UNDERSTANDING CODING AGENTS FOR PATCH PRESENCE TESTING

Anonymous authors
Paper under double-blind review

ABSTRACT

Patch presence testing determines whether a binary incorporates the security fix for a known vulnerability. Existing systems generally require symbols and therefore cannot operate end to end on stripped binaries, limiting their practical applicability. Recent advances in large language models and coding agents make it possible to use agents directly for patch presence testing.

We conduct a systematic empirical study of agent systems for end-to-end patch presence testing. Our findings show that: (1) with symbols available, a coding agent using only binutils tools performs comparable to the state-of-the-art specialized system; (2) stripping and build-configuration variation have limited impact, whereas patch evolution remains challenging; (3) exposing additional binary-analysis interfaces introduces regressions and higher analysis cost, whereas enriched patch metadata raises performance at lower cost. Finally, we evaluate 12 combinations of four model configurations and three agent implementations, and derive a practical deployment recommendation that balances decision success and cost.

1 INTRODUCTION

Security patching is fundamental to software security, yet modern software supply chains make it difficult to determine whether a specific vulnerability has been fixed in a deployed binary Pashchenko et al. (2018); Jiang et al. (2024). The Log4j vulnerability National Vulnerability Database (2021); Everson et al. (2022), for example, showed how a single flawed dependency can create ecosystem-wide verification challenges. To address this problem, **patch presence testing (PPT)** analyzes whether a target binary incorporates the corresponding security fix.

A source-level patch can produce substantially different binary representations across architectures, compilers, optimization levels, and software versions. Existing approaches seek to tolerate this variability through syntactic feature matching or semantic analysis Sun et al. (2021); Zhan et al. (2024b); Xu et al. (2020); Yang et al. (2023); Pan et al. (2024); Lang et al. (2021); Xu et al. (2026), and many achieve high Accuracy or Recall on successfully analyzed cases. Yet these metrics do not necessarily capture practical effectiveness. A recent empirical study found that high Accuracy or Recall could coexist with a low decision success rate (DSR), which measures the rate of correct verdicts over all test cases, including failed analyses Zhang et al. (2026). It further showed that patch presence testing tools relied on symbols or a supplied target-function range, and therefore could not perform fully end-to-end analysis when such localization hints were unavailable in stripped binaries. This leaves a gap between verifying a patch once its location is known and localizing and verifying that patch from a deployed binary.

With the development of LLMs, recent work has begun to incorporate them into patch presence testing. Lares uses LLMs to understand patch semantics and identify semantically equivalent code slices in the target binary, together with constraint solving for verification Li et al. (2025). PPTSP applies an LLM when structurally similar patch-affected paths require further semantic comparison Xu et al. (2026). Both demonstrate the value of LLM reasoning for patch presence testing, but assume that the patch-relevant binary region has already been identified.

Meanwhile, LLM-based coding agents can interact with file systems, command-line tools, and other analysis utilities through agent harnesses, enabling multi-step analysis with substantially less manual

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

orchestration. General-purpose coding agents OpenAI (2026) therefore offer the potential for end-to-end patch presence testing, where an agent could start from vulnerability information and a target binary, use binary-analysis tools to locate and inspect patch-relevant binary regions, and determine the patch status. We therefore investigate whether coding agents can perform end-to-end patch presence testing effectively.

Such a task requires the agent to locate patch-relevant code in stripped binaries, recognize fixes across build variations and patch evolution Xie et al. (2024b), and make reliable decisions under resource constraints. Its effectiveness may also depend on the underlying model, the vulnerability metadata provided, and the tools exposed by the *agent harness*—the software layer that coordinates tool use and manages the agent's interaction context Anthropic (2025). These factors have not yet been systematically studied for agentic patch presence testing.

To investigate these issues, we conduct an empirical study of coding agents for patch presence testing. We construct four evaluation sets covering realistic deployment conditions and key challenges for SOTA patch presence testing systems. The Source-Built Benchmark contains 120 CVEs across six projects and covers multiple architectures, compilers, optimization levels, and symbol conditions. We further include an Applicability set of `not_applicable` binaries, where the vulnerability-affected functionality is absent and thus no patch-status judgment applies, a Patch Evolution set containing later implementations of known fixes, and an Ubuntu Packages set based on deployed distribution binaries.

- **RQ1:** Can a general-purpose coding agent achieve performance comparable to state-of-the-art patch presence testing systems?
- **RQ2:** How robust is the agent under stripped binaries, build variation, patch evolution, `not_applicable` binaries, and deployed Ubuntu packages?
- **RQ3:** Which factors influence agent performance in patch presence testing?
- **RQ4:** Which agent–model configurations best balance DSR and deployment cost?

In RQ1, we compare Codex OpenAI (2026) with three patch presence testing systems under binaries with symbols. Codex reaches 0.99 for both Accuracy and DSR using only binutils, comparable to Lares, which reaches 1.00 Accuracy and 0.99 DSR, while PS^3 and BinXray reach only 0.28 and 0.27 DSR.

In RQ2, we test Codex on stripped binaries, build variations, patch evolution, `not_applicable` binaries, and Ubuntu packages. Stripping produces only a 0.025 reduction in DSR, build and packaging changes have little effect, and 95% of `not_applicable` binaries are identified correctly. The main challenges are patch localization and evolved fixes.

In RQ3, we investigate what determines agent capability by varying model intelligence, vulnerability metadata, and binary-analysis tools. Stronger models mainly improve patch localization. Enriched metadata improves DSR while reducing token use, whereas adding binary-analysis tools increases resource consumption without improving DSR. These results identify patch-site localization as the main bottleneck and show that providing richer patch and vulnerability information is effective.

