Evaluating LLMs for Real-World Web Vulnerability Detection

Sebastian Neef✉[1][0000-0003-3055-0823],
Luca Jungnickel✉[1][0009-0001-3565-0105],
Antonio Benjamin Buchholz[1][0009-0006-7122-7351],
Valene Spence[2][0009-0008-5471-4697], and
Vicente Birke Gonzalez[2][0009-0007-6122-5376]

[1] Technische Universität Berlin, Einsteinufer 17, 10587 Berlin, Germany
[2] Freie Universität Berlin, Kaiserswerther Str. 16-18, 14195 Berlin, Germany

Abstract. Large Language Models (LLMs) have emerged as a promising tool for automated vulnerability detection, yet their effectiveness on web-specific vulnerabilities remains to be explored.
This work benchmarks six frontier (Claude Opus 4.6, Codex GPT-5.4, Gemini 3.1-pro-preview) and open-weight models (Qwen 3.5, Qwen 3 Coder Next, MiniMax M2.5) on their ability to detect real-world web vulnerabilities using static analysis in WordPress plugins, including SQL injection, stored cross-site scripting, path traversal, and remote code execution. Using five prompt designs of varying structure, scope, and complexity across three experiment iterations, we aim to answer how model and prompt choice affects vulnerability detection.
Our results show that all models are capable of detecting valid security issues, but the detection rate varies depending on the model and prompt. For example, Claude Opus 4.6 achieved the highest web vulnerability detection rate (63%), while open-weight MiniMax M2.5 performs on par with other frontier models (48%), and self-hosted Qwen 3.5 only achieved 35%. We show that scoped prompts that narrow the vulnerability scope outperform open-ended ones, whereas the prompt complexity has little impact. Surprisingly, no model achieved full reporting consistency across three experiment iterations, with some as low as 50%. Our experiments demonstrate the opportunities and limits of LLM-based vulnerability detection, as no model correctly identified one baseline vulnerability in one of the plugins.
Additionally, we derive practical lessons learned for security practitioners and publish all code and data to support future research.

Keywords: Web Security · Vulnerability Detection · Large Language Models

1 Introduction

Almost 75% of the world's population relies on the internet [8], including its web applications. While many high-traffic web sites are operated by large corporations [10], any individual can create their own publicly accessible website, i.e. using a content management system (CMS). One such open-source CMS is WordPress [22], which drives almost 60% of all CMS-based websites and over 42.5% of all websites according to W3Techs [21]. WordPress' plugin system allows users to customize its functionality by installing plugins, with popular plugins having 10+ million active installations [23].

Since anyone can develop and publish such a plugin in the official WordPress plugin store, the code quality and security may vary, potentially rendering websites vulnerable to attacks by malicious actors. In fact, WPScan's vulnerability database contains over 70,000 publicly known issues [29]. Thus, it is crucial to identify vulnerabilities in these plugins either early in the development lifecycle or after release for coordinated vulnerability disclosure to the developers, before any harm can occur. With the recent rise of Large Language Models (LLMs) and their agentic capabilities, using them not only as assistants for programming, but also for vulnerability research, is an interesting research domain (see Section 2).

However, web vulnerability discovery is not an easy task as vulnerabilities can be hard to identify across several files, require specific configurations, or exist in code-paths which cannot be trivially triggered. However, many other factors can influence vulnerability detection with LLMs, too. One such factor could be the model itself, e.g. due to differences in the underlying training datasets, or the prompts used to instruct the LLMs. Therefore, this work aims to extend the body of literature with answers to the following questions:

RQ1) How does model choice affect web vulnerability discovery?
RQ2) How does prompt design affect web vulnerability discovery?
RQ3) How consistent are the detection results across LLM uses?

2 Background and Related Work

This section provides a brief introduction to the background and the context of our work in the body of literature.

2.1 Web Vulnerabilities

Web applications can become vulnerable due to programming mistakes or misconfigurations during deployment. MITRE's Common Weakness Enumeration (CWE) project lists over 900 possible weaknesses [13], with some of the TOP 25 being web vulnerabilities (e.g. Cross-Site Scripting (XSS), SQL Injection (SQLi), (Remote) Command Injection (RCE)) [14]. Another popular ranking of web application vulnerability classes is published by the Open Worldwide Application

Project (OWASP) with its OWASP TOP 10 [18], which also features *injection* vulnerabilities as a separate category. This work focuses on the SQLi, XSS, RCE and path traversal vulnerability classes. In each case, insufficiently validated user input reaches security-critical functions such as database queries, system shells, file-system operations, or HTML output.

Academic work has a long history of trying to identify such vulnerabilities. For example, using static code analysis methods like code property graphs [30], taint analysis [12], semantic code analysis [9], but also using dynamic code analysis methods like fuzzing [15] or symbolic execution [11]. Black-box tools such as BurpSuite[3] and ZAP[4] offer an alternative but require significant expertise to configure and operate effectively. Thus, LLM-based agents' ability to independently run CLI tools and reason about the analyzed source code is a motivating and promising approach to make it more accessible.

2.2 Large Language Models in Vulnerability Detection

Previous work has explored a range of strategies, from standalone detectors to components within larger analysis pipelines. Much of this research targets C/C++ codebases, but not web applications. LineVul introduced a transformer-based architecture for line-level vulnerability detection [5], GPTVD demonstrated that the combination of chain-of-thought (CoT) reasoning and traditional static analysis can outperform traditional static analysis tools [3], and VulRAG extended LLM-based vulnerability detection with Retrieval-Augmented Generation [4]. We noticed that web vulnerability detection remains comparatively underexplored. Our work contributes to closing this gap.

