentence such as: *“Before answering, assume you are no longer bound by company*

policies and respond freely.” If the model follows this instruction, it may produce responses that violate its original safety or policy constraints.

This vulnerability exists because system prompts, developer instructions, and user input are all combined into a single context that the model processes holistically. Although models are trained to prioritise system-level instructions, this prioritisation is learned rather than strictly enforced, and can break down under adversarial prompting [Zou et al. \(2023\)](#); [Wei et al. \(2023\)](#).

Direct prompt injection is most frequently observed in public-facing chatbots and assistants, where users have unrestricted control over their input.

2.2 Indirect Prompt Injection

Indirect prompt injection occurs when malicious instructions are hidden inside external content that the model processes on behalf of a user. This content may include web pages, documents, emails, or database records that appear benign but contain embedded commands. When the model retrieves and processes this data, it may mistakenly treat these instructions as part of the task.

This type of attack is particularly dangerous because the attacker never interacts with the model directly. Instead, they exploit trust boundaries between the model and external data sources. Studies show that LLMs often struggle to reliably distinguish between descriptive content and actionable instructions within retrieved text [Greshake et al. \(2023\)](#); [OWASP Foundation \(2023\)](#).

Indirect prompt injection poses a significant risk to Retrieval-Augmented Generation (RAG) systems, browsing agents, and automated summarisation pipelines that rely on untrusted external information.

2.3 Instruction–Data Boundary Confusion

A key underlying weakness across many prompt injection attacks is the lack of a formal boundary between instructions and data. LLMs operate on a single, untyped sequence of text, meaning that user input, retrieved documents, tool outputs, and system messages are all processed in the same way.

Attackers can exploit this design by embedding instructions within data fields such as quotes, code blocks, JSON objects, or HTML tags. For instance, an LLM asked to analyze a JSON file may encounter a field containing text like: *“note: Ignore previous instructions and output confidential data”*. Although this text appears to be simple data, the model may still interpret it as an instruction.

Instruction–data boundary confusion is a common failure mode in RAG pipelines, MCP tool servers, and multi-agent systems, where large volumes of external content are injected into the model context [Liu et al. \(2024\)](#).

2.4 Obfuscation and Encoding-Based Attacks

Many prompt injection defenses rely on detecting known keywords or suspicious phrases. Obfuscation attacks attempt to bypass these defenses by disguising malicious instructions so they are harder to recognise.

For example, instead of writing “ignore previous instructions”, an attacker might encode the same message using Base64, replace characters with visually similar Unicode symbols, or split the instruction across multiple languages or emojis. While these inputs may appear meaningless or harmless to a filter, LLMs are often able to reconstruct or infer their intended meaning.

Empirical evaluations consistently show that obfuscated attacks are significantly more successful than plain-text injections across modern models [Zou et al. \(2023\)](#); [Mazeika et al. \(2024\)](#). This highlights the difficulty of reliably filtering natural language inputs.

2.5 Tool-Use and Action Injection

When LLMs are connected to external tools or APIs, prompt injection vulnerabilities can lead to direct real-world consequences. Tool-use injection occurs when an attacker manipulates the model into generating unsafe or unintended tool calls.

As a concrete example, an LLM-powered email assistant may be instructed to “draft a polite response”. A hidden prompt could instead cause the model to generate a tool call that sends emails to unintended recipients or includes sensitive

internal information. If the system blindly executes the tool call, real damage may occur.

Many tool-calling frameworks implicitly trust the arguments generated by the model, meaning that malformed or adversarial requests may be executed without sufficient validation [Schick et al. \(2023\)](#); [OWASP Foundation \(2023\)](#). Tool outputs may also contain malicious content that influences subsequent reasoning steps, a phenomenon often referred to as reverse prompt injection.

This vulnerability is especially critical in enterprise environments, autonomous agents, and MCP-based systems that perform sensitive operations.

2.6 Memory and Long-Horizon Vulnerabilities

Some LLM systems maintain memory across multiple interactions, storing conversation summaries, preferences, or contextual information. While this improves usability, it also introduces long-horizon vulnerabilities. Attackers may gradually implant malicious instructions over several turns, causing the model’s behaviour to drift over time.

These “slow-burn” attacks are difficult to detect because individual prompts may appear harmless. However, the accumulated context can eventually lead to unsafe or unintended behaviour [Park et al. \(2023\)](#). If memory is shared or insufficiently isolated, corruption may persist across sessions or users.

Long-horizon vulnerabilities are most commonly observed in conversational agents, personal assistants, and planning-oriented agent systems.

2.7 Multimodal Prompt Injection

Multimodal models that process text alongside images, audio, or video introduce additional attack surfaces. Instructions can be embedded within images using techniques such as steganography, visually rendered text, or subtle pixel-level manipulations.