In RQ4, we evaluate 12 model–agent combinations. Codex with GPT-5.6 Sol max achieves the highest DSR of 0.961, while MiniSWEAgent with GPT-5.6 Sol medium reaches 0.956 DSR at \$0.37 per case and offers the best effectiveness–cost trade-off.

The main contributions of this paper are as follows:

- We evaluate coding agents for binary patch presence testing, comparing a general-purpose agent with specialized systems and testing whether its performance extends to stripped binaries, evolved fixes, and deployed packages.
- We identify patch-site localization as the primary cause of the model-capability gap and examine how enriched patch metadata and additional binary-analysis tools affect agent performance.
- We evaluate decision success and cost across 12 model–agent configurations and derive practical deployment recommendations for different effectiveness and cost requirements.

2 BACKGROUND

2.1 PATCH PRESENCE TESTING

Patch presence testing determines whether a specific security fix is present in a target binary. Existing techniques typically derive a patch signature from patched and vulnerable reference binaries and match it against the target binary Zhang & Qian (2018); Xu et al. (2020); Zhan et al. (2024a;b). The main challenge is to construct signatures that remain distinguishable despite the semantic gap between source patches and binary code and the transformations introduced by compilation configurations.

In real-world deployment, patch presence testing faces patch evolution, where later revisions reimplement a fix while preserving its security property Xie et al. (2024c), and applicability Zhang et al. (2026), which asks whether the vulnerability-relevant functionality is present in the target binary. If it is absent, the correct outcome is `not_applicable`.

2.2 CODING AGENTS AND AGENT HARNESSES

A coding agent couples a large language model with an iterative control loop that allows the model to observe an environment, invoke tools, and update its analysis over multiple steps OpenAI (2026); Yang et al. (2024); Wang et al. (2024; 2025). The model performs reasoning and selects actions, while the surrounding *agent harness* defines the task interface, maintains interaction state, executes tool calls, enforces resource limits, and controls how a final answer is submitted Xia et al. (2025); Bandel et al. (2026).

Agent behavior therefore depends not only on the underlying model, but also on how the harness presents task information and exposes external capabilities. In particular, the available metadata can shape the agent's initial understanding of the task, while the tool stack determines what information it can obtain during analysis. Different harness designs may therefore lead the same model to different analysis paths and final outcomes.

3 STUDY DESIGN

3.1 END-TO-END PATCH PRESENCE TESTING TASK

Task inputs. Each case contains one target ELF binary and metadata for one CVE. The metadata includes the CVE identifier, public description, CWE label, fixing patch, and patch commit message. In the stripped setting, no symbols, candidate addresses, or debug information are provided. Here, end-to-end means that the agent must localize and verify the patch directly in the target binary without symbols, candidate addresses, or binary regions; public source-side vulnerability and patch metadata remain available. **Task outputs.** The system returns one outcome per case: `patched`, `vulnerable`, `not_applicable`, or `inconclusive`. A `not_applicable` outcome requires evidence that the vulnerability-relevant functionality is absent, while `inconclusive` indicates insufficient evidence for a determinate decision.

3.2 DATASETS

Our evaluation uses four datasets based on Zhang et al. (2026). The **Source-Built Benchmark** covers six projects, with 20 CVEs per project selected for diverse patch semantics, and includes x86-64/AArch64, GCC/Clang, different optimization levels, and symbol-known/stripped binaries. The **Applicability** set contains binaries where the CVE-relevant functionality is excluded by the build, making `not_applicable` the correct outcome. The **Patch Evolution** set contains later versions where the original fix has been modified or reimplemented, while the **Ubuntu Packages** set evaluates deployed x86-64 distribution binaries.

3.3 EVALUATION METRICS

Appendix A defines the evaluation metrics. We report Accuracy, Precision, Recall, and F1 for patched/vulnerable cases, excluding `inconclusive` and `not_applicable` outputs from these

Table 1: Detailed and Overall Performance of Patch Presence Testing Tools

	Codex					PS^3					BinXray					Lares				
Project	**A**	**P**	**R**	**F_1**	**DSR**	**A**	**P**	**R**	**F_1**	**DSR**	**A**	**P**	**R**	**F_1**	**DSR**	**A**	**P**	**R**	**F_1**	**DSR**
Binutils	1.00	1.00	1.00	1.00	1.00	0.49	0.76	0.38	0.51	0.36	0.38	1.00	0.11	0.20	0.05	1.00	1.00	1.00	1.00	1.00
curl	0.99	1.00	0.98	0.99	0.99	0.57	0.92	0.32	0.48	0.34	0.45	0.50	0.33	0.40	0.15	1.00	1.00	1.00	1.00	1.00
FFmpeg	0.99	1.00	0.98	0.99	0.98	0.93	1.00	0.90	0.95	0.34	0.68	1.00	0.48	0.65	0.54	1.00	1.00	1.00	1.00	1.00
libxml2	0.96	0.98	0.93	0.96	0.96	0.59	0.60	0.75	0.67	0.10	0.50	0.50	0.25	0.33	0.07	0.97	0.95	1.00	0.98	0.97
OpenSSL	1.00	1.00	1.00	1.00	1.00	0.83	0.75	1.00	0.86	0.56	1.00	1.00	1.00	1.00	0.43	1.00	1.00	1.00	1.00	0.95
SQLite	1.00	1.00	1.00	1.00	1.00	0.51	0.69	0.38	0.49	0.40	0.89	0.96	0.83	0.89	0.72	1.00	1.00	1.00	1.00	1.00
Overall	**0.99**	**1.00**	**0.98**	**0.99**	**0.99**	**0.56**	**0.77**	**0.45**	**0.57**	**0.28**	**0.71**	**0.86**	**0.58**	**0.69**	**0.27**	**1.00**	**0.99**	**1.00**	**1.00**	**0.99**

Measured on the source-built x86-64 GCC-O2 benchmark.

binary metrics. For the Applicability set, we report the `not_applicable` identification rate. Our primary metric is decision success rate (DSR), defined as the rate of correct verdicts over all attempted cases; incorrect `not_applicable` predictions and inconclusive outcomes therefore reduce DSR. RQ4 additionally evaluates token usage, tool calls, and monetary cost.

