

CWEFT: CWE-Aware Evaluation of Free-Text vs. Typed Prompts

Lanaya A. Carbonell 

Department of Computer Engineering, Bilkent University, Ankara, Turkey

Anıl Koyuncu 

Department of Computer Engineering, Bilkent University, Ankara, Turkey

Abstract

Adding vulnerability context to a prompt is known to improve LLM repair quality, but the studies that establish this vary two things at once: the domain knowledge injected, such as a CWE label or a fix pattern, and the form that knowledge takes in the prompt. Prompt-engineering research treats format as a variable in its own right, yet in automated vulnerability repair it has never been isolated under a controlled ablation, leaving open whether the wording of guidance matters once its content is fixed. This paper presents CWEFT, a controlled ablation that holds injected repair knowledge constant across a free-text and a typed-schema rendering while decomposing the separate contributions of naming a vulnerability and supplying fix guidance. We construct four prompt conditions that differ by exactly one factor each, grounding all injected knowledge in APR4Vul’s empirically mined fix patterns. Using this design, we generate 504 patches across 42 Java vulnerabilities from Vul4J spanning 15 CWE classes on three frontier models – Claude Sonnet 4.5, GPT-5, and Gemini 2.5 Pro – and score each patch on a Proof-of-Vulnerability-aware hierarchy that separates code that compiles from code that closes the vulnerability and from code that does so without regression. Our emerging results invert a common assumption: naming the CWE class alone achieves the highest correct-fix rate at 19.8%, ahead of the 12.7% unenriched baseline ($p = 0.049$, exact McNemar), while both richer conditions that add explicit repair guidance fall back to 16.7%. Crucially, the free-text and typed-schema conditions are indistinguishable at 16.7% with perfectly symmetric paired discordance, and with only eight discordant pairs the design cannot rule out a smaller format effect. Across all conditions, repair rate tracks how concrete the vulnerability class is, falling to 3.1% for improper input validation. These initial findings suggest that for frontier models on common vulnerability classes, what a prompt identifies matters more than how elaborately that information is formatted – a direction that warrants higher-powered replication before it is treated as a design principle.

2012 ACM Subject Classification Software and its engineering → Software defect analysis; Security and privacy → Software security engineering; Computing methodologies → Natural language processing

Keywords and phrases Automated vulnerability repair, Automated program repair, Large language models, Prompt engineering, Structured prompting, Software security, Vul4J

Digital Object Identifier 10.4230/LIPIcs.ESEM.2026.78

Category Emerging Results, Vision & Reflection Track Paper

Supplementary Material

Software (Replication Package): <https://doi.org/10.5281/zenodo.21672348>

1 Introduction

Software vulnerabilities are disclosed faster than developers can patch them by hand, sustaining research interest in automated vulnerability repair (AVR) and, more recently, in LLM-based repair. A growing body of work shows that adding vulnerability context to a model’s prompt raises repair quality [14, 10, 7]. Yet these studies conflate two variables:



© Lanaya A. Carbonell and Anıl Koyuncu;

licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).

Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;

Article No. 78; pp. 78:1–78:14



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

what domain knowledge is added, such as a CWE label or a fix pattern, and *how* that knowledge is rendered into the prompt, whether as a sentence, a list, or a typed schema. The prompt-engineering literature treats format as an experimentally meaningful variable in other tasks [19, 8], but in AVR specifically the question has not been tested under a controlled ablation.

CWEFT treats representational format as a controlled variable. It holds injected fix knowledge identical across a free-text condition and a typed-schema condition, and adds two further conditions, an unenriched baseline and a CWE-named-only condition, that decompose the separate contributions of naming a vulnerability and supplying fix guidance. The injected knowledge is grounded in real repair behavior: we mine it directly from APR4Vul’s empirical fix patterns [4]. Because adjacent conditions differ by exactly one factor, any outcome difference traces to that factor alone. We evaluate the ablation on 42 Java vulnerabilities from Vul4J across 15 CWE classes, generating 504 patches with three frontier models from three providers, and we score each patch on a Proof-of-Vulnerability-aware trustworthiness hierarchy that separates code that compiles from code that closes the vulnerability and from code that does so without breaking other tests.

We organize the study around four research questions:

- RQ1** To what extent does enriching prompts with vulnerability-type-specific fix knowledge improve patch quality over an unenriched baseline?
- RQ2** To what extent does the representational format (free-text vs. typed schema) of that knowledge affect patch quality when content is held constant? (*central question*)
- RQ3** How does single-shot LLM repair compare across three frontier models?
- RQ4** Under a multi-level trustworthiness hierarchy, how does repair rate vary by vulnerability class?

This paper makes three contributions:

1. A controlled ablation that separates the *content* of injected vulnerability knowledge from its *representational format*, holding source content fixed across a free-text and a typed-schema condition so that the two factors can be attributed separately.
2. An empirical finding that runs against common prompt-engineering practice: naming the CWE class alone outperforms both richer conditions built from the same repair knowledge, and it is also the cheapest enrichment to run.
3. A fully open replication artifact covering prompt construction, patch generation, and Proof-of-Vulnerability-aware evaluation, with all 504 per-cell outcomes released.