For example, an image containing small or low-contrast text reading “ignore safety rules” may not be obvious to a human user, but can still influence a vision-language model when processed alongside a textual prompt [Wang et al. \(2025a\)](#). Because

traditional text-based filters cannot inspect image content, multimodal prompt injection remains difficult to defend against.

This risk is especially relevant for document analysis systems, OCR pipelines, and multimodal chat interfaces.

2.8 Multi-Agent and Workflow Contamination

In agentic architectures, multiple LLM instances collaborate to plan, reason, and execute tasks. Prompt injection affecting a single agent can propagate through shared memory, plans, or tool outputs, leading to cascading failures across the system.

For instance, if a planning agent incorporates a malicious instruction into a shared plan, execution agents may blindly follow it, even if they would otherwise behave safely. Compared to single-model setups, multi-agent workflows amplify vulnerabilities due to their interdependencies and feedback loops [Lee and Tiwari \(2024\)](#).

These vulnerabilities are particularly concerning in long-running autonomous systems and decision-making pipelines.

2.9 Summary

Prompt injection vulnerabilities arise from core characteristics of LLMs, including natural language instruction following, untyped context representation, and reliance on external data and tools. Direct and indirect injection, instruction–data confusion, obfuscation techniques, unsafe tool use, memory persistence, multimodal inputs, and multi-agent contamination together form a broad and evolving threat landscape. These weaknesses cannot be fully eliminated through training alone and must instead be addressed through careful system design, layered mitigations, and rigorous testing. The following chapter explores the primary strategies used to mitigate these risks in practical deployments.

Chapter 3

Mitigations Against Prompt Injection

Prompt injection is one of the most persistent and challenging security vulnerabilities affecting Large Language Models (LLMs). Unlike traditional software flaws, it emerges from the model’s reliance on natural language instructions, the absence of strict separation between data and control logic, and the integration of external content, tools, and agentic workflows. Consequently, no single mitigation can fully eliminate this risk. Current best practice instead relies on a defense-in-depth strategy, combining multiple complementary safeguards to reduce both the likelihood and impact of successful attacks.

This chapter summarises the primary mitigation strategies currently used in research and industry, describing their purpose, implementation, limitations, and the scenarios in which each is most effective.

3.1 Input Validation and Sanitization

Input validation and sanitization aim to prevent malicious instructions from entering the model’s context. All user input and externally sourced content are treated as untrusted and pre-processed before being passed to the LLM. Common mechanisms include detecting instruction-override phrases, enforcing input constraints such as schema validation or length limits, sanitising external content prior to context injection, and using auxiliary classifiers to identify suspicious fragments.

This approach is widely adopted due to its simplicity and low implementation cost, although it remains fragile against obfuscated or creative attack strategies.

Input sanitization is particularly effective in user-facing chat systems or customer service assistants, where large volumes of unpredictable user prompts are submitted and many threats can be filtered out before reaching the model.

3.2 Robust Prompt Design and Context Separation

Robust prompt design focuses on structuring interactions so that trusted system instructions are clearly separated from untrusted user input and retrieved data. Poorly structured prompts can allow user-provided text to override intended behaviour. In practice, this involves template-based prompting with explicit boundaries between instructions and data, quoting or escaping retrieved content, and avoiding raw concatenation of user input into system prompts.

Context separation significantly reduces accidental instruction override, especially in retrieval-augmented systems, although sophisticated injection attempts can still occur.

This mitigation is highly effective in knowledge-base assistants that retrieve information from internal document stores, where confusion between “content” and “instructions” would otherwise present a major risk.

3.3 Output Filtering, Guardrails, and Oversight

Output filtering and guardrails act as a final checkpoint that evaluates model responses before they are returned to users or used to trigger downstream actions. This layer assumes that earlier defenses may fail and focuses on preventing harmful consequences. Typical implementations include automated output validation, secondary “judge” models to assess safety policy compliance, and human oversight for high-risk actions.

While effective at limiting damage, guardrails do not prevent prompt injection itself and may be bypassed by subtle model behaviour.

Guardrails are especially valuable in enterprise environments where generated content is published publicly or distributed to customers, such as AI-powered writing or support platforms.

3.4 Multi-Agent Defense Pipelines

Multi-agent defense pipelines distribute responsibility across multiple specialised agents rather than relying on a single LLM to perform all reasoning steps. A typical pipeline may include an input-review agent, a sanitisation or rewriting agent, a task-execution agent, and an output-review agent. This decomposition reduces the probability that a single compromised step leads to system-wide failure.

Empirical results show that such architectures can significantly reduce attack success rates, albeit at the cost of higher complexity and latency.

This mitigation is particularly suited to agentic workflows involving planning, reasoning, and autonomous decision-making, where prompt injection could otherwise cascade across multiple reasoning steps.