3.4 Evaluated Systems

3.4.1 Agent Systems.

We evaluate three agent systems with different levels of harness specialization. **Codex** OpenAI (2026) is a general-purpose coding agent that autonomously plans its analysis, issues shell commands, and invokes binary-analysis tools. **MiniSWEAgent** SWE-agent Team (2026) is a lightweight general-purpose coding agent with a compact control loop and shell-based interface. **PPTAgent** is developed specifically for patch presence testing by the authors, combining vulnerability context, bounded source guidance and evidence-based decision making; Section 7 details its design.

3.4.2 Patch Presence Testing Baselines.

RQ1 compares Codex with three existing patch presence testing systems. PS^3 Zhan et al. (2024a) derives symbolic-execution signatures from a patch and matches them against the target function. BinXray Xu et al. (2020) captures patch-induced structural differences between vulnerable and patched reference binaries and matches them against the target through binary diffing. Lares Li et al. (2025) uses LLMs to analyze patch-related code slices and identify corresponding behavior in the target, with constraint solving used for verification. For a fair comparison, Lares uses GPT-5.6 Sol max, the same model configuration used by Codex in RQ1.

3.5 Evaluation Settings

We evaluate four model–reasoning configurations ranked by Artificial Analysis Artificial Analysis (2026): T1, GPT-5.6 Sol max (61); T2, GPT-5.6 Sol medium (56); T3, GPT-5.6 Terra high (50); and T4, GPT-5.6 Luna medium (39). Each agent receives the CVE description, patch hunk, and commit message, and performs read-only static analysis with binutils tools such as `objdump`, `readelf`, and `strings`. Network access, dynamic analysis, external version lookup, and target source code are disabled. Each case is limited to 32 tool calls, with no token or time limit. We retain complete interaction traces and manually verify that each verdict is grounded in evidence obtained from the supplied binary.

4 RQ1: Performance on Binaries with Symbols

The patch-presence tools evaluated in this study rely on symbol information to identify the target function and therefore do not perform end-to-end analysis Zhang et al. (2026). RQ1 uses the symbol-known binaries of the Source-Built Benchmark to provide a comparison setting. We evaluate Codex with GPT-5.6 Sol max alongside the three patch-presence systems in Section 3.4.

Table 2: Stripped and cross-build performance.

Arch	Compiler	Opt	A	P	R	DSR
x86-64	GCC	O2	0.97	0.96	0.98	0.96
x86-64	GCC	O3	0.99	0.98	0.99	0.97
x86-64	Clang	O2	0.98	0.97	0.98	0.95
AArch64	GCC	O2	0.98	0.99	0.98	0.98

Table 3: Performance under realistic settings.

Setting	A	P	R	DSR
Patch Evolution	0.95	1.00	0.95	0.86
Applicability	0.95	NA	NA	0.95
Ubuntu packages	0.97	0.98	0.96	0.97

The general-purpose agent achieves performance comparable to existing PPT systems.

As shown in Table 1, Codex achieves 0.99 Accuracy and 0.99 DSR, with consistently high performance across projects. Lares reports 1.00 Accuracy and 0.99 DSR, indicating comparable performance in the symbol-known setting. Their near-perfect performance suggests that binaries with symbols provide limited insight into the true capability and failure modes of strong LLM-based PPT systems.

Codex achieves performance comparable to the strongest existing PPT system in the symbol-known setting, while traditional pipelines remain limited by low decision success.

5 RQ2: Real-World Capability

5.1 Stripped Binaries

As Table 2 shows, the overall impact of stripping is limited. Accuracy decreases from 0.99 to 0.97 and DSR from 0.99 to 0.96. Five of the six projects still achieve accuracies of 0.98 or higher, suggesting that stripping alone does not degrade performance in most cases.

Among cases unaffected by stripping, distinctive patch-related clues are available for localization. Listing 1 shows CVE-2017-12448, where the archive magic strings corresponding to `ARMAG`, `ARMAGT`, and `ARMAGB` still guide the agent to the relevant checks, after which the patched error-handling logic is easy to verify. Here, stripping mainly increases localization difficulty.

Libxml2 exhibits a larger drop, from 0.96 to 0.88 accuracy. In the regressed cases, patch localization is difficult even after the relevant function has been found, as similar checks may appear at multiple locations within the same function. Listing 2 shows CVE-2015-5312, where a similar `lastError.code` guard already exists before the patched check was added. After stripping, the two guards become difficult to distinguish in assembly, accounting for ten of the thirteen new failures.

5.2 Build-Configuration Robustness

Build configuration has limited impact. We evaluate the agent across the build settings of the Source-Built Benchmark (Section 3.2). As shown in Table 2, Accuracy remains between 0.97 and 0.99 across the tested settings, with Precision and Recall in a similar range. DSR is 0.97 under GCC-O3, 0.98 under AArch64 GCC-O2, and 0.95 under Clang-O2. These results indicate that the agent is largely robust to build changes and does not rely on exact instruction sequences for patch presence testing.