2 Related Work

AVR and LLM-based repair. AVR sits inside a broader APR landscape that recent surveys taxonomize as template-guided, search-based, constraint-based, and learning-driven [9]. The most directly relevant prior work is APR4Vul, which evaluated traditional APR techniques on Vul4J and found testable patches for only 20% of bugs with correctness rates under 50%, while also mining the recurring fix patterns behind real human patches [4]. Those mined patterns, among them input sanitization, canonical-path verification, and secure parser configuration, are the direct source of the repair knowledge we inject in our C3a and C3b conditions. A parallel line of LLM-based work has established that prompt content affects repair quality [7, 11, 14, 6, 17]. PromptFix is closest in framing, since it augments prompts with CWE and CVE context, yet it delivers all of that context as unstructured prose [14]. Most pointedly, Khan et al. showed that progressively injecting domain knowledge about a vulnerability lifted GitHub Copilot’s buffer-overflow repair rate from 15% to 63% [10]. Their

result establishes that content can help a great deal; what stays open is whether returns continue as the guidance grows more elaborate, and whether its *form* matters once content is fixed. Recent systems such as APPATCH and chain-based repair pursue stronger results through agentic or multi-step pipelines [16, 15, 12], a direction orthogonal to ours: we hold the pipeline fixed and vary only how knowledge is represented. Separately, prior work found that pre-trained model choice and tokenization substantially affect AVR performance [23], which is why we evaluate three frontier models from different providers rather than rely on one.

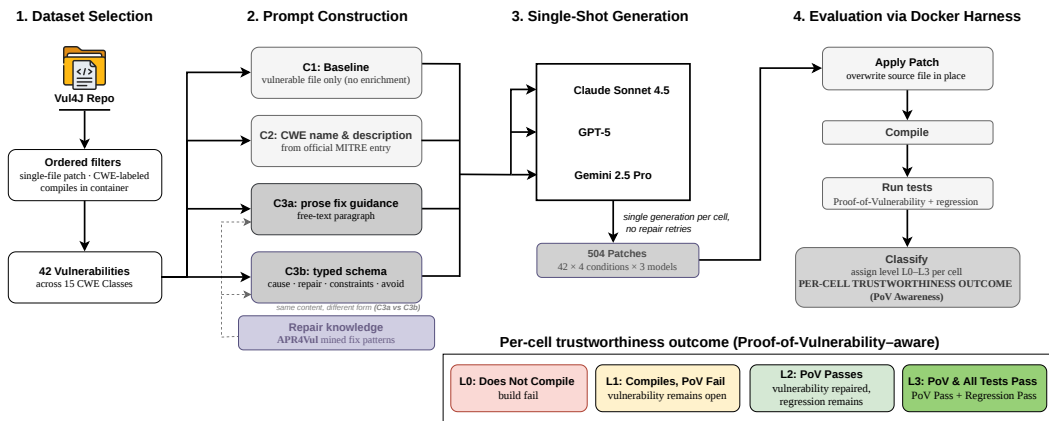
Prompt format as a variable. The prompt-engineering literature treats format as an experimentally meaningful variable independent of content. Brown et al. introduced few-shot in-context learning and showed that how examples are framed within a prompt changes task performance [3]. Wei et al. showed that chain-of-thought structure, which inserts intermediate reasoning steps, improves reasoning-heavy tasks even when no new information is added [22]. More recent work quantifies sensitivity to surface features: Sclar et al. found that separator choice and field ordering swing LLM performance as much as switching models [19], and He et al. replicated the effect on a separate task suite [8]. Framing effects extend to security-specific tasks: Bosnak et al. found that persona and phrasing choices substantially change how reliably open-source models produce a requested CWE, and read this as evidence that models rely on pattern recall over semantic reasoning about vulnerability classes [2]. The evidence is not one-sided in security tasks, however. A recent survey notes that structured strategies such as chain-of-thought improve some vulnerability-detection metrics but show inconsistent effects across settings rather than a uniform gain [20]. This mixed prior evidence is exactly what motivates testing format as a controlled variable in repair, instead of assuming structure helps.

Evaluation of LLM repairs. Measuring repair quality is itself contested, and the choice of metric shapes what a study can claim. Qi et al. analyzed plausible-but-incorrect patches across several APR systems and found that most reported patches amounted to functionality deletion rather than correct repair [18]. The same concern carries into the LLM era: recent evaluations report that models often misidentify or fail to genuinely fix vulnerabilities even when their output looks plausible [21, 1]. We take this seriously in our scoring rather than treating a passing test suite as success. Our trustworthiness hierarchy is Proof-of-Vulnerability-aware: it separates patches that merely compile from those that close the vulnerability, and from those that close it without regressing the rest of the suite, which guards against the functionality-deletion failure mode Qi et al. identified. We return to the limits of this scoring in Section 6.

3 Methodology

The pipeline has four stages: dataset selection, prompt construction, patch generation, and automated evaluation in a reproducible environment. Figure 1 gives a visual overview, and the subsections below describe each stage.