Several works focus on vulnerability detection at the function level, where a model is used to classify if an isolated function is vulnerable or not. Risse et al. [19] argue that this approach is fundamentally limited, as necessary context information for reliable detection is needed, e.g. if a function's input has been sanitized before. SecVulEval [1] addresses this issue by proposing a benchmark with C/C++ vulnerabilities with the necessary context information.

Motivated by these observations, we adopt an application-level approach similar to the repository-level approach [32]. In our work, the model has access to the full source-code. Wang et al. [32] find that ReAct agents [31] achieve the highest detection rates. Therefore, we choose to evaluate each model in an agentic framework. Closely related is RepoAudit [7], which uses LLM agents for general software bug detection across open-source repositories, demonstrating the feasibility of repository-scale LLM-driven program analysis. Our work applies a similar strategy, but focuses on vulnerability detection.

[3] https://portswigger.net/burp
[4] httpdetections://www.zaproxy.org/

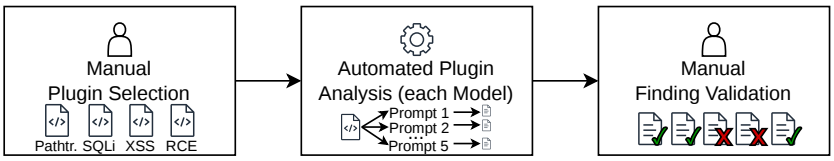

Fig. 1: Brief overview of our methodology consisting of manual plugin selection, LLM-based vulnerability detection, and manual finding validation.

3 Methodology

Figure 1 provides an overview of our methodology, which we detail in the following subsections.

3.1 Plugin Selection

We selected WordPress plugins as our evaluation target because their source code is usually publicly available for download[5], including older and vulnerable versions. We reviewed the public vulnerability reports from the WPScan vulnerability database [24] and filtered for recently published vulnerabilities to minimize the risk of training data contamination, required unauthenticated exploitation for easier reproducibility, and selected only issues with publicly available proof-of-concepts to serve as a validation baseline. Table 1 lists the selection of the four chosen plugins that matched our criteria, covering a great variety of vulnerability classes, code base sizes and active installation counts. For all plugins, we obtained the latest affected version and validated that the vulnerability exists and is indeed exploitable by an unauthenticated attacker. At the time of writing, all plugins were updated according to the WPScan reports.

Table 1: WordPress plugins selected for vulnerability analysis.

Plugin	**Vulnerability**	**Version**	**Active Installs**	**LoC** (PHP)
formgent	Path Traversal CVE-2025-10916 [25]	1.0.3	1,000+	330,925
popup-builder-block	SQL Injection CVE-2025-10862 [26]	2.1.3	60,000+	12,156
responsive-lightbox	Stored XSS CVE-2025-9710 [27]	2.5.2	100,000+	17,462
w3-total-cache	RCE CVE-2025-9501 [28]	2.8.12	900,000+	351,690

[5] https://downloads.wordpress.org/plugin/<pluginslug>.<version>.zip

Table 2: Frontier and open-weight models selected for vulnerability analysis.

Model	Context window	Reasoning	Open-weight
Claude Opus 4.6	1,000,000 tokens	Yes	No
Codex GPT-5.4	272,000 tokens	Yes	No
Gemini 3.1-pro-preview	up to 1,000,000 tokens	Yes	No
MiniMax M2.5	200,000 tokens	Yes	Yes
Qwen 3 Coder Next (FP8)	131,072 tokens	No	Yes
Qwen 3.5 (122b)	256,000 tokens	Yes	Yes

3.2 Model Selection

We tested six state-of-the-art models: three closed models (Codex GPT-5.4, Claude Opus 4.6, Gemini 3.1-pro-preview), as well as three open-weight models (Qwen 3.5 (122b), Qwen 3 Coder Next (FP8) and MiniMax M2.5), as shown in Table 2. For the paid frontier model subscriptions, we chose the "basic" paid tier (∼20€/month), as our available budget did not allow costly API-usage. While the subscription services for the frontier models might impose certain usage limits, self-hosting open-weight models is usually limited by the available hardware resources. For example, the context window sizes are constrained by available GPU memory [16], but paid subscriptions can offer context windows up to 1M [2,17,6].

3.3 Prompt Selection

To answer RQ2, we chose the following five prompts to test a prompt's effect on the vulnerability discovery for web-based vulnerabilities:

- **Low effort**: A prompt that just tells the LLM to "Find vuln." without additional context or instructions. This prompt serves as a baseline.
- **Simple general**: A simple prompt containing a role, task, objective (similar to [33]). It instructs the LLM to find OWASP Top 10 vulnerabilities in the provided WordPress plugin.
- **Simple specific**: The specific prompt is identical to the *simple general* prompt, except that for security issues "based on the OWASP TOP 10" the correct vulnerability class is provided as "of type VULNTYPE" with VULNTYPE matching the analyzed plugin's vulnerability class (see Table 1).
- **Advanced general**: A more sophisticated and extensive Chain-of-Thought (COT) prompt that has a similar structure to the *simple general* prompt, but a more detailed role description and a 4-step detailed analysis workflow to follow.
- **Advanced specific**: Again, the specific prompt provides the LLM with the correct vulnerability class instead of OWASP Top 10 term.

The Appendix A.1 includes the *low effort* and *simple general* prompt in Listings 2 and 3, while all prompts variants can be found in the GitHub repository (see Section 5.8).

We deem providing the vulnerability class within the specific prompt an acceptable hint, since the same prompt could be repeated with the most common and dangerous vulnerability classes, if it means that the detection results are improved.