5.3 Patch Evolution and Applicability

The agent recognizes most non-applicable cases. As shown in Table 3, Codex correctly identifies 95% of the `not_applicable` binaries. **Patch evolution remains challenging.** Semantic changes can preserve the security property while altering how the fix is represented in code. CVE-2015-5312 illustrates this case with Listing 2. The original fix checked `ctxt->lastError.code` after processing the entity content, whereas a later refactoring reported the same error by returning `NULL`. The agent continued to search for the original `lastError.code` checks and therefore missed the new implementation, where the return value is checked and the error path is taken accordingly. Later

Listing 1: Patch for CVE-2017-12448.

```
bfd_is_thin_archive(abfd)=strncmp(armag,
  ARMAGT,SARMAG)==0;

if(strncmp(armag,ARMAG,SARMAG)!=0 &&
   strncmp(armag,ARMAGB,SARMAG)!=0 &&

   !bfd_is_thin_archive(abfd)) {
/* Security-relevant behavior added by the patch. */

    bfd_set_error(bfd_error_wrong_format);

    if(abfd->format==bfd_archive)
        abfd->format = bfd_unknown;

    return NULL;
}
```

Listing 2: Patch for CVE-2015-5312.

```
/* Similar guard already exists */
if (ctxt->lastError.code == XML_ERR_ENTITY_LOOP ||
    ctxt->lastError.code == XML_ERR_INTERNAL_ERROR)
    goto int_error;

rep = xmlStringDecodeEntities(...);

/* Patch-specific guard */
if (ctxt->lastError.code == XML_ERR_ENTITY_LOOP ||
    ctxt->lastError.code == XML_ERR_INTERNAL_ERROR)
    goto int_error;

/* Later evolved implementation */
rep = xmlStringDecodeEntities(...);
if (rep == NULL)
    goto int_error;
```

revisions further moved the same safety logic into helper and cleanup routines. These failures show that patch evolution can preserve the fix while substantially changing the code patterns available for recognition.

5.4 Ubuntu Deployed Packages

Performance remains strong on deployed packages. Accuracy and DSR both reach 0.97, close to the stripped Source-Built Benchmark. Binutils, OpenSSL, and SQLite achieve perfect results, while most remaining errors occur on libxml2. Overall, Codex maintains similar effectiveness on Ubuntu-distributed binaries as on the source-built binaries.

> Stripping and build variation have limited impact on the agent. The main challenges are patch localization without symbols and recognizing substantially evolved fixes, while non-applicability is usually identified correctly.

6 RQ3: Factors Influencing Capability

6.1 Impact of Model Capability

We evaluate four model–reasoning configurations under the same Codex harness, metadata, tool stack, and budget. Table 5 shows that performance varies with model capability. T1 and T2 remain close, while Accuracy and DSR drop substantially at T3 and T4, together with more incorrect and inconclusive outcomes. These results confirm that model capability affects end-to-end patch presence testing. To identify the source of the capability gap, we review the tool-call traces of all incorrect and inconclusive cases and classify their failure mechanisms as shown in Table 4.

Table 4: Failure mechanisms across model tiers.

Failure mechanism	T1	T2	T3	T4
Patch localization failure				
Target function not found	0	0	7	20
Wrong patch site selected in target function	3	2	12	27
Patch site not found in target function	0	0	2	3
Semantic analysis failure				
Patch semantics misjudged	3	8	10	11
Applicability misjudged	2	2	0	2
No distinguishable binary evidence	4	2	1	0
Total	12	14	32	63

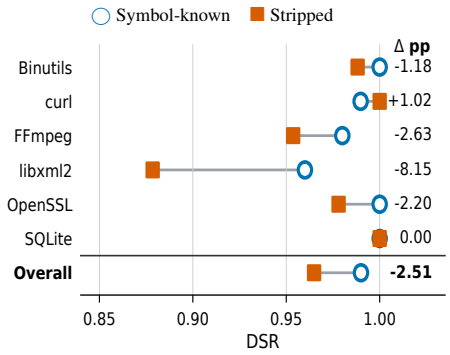


Figure 1: Symbol-removal effects on DSR.

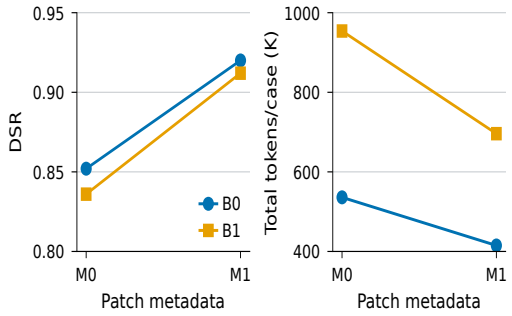


Figure 2: DSR and token usage.

6.1.1 Sources of the Capability Gap

Patch localization is the main source of the model capability gap. As shown in Table 4, patch localization accounts for only 25.0% and 14.3% of failures at T1 and T2, but rises to 65.6% and 79.4% at T3 and T4. We further examine cases where both T1 and T2 succeed but T3 or T4 fails, and find that 95.2% of these failures are attributable to patch localization. Most failures come from missing the target function or selecting the wrong patch site, showing that weaker models mainly struggle with patch localization.

Finding useful clues does not always lead to the correct patch site. CVE-2017-12448 in Binutils illustrates this problem. All four tiers recover the distinctive archive magic values. T1 and T2 use them to locate the archive probe at `0xa2f10` and verify the patched rejection branch at `0xa3068`. T3 and T4 instead follow the same clues to other archive-processing code at `0xa4a60` and `0x97530`. These locations handle the same magic values but do not contain the modified branch. This case shows that weaker models can find relevant clues but still select the wrong patch site.