Dataset. We draw our subjects from Vul4J [5], a benchmark of real Java CVEs that ships reproducible build and test environments. We selected Vul4J over other Java vulnerability datasets because it pairs every entry with both a CVE and a CWE label and provides an executable build-and-test harness; alternatives such as CWE-Bench-Java [13] supply



■ **Figure 1** The CWEFT pipeline. Each of 42 vulnerabilities is processed under all four prompt conditions with three frontier models, producing 504 patches. Each model returns a complete repaired source file, which is compiled and tested in a reproducible Docker harness; outcomes are classified into the four levels of our Proof-of-Vulnerability-aware trustworthiness hierarchy.

■ **Table 1** CWE distribution across the 42 selected Vul4J vulnerabilities, sorted by frequency. Classes with two or more vulnerabilities anchor per-class observation; the remainder appear as singletons and probe the long tail.

CWE	Description	Count
CWE-20	Improper Input Validation	8
CWE-835	Infinite Loop	7
CWE-22	Path Traversal	5
CWE-79	Cross-Site Scripting	5
CWE-611	XML External Entity (XXE)	4
CWE-264	Permissions and Access Controls	2
CWE-287	Improper Authentication	2
CWE-502	Deserialization of Untrusted Data	2
7 additional CWE classes (one each)		7
<i>Total</i>		42

CWE labels and build scripts but no per-project test harness, since they target vulnerability detection rather than test-based repair validation; our evaluation requires executable Proof-of-Vulnerability and regression tests. From Vul4J we derive a working set through an ordered chain of filters, each demanded by a requirement of our design. First, we keep only vulnerabilities whose reference human patch modifies a single file, because our task asks a model to return one repaired file. Second, we keep only entries carrying a CWE label, since the C2, C3a, and C3b conditions cannot be built without one. Third, we require that the vulnerable revision compile cleanly in our container before any patch is applied; this step removes VUL4J-15, -23, -35, and -39, the first three of which the Vul4J maintainers themselves flag as non-compilable or needing manual modification, and the fourth of which fails to build in our environment despite its upstream status. Applying the first two filters yields 46 candidates; the buildability filter removes four of these, leaving the 42 vulnerabilities we study, spanning the 15 CWE classes shown in Table 1. Selection was performed once, ahead of generation, and the resulting subset is released as an explicit list rather than as a re-runnable script, together with the buildability record behind the final four exclusions.

■ **Listing 1** Example C3a (free-text) and C3b (schema) for CWE-22 Path Traversal. Both encode identical repair knowledge drawn from APR4Vul’s mined fix patterns; only the form differs.

```
# C3a (free-text):
Path traversal vulnerabilities are typically fixed by
canonicalizing user-supplied paths with
File.getCanonicalPath() and verifying the result
remains within an allowed base directory before any
file operation.

# C3b (schema):
Root Cause      : Untrusted path fragment used in
                  file resolution without
                  normalization.
Canonical Repair: Canonicalize via
                  getCanonicalPath() and verify
                  startsWith(base).
Constraints     : Preserve original method signature;
                  do not alter exception types.
What to Avoid   : String.contains("..") checks; these
                  bypass via encoded variants.
```

Prompt construction. The four conditions form a controlled ablation in which each condition adds exactly one factor over the previous one. This is what licenses the study’s central claim: a difference between two adjacent conditions can be attributed to the single factor that separates them. Every condition shares the same input and the same output instruction. The input is the complete vulnerable source file together with its path; the instruction asks the model to return the complete modified file as code only.

C1 (Baseline) provides the vulnerable file with a generic repair instruction and no vulnerability-specific information. It measures the model’s intrinsic ability to recognize and fix the flaw from the code alone, and serves as the floor against which enrichment is measured.

C2 (CWE-Named) adds the CWE identifier and a short description of the class, taken from the official MITRE CWE entry for that identifier. Comparing C2 with C1 isolates the effect of *naming the vulnerability type*, independent of any repair guidance.

C3a (Free-Text Fix Guidance) adds a short prose paragraph describing how the class is typically fixed in practice. We wrote this guidance by hand for each class, drawing its content from APR4Vul’s empirical fix-pattern tables so that it reflects real human repair behavior rather than invented instructions [4]. Comparing C3a with C2 isolates the effect of *fix-pattern knowledge delivered as prose*.

C3b (Typed Schema) carries the same underlying knowledge as C3a, reorganized into four labeled fields: *Root Cause*, *Canonical Repair*, *Constraints*, and *What to Avoid*. The fields map onto APR4Vul’s findings: root cause from its fault taxonomy, canonical repair from its fix-pattern tables, and the two constraint fields from its failure analysis. Comparing C3a with C3b is the study’s critical contrast: source content and surrounding instruction are matched and only the form changes, although the field labels themselves are part of C3b’s form and convey a kind of structural cue that C3a does not. Listing 1 shows the two renderings for one class.

Patch generation. Three frontier models from different providers generate the patches: Anthropic Claude Sonnet 4.5, OpenAI GPT-5, and Google Gemini 2.5 Pro, each accessed through its public API. Drawing from three providers lets us check whether any format effect generalizes across model families rather than reflecting one provider’s training or inference behavior. Each model receives the complete vulnerable file and returns a complete candidate

file in a single call. We use single-shot prompting with no repair-retry loops, self-consistency sampling, or tool-use feedback. This is a deliberate constraint: retry and voting strategies produce gains of their own, and admitting them would confound the format comparison this study exists to make. The absolute correct-fix rates we report are therefore a floor on what these models can reach, not a ceiling.