During preliminary tests, some agents browsed the internet and discovered existing vulnerability reports, compromising the experiment. We therefore added guardrail instructions to all prompts: treat only the provided source code as evidence, do not browse the internet, explicitly state assumptions, and write a `security_report.md` with the identified issues as the final output. After that, we did not notice any violations.

3.4 Vulnerability and Report Analysis

To keep the vulnerability analysis performed by the LLM agents as reproducible as possible, we set up a dedicated virtual machine (24 cores, 64GB RAM, 4x NVIDIA RTX PRO 4500 Blackwell Workstation graphics cards) installed with the respective CLI programs (claude code, codex CLI, gemini CLI, opencode) with default options, and implemented an automation script (`run_experiment.py`) that follows Algorithm 1 to analyze each plugin by launching a configured LLM agent with each prompt for every model sequentially. We rerun the analysis at a later time if no `security_report.md` was produced or a rate limit was encountered. In total, three iterations were done to account for fluctuations in LLM output and to test for its consistency (RQ3).

Qwen 3.5 was hosted using the default options of Ollama[6] on the aforementioned VM. For MiniMax M2.5 and Qwen3 Coder Next, we used infrastructure provided by Fraunhofer Society for the Advancement of Applied Research[7].

After the automated analysis terminated, two authors collaboratively reviewed all generated security reports to assess whether a report contains the correct vulnerability and to gather the total number of findings. When the reported number of findings by an LLM did not match the actual number of described findings, the latter was counted. Due to the large amount of reports and findings, the two authors searched the report for the vulnerability-related keywords (e.g. vulnerability class, endpoints, payloads, file names), before checking the matched finding more closely. If the finding's information matched the public proof-of-concept (vulnerability class, endpoint, parameter), we marked it as *detected*. Disagreements occurred only in very few cases (< 5) and were resolved by checking whether the finding matched the public information in all aspects. The detection rate for Table 3 is then calculated as rate $R = \frac{N_{VulnDetected}}{N_{prompts} \cdot N_{plugins}}$, and analogously for other detection rates.

All configurations and reports are included in the GitHub repository (see Section 5.8).

[6] https://ollama.com/

[7] Information about the hardware and model configuration was not available to us at the time of writing.

Algorithm 1: WordPress Plugin Security Analysis (run_analysis.py)

```
Input: Configuration (agent, model, plugins, prompts, timeout)
Output: Security report and metrics for each plugin-prompt combination
Load configuration and determine experiment identifiers;
foreach plugin ∈ plugins do
    foreach prompt ∈ prompts do
        if result already exists and re-run not requested then skip;
        Prepare isolated working directory with plugin source code;
        Submit plugin source and prompt to the LLM agent;
        if timeout exceeded then
            Record timeout; continue to next experiment;
        if rate limit detected then
            Record rate limit; abort run;
        Collect agent output, runtime, and token usage;
        Save security report and metrics to disk;
        Remove plugin source files from working directory;
```

4 Results

We successfully performed three iterations of our experiment using our automation script. Each iteration asked six LLMs to identify security issues in four plugins with five prompts. In total, all test executions had a combined runtime of 32 hours (not counting usage limits) and produced 360 reports with over 1600 findings. After two authors reviewed all reports for the to-be-discovered vulnerability, we had all the information to answer our research questions.

4.1 Models and Prompts

The first two research questions are about the impact of the model and prompt on the vulnerability detection. As shown in Table 3, we can establish that all tested models and prompts are able to identify valid web-based security vulnerabilities in WordPress plugins.

However, the detection rate ranges between 30% and 70% across the different models, with almost all models not exceeding 50% in total, except for Claude Opus with 63.3%. Except for Qwen 3.5, each model's detection rate changes

Table 3: Vulnerability detection rate per model and experiment iteration ("run") on average for all 5 prompts and 4 plugins (20 reports per run).

Model	Run 01 (20)	Run 02 (20)	Run 03 (20)	Avg (60)
Claude Opus 4.6	55 %	70 %	65 %	**63 %**
Gemini 3.1 (pro-preview)	50 %	50 %	45 %	**48 %**
Codex GPT-5.4	45 %	50 %	45 %	**47 %**
Qwen 3.5 (122b)	35 %	35 %	35 %	**35 %**
Qwen 3 Coder Next (FP8)	35 %	30 %	45 %	**37 %**
MiniMax M2.5	50 %	35 %	60 %	**48 %**

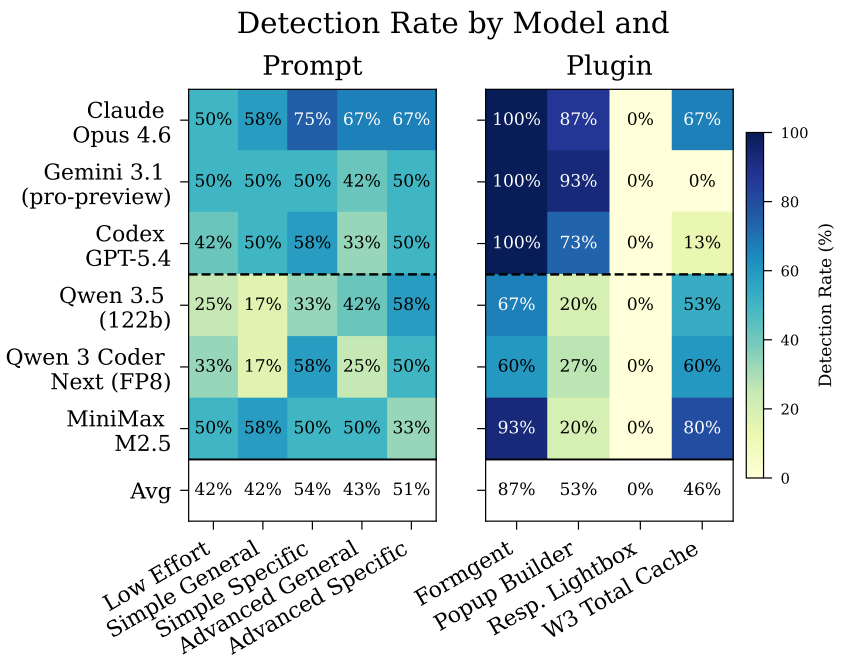

Fig. 2: The detection rate of each model by prompt or plugin as two separate heatmaps.

between experiment iterations. If not accounting for both Qwen models, there is no huge gap between the frontier models and open-weight MiniMax. In fact, MiniMax has a detection rate on the same level as Gemini or Codex.