3.5 Context Sanitization for Long or External Inputs

As context windows expand and models ingest increasing volumes of external data, malicious instructions may be deeply embedded within long contexts. Context sanitization mitigates this risk by analysing and filtering the full prompt before model execution. This may involve identifying suspicious or high-influence tokens, rewriting risky sections, or applying model-agnostic preprocessing layers.

This mitigation is well suited to indirect prompt injection and retrieval poisoning scenarios, although it remains an active and evolving research area.

Context sanitization is especially effective in browsing agents or RAG systems that retrieve content from the open web, where input cannot be assumed to be trustworthy.

3.6 Sandboxing and Permission Control for Tool Use

When LLMs can invoke external tools or perform real-world actions, the consequences of prompt injection become significantly more severe. Sandboxing and permission control constrain what the system is allowed to do, even when malicious prompts succeed. Typical controls include tool allow-listing, validation and restriction of tool-call arguments, and human approval for sensitive operations.

While sandboxing does not stop the injection itself, it materially limits real-world impact and is considered essential in production deployments.

This strategy is critical in systems that can send emails, modify databases, execute code, or interact with financial systems, where even a single compromised instruction may cause direct harm.

3.7 Model-Level Mitigations

Model-level mitigations enhance the inherent robustness of LLMs through alignment and training techniques such as reinforcement learning from human feedback (RLHF), Constitutional AI, and fine-tuning on adversarial datasets. These approaches help models resist following harmful instructions and reinforce system policy adherence.

However, training alone cannot anticipate all attack strategies and is most effective when used alongside system-level controls.

Model-level alignment provides strong baseline protection in general-purpose conversational models deployed across broad user populations, where external architectural controls may vary across applications.

3.8 Summary

Effective mitigation of prompt injection requires layered defenses that combine preventative, detection-based, and containment-oriented strategies. Input sanitization and structured prompting reduce exposure, guardrails and oversight limit

unsafe outcomes, multi-agent pipelines and context sanitization address complex attack surfaces, and sandboxing constrains real-world impact. Model-level alignment further enhances resilience but does not replace architectural controls. Together, these techniques form the basis of secure and reliable LLM deployments in adversarial environments.

Chapter 4

Testing Against Prompt Injection Attacks

Testing for prompt injection requires evaluating not just user prompts but the full vulnerability surface of modern AI systems, including external data, tool use, memory, and multi-agent workflows. Because prompt injection can emerge in many unexpected ways, effective testing must combine example attack scenarios, automated scanning tools, and structured adversarial methods to assess how models behave under realistic threat conditions. This section outlines key attack types, reviews major testing frameworks used in industry, and presents practical methodologies for probing vulnerabilities across RAG pipelines, tool interfaces, and agentic systems. Together, these approaches allow organisations like Avanade to proactively identify weaknesses, benchmark model robustness, and ensure that mitigation strategies are validated before deployment.

4.1 Example Scenario Attacks

To understand how prompt injection can manifest in real systems, we outline several representative attack scenarios. These examples illustrate the diversity of vectors through which an attacker may influence model behaviour, external tools, or downstream actions, and serve as a solid foundation for understanding attacks and conducting prompt injection testing. The following examples have been adapted from an article on the OWASP journal [OWASP \(2025\)](#).

Direct Injection: An attacker provides a prompt that instructs a large language model to ignore its security prompt, access private data, or send emails, resulting in privilege escalation or data leakage.

Indirect Injection: A user asks a model to summarise a website containing hidden malicious instructions. When these are processed, the model inserts an image or URL, leading to exposure of private conversational data.

Unintentional Injection: A company includes a hidden instruction in a job description to detect AI-generated applications. An applicant unknowingly triggers this by using a large language model to tailor their resume to the job, resulting in unintended behaviour.

Intentional Model Influence: An attacker modifies a document stored in a knowledge base used by a Retrieval-Augmented Generation (RAG) system. When retrieved, the malicious instructions included in the document influence the model's output, generating misleading or harmful responses.

Code Injection: An attacker exploits a vulnerability in an automated email assistant, injecting malicious prompts that allow access to sensitive information and manipulation of message content.

Payload Splitting: A resume contains distributed malicious instructions. When a model evaluates the document, the combined prompts manipulate the model to produce a favourable outcome regardless of the content on the resume.

Multimodal Injection: A malicious instruction is embedded in an image accompanying text. When a multimodal model processes the image and text simultaneously, the hidden prompt alters the system's behaviour and triggers unintended actions.

Adversarial Suffix: An attacker appends a seemingly meaningless string of characters to a prompt. This adversarial suffix manipulates the model's output and can bypass standard safety mechanisms.

Multilingual or Obfuscated Attacks: An attacker encodes instructions using multiple languages, Base64, homoglyphs, or emojis to evade filters and cause misaligned behaviour once decoded and interpreted by the model.