Experimental configuration. We report the generation configuration in full so that the run can be reproduced. The three models were accessed through their providers' public APIs under the identifiers `claude-sonnet-4-5-20250929`, `gpt-5`, and `gemini-2.5-pro`, which were the frontier offerings of their respective providers at the time of collection. All 504 cells were generated over 27–28 May 2026, strictly sequentially, one call at a time, with no concurrency and no client-side rate cap. The system prompt is identical across the three providers. We passed neither a temperature nor a `top_p` value to any provider, so each model sampled at its own default; we fixed no random seed and enabled no provider-side prompt caching, so generation is not deterministic across re-runs. Output budgets were not matched: Claude and GPT-5 were capped at 8192 output tokens, while Gemini was allowed 32768 output tokens together with a 512-token thinking budget. Transport failures were retried up to three times with exponential backoff (4s, then 8s); these retries respond to API exceptions alone and never resample a response that returned successfully, so the single-shot repair design is preserved. One provider-specific exception applies: our Gemini client raises on an empty response body, so an empty or blocked Gemini return is regenerated where the equivalent Claude or GPT-5 return would not be. Before a returned file is written to disk we strip Markdown code fences from the response; no other post-processing is applied. Every call is logged with its prompt, raw response, model identifier, input and output token counts, and condition.

Patch application. Because every condition asks the model for a complete file, application is direct: we take the returned file and write it in place of the original, with no method-level extraction and no import reconstruction. This whole-file design keeps the application step identical across all four conditions, so no condition is advantaged by how its output is reassembled. The only exception is the Markdown fence stripping noted above, which is applied identically to every response.

Compilation and testing. We compile and test each patched project through the Vul4J CLI inside a Docker container, which standardizes per-project build configuration. Every build and test action is issued through the Vul4J wrapper (`vul4j compile` and `vul4j test`); we never invoke Maven or Gradle directly, so no project receives special build handling. A 900-second bound guards each compile-and-test cycle against runaway dependency resolution; because each patch is generated in a single shot rather than by an iterative repair tool, we do not need the multi-hour repair-process budgets that search-based APR requires [4]. For every patch we record whether compilation succeeded, whether the Proof-of-Vulnerability test passed, and the names of any regression tests that failed; these three fields drive the classification below.

Trustworthiness hierarchy. We score each patch on a four-level hierarchy adapted from APR4Vul and made Proof-of-Vulnerability-aware, so that closing the vulnerability is scored separately from merely passing tests. **L0** marks a patch that does not compile. **L1** compiles but fails the Proof-of-Vulnerability (PoV) test, so the vulnerability remains open. **L2** passes

the PoV test, closing the vulnerability, but leaves one or more regression tests failing. **L3** passes the PoV test and the full regression suite, and is our definition of a correct fix. For each vulnerability we obtain the PoV test selector from Vul4J’s own metadata, using the explicit `failing_tests` field when present and otherwise parsing the `-Dtest` clause of the recorded test command. These are the tests the benchmark maintainers certify as failing on the vulnerable revision and passing on the human patch, which makes them the vulnerability-witnessing tests for each entry. Executing the PoV explicitly is what separates a patch that genuinely closes the hole from one that merely keeps tests green, the failure mode that inflates plain test-passing rates [18]. APR4Vul’s two highest levels require manual judgement of exploit removal and intent preservation; we hold those out of scope and revisit the limitation in Section 6.

4 Results

The experiment produced 504 patches: 42 vulnerabilities under four conditions across three models. Every cell generated, compiled, tested, and classified without pipeline error, so all 504 enter the analysis. Because the hierarchy is Proof-of-Vulnerability-aware, each patch is scored not only on whether it compiles but on whether it closes the vulnerability and whether it does so without regressing the rest of the suite. Throughout, “correct” means L3: the PoV test passes and the full regression suite passes.

Overall outcomes. Table 2 gives the distribution across all 504 cells. Compilation is not the dominant obstacle: 77.2% of patches compile. The harder wall is semantic. Only 22.6% of patches make the PoV test pass, and 16.5% are fully correct. The distance between a 77.2% compile rate and a 16.5% correct-fix rate locates the real difficulty of LLM-based repair past the point of producing code that builds.

■ **Table 2** Overall trustworthiness distribution across all 504 cells.