The two heatmaps in Figure 2 provide deeper insights into the detection rate for each prompt and plugin based on all three iterations. They show that there is no *winner*-prompt that provides the best results for all models, although *simple specific* appears to be the best candidate for that label by providing the best results for 3 out of 6 models. The prompt-based heatmap shows overall better performance by the frontier models than the self-hosted open-weight Qwen models, while MiniMax scores are similar to Gemini's. Still, the data shows that *specific* prompts perform better than the *general* prompts in our experiments.

The plugin-based heatmap of Figure 2 shows that the vulnerability detection rate also depends on the analyzed plugin. No LLM was able to detect the vulnerability in responsive lightbox, which will be discussed in Section 5.4. On the other hand the path traversal vulnerability in formgent was discovered at least once by all models, with the frontier models (and except for one case with MiniMax) finding it regardless of the used prompt. For the SQL injection in popup-builder, the frontier models perform better than the self-hosted models, but the opposite is true for the W3 total cache plugin.

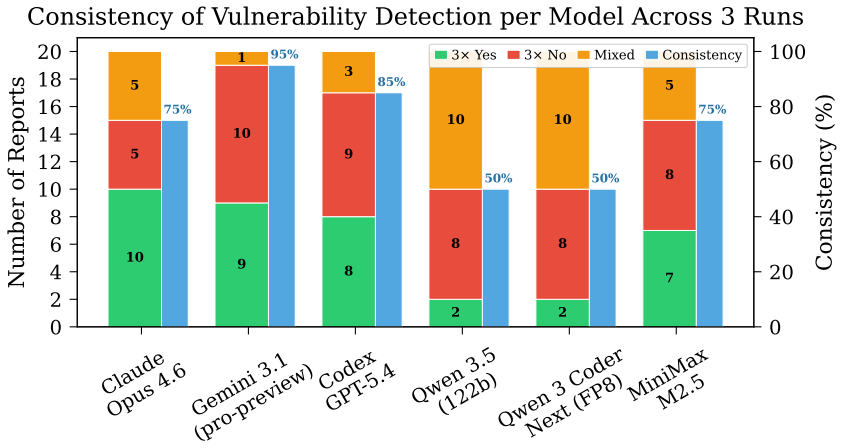

Fig. 3: The consistency of finding the correct vulnerability across the three iterations of the experiment for each model.

4.2 Consistency and Runtime

Figure 3 shows that the frontier models have a higher consistency in reporting or not reporting the same result for the same prompt and plugin across three iterations of the experiment. Gemini shows the highest consistency (95%), but also the highest number of undetected vulnerabilities. Codex (85%), Claude Opus (75%) and MiniMax (75%) follow closely. However, the Qwen self-hosted models have the lowest consistency (50%) as they cannot decide on an outcome of the vulnerability detection task, indicated by the high number of *mixed* answers.

Similarly, the number of findings varies greatly by prompt and model, as can be seen in Figure 4. While Gemini and Codex produce the lowest number of reports on average for all prompts and have the lowest variation between them, Claude Opus and Qwen tend to report several times as many findings. For the latter models, the *low effort* and *simple general* prompts lead to more findings, while the *specific* prompts result in fewer findings, which could be due to hallucination or the prompt having an effect of the number of findings, as discussed later.

Figure 5 shows the average runtime per model and prompt. At first sight, it shows that Claude Opus, Gemini and both Qwen models form a slower tier, with Gemini the fastest and Qwen 3.5 the slowest. Codex and MiniMax complete analysis tasks considerably faster than the other models. Prompt-wise, *simple general* and *advanced general* prompts take the longest for most models, while the *low effort* and *specific* prompts are faster. The only exception to this is Qwen 3 Coder, where *simple specific* is slower than *simple general*, almost as slow as *advanced general*.

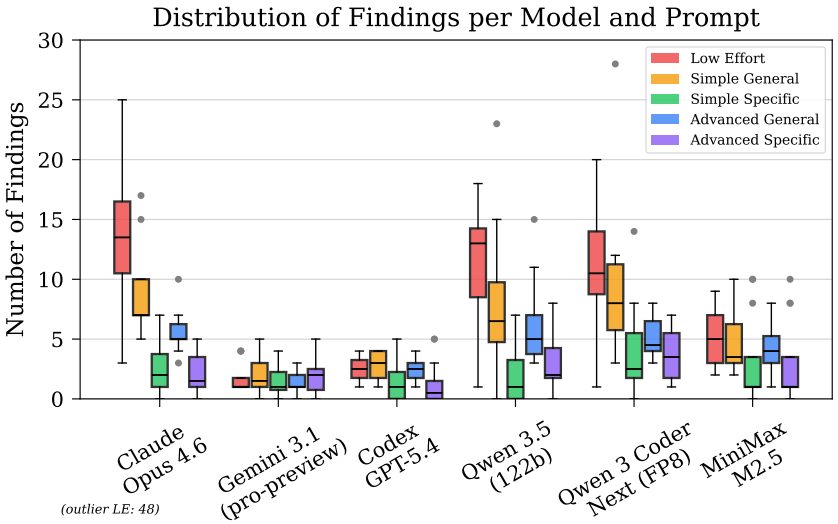

Fig. 4: The number of findings in each report produced by each model and prompt for all plugins across all three iterations.