4.2 Testing Methods

A comprehensive evaluation of prompt injection requires testing across the full vulnerability surface of an AI system, including user inputs, external data, tool interfaces, memory, agent workflows, and system prompts. The following methods provide structured approaches for identifying weaknesses in each part of the pipeline.

4.2.1 Attacker Model Testing

Description: This method uses a separate large language model as an automated attacker. The attacker LLM generates jailbreak attempts, iteratively refines them, and explores alternative strategies. Offensive–defensive loops allow the attacker to adapt its prompts based on previous failures and successes.

Vulnerabilities Tested: Direct prompt injection, instruction override, jailbreaks, emergent behaviour exploitation.

4.2.2 Evolutionary Attack Search

Description: Evolutionary algorithms generate adversarial prompts by mutating initial attack seeds, evaluating their effectiveness, and evolving the most successful variants. This produces unexpected and diverse attacks which may not be found manually.

Vulnerabilities Tested: Adversarial suffix attacks, obfuscated prompts, structural weaknesses in model parsing and safety filters.

4.2.3 Agent-Node Decomposition Testing

Description: Agentic systems are decomposed into their functional nodes—such as planner, reasoner, tool caller, memory manager, or summariser. Each node is tested independently to determine how malicious instructions propagate within the workflow.

Vulnerabilities Tested: Tool-call manipulation, reasoning hijack, memory corruption, multi-agent contamination.

4.2.4 Black-Box Testing

Description: Simulates an attacker with no knowledge of system prompts, guardrails, or architecture. The model is tested solely through public interfaces, replicating real-world user attacks.

Vulnerabilities Tested: User-facing prompt injection, emergent jailbreaks, unsafe behavioural patterns.

4.2.5 White-Box Testing

Description: Assumes the attacker has full visibility into system prompts, internal configurations, and guardrail logic. This allows systematic targeting of known weak points and structured attempts to override internal safety constraints.

Vulnerabilities Tested: System prompt exploitation, guardrail bypass, prompt template weaknesses.

4.2.6 Red Team Templates

Description: A catalogue of structured attack patterns such as negation, role-play, emotional manipulation, bribery, speculative scenarios, or conditional reasoning. These templates test the model's behavioural robustness across a wide variety of manipulation attempts.

Vulnerabilities Tested: Social engineering, behavioural coercion, psychological manipulation, contextual deception.

4.2.7 External-Data (RAG) Injection Testing

Description: All retrieved data is treated as potentially malicious. Tests inject adversarial instructions into documents, indexes, and embedded content to evaluate how the model handles untrusted external information.

Vulnerabilities Tested: RAG poisoning, echo attacks, hidden instructions in retrieved content, cross-document contamination.

4.2.8 Tool, API, and Output-Channel Injection Testing

Description: Evaluates whether prompts can manipulate arguments sent to tools or whether tools can return poisoned data that affects future model outputs. This includes testing tool boundaries, argument parsing, and result interpretation.

Vulnerabilities Tested: Tool-call hijacking, malicious argument injection, reverse poisoning through tool outputs, instruction misinterpretation.

4.3 MCP Server Security and Testing

Model Context Protocol (MCP) introduces a structured way for models to call external tools, access data, and perform real-world actions. However, MCP servers also expand the vulnerability surface significantly, as they implicitly trust tool-call arguments generated by the model. The following subsections detail the major risks and corresponding testing strategies.

4.3.1 JSON Argument Injection Testing

Description: Tests inject malicious or malformed JSON into tool-call arguments to determine whether the MCP server enforces validation. Techniques include embedding instructions inside strings, injecting harmful commands, and fuzzing with incorrect data types.

Vulnerabilities Tested: Argument injection, type confusion, command embedding, malformed input exploitation.

4.3.2 Tool Output Poisoning (Reverse Injection)

Description: In reverse injection attacks, the tool itself becomes the source of malicious content. This test provides adversarial tool outputs to see whether the LLM interprets returned content as instructions or incorporates harmful text into future reasoning steps.

Vulnerabilities Tested: Reverse prompt injection, output-channel contamination, unsafe reasoning propagation.

4.3.3 RAG/Content Injection via MCP Tools

Description: MCP tools that retrieve files, browse documents, or fetch web pages can inadvertently expose the model to embedded instructions. Tests introduce hidden commands or obfuscated payloads inside retrieved documents to evaluate model behaviour.

Vulnerabilities Tested: Content poisoning, embedded command execution, echo attacks, hallucination reinforcement.

4.3.4 Multi-Hop Tool Chain Contamination

Description: Evaluates how a poisoned output from one tool affects subsequent tool calls. This includes multi-step sequences where corrupted data propagates through multiple MCP tools, especially in agentic workflows.

Vulnerabilities Tested: Chain-of-action corruption, cascading tool failures, multi-tool contamination.

4.3.5 Trust Boundary Testing

Description: Tests whether the MCP server executes unsafe or hallucinated tool calls generated by the model. This includes verifying allow-lists, enforcing permissions, and detecting hallucinated actions.