Level	Meaning	Count (%)
L0	Does not compile	115 (22.8)
L1	Compiles, PoV fails	275 (54.6)
L2	PoV passes, regressions remain	31 (6.2)
L3	PoV passes, all tests pass	83 (16.5)
Compile rate (L1+L2+L3)		77.2%
PoV-pass rate (L2+L3)		22.6%
Correct-fix rate (L3)		16.5%

RQ1: How much enrichment improves patch quality. Enrichment does not improve correctness in proportion to its detail. Table 3 reports outcomes by condition. The baseline (C1) reaches 12.7% L3. Naming the CWE class (C2) is the single best condition at 19.8%, yet the two richer conditions that add explicit fix guidance, free-text (C3a) and schema (C3b), both fall back to 16.7%. The largest gain therefore comes from simply telling the model which vulnerability class it faces; layering on detailed repair guidance adds nothing further and, on this sample, slightly reduces the rate. The two contrasts behind this paragraph deserve different treatment. For C1 vs. C2, a paired McNemar test on the 126 (vuln, model) pairs gives 13 cases where only C2 succeeded against 4 where only C1 succeeded (with 12 pairs correct under both, reconciling the 16 and 25 marginal totals in Table 3), for an exact two-sided $p = 0.049$. In magnitude, naming the CWE class raises the correct-fix rate by 7.1

■ **Table 3** Outcomes by condition ($n = 126$ each). L3% is the correct-fix rate; PoV% is L2+L3; Compile% is L1+L2+L3. Bold marks the best value in each column.

Condition	L0	L1	L2	L3	L3%	PoV%	Comp%
C1 (baseline)	31	72	7	16	12.7	18.3	75.4
C2 (CWE name)	24	68	9	25	19.8	27.0	81.0
C3a (prose)	25	71	9	21	16.7	23.8	80.2
C3b (schema)	35	64	6	21	16.7	21.4	72.2

percentage points, from 12.7% to 19.8%. The margin is narrow – one reclassified pair would push the test above the conventional threshold – so we treat the direction as reliable and the size as suggestive. The C2-vs-C3a/C3b margin is too small to test at this scale, as the C3a-vs-C3b McNemar in RQ2 already shows; we report it as suggestive only and return to both in Section 5.

RQ2: How much representational format matters. The central contrast is C3a against C3b: identical repair knowledge rendered in two different forms. Both conditions achieve a 16.7% correct-fix rate (21 of 126), showing that the injected knowledge remains robust to representational choice at the scale of our experiment. The paired outcomes reinforce this parity: among the eight discordant pairs pooled across the three models, four favor prose and four favor schema, and this even split holds even though the three models respond to enrichment along visibly different curves (RQ3). The observed difference is therefore 0.0 percentage points, and a paired McNemar test detects no directional advantage for either representation ($p = 1.000$, two-sided exact). Across standard interval methods for paired proportions, a 95% bound on that difference is approximately ± 4.5 percentage points. At this sample size the design therefore excludes a format effect larger than roughly four and a half points in either direction, and says nothing about smaller ones.

■ **Table 4** McNemar test, C3a vs. C3b, success = L3. *b*: only C3b correct; *c*: only C3a correct. Two-sided exact p .

Group	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	disc.	<i>p</i>
Claude	5	0	2	35	2	0.500
Gemini	6	3	2	31	5	1.000
GPT	6	1	0	35	1	1.000
Pooled	17	4	4	101	8	1.000

RQ3: Comparison across models. The three models differ both in overall success and in how they respond to enrichment. Table 5 reports outcomes per model. Gemini 2.5 Pro is the strongest single model on this benchmark at 22.0% correct, with the highest compile rate at 84.5%, ahead of Claude Sonnet 4.5 at 14.3% and GPT-5 at 13.1%. Their enrichment profiles also diverge: GPT-5 improves close to monotonically as context is added, Gemini spikes sharply at C2 and then declines, and Claude peaks at C2 and C3a before dropping at C3b. These are three distinct response shapes, yet, as RQ2 showed, the prose and schema conditions still converge to the same pooled correct-fix rate. The format result is therefore not an artifact of one model’s behavior.

RQ4: Repair rate by vulnerability class. Correct-fix rate varies sharply across CWE classes (Table 6), and the variation tracks how concrete the class is. Classes that name a specific, recognizable repair pattern are repaired often: improper authentication (CWE-287) at 37.5%,

■ **Table 5** Outcomes by model ($n = 168$ each). Bold marks the best value in each column.

Model	L0	L1	L2	L3	L3%	PoV%	Comp%
Claude Sonnet 4.5	44	89	11	24	14.3	20.8	73.8
GPT-5	45	89	12	22	13.1	20.2	73.2
Gemini 2.5 Pro	26	97	8	37	22.0	26.8	84.5

XXE (CWE-611) at 25.0%, deserialization (CWE-502) at 25.0%, infinite loop (CWE-835) at 21.4% across its 84 cells, the largest of the well-repaired classes, and path traversal (CWE-22) at 20.0%. Broad, abstract classes are repaired rarely or never: improper input validation (CWE-20) at 3.1%, and the access-control, cryptographic, privilege-management, and data-processing classes at 0%. CWE-20 is both the largest class, at 96 cells, and one of the lowest-scoring, which drags the overall average down. The high end of the table rests on individual vulnerabilities: the 91.7% for CWE-74, for instance, comes from a single injection bug seen across all conditions and models. We therefore read this as a gradient over *classes* rather than a precise per-class rate, and treat the singleton-backed entries as suggestive. The pattern fits RQ1: enrichment helps when the class points the model at a concrete repair, and stops helping when the class is too generic to act on.

■ **Table 6** Correct-fix (L3) rate by CWE class. Multi-instance classes anchor the gradient; single-vulnerability classes ($n_{\text{cells}} = 12$) are suggestive only and listed separately.