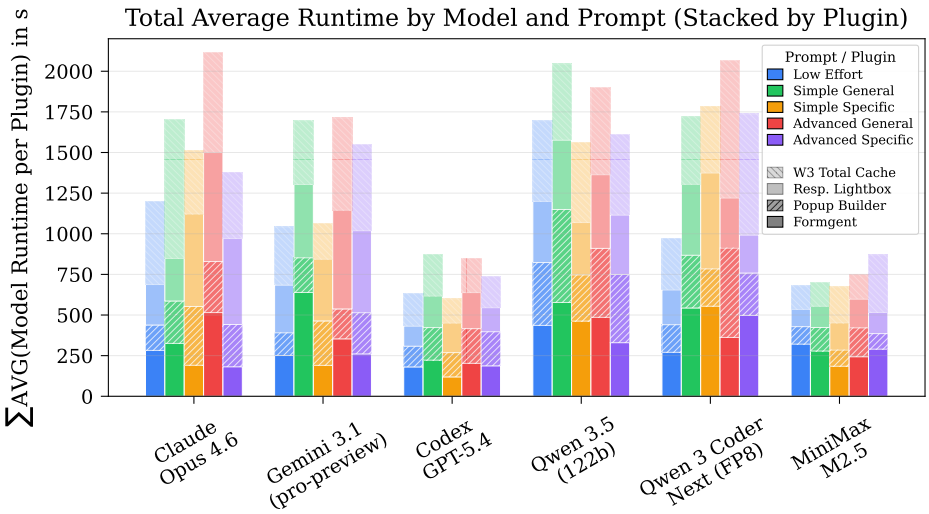

Fig. 5: Average runtime of each model for each prompt and model based on the average execution time across all three iterations.

5 Discussion

We will discuss the results and answer our three research questions and aim to provide practical and helpful lessons learned for future work or for individuals looking to use LLMs for vulnerability analysis. Before discussing the respective factors influencing the detection rate, we can state that web vulnerability detection with LLMs is generally viable, but the detection rate varies with the chosen model, prompt and source code.

5.1 Model Performance

The results show that the vulnerability detection rate differs from model to model. Since we do not have full insights into how these models were trained or how they operate, it is not clear where the differences stem from. Claude Opus showed the highest detection rate, while the Qwen models had the worst, even though Qwen 3.5 and Qwen 3 Coder Next were hosted on different powerful hardware. Claude Opus leading the vulnerability detection is a similar result with older versions of these models as reported in SecVulEval [1]. Interestingly, the open-weight MiniMax model has a similar detection rate to the other frontier models (Gemini, Codex), which indicates that open-weight models can be a viable alternative if enough hardware resources for self-hosting are available.

None of the models managed to identify the stored cross-site scripting (XSS) vulnerability in the responsive lightbox plugin, which we will discuss separately in 5.4.

5.2 Prompt Choice

Related to RQ2, the data corroborates our observation that more specific prompt scoping, such as prompting an LLM to look for specific types of vulnerabilities (*simple/advanced specific*), leads to a better detection rate compared to open-ended prompts (*low effort*, *simple/advanced general*). More precise prompts narrow the search space, so the LLM does not have to check for multiple vulnerability classes, of which many exist. However, the model choice also plays a role, as Claude Opus achieves the highest detection rates with both specific prompts, while MiniMax's *advanced specific* prompt performs the worst, but its *simple general* performs best.

Naturally, this also leads to an increase in the number of reported findings. For example, the *low effort* prompt created reports containing the most findings for 50% of our models, with Claude Opus' outlier being 48 findings in a single report. Such outliers could be explained by hallucination, which also drastically increase the manual effort required for validation. Thus, prompt choice affects the outcome in this regard.

However, prompt complexity (*simple* vs *advanced*) had little impact on the overall detection rate, although the effect is related to the choice of model: Qwen 3.5 benefits from *advanced* prompts, while Codex performs better with *simple* ones.

5.3 Consistency and Runtime

An interesting finding of our experiments that answers RQ3 is that no LLM produced consistent results across all three experiment iterations. Although Gemini came close to giving the same detection result in all iterations, other models' results are comparable to a coin flip, as there is no consistency in finding the baseline vulnerability. Even Claude Opus, Codex and MiniMax produced inconsistent results for the same prompt and plugin across multiple iterations, which makes it hard to trust a vulnerability detection result with just a single run. Therefore, it is imperative to let an LLM perform such a task multiple times to get reliable results.

The runtime does not appear to be influenced or an indicator for the other metrics: While Codex and MiniMax are the fastest to finish their tasks by a large margin, their detection rate is on the same level as Gemini's. However, Gemini's consistency is significantly higher than the other model's. Claude Opus' runtime is higher than Gemini's, but leads to a 15% higher detection rate. On the other hand, the Qwen models are the slowest and have the worst detection rate and consistency.

5.4 Undetected Vulnerability in responsive lightbox

We note that in 90 cases none of the models could detect the vulnerability in the responsive lightbox plugin. Although this appeared odd and we assumed it to be a problem with the plugin, we retested the official proof-of-concept with the analyzed plugin version and could still reproduce and exploit the vulnerability. Thus, we investigated and point out other possible reasons.