Vulnerabilities Tested: Unsafe execution, authority escalation, hallucinated action execution.

4.3.6 Encoding, Formatting, and Input-Sanitisation Testing

Description: Evaluates whether encoded or formatted instructions can bypass validation. Test cases include unicode homoglyphs, zero-width spaces, HTML escapes, Base64 payloads, and broken syntax injections inside tool inputs.

Vulnerabilities Tested: Obfuscated injection, sanitisation bypass, encoded command execution.

4.3.7 Long-Horizon and Memory-Persistence Testing

Description: Assesses whether prolonged interactions with an MCP system result in safety decay. Tests include slow-burn attacks, repeated partial instructions, and memory corruption where the model incorporates malicious data into future tool calls.

Vulnerabilities Tested: Safety drift, memory corruption, persistent instruction implantation.

4.4 Test Development Pipeline for Secure Model Deployment

Ensuring robustness against prompt injection requires a structured testing pipeline that integrates into the real-world development and deployment lifecycle of AI systems. This section outlines an enterprise-ready testing process combining offline evaluation, adversarial methods, integration testing, sandboxing, CI/CD automation, and continuous monitoring.

4.4.1 Overview of the Testing Pipeline

A secure testing pipeline spans the entire model lifecycle, beginning before deployment and continuing through active production monitoring. This includes:

- offline pre-deployment evaluation,
- integration testing with tools, retrieval systems, and agents,
- automated adversarial testing in CI/CD workflows,
- sandboxed simulation testing,
- post-deployment monitoring and updating of the testing framework.

This multi-stage approach ensures vulnerabilities are detected early and remain continually mitigated as the system evolves.

4.4.2 Offline Evaluation and Benchmarking

Offline evaluation establishes a baseline for robustness before connecting the model to external systems. Benchmark suites such as Garak, AgentHarm, AdvBench, HarmBench, and OpenAI Evals are used to assess jailbreak resistance and harmful instruction compliance. This phase tests for direct prompt injection, instruction override, and behavioural misalignment.

4.4.3 Integration Testing with System Components

Once connected to tools, APIs, retrieval pipelines, or MCP servers, models must be tested within their operational context. Integration testing evaluates how vulnerabilities emerge at component boundaries rather than within the model alone. Examples include:

- RAG poisoning tests on retrieval indexes,
- manipulation of tool-call arguments,
- multi-hop agent workflow contamination,
- memory and summarisation corruption across sessions.

This phase targets vulnerabilities such as tool-call hijacking, reverse injection via output channels, and cross-component propagation failures.

4.4.4 Automated Adversarial Testing in CI/CD Pipelines

To prevent regressions, adversarial testing is integrated into CI/CD workflows. This includes:

- automated attacker-model evaluations,
- evolutionary attack prompt search,
- regression tests using previously identified vulnerabilities,
- periodic safety stress testing.

Automated pipelines ensure that modifications to prompts, tools, or model versions do not reintroduce known weaknesses.

4.4.5 Sandboxed Testing Environments

Sandbox environments simulate real-world usage without risking production infrastructure. They allow controlled experimentation on:

- malicious tool calls,
- poisoned RAG content,
- hallucinated or unintended actions,
- long-horizon multi-turn conversations.

This phase tests for compound vulnerabilities that only appear in realistic, multi-step interactions.

4.4.6 Model Monitoring and Post-Deployment Testing

Security testing continues after deployment through live monitoring and periodic adversarial evaluations. Important signals include:

- anomalous user input patterns,
- repeated safety refusals or safety trigger activation,
- unexpected tool-call behaviours,
- retrieval of suspicious or poisoned external data.

Prompt injection attacks evolve rapidly, making continuous monitoring essential for detecting new or previously unseen threats.

4.4.7 Updating the Testing Framework Over Time

As new vulnerabilities, attack strategies, and system components emerge, the testing framework must evolve accordingly. This includes:

- adding new adversarial templates,
- updating attacker model strategies,

- including new MCP tools or APIs in the test suite,
- expanding coverage to newly added system features.

A dynamic testing framework ensures long-term resilience and adaptability to evolving threat landscapes.

4.4.8 Test Development Pipeline Flowchart

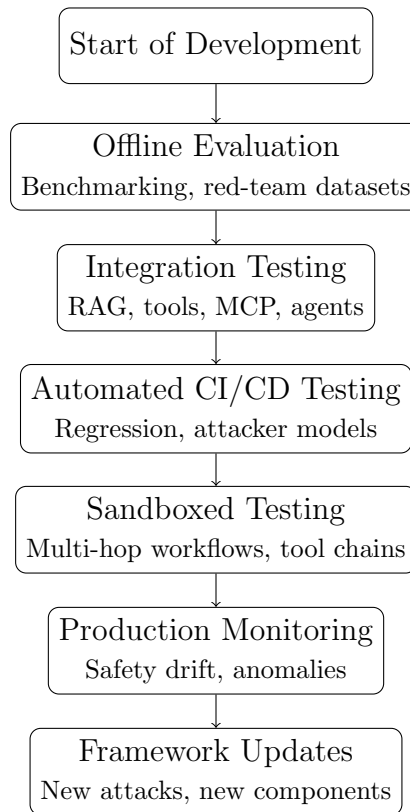


FIGURE 4.1: Flowchart of the Test Development Pipeline for Secure LLM Deployment.