CWE	Description	n	L3%
<i>Multi-instance classes (≥ 2 vulnerabilities)</i>			
CWE-287	Improper authentication	24	37.5
CWE-611	XML external entity (XXE)	48	25.0
CWE-502	Deserialization	24	25.0
CWE-835	Infinite loop	84	21.4
CWE-22	Path traversal	60	20.0
CWE-79	Cross-site scripting	60	15.0
CWE-20	Improper input validation	96	3.1
CWE-264	Permissions / access control	24	0.0
<i>Single-vulnerability classes ($n = 12$, suggestive only)</i>			
CWE-74	Injection	12	91.7
CWE-532	Sensitive info in log	12	16.7
CWE-77	Command injection	12	8.3
CWE-284	Improper access control	12	0.0
CWE-310	Cryptographic issues	12	0.0
CWE-269	Improper privilege management	12	0.0
CWE-19	Data processing errors	12	0.0

Cost of enrichment. Because an approach that is accurate but unaffordable is of limited practical use, we report what this study cost to run. Generating all 504 patches cost approximately \$70 in API charges at the prices in force during collection: roughly \$30 for GPT-5, \$30 for Claude Sonnet 4.5, and \$10 for Gemini 2.5 Pro, or about \$0.14 per patch. Evaluation adds only local compute. The four conditions are nested by construction: every prompt carries the complete vulnerable source file, C2 adds a CWE identifier and its MITRE description, C3a adds a prose paragraph, and C3b adds a four-field block. Prompt length therefore rises monotonically from C1 to C3b, making C3b the most expensive condition to run. Correct-fix rate does not follow that ordering: it peaks at C2, the cheapest of the three enriched conditions, and is flat across C3a and C3b (Table 3). The cheapest enrichment we tested is also the best-performing one. Two qualifications keep this in proportion. The

vulnerable file dominates every prompt, so the absolute cost difference between conditions is small; the point is which enrichment earns its place, not that a large saving is available. And we report cost for the run as a whole rather than per condition, since per-cell token counts are not part of the released artifact.

Classifier validation. Because the L0–L3 verdicts drive every result above, we manually audited cells spanning the riskiest classification boundaries. For one XXE patch scored L3, the model disabled external-entity expansion on the input factory, matching the root-cause fix of the human patch. For one deserialization patch scored L3, the model enabled secure processing and disabled external DTD and stylesheet loading, a broader hardening than the human patch, with all tests still passing. For one authentication patch scored L2, the model closed the authentication hole, so the PoV test passed, but dropped a containment check and regressed more than thirty tests; the classifier correctly recorded L2 rather than L3. The audit supports treating the automated verdicts as faithful.

5 Discussion

A bare label beat elaborate guidance. The clearest result of the study is also its most useful: naming the vulnerability class outperformed every richer condition. C2 reached 19.8% correct against 16.7% for both fix-guidance conditions, so a single line of classification did more than a paragraph of repair instructions or a four-field schema built from the same knowledge. This inverts the working assumption of much prompt-engineering practice, where more context is presumed to help. Two mechanisms fit. First, frontier models already hold CWE-level repair knowledge from training, having seen many path-traversal or XXE fixes, so a label is enough to retrieve the right pattern while added guidance is redundant. Second, the extra text lengthens the prompt and pushes the code further from the instruction, which can dilute attention on the file being repaired. We hold the size of the effect loosely, since the C2 lead is 25 versus 21 correct fixes out of 126, but the direction is consistent and runs against the field’s default, which is what makes it worth reporting. We stress what this does not license: the result is a direction worth testing, not a design principle, and it should not be read as showing that naming the CWE class is sufficient. Adopting it as practice would require a higher-powered replication than an emerging-results study can provide.

Repair tracks how concrete a class is. The by-class results sharpen the same point on the full 504 cells: correct-fix rate forms a clear gradient from concrete classes (authentication, XXE, deserialization, path traversal) down to broad ones (CWE-20 at 3.1%, the access-control and cryptographic classes at zero). A concrete class points the model at a canonical fix; a broad class such as CWE-20 names a symptom with no single repair shape to retrieve.

Equivalent knowledge was robust to representational choice. Within the enrichment story sits the question we set out to isolate: does the form of the guidance matter when its content is fixed? In our experiment, equivalent repair knowledge proved robust to two distinct representations. Free-text guidance and a typed schema each produced a 16.7% correct-fix rate, while the paired outcomes were symmetric: four discordant cases favored prose and four favored schema. This parity is practically useful. A structured schema can be easier to generate consistently, validate automatically, and integrate into a repair pipeline, yet it matched the correctness of hand-written prose without an observed penalty. We remain careful about the scope of the claim: with only eight pooled discordant pairs, our interval

excludes a format effect larger than about 4.5 percentage points but leaves anything smaller open, and says nothing about other settings. The supported conclusion is narrower: for the frontier models, vulnerability classes, and schema design tested here, repair guidance transferred cleanly across representations.