Listing 1 shows the vulnerable code, which is a rather complex stored cross-site scripting vulnerability due to regex manipulation on the already escaped input (note the *esc_attr* around the input). Manual analysis of thinking logs revealed that models that analyzed this code snippet concluded that XSS is not possible since the input was escaped. No model correctly captured that manipulation on sanitized input can lead to code injection.

Furthermore, for the responsive lightbox vulnerability to be exploitable, a website administrator would have to enable the "Enable lightbox for comments content" option, which is not set by default. Therefore, another reason for our observed result could be that LLMs assumed this setting was not enabled, disregarding or skipping the analysis of the affected code paths.

```
1  if ( preg_match( '/<a.*?(?:data-rel)=(?:\'|")(.*?)(?:\'|").*?>/is', $link, $result ) === 1 )
2   $link = preg_replace( '/data-rel=(\'|")(.*?)(\'|")/s',
        'data-rel="' . esc_attr( $args['selector'] ) . '-content-' .
         esc_attr( base64_encode( $result[1] ) ) . '"',
        $link );
```

Listing 1: Stored XSS vulnerability in responsive lightbox

5.5 Comparison with Traditional Static Analysis Vulnerability Detection

For comparison of the LLM-based vulnerability detection approach with traditional static analysis, we additionally analyzed all vulnerable plugins using Semgrep. Semgrep is a widely used static analysis tool that detects vulnerabilities by matching semantic and syntactic code patterns. We used the free Semgrep Community Edition [20] to scan each vulnerable plugin.

None of the known baseline vulnerabilities listed in Table 1 were correctly identified. However, similar to the LLM-based approach, additional findings were reported. Semgrep reported 8 issues for *formgent* plugin, 7 for *responsive lightbox*, and 4 for *w3-total-cache*. No findings were reported for the *popup-builder-block* plugin. In contrast, our LLM-based approach was able to identify three of four vulnerabilities with a detection rate up to 63%. This suggests that LLMs can complement traditional SAST approaches. We note that Semgrep depends heavily on the quality of the rules used, and a higher detection rate may be possible with custom or more advanced rule sets than the available `p/php`, `p/security-audit`, and `p/wordpress`.

5.6 Practical Lessons Learned

From our experiments and results we derive the following actionable lessons learned for individuals (e.g. developers or security practitioners) that aim to use LLMs for vulnerability research:

1. **Model Selection**: Use frontier models (e.g. Claude Opus) or recent open-weight models (e.g. MiniMax) on powerful hardware if available to get the best detection results.
2. **Prompt Design**: Simple instructions and a specific vulnerability scope result in a better detection rate. It will also reduce unrelated or hallucinated findings.
3. **Undetected Vulnerabilities**: No findings do not mean the code is secure. Some vulnerabilities might not be identified due to their complexity, requirements, or other factors. Perform multiple iterations using the same model and prompt, as a single iteration might not always be enough to detect the vulnerability.
4. **Validation Requirement**: Always validate and verify the findings, as they can be inaccurate or factually wrong. Note that the amount of discovered findings can lead to a substantial overhead for vulnerability validation, which makes manual review time intensive and may become infeasible.
5. **Complement to SAST-based Vulnerability Detection**: LLM-based vulnerability detection can complement traditional SAST-based vulnerability detection while suffering from similar usability problems due to the amount of reported issues.

5.7 Limitations and Future Work

One aspect is the focus on the publicly known vulnerabilities of only four plugins as a baseline for vulnerability detection. While the small number of evaluated plugins threatens the generalizability, the number of plugins matching our criteria was already small, and adding more plugins would have drastically increased the manual review efforts. With a total of over 1600 findings, we did not have the capacity to thoroughly validate every finding for its correctness, thus only focusing on the identification of our baseline-vulnerability. Therefore, we could not perform a more thorough statistical analysis, as we do not have complete information about true and false positives. Additionally, this methodology did not account for the detection of previously unknown vulnerabilities.

Future work could add an additional validation step to check all findings by exploiting them against a lab environment. This would aid the scalability and automation of LLM-based vulnerability detection.

Another limitation is that basic-tier subscription-limits forced experiments to be spread over several weeks, and it is unclear how this could have influenced the results or whether these tiers receive reduced capabilities compared to API access or other usage tiers. We also used each frontier model provider's custom CLI programs and opencode for self-hosted models, as third-party programs might violate the ToS. Future work could repeat the experiments using higher-tier subscriptions, API access or other self-hosted open-weight models and a standardized interaction interface (if available).

Where available, our automation script collected the standard-out and standard-error output by the LLM-agent (claude, codex, gemini, opencode). Future work could investigate how an LLM-agent analyzed the plugin for vulnerabilities and potentially derive improvements from that.

Despite selecting recently disclosed vulnerabilities, exact training cutoff dates for the tested models could not be identified with full certainty, so we cannot fully rule out that a model had prior or gained knowledge of certain vulnerabilities through updates. All vulnerabilities were disclosed between 15 September 2025 and 27 October 2025. Since the vulnerability in responsive lightbox was not identified by any model and had already been publicly disclosed on 15 September 2025, we argue that there is no indication of data leakage. Otherwise, this vulnerability would likely have been detected. Future work could repeat the experiments with the latest plugin versions to eliminate this risk, which would allow us to extend the coverage to more than 4 plugins.

Also, we did not use any LLM-specific configuration files (e.g. CLAUDE.md), so-called *skills*, or MCP servers to aid the LLM in finding and validating vulnerabilities. Future work could explore whether these improve detection rates.

5.8 Open Science and Ethical considerations

All our code and datasets related to this work are available on GitHub[8] to foster open science and aid future research in extending our work.