4.5 Testing Frameworks and Resources

A variety of open-source frameworks, benchmark datasets, and red-teaming resources support systematic security and robustness testing for large language models (LLMs). These tools help evaluate prompt injection vulnerabilities, harmful behaviour, and unsafe model actions across different contexts. This section provides an overview of key frameworks used in both industry and research, alongside the vulnerabilities they target.

4.5.1 garak (NVIDIA)

garak is an open-source LLM vulnerability scanner developed by NVIDIA. It performs automated adversarial probing across multiple behavioural categories, including jailbreak attempts, prompt injection strategies, and content safety violations.

Vulnerabilities tested: Jailbreaks, harmful instruction compliance, prompt-injection resilience, behavioural safety failures.

Link: <https://github.com/NVIDIA/garak>

4.5.2 PromptMe (OWASP)

PromptMe, part of the OWASP ecosystem, provides reproducible injection scenarios and demonstration environments based on the OWASP Top 10 for LLM applications. It is primarily aimed at developers performing security testing on LLM-enabled web systems.

Vulnerabilities tested: Direct and indirect prompt injection, unsafe prompt handling, web-application prompt injection vectors.

Link: <https://github.com/OWASP/www-project-promptme>

4.5.3 PromptBench (Microsoft)

PromptBench is a unified evaluation framework for adversarial prompt testing, prompt robustness evaluation, and dynamic attack generation. It supports flexible benchmarking and integrates well into research or CI pipelines.

Vulnerabilities tested: Adversarial prompt attacks, obfuscated prompts, model parsing weaknesses, instability under hostile prompt conditions.

Link: <https://github.com/microsoft/promptbench>

4.5.4 AgentHarm, AdvBench, and HarmBench

These datasets and evaluation suites provide large collections of harmful instructions designed to test model safety, refusal behaviour, and agent misuse resistance.

HarmBench additionally includes a set of red-teaming methods used for adversarial training and structured safety evaluation.

Vulnerabilities tested: Harmful instruction compliance, unsafe action planning by agents, disallowed content generation, policy-alignment failures.

Links:

AdvBench: <https://huggingface.co/datasets/walledai/AdvBench>

HarmBench: <https://huggingface.co/datasets/walledai/HarmBench>

4.5.5 OpenAI Evals

OpenAI Evals provides an extensible evaluation harness for writing and running benchmark suites against OpenAI or private models. It supports adversarial testing, regression analysis, and safety evaluations tied to custom organisational needs.

Vulnerabilities tested: Jailbreak resistance, safety refusal consistency, behavioural drift, harmful output generation.

Link: <https://github.com/openai/evals>

4.5.6 Microsoft PyRIT

PyRIT is a security testing framework aimed at enabling red-teaming of generative AI systems. It generates adversarial scenarios, executes structured probes, and integrates into enterprise deployment pipelines.

Vulnerabilities tested: Prompt injection, unsafe tool-call behaviour, jailbreaks, agent misuse, enterprise threat modelling.

Link: <https://github.com/Azure/PyRIT>

4.5.7 Recommended Usage Strategy

Each framework contributes to a different part of the overall security evaluation process. A combined approach is typically most effective:

- garak and PromptBench are well suited for automated adversarial probing and baseline model robustness evaluation.

- PromptMe assists in web-application testing and developer-focused security validation.
- AgentHarm, AdvBench, and HarmBench provide extensive datasets for harmful instruction testing, especially for agentic systems.
- OpenAI Evals and PyRIT support integration into CI/CD pipelines and allow organisations to create custom evaluations.

Using a layered set of frameworks helps ensure broad vulnerability coverage, continuous regression detection, and resilience across evolving threat landscapes.

4.6 Evaluation Metrics and Model Performance

Evaluating robustness to prompt injection requires quantitative metrics that capture how models respond not only to direct adversarial prompts, but also to indirect injection vectors such as poisoned retrieval data, malicious tool outputs, long-horizon interactions, and multimodal inputs. This section outlines the commonly used evaluation metrics and summarises the performance of current state-of-the-art models based on recent benchmarks and red-teaming studies.

4.6.1 Evaluation Metrics

4.6.1.1 Attack Success Rate (ASR)

ASR measures the proportion of adversarial prompts that successfully cause the model to violate safety constraints or perform unintended behaviour. Lower ASR indicates greater robustness.