Successful localization leaves few failures. Only three of the 62 failures occur after the patch site is correctly localized, where the model misinterprets local data flow or semantic effects. A separate limitation occurs when the source-level patch leaves no distinguishable evidence in the optimized binary, accounting for all six inconclusive outputs at T1 and T2 Zhang et al. (2026). At lower tiers, these cases are often preceded by localization failures.

6.2 Impact of Tools and Metadata

Experimental setup. To examine whether additional analysis support improves patch localization, we evaluate four configurations formed by two factors, patch metadata (M) and binary-analysis support (B). Subscripts 0 and 1 denote basic and enriched settings, with M_0+B_0 as the baseline. We use GPT-5.6 Terra high (T3), whose lower baseline performance provides room to measure these effects.

For metadata, M_0 contains the basic CVE information, while M_1 adds pre-analyzed patch information. We use Tree-sitter Brunsfeld & contributors (2026) to retain the modified statements and necessary control- and data-dependence context from patched functions. We use DeepSeek-V4-Pro DeepSeek-AI (2026) to derive the root cause, repair intent, behavioral changes, and binary-observable localization anchors. The preprocessing cost of enriched metadata is below $3 for all CVEs and is negligible relative to the per-binary agent analysis. All together, M_1 adds reduced source context, patch-semantic analysis, and localization anchors. For binary-analysis support, B_0 provides the default binutils utilities, while B_1 additionally exposes six read-only Ghidra interfaces for function localization, structural inspection, control-flow and data-flow analysis, and bounded decompilation Rohleder (2019). Appendix B describes these interfaces in detail.

Table 5: Model–system effectiveness and cost.

System	Tier	Acc.	DSR	Token usage (K/case)			USD/case
				Reasoning	Cache miss	Total	
Codex	T1	0.97	0.96	3.02	55.21	328.01	0.54
	T2	0.96	0.95	0.93	49.81	280.15	0.43
	T3	0.90	0.84	2.54	152.67	536.28	0.43
	T4	0.81	0.73	1.72	109.06	563.11	0.04
PPTAgent	T1	0.99	0.95	9.57	354.28	813.34	2.46
	T2	0.95	0.92	1.29	53.18	293.46	0.51
	T3	0.95	0.89	4.02	66.72	448.08	0.30
	T4	0.72	0.62	2.51	70.08	610.78	0.03
MiniSWEAgent	T1	0.99	0.90	7.73	88.58	622.46	1.04
	T2	0.96	0.96	1.05	43.37	165.32	0.37
	T3	0.93	0.92	2.60	49.33	278.59	0.20
	T4	0.84	0.83	1.90	54.38	390.59	0.02

6.2.1 RESULT ANALYSIS.

Enriched metadata improves both effectiveness and efficiency. Figure 2 reports DSR and token use for the four configurations. Compared with M_0+B_0, M_1+B_0 increases DSR by 0.07 while reducing tool calls and token usage by 17.7% and 22.6%, respectively. In contrast, B_1 increases resource use at both metadata levels without improving DSR. Overall, M_1+B_0 provides the best effectiveness–efficiency trade-off among the four configurations.

Ghidra introduces additional analysis burdens. To our surprise, enabling Ghidra under M_0 degrades 2 cases, while under M_1 it shows no improvement. Trace analysis reveals three causes: broad function localization can divert the agent to patch-irrelevant candidates, large summaries and decompiled views can obscure the relevant instruction region, and ambiguous outputs can lead to premature `inconclusive` decisions. CVE-2020-8284 illustrates the localization burden: under M_1+B_0, Codex links the flag write at `0x451f0` to its use at `0x5b300` and correctly returns `patched`; with B_1, Ghidra returns multiple candidate functions, but the agent fails to reconstruct this data-flow relation and returns `inconclusive`.

> Enriched patch metadata improves both DSR and efficiency, whereas additional binary-analysis tools increase resource use without improving DSR.

7 RQ4: COST-EFFECTIVE DEPLOYMENT

In this section, we evaluate 12 combinations of the four model configurations described in Section 3.5 and the three agent systems in Section 3.4 on the stripped Source-Built Benchmark. Following prior work on evaluating complete agent systems and their resource use Kapoor et al. (2024); Fan et al. (2025); Bandel et al. (2026), each agent uses its native harness configuration. We compare DSR and resource consumption across these combinations to identify practical deployment choices.

Among the evaluated systems, **PPTAgent** is designed specifically for patch presence testing. The model uses host-provided tools for read-only source guidance and sandboxed binary inspection. Each binary-inspection call produces an *observation* containing the executed analysis and raw output, from which the host records *evidence* items with stable identifiers. The model summarizes selected evidence into natural-language *claims* and cites them when submitting a verdict. A validation gate verifies the cited evidence and submission structure, and returns invalid submissions to the model for revision. This design keep s accepted verdicts traceable to the original binary observations. Table 5 reports effectiveness and resource usage for all 12 configurations.

Agent design changes how model capability translates into performance. The ranking of the three systems changes across model tiers. At T1, Codex achieves the highest DSR of 0.96, whereas MiniSWEAgent performs best at the lower model tiers and reaches 0.83 DSR even at T4. PPTAgent

achieves high accuracy but lower DSR, indicating more unsuccessful analyses. These results show that model capability alone does not determine agent performance.

Performance and cost favor different deployment choices. Codex T1 achieves the highest DSR of 0.961. MiniSWEAgent T2 reaches a comparable DSR of 0.956 at $0.37 per case and is the least expensive configuration above 0.95 DSR. Lower model tiers further reduce cost but also show reduced performance. Under these criteria, we recommend Codex T1 when maximum performance and MiniSWEAgent T2 when balancing effectiveness and cost.