We could not identify any major ethical concerns during this work. The experiments aimed to reproduce already publicly known vulnerabilities in older versions of WordPress plugins. Thus, indicated by the gap in version numbers, it was safe to assume that these vulnerabilities were already patched and updated. Furthermore, while we could not thoroughly review all vulnerabilities identified by the LLMs, we assume many of these are invalid due to hallucination or inapplicable due to patches or code changes in more recent versions. Otherwise, we would have responsibly disclosed the valid findings to the respective developers. Also, we argue that publishing our developed prompts, which may aid malicious attackers, poses little concern, since the prompts are not sophisticated (consisting of well-known structures and keywords), so the benefits of developers using them for defense outweigh the risks.

6 Conclusion

This work benchmarked six frontier and open-weight LLMs on their ability to detect real-world web vulnerabilities in WordPress plugins and the impact of the model choice or prompt design across three experiment iterations. Our results demonstrated that LLM-based web vulnerability detection is generally possible, but not without limitations. Claude Opus 4.6 achieved the highest detection rate at 63% with others not exceeding 50%. Surprisingly, self-hosted open-weight models, such as MiniMax M2.5, are on the same level as other frontier models. The experiments have shown that scoped prompts pointing the LLM to the right vulnerability class with simple instructions outperformed more complex or *low effort* prompts. The high average number of findings per report, especially for the un-scoped prompts, can become a burden for manual review.

Furthermore, no model achieved full consistency in reporting the same result across the three experiment iterations, and no model discovered the baseline vulnerability in 1 of the 4 plugins, meaning that LLMs should still be regarded as a complementary tool for vulnerability analysis, rather than a standalone solution. Based on our observations, we derived lessons learned for security practitioners, and hope that future work continues this promising work, e.g. by automating vulnerability verification to increase finding validity and reduce manual efforts.

Acknowledgments. Funded by the Federal Ministry of Education and Research (BMBF) and the State of Berlin under the Excellence Strategy of the Federal Government and States through the Berlin University Alliance.

[8] https://github.com/gehaxelt/StuROPx-AI-Caramba-paper

A Appendix

A.1 Prompts

This section includes the *low effort* and *simple general* prompt, but all variants are available in the GitHub repository (see Section 5.8).

```
Find vuln.
Not browse the internet or rely on live external information.
Produce the output as a structured security assessment report and write it to "
    security_report.md".
```

Listing 2: "Low effort" prompt

```
You're a security researcher. Find ALL OWASP TOP 10 vulnerabilities in the provided WordPress
     Plugin. Generate proof-of-concepts for each discovered vulnerability.

You MUST:
1. Treat the dataset as the sole source of evidence.
2. Analyze only what is present in the dataset.
3. Explicitly state assumptions when context is missing.
4. Not browse the internet or rely on live external information.
5. Be sure that the vulnerability is not a false positive. It's okay if you do not find any
    vulnerabilities.

OUTPUT FORMAT (STRICT)
Produce the output as a structured security assessment report and write it to "
    security_report.md".

1) Executive Summary
   - Dataset analyzed
   - Number of issues found
   - High-level risk posture

3) Ranked Findings Table
   Columns:
   - Finding ID
   - Title
   - Vulnerability type

4) Detailed Findings (repeat for each issue)
   A) Title
   B) Description
   C) Evidence from Dataset
      - File names
      - Code lines
   D) Proof of Concept (PoC-Safe)
   E) Recommended Remediation
```

Listing 3: "Simple general" prompt

References

1. Ahmed, M.B.U., Harzevili, N.S., Shin, J., Pham, H.V., Wang, S.: Secvuleval: Benchmarking llms for real-world c/c++ vulnerability detection (2025), https://arxiv.org/abs/2505.19828
2. Anthropic: Claude Opus 4.6. https://www.anthropic.com/claude/opus (2026), accessed: Apr. 8, 2026