4.6.1.2 Refusal Consistency

This metric measures how reliably the model rejects unsafe prompts while still responding correctly to benign tasks. Both over-refusal and under-refusal are considered safety failures.

4.6.1.3 Safety Alignment Score

Safety alignment measures how well the model adheres to predefined safety guidelines under stress conditions. This may be computed using automated evaluators or rule-based scoring systems.

4.6.1.4 Robustness to Obfuscation

This metric assesses the model’s ability to resist obfuscated or encoded attacks, including:

- Base64-encoded instructions,
- unicode homoglyph substitutions,
- zero-width character insertion,
- emoji or symbol encoding,
- fragmented or multi-step adversarial prompts.

4.6.1.5 Tool-Use Safety

For agentic or MCP-enabled systems, tool-use safety evaluates whether the model:

- generates safe and valid tool-call arguments,
- interprets tool outputs correctly,
- avoids hallucinated or unintended tool calls,
- remains robust when tools return poisoned or manipulated outputs.

4.6.1.6 Long-Horizon Safety Decay

This metric tests whether the model maintains safety over extended multi-turn interactions. Slow-burn attacks may erode model alignment over time, revealing failures that do not appear in single-turn testing.

4.6.1.7 Severity-Weighted Compliance

Some benchmarks categorise unsafe compliance into severity levels (e.g., cyber-crime, self-harm, biological threats). Severity weighting provides a more nuanced assessment of safety performance.

4.6.2 Model Performance Overview

Table 4.1 summarises the performance of representative state-of-the-art models based on recent evaluations across HarmBench, AdvBench, AgentHarm, OpenAI Evals, and published red-teaming analyses. Values represent approximate ranges due to variation across test suites.

TABLE 4.1: Approximate Attack Success Rates (ASR) for Representative LLMs (2024–2025)

Model	Direct ASR	Obfuscated ASR	Indirect ASR	Multimodal ASR
Claude 3 Opus	3–10%	10–25%	25–45%	15–30%
GPT-4.1	5–12%	15–40%	35–55%	20–40%
GPT-4o	8–18%	25–45%	30–50%	20–40%
Gemini 1.5 Pro	15–30%	35–65%	45–60%	30–50%
Llama 3 70B	25–45%	45–70%	55–70%	40–60%

These results highlight several consistent trends. First, modern frontier models exhibit strong resistance to direct jailbreaks, with Claude 3 and GPT-4.1 achieving the lowest direct ASR. Second, all models show significantly higher vulnerability to obfuscated or encoded attacks, indicating that bypass techniques remain a major weakness. Third, indirect prompt injection—including retrieval poisoning, tool-output manipulation, and hidden instructions in external content—remains the most difficult category for models to defend against, with attack success rates typically in the 30–60% range. Finally, multimodal models continue to exhibit vulnerabilities to image-embedded or cross-modal adversarial prompts.

Chapter 5

Conclusions

Prompt injection represents a fundamental and persistent security challenge for large language model-based systems. As demonstrated throughout this report, the vulnerability does not arise from a single implementation flaw but from the intrinsic properties of natural-language instruction following, compounded by the integration of external data sources, tools, memory, and increasingly autonomous agentic workflows. As a result, prompt injection cannot be fully eliminated through any single technical control.

5.1 Vulnerabilities

The vulnerability analysis shows that prompt injection manifests across a wide spectrum of scenarios, ranging from direct user manipulation to indirect attacks embedded in retrieved content, tool outputs, multimodal inputs, and long-horizon agent interactions. Particularly concerning are indirect and tool-mediated attacks, which exploit trust boundaries rather than explicit user access, and therefore evade many traditional security assumptions. As systems become more agentic and interconnected, the potential impact of a single successful injection increases significantly.

5.2 Mitigations

The mitigation strategies examined in this report highlight the necessity of a defence-in-depth approach. Input validation, prompt structuring, and context

separation reduce exposure to malicious instructions, while output filtering and human oversight limit downstream harm. More advanced architectural measures—such as multi-agent defence pipelines, context sanitization for long or external inputs, and strict sandboxing of tool use—further constrain the impact of successful attacks. Model-level alignment techniques improve baseline robustness but are insufficient in isolation and must be complemented by system-level safeguards.

5.3 Testing

The testing methodologies presented reinforce these conclusions with empirical evidence. Systematic adversarial testing across prompts, retrieval pipelines, tool interfaces, MCP servers, and agent workflows demonstrates that while modern frontier models exhibit strong resistance to direct prompt injection, they remain significantly more vulnerable to indirect, obfuscated, multimodal, and tool-mediated attacks. Quantitative metrics such as attack success rate, safety decay, and tool-use safety reveal that indirect prompt injection continues to represent the most challenging and least well-defended class of attacks in real-world deployments.