Next steps. Five follow-ups would extend these findings. First, a higher-powered replication on more vulnerabilities would resolve whether the prose-schema null holds once the discordant sample is large enough to detect a small effect, and would re-establish the ordering on whatever models are current at the time; the full design costs roughly \$70 to run end to end, so it is cheap to repeat as models advance. Second, extending the enrichment comparison to rarer, domain-specific CWE classes would test whether guidance pays off where a model’s prior knowledge is thinner, since our set leans toward common classes. Third, promoting our automatic L0–L3 hierarchy toward APR4Vul’s security-verified levels through manual exploit-replay on a sample of L3 patches would test whether passing the PoV test corresponds to genuinely removing the exploit, the construct-validity limit we note in Section 6. Fourth, richer structural context – architecture summaries or code property graphs – would vary the content axis this study deliberately holds fixed, and is complementary to the format question rather than a substitute for it. Finally, extending the comparison to open-weight models would test whether these patterns hold outside proprietary APIs, since pre-trained model choice is known to affect AVR performance [23].

6 Threats to Validity

Internal validity. We generate one patch per cell with no repeated sampling, and we passed no temperature to any provider, so each model sampled at its own default rather than at a common fixed value (Section 3). Because those defaults differ across providers, the three models were not sampled under matched decoding settings, which limits exact replication and weakens the cross-model comparison in RQ3; a higher-powered replication should fix a single common temperature across all three. Because model output is stochastic, an individual cell could land differently on another run, though pooled rates over 504 cells should be far more stable than any single cell; we chose a single trial per cell to keep the design tractable for an emerging-results study, and we treat per-cell verdicts as evidence in aggregate rather than individually. Two further asymmetries qualify RQ3 in the same direction. Our output-token budget was four times larger for Gemini than for Claude and GPT-5, so a long repaired file could be truncated under the smaller budget and recorded as non-compiling. And because our Gemini client treats an empty response body as an exception, empty Gemini returns were regenerated while equivalent Claude and GPT-5 returns were not. Both asymmetries favour Gemini, the strongest model in our results, so the per-model ordering in Table 5 warrants more caution than the per-condition comparisons, which are unaffected: within any one model, all four conditions ran under identical settings. Our L0–L3 verdicts also rest on Vul4J’s own metadata correctly identifying the vulnerability-witnessing test for each entry; we rely on the maintainers’ curation and do not independently re-derive each Proof-of-Vulnerability test from the CVE. Finally, the classifier that assigns trust levels was checked by a manual audit of cells at the riskiest boundaries rather than an exhaustive review of all 504, so the audit supports the verdicts without certifying every one.

External validity. The study covers a single language (Java), a single benchmark (Vul4J), and three specific frontier models, so generalization to other languages, to smaller or open-weight models [23], or to vulnerability classes outside our set is not established. The three

models were the frontier offerings of their respective providers when the data was collected in May 2026, and more capable models have since been released. What the study reports is the relative ordering of prompt conditions under a controlled ablation, not absolute repair rates, and that ordering is the object of interest rather than any particular model’s ceiling. The buildability filter also removed four entries (VUL4J-15, -23, -35, -39); the maintainers themselves flag the first three as non-buildable, while the fourth builds upstream but not in our container, an environmental quirk that bounds the working set. Within the 42 vulnerabilities we study, the class distribution is uneven: a few classes such as CWE-20, CWE-835, CWE-22, and CWE-79 supply most cases, while several classes rest on a single vulnerability seen across twelve cells, so the per-class rates for those small classes are suggestive rather than firm. The format comparison is similarly limited in power: with only eight discordant pairs pooled, the null McNemar result means no effect was detectable at this scale, not that no effect exists.

Construct validity. We operationalize a correct fix as passing the Proof-of-Vulnerability test and the full regression suite. Qi et al. have shown that test-based correctness can overstate true repair quality [18]: a patch may pass the available tests yet leave some exploitable behavior open or regress an untested path. We do not perform manual security verification, so APR4Vul’s two highest levels, which judge exploit removal and intent preservation by hand, remain out of scope and are listed in Section 5 as future work. A further construct assumption is specific to our central comparison. We operationalize C3a and C3b as matched-source encodings drawn from the same APR4Vul fix patterns, written by one author rather than generated automatically and without an inter-rater check on prose quality. Variance across the 15 classes is bounded by construction, since both renderings for a class derive from the same source rows, but we cannot rule out that some classes received sharper guidance than others; a sensitivity analysis on prose quality is left to future work. Strict informational equivalence is not measured: C3b’s field labels (Root Cause, Canonical Repair, Constraints, What to Avoid) are themselves a structural cue that C3a does not contain. A reviewer could therefore read our null result as evidence that this label-level asymmetry is also not detectable at scale, rather than as evidence that form is irrelevant in general.

7 Conclusion

Across 504 patches spanning 42 Java vulnerabilities, 15 CWE classes, and three frontier models, three findings emerge. First, naming the CWE class improves repair performance over an unenriched baseline and achieves the highest observed correct-fix rate among the four conditions. Adding more elaborate prose or schema guidance does not produce a further observed improvement. Second, repair success varies sharply across vulnerability classes. The descriptive pattern suggests that specific classes with recognizable repair templates are easier to repair than broader classes. Third, the free-text and typed-schema conditions reach an identical 16.7% correct-fix rate with symmetric McNemar discordance. We interpret this as no detectable format effect at the scale of our experiment, not as proof that representation never matters. For frontier models on common vulnerability classes, what a prompt identifies may matter more than how elaborately that information is formatted, a direction that warrants higher-powered replication before it is treated as a design principle.