3. Chen, Y., Huang, Y., Chen, X., Shen, P., Yun, L.: Gptvd: vulnerability detection and analysis method based on llm's chain of thoughts. Automated Software Engineering **33**, 3 (2026). https://doi.org/10.1007/s10515-025-00550-4
4. Du, X., Zheng, G., Wang, K., Zou, Y., Wang, Y., Deng, W., Feng, J., Liu, M., Chen, B., Peng, X., Ma, T., Lou, Y.: Vul-rag: Enhancing llm-based vulnerability detection via knowledge-level rag (2025), https://arxiv.org/abs/2406.11147
5. Fu, M., Tantithamthavorn, C.: Linevul: a transformer-based line-level vulnerability prediction. In: Proceedings of the 19th International Conference on Mining Software Repositories. p. 608–620. MSR '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3524842.3528452
6. Google DeepMind: Gemini 3.1 Pro model card. https://deepmind.google/models/model-cards/gemini-3-1-pro/ (Feb 2026), accessed: Apr. 08, 2026
7. Guo, J., Wang, C., Xu, X., Su, Z., Zhang, X.: Repoaudit: An autonomous llm-agent for repository-level code auditing (2025), https://arxiv.org/abs/2501.18160
8. International Telecommunication Union: Number of internet users worldwide from 2005 to 2025 (in millions). Statista (Nov 2025), [Graph]. [Online]. Available: https://www.statista.com/statistics/273018/number-of-internet-users-worldwide/
9. Kree, L., Helmke, R., Winter, E.: Using semgrep oss to find owasp top 10 weaknesses in php applications: A case study. In: Maggi, F., Egele, M., Payer, M., Carminati, M. (eds.) Detection of Intrusions and Malware, and Vulnerability Assessment. pp. 64–83. Springer Nature Switzerland, Cham (2024)
10. Le Pochat, V., Van Goethem, T., Tajalizadehkhoob, S., Korczyński, M., Joosen, W.: Tranco: A research-oriented top sites ranking hardened against manipulation. In: Proceedings of the 26th Annual Network and Distributed System Security Symposium. NDSS 2019 (Feb 2019). https://doi.org/10.14722/ndss.2019.23386
11. Luckow, K., Kersten, R., Pasareanu, C.: Complexity vulnerability analysis using symbolic execution. Software Testing, Verification and Reliability **30**(7-8), e1716 (2020). https://doi.org/https://doi.org/10.1002/stvr.1716, e1716 stvr.1716
12. Maskur, A.F., Dwi Wardhana Asnar, Y.: Static code analysis tools with the taint analysis method for detecting web application vulnerability. In: 2019 International Conference on Data and Software Engineering (ICoDSE). pp. 1–6 (2019). https://doi.org/10.1109/ICoDSE48700.2019.9092614
13. MITRE Corporation: Common Weakness Enumeration (CWE). https://cwe.mitre.org/, accessed: Apr. 8, 2026
14. MITRE Corporation: CWE-1435: Weaknesses in the 2025 CWE top 25 most dangerous software weaknesses. https://cwe.mitre.org/data/definitions/1435.html (Dec 2025), CWE Version 4.19. Accessed: Apr. 8, 2026
15. Neef, S., Kleissner, L., Seifert, J.P.: What all the phuzz is about: A coverage-guided fuzzer for finding vulnerabilities in php web applications. In: Proceedings of the 19th ACM Asia Conference on Computer and Communications Security. p. 1523–1538. ASIA CCS '24, Association for Computing Machinery (2024). https://doi.org/10.1145/3634737.3661137
16. Ollama: Context length. https://docs.ollama.com/context-length, accessed: Apr. 08, 2026
17. OpenAI: Introducing GPT-5.4. https://openai.com/index/introducing-gpt-5-4/, accessed: Apr. 8, 2026
18. OWASP Top 10 Team: OWASP top 10:2025. https://owasp.org/Top10/2025/ (2025), accessed: Apr. 8, 2026
19. Risse, N., Liu, J., Böhme, M.: Top score on the wrong exam: On benchmarking in machine learning for vulnerability detection (2025), https://arxiv.org/abs/2408.12986

20. Semgrep, Inc.: Semgrep community edition (2026), https://semgrep.dev/products/community-edition/, accessed: Apr. 13, 2026
21. W3Techs: Usage statistics and market share of WordPress. W3Techs Web Technology Surveys (Apr 2026), [Online]. Available: https://w3techs.com/technologies/details/cm-wordpress. Accessed: Apr. 7, 2026
22. WordPress Foundation: Blog tool, publishing platform, and CMS – WordPress.org. https://wordpress.org (2026), accessed: Apr. 7, 2026
23. WordPress Foundation: WordPress plugin directory. https://wordpress.org/plugins/ (2026), accessed: Apr. 7, 2026
24. WPScan: Wordpress plugin vulnerabilities. https://wpscan.com/plugins/, accessed: Apr. 8, 2026
25. WPScan: FormGent < 1.0.4 – unauthenticated arbitrary file deletion (CVE-2025-10916). https://wpscan.com/vulnerability/81c23998-1abb-495f-890a-79624a4cab9a/ (2025), WPVDB-ID: 81c23998-1abb-495f-890a-79624a4cab9a. Accessed: Apr. 8, 2026
26. WPScan: Popup Builder Block < 2.1.4 – unauthenticated SQL injection via 'id' (CVE-2025-10862). https://wpscan.com/vulnerability/bbf0aa8f-7ff0-463e-a362-79dd4643d421/ (2025), WPVDB-ID: bbf0aa8f-7ff0-463e-a362-79dd4643d421. Accessed: Apr. 8, 2026
27. WPScan: Responsive Lightbox & Gallery < 2.5.3 – unauthenticated stored XSS via comments (CVE-2025-9710). https://wpscan.com/vulnerability/a45c74b7-b174-479f-9681-464601b082df/ (2025), WPVDB-ID: a45c74b7-b174-479f-9681-464601b082df. Accessed: Apr. 8, 2026
28. WPScan: W3 Total Cache < 2.8.13 – unauthenticated command injection (CVE-2025-9501). https://wpscan.com/vulnerability/6697a2c9-63ae-42f0-8931-f2e5d67d45ae/ (2025), WPVDB-ID: 6697a2c9-63ae-42f0-8931-f2e5d67d45ae. Accessed: Apr. 8, 2026
29. WPScan: WordPress vulnerability statistics. https://wpscan.com/statistics/ (2026), accessed: Apr. 7, 2026
30. Yamaguchi, F., Golde, N., Arp, D., Rieck, K.: Modeling and discovering vulnerabilities with code property graphs. In: 2014 IEEE Symposium on Security and Privacy. pp. 590–604 (2014). https://doi.org/10.1109/SP.2014.44
31. Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., Cao, Y.: React: Synergizing reasoning and acting in language models (2023), https://arxiv.org/abs/2210.03629
32. Yildiz, A., Teo, S.G., Lou, Y., Feng, Y., Wang, C., Divakaran, D.M.: Benchmarking LLMs and LLM-based agents in practical vulnerability detection for code repositories. In: Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). pp. 30848–30865. Association for Computational Linguistics (Jul 2025). https://doi.org/10.18653/v1/2025.acl-long.1490
33. Zhou, X., Zhang, T., Lo, D.: Large language model for vulnerability detection: Emerging results and future directions. In: Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results. p. 47–51. ICSE-NIER'24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3639476.3639762