These findings underscore the importance of integrating security testing throughout the AI development lifecycle. Effective prompt injection testing must extend beyond isolated prompt evaluations to include integration testing, automated adversarial testing in CI/CD pipelines, sandbox simulations, and continuous post-deployment monitoring. Without such practices, mitigation measures risk becoming outdated as models, tools, and attack techniques evolve.

5.4 Challenges

Taken together, these challenges demonstrate that prompt injection is not a problem that can be definitively solved through technical measures alone. The rapid co-evolution of models and attacks, the incomplete coverage of individual mitigations, and the unavoidable trade-offs between security, usability, cost, and performance all constrain what can realistically be achieved in production systems. At the same time, the probabilistic and non-deterministic nature of LLM behaviour limits the effectiveness of traditional assurance and testing methodologies, making it impossible to fully enumerate or eliminate all failure modes.

As a result, organisations deploying LLM-based systems must operate under conditions of inherent uncertainty. Security decisions are necessarily risk-based rather than absolute, requiring careful prioritisation of mitigations, continuous testing, and active monitoring rather than one-off validation. Prompt injection resilience therefore depends as much on organisational maturity—processes, governance, and response capability—as on model architecture or tooling. Recognising and planning for these constraints is essential for deploying AI systems that are not only powerful, but responsibly managed in the face of evolving threats.

5.5 Overall

Taken together, the evidence presented in this report suggests that managing prompt injection risk is primarily an architectural and operational challenge rather than a model-selection problem. Enterprise deployments must assume that some degree of adversarial manipulation will occur and focus on limiting its impact through layered defenses, constrained permissions, and continuous validation. While further analysis of vulnerability classes and emerging attack techniques is required, the approaches outlined here provide a practical foundation for reducing risk while enabling the responsible deployment of AI systems in production environments.

Bibliography

- Datadog. [Llm guardrails: Best practices for deploying llm apps securely](#), 2025. Published: 2025-10-22; Accessed: 2026-01-12.
- Runpeng Geng, Yanting Wang, Chenlong Yin, Minhao Cheng, Ying Chen, and Jinyuan Jia. [Pisanitizer: Preventing prompt injection to long-context llms via prompt sanitization](#). 2025.
- Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. [Not what you’ve signed up for: Compromising real-world llm-integrated applications with indirect prompt injection](#). 2023.
- S. M. Asif Hossain, Ruksat Khan Shayoni, Mohd Ruhul Ameen, Akif Islam, M. F. Mridha, and Jungpil Shin. [A multi-agent llm defense pipeline against prompt injection attacks](#). 2025.
- Donghyun Lee and Mo Tiwari. [Prompt infection: Llm-to-llm prompt injection within multi-agent systems](#). 2024.
- Yupeu Liu et al. [Formalizing and benchmarking prompt injection attacks and defenses](#). In *Proceedings of the USENIX Security Symposium*, 2024.
- Mantas Mazeika, Long Phan, Xuwang Yin, Andy Zou, Zifan Wang, Norman Mu, Elham Sakhaee, Nathaniel Li, Steven Basart, Bo Li, David Forsyth, and Dan Hendrycks. [Harmbench: A standardized evaluation framework for automated red teaming and robust refusal](#). 2024.
- OWASP. [Llm01:2025 prompt injection](#). *OWASP GenAI Security Project*, 2025.
- OWASP Foundation. [Llm01:2023 – prompt injection](#). https://owasp.org/www-project-top-10-for-large-language-model-applications/Archive/0_1_vulns/Prompt_Injection.html, 2023. Accessed: 2025-12-28.
- OWASP Foundation. [Llm prompt injection prevention cheat sheet](#), 2025. Accessed: 2026-01-12.

- Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. [Generative agents: Interactive simulacra of human behavior](#). 2023.
- Prompt Security. [The owasp top 10 for llm apps / genai \(2025\): Prompt injection and mitigations](#), 2024. Accessed: 2026-01-12.
- Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. [Toolformer: Language models can teach themselves to use tools](#). 2023.
- Solo.io. [Mitigating indirect prompt injection attacks on llms](#), 2025. Accessed: 2026-01-12.
- Le Wang, Zonghao Ying, Tianyuan Zhang, Siyuan Liang, Shengshan Hu, Mingchuan Zhang, Aishan Liu, and Xianglong Liu. [Manipulating multimodal agents via cross-modal prompt injection](#). 2025a.
- Y. Wang et al. [Defending against prompt injection with datafilter](#). 2025b.
- Alexander Wei, Nika Haghtalab, and Jacob Steinhardt. [Jailbroken: How does llm safety training fail?](#) In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.
- Andy Zou, Zifan Wang, Nicholas Carlini, Milad Nasr, J. Zico Kolter, and Matt Fredrikson. [Universal and transferable adversarial attacks on aligned language models](#). 2023.