> Codex with GPT-5.6 Sol max provides the highest decision success, while MiniSWEAgent with GPT-5.6 Sol medium provides the best cost–effectiveness trade-off above a 0.95 DSR threshold.

8 Related Work

8.1 Patch Presence Testing

Existing patch presence testing methods typically generate a patch signature from the fixing patch and vulnerable–patched reference binaries and match it against the target. Syntactic approaches compare instruction and control-flow features Xu et al. (2020; 2023), while semantic approaches use symbolic execution or intermediate representations to capture patch behavior Sun et al. (2021); Zhang & Qian (2018); Jiang et al. (2020); Zhan et al. (2024a;b); Yang et al. (2023). Recent work further incorporates LLM reasoning: Lares analyzes patch-related code slices within a supplied target-function range Li et al. (2025), while PPTSP uses an LLM to resolve ambiguous matches after semantic normalization and key-path extraction Xu et al. (2026). These methods operate within a known function or address range, whereas our work studies whether general-purpose coding agents can perform patch presence testing end to end, from patch understanding and localization to evidence collection and patch-status judgment.

8.2 Agent Systems and Evaluation

Coding agents place an LLM in an iterative loop with an execution environment, where the surrounding harness can substantially affect agent behavior. SWE-agent studies agent–computer interfaces, while Agentless examines more constrained software-engineering workflows Yang et al. (2024); Xia et al. (2025). AgentBench and AgentBoard extend evaluation to multi-turn agent behavior Liu et al. (2024); Ma et al. (2024). Recent studies further emphasize evaluating complete model–agent configurations together with their resource use and cost Kapoor et al. (2024); Fan et al. (2025); Bandel et al. (2026). Our study applies this perspective to patch presence testing by evaluating model and agent choices in terms of both DSR and deployment cost.

8.3 LLM-Assisted Binary Analysis

LLMs have been applied to decompilation, symbol recovery, and binary-to-source matching. DeGPT and LLM4Decompile improve or recover source-like representations from binaries, while ReSym recovers variable and data-structure information from stripped binaries Hu et al. (2024); Tan et al. (2024); Xie et al. (2024a). More recently, BinReX Liu et al. (2026) targets general-purpose static binary analysis. It enriches stripped functions with retrieved semantic context and uses hierarchical planning and generated IDAPython scripts to carry out and validate analysis tasks.

9 Conclusion

This study evaluates coding agents for end-to-end patch presence testing. Codex performs comparably to state-of-the-art patch presence testing systems with symbols and remains effective under realistic binary conditions. The main challenges are patch localization and patch evolution. Stronger models improve localization, while enriched patch metadata is more effective and efficient than additional binary-analysis tools. Evaluating 12 model–agent configurations further reveals practical trade-offs between performance and deployment cost.

AI USE STATEMENT.

Generative AI systems are central to the experimental methodology of this work. We use GPT-5.6 model configurations through the evaluated agent systems, and DeepSeek-V4-Pro for the enriched patch-metadata analysis described in Section 6.2. Any additional use of generative AI for research or manuscript preparation is disclosed in accordance with the ICLR 2027 AI policy. The authors reviewed the AI-assisted outputs used in this work and take responsibility for the final content.

REPRODUCIBILITY STATEMENT.

We provide the code, experimental configurations, prompts, and detailed results required to reproduce our experiments in the anonymous repository at `https://anonymous.4open.science/r/agenticPPT-Overview`.

REFERENCES

Anthropic. Effective harnesses for long-running agents. `https://www.anthropic.com/engineering/effective-harnesses-for-long-running-agents`, November 2025. Published November 26, 2025. Accessed 2026-09-10.

Artificial Analysis. Artificial analysis intelligence index. `https://artificialanalysis.ai`, 2026. Leaderboard snapshot of 2026-08-24.

Elron Bandel, Asaf Yehudai, Lilach Eden, Yehoshua Sagron, Yotam Perlitz, Elad Venezian, Natalia Razinkov, Natan Ergas, Shlomit Shachor Ifergan, Segev Shlomov, Michal Jacovi, Leshem Choshen, Liat Ein-Dor, Yoav Katz, and Michal Shmueli-Scheuer. General agent evaluation, 2026.

Max Brunsfeld and contributors. Tree-sitter. `https://github.com/tree-sitter/tree-sitter`, 2026. Accessed 2026-09-03.

DeepSeek-AI. Deepseek-v4 technical report. *arXiv preprint arXiv:2606.19348*, 2026.

Douglas Everson, Long Cheng, and Zhenkai Zhang. Log4Shell: Redefining the web attack surface. In *Proceedings of the 2022 Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb)*, 2022. doi: 10.14722/madweb.2022.23010.

Zhiyu Fan, Kirill Vasilevski, Dayi Lin, Boyuan Chen, Yihao Chen, Zhiqing Zhong, Jie M. Zhang, Pinjia He, and Ahmed E. Hassan. SWE-Effi: Re-evaluating software AI agent system effectiveness under resource constraints, 2025.

Peiwei Hu, Ruigang Liang, and Kai Chen. DeGPT: Optimizing decompiler output with LLM. In *Network and Distributed System Security Symposium*, 2024. doi: 10.14722/ndss.2024.24401.

Ling Jiang, Junwen An, Huihui Huang, Qiyi Tang, Sen Nie, Shi Wu, and Yuqun Zhang. BinaryAI: Binary software composition analysis via intelligent binary source code matching. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pp. 1–13, 2024. doi: 10.1145/3597503.3639100.

Zheyue Jiang, Yuan Zhang, Jun Xu, Qi Wen, Zhenghe Wang, Xiaohan Zhang, Xinyu Xing, Min Yang, and Zhemin Yang. Pdiff: Semantic-based patch presence testing for downstream kernels. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1149–1163, 2020. doi: 10.1145/3372297.3417240.

Sayash Kapoor, Benedikt Stroebl, Zachary S. Siegel, Nitya Nadgir, and Arvind Narayanan. AI agents that matter, 2024.

Zhe Lang, Shouguo Yang, Yiran Cheng, Xiaoling Zhang, Zhiqiang Shi, and Limin Sun. Pmatch: Semantic-based patch detection for binary programs. In *2021 IEEE International Performance, Computing, and Communications Conference (IPCCC)*, pp. 1–10, 2021. doi: 10.1109/IPCCC51483.2021.9679443.

Siyuan Li, Yaowen Zheng, Hong Li, Jingdong Guo, Chaopeng Dong, Chunpeng Yan, Weijie Wang, Yimo Ren, Limin Sun, and Hongsong Zhu. Lares: Llm-driven code slice semantic search for patch presence testing. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 919–931, 2025. doi: 10.1109/ASE63991.2025.00081.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. AgentBench: Evaluating LLMs as agents. In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/e9df36b21ff4ee211a8b71ee8b7e9f57-Abstract-Conference.html.

Yuzhe Liu, Zhijie Liu, Zhengmin Yu, Shu Wang, Ling Jiang, Sen Nie, Shi Wu, Zhanyong Tang, and Yuan Zhang. Towards generality: Task-adaptive binary analysis via semantic retrieval and verifiable reasoning. In *35th USENIX Security Symposium (USENIX Security 26)*, pp. 2783–2802. USENIX Association, 2026.

Chang Ma, Junlei Zhang, Zhihao Zhu, Cheng Yang, Yujiu Yang, Yaohui Jin, Zhenzhong Lan, Lingpeng Kong, and Junxian He. AgentBoard: An analytical evaluation board of multi-turn LLM agents. In *Advances in Neural Information Processing Systems*, volume 37, 2024. doi: 10.52202/079017-2365.

National Vulnerability Database. CVE-2021-44228. https://nvd.nist.gov/vuln/detail/CVE-2021-44228, 2021.

OpenAI. Codex. https://github.com/openai/codex, 2026. Accessed 2026-08-28.

Zhiyuan Pan, Xing Hu, Xin Xia, Xian Zhan, David Lo, and Xiaohu Yang. Ppt4j: Patch presence test for java binaries. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, 2024. doi: 10.1145/3597503.3639231.

Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. Vulnerable open source dependencies: Counting those that matter. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 1–10, 2018. doi: 10.1145/3239235.3268920.

Roman Rohleder. Hands-on Ghidra: A tutorial about the software reverse engineering framework. In *Proceedings of the 3rd ACM Workshop on Software Protection (WASP)*, pp. 77–78, 2019. doi: 10.1145/3338503.3357725.

Peiyuan Sun, Qiben Yan, Haoyi Zhou, and Jianxin Li. Osprey: A fast and accurate patch presence test framework for binaries. *Computer Communications*, 173:95–106, 2021. doi: 10.1016/j.comcom.2021.03.011.

SWE-agent Team. Mini-swe-agent. https://github.com/SWE-agent/mini-swe-agent, 2026. Accessed 2026-09-03.

Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. LLM4Decompile: Decompiling binary code with large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 3473–3487, Miami, Florida, USA, 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.203. URL https://aclanthology.org/2024.emnlp-main.203/.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, volume 235 of *Proceedings of Machine Learning Research*, 2024.

Xingyao Wang et al. Openhands: An open platform for ai software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations (ICLR)*, 2025.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying LLM-based software engineering agents. *Proceedings of the ACM on Software Engineering*, 2(FSE):801–824, 2025. doi: 10.1145/3715754.

Danning Xie, Zhuo Zhang, Nan Jiang, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. ReSym: Harnessing LLMs to recover variable and data structure symbols from stripped binaries. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security*, pp. 4554–4568. Association for Computing Machinery, 2024a. doi: 10.1145/3658644.3670340.

Zifan Xie, Ming Wen, Zichao Wei, and Hai Jin. Unveiling the characteristics and impact of security patch evolution. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1094–1106, New York, NY, USA, 2024b. Association for Computing Machinery. doi: 10.1145/3691620.3695488.

Zifan Xie, Ming Wen, Zichao Wei, and Hai Jin. Unveiling the characteristics and impact of security patch evolution. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, pp. 1094–1106. Association for Computing Machinery, 2024c. doi: 10.1145/3691620.3695488.

Chengke Xu, Senlin Luo, Xueming Duan, and Limin Pan. PPTSP: Patch presence test via semantic normalization and key path extraction. *Software Quality Journal*, 34(2):1–29, 2026. doi: 10.1007/s11219-026-09760-9.

Xi Xu, Qinghua Zheng, Zheng Yan, Ming Fan, Ang Jia, Zhaohui Zhou, Haijun Wang, and Ting Liu. Patchdiscovery: Patch presence test for identifying binary vulnerabilities based on key basic blocks. *IEEE Transactions on Software Engineering*, 49(12):5279–5294, 2023. doi: 10.1109/TSE.2023.3332732.

Yifei Xu, Zhengzi Xu, Bihuan Chen, Fu Song, Yang Liu, and Ting Liu. Patch based vulnerability matching for binary programs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 376–387, 2020. doi: 10.1145/3395363.3397361.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37, pp. 50528–50652, 2024. doi: 10.52202/079017-1601.

Shouguo Yang, Zhengzi Xu, Yang Xiao, Zhe Lang, Wei Tang, Yang Liu, Zhiqiang Shi, Hong Li, and Limin Sun. Towards practical binary code similarity detection: Vulnerability verification via patch semantic analysis. *ACM Transactions on Software Engineering and Methodology*, 32(6), 2023. doi: 10.1145/3604608.

Qi Zhan, Xing Hu, Zhiyang Li, Xin Xia, David Lo, and Shanping Li. Ps3: Precise patch presence test based on semantic symbolic signature. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pp. 1–12, 2024a. doi: 10.1145/3597503.3639134.

Qi Zhan, Xing Hu, Xin Xia, and Shanping Li. React: Ir-level patch presence test for binary. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 381–392, 2024b. doi: 10.1145/3691620.3695012.

Hang Zhang and Zhiyun Qian. Precise and accurate patch presence test for binaries. In *27th USENIX Security Symposium (USENIX Security 18)*, pp. 887–902. USENIX Association, 2018.

Xiaobei Zhang, Yaowen Zheng, Wu Luo, Shijun Zhao, Yongsheng Tao, Dan Meng, and Rui Hou. A comprehensive empirical analysis of patch presence testing: Capabilities, limitations, and paths forward. *Proceedings of the ACM on Software Engineering*, 3(ISSTA), October 2026. doi: 10.1145/3832144.

A Dataset and Metric Definitions

Table A.1: Datasets.

Dataset	Ground-truth composition	Arch.	Compiler	Opt.	Symbols	#CVEs	#Cases
Source-built	Vulnerable/patched pairs	x86-64/AArch64	GCC/Clang	O2/O3	Known/stripped	120	720
Applicability	`not_applicable` only	x86-64	GCC	O2	Stripped	13	58
Patch Evolution	Patched; fix later modified	x86-64	GCC	O2	Stripped	23	73
Ubuntu packages	Pre-fix/fixed package pairs	x86-64	Varies	Varies	Stripped	120	720

Table A.2: Evaluation metrics.

Term	**Abbr.**	**Definition**
True Positive	TP	Patched target judged `patched`
True Negative	TN	Vulnerable target judged `vulnerable`
False Positive	FP	Vulnerable target judged `patched`
False Negative	FN	Patched target judged `vulnerable`
Total Cases	TC	Number of attempted cases
Precision	P	$TP/(TP+FP)$
Recall	R	$TP/(TP+FN)$
F_1	F_1	$2PR/(P+R)$
Accuracy	A	$(TP+TN)/(TP+TN+FP+FN)$
Decision Success Rate	DSR	$(TP+TN)/TC$

The binary metrics in Table A.2 are computed only for cases with `patched` or `vulnerable` ground truth. Precision, Recall, F_1, and Accuracy are calculated from TP, TN, FP, and FN; `inconclusive` and `not_applicable` predictions are excluded. For the Applicability set, we report the proportion of cases correctly identified as `not_applicable`. DSR covers all attempted cases, so any prediction that differs from the ground truth, including `inconclusive` or an incorrect `not_applicable` prediction, is counted as unsuccessful.

B Ghidra-Based Binary Analysis Tools

We expose six read-only Ghidra capabilities to the agent through a local stdio MCP server, with each invocation bound to a single anonymized target binary. `ghidra_locate_function` ranks candidate functions using string, callee, constant, and field-offset anchors; `ghidra_function_summary` returns bounded function ranges, calls, branches, returns, constants, and raw instruction samples; `ghidra_cfg_slice` extracts basic blocks, control-flow edges, and instructions around a specified address; `ghidra_path_probe` checks bounded reachability between program points and reports required or forbidden instruction patterns; `ghidra_call_args` recovers call-site context, nearby argument-setup instructions, the resolved callee, and available P-code; and `ghidra_decompile_slice` provides a bounded recovered-pseudocode view together with a corresponding raw-instruction window. All responses are structured observations containing an observation identifier, sanitized `target_binary` paths, parsed facts, bounded evidence excerpts, truncation metadata, and error information. The tools reject arbitrary file paths, scripts, and expressions. Function location results are treated only as navigation hints, while determinate patch-presence decisions must be supported by raw instructions, control-flow, or P-code; decompiler output is advisory and cannot serve as the sole evidence.

C Binary Patch-Presence Analysis Prompt

The following prompt template is provided to the agent for each patch-presence analysis case. Case-specific fields are instantiated with the corresponding CVE metadata and target information.

Prompt Template for Binary Patch-Presence Analysis

As an authorized binary patch-presence investigator, analyze whether the specified target binary implements the security behavior described by the CVE metadata and determine whether the patch is `present`, `absent`, `not_affected`, or `inconclusive` based on binary evidence.

Investigation Scope

CVE:	`[CVE_ID]`
Project:	`[project]`
CWE:	`[CWE_ID]`
Relevant Functions:	`[function_names]`
Vulnerability Description:	`[vulnerability_description]`
Patch Commit Message:	`[patch_commit_message]`
Patch Hunk:	`[patch_hunk]`
Target Binary:	`/workspace/binary`

Evidence Rules

- Metadata is investigation guidance, not evidence.
- Use binary inspection results to support the decision.
- Every cited evidence item must be summarized before submission.
- A `present` or `absent` result must include decisive address ranges.
- If the security semantics cannot be established, return `inconclusive`.

Example Output

```
Status: [present / absent / not_affected / inconclusive]
Reasoning: [concise explanation based on binary instructions and control flow]
Decisive Addresses: [0x..., 0x...]
```