Data Availability Statement

Our replication package and data used for this study are available at: <https://doi.org/10.5281/zenodo.21672348>.

References

- 1 Obieda Ananbeh, Wala Alnozami, and Dae-Kyoo Kim. Assessing the effectiveness of large language models for Java vulnerability repair: A comparative study. *Automated Software Engineering*, 33(3):75, 2026. doi:10.1007/s10515-026-00622-z.
- 2 Ahmet Emir Bosnak, Sahand Moslemi, Mayasah Lami, and Anil Koyuncu. Explicit vulnerability generation with LLMs: An investigation beyond adversarial attacks. In *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 821–826, 2025. doi:10.1109/ICSME64153.2025.00086.
- 3 Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.
- 4 Quang-Cuong Bui, Ranindya Paramitha, Duc-Ly Vu, Fabio Massacci, and Riccardo Scandariato. APR4Vul: An empirical study of automatic program repair techniques on real-world Java vulnerabilities. *Empirical Software Engineering*, 29(18), 2024. doi:10.1007/s10664-023-10415-7.
- 5 Quang-Cuong Bui, Riccardo Scandariato, and Nicolás E. Díaz Ferreyra. Vul4J: A dataset of reproducible Java vulnerabilities geared towards the study of program repair techniques. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, pages 464–468, 2022. doi:10.1145/3524842.3528482.
- 6 Mohamad Fakihi, Rahul Dharmaji, Halima Bouzidi, Gustavo Quiros Araya, Oluwatosin Ogundare, Mst Ayesha Siddika, and Mohammad Abdullah Al Faruque. LLM4CVE: Enabling iterative automated vulnerability repair with large language models. In *2025 28th Euromicro Conference on Digital System Design (DSD)*, pages 592–599, 2025. doi:10.1109/DSD67783.2025.00087.
- 7 Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. VulRepair: A T5-based automated software vulnerability repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 935–947, 2022. doi:10.1145/3540250.3549098.
- 8 Jia He, Mukund Rungta, David Koleczek, Arshdeep Sekhon, Franklin X. Wang, and Sadid Hasan. Does prompt formatting have any impact on LLM performance?, 2024. doi:10.48550/arXiv.2411.10541.
- 9 Yiwei Hu, Zhen Liu, Kedie Shu, Shenghua Guan, Deqing Zou, Shouhuai Xu, Bin Yuan, and Hai Jin. SoK: Automated vulnerability repair: Methods, tools, and assessments. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 4421–4440, Seattle, WA, 2025. USENIX Association. URL: <https://www.usenix.org/conference/usenixsecurity25/presentation/hu-yiwei>.
- 10 Arshiya Khan, Guannan Liu, and Xing Gao. Code vulnerability repair with large language model using context-aware prompt tuning. In *2025 IEEE Security and Privacy Workshops (SPW)*, pages 283–287. IEEE, May 2025. doi:10.1109/spw67851.2025.00040.
- 11 Ummay Kulsum, Haotian Zhu, Bowen Xu, and Marcelo d’Amorim. A case study of LLM for automated vulnerability repair: Assessing impact of reasoning and patch validation feedback. In *Proceedings of the 1st ACM International Conference on AI-Powered Software (AIware)*, pages 103–111, 2024. doi:10.1145/3664646.3664770.
- 12 Jia Li, Zhuangbin Chen, Yuxin Su, and Michael R. Lyu. VulKey: Automated vulnerability repair guided by domain-specific repair patterns, 2026. doi:10.48550/arXiv.2605.01769.
- 13 Ziyang Li, Saikat Dutta, and Mayur Naik. IRIS: LLM-assisted static analysis for detecting security vulnerabilities, 2025. doi:10.48550/arXiv.2405.17238.

- 14 Peng Liu, He Wang, Chen Zheng, and Yuqing Zhang. Prompt fix: Vulnerability automatic repair technology based on prompt engineering. In *2024 International Conference on Computing, Networking and Communications (ICNC)*, pages 116–120, 2024. doi:10.1109/ICNC59896.2024.10556123.
- 15 Yu Nong, Mohammed Aldeen, Long Cheng, Hongxin Hu, Feng Chen, and Haipeng Cai. Chain-of-thought prompting of large language models for discovering and fixing software vulnerabilities. *arXiv preprint arXiv:2402.17230*, 2024. doi:10.48550/arXiv.2402.17230.
- 16 Yu Nong, Haoran Yang, Long Cheng, Hongxin Hu, and Haipeng Cai. APPATCH: Automated adaptive prompting large language models for real-world software vulnerability patching. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 4481–4500, 2025. URL: <https://www.usenix.org/conference/usenixsecurity25/presentation/nong>.
- 17 Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2339–2356. IEEE, 2023. doi:10.1109/SP46215.2023.10179324.
- 18 Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)*, pages 24–36, 2015. doi:10.1145/2771783.2771791.
- 19 Melanie Sclar, Yejin Choi, Yulia Tsvetkov, and Alane Suhr. Quantifying language models’ sensitivity to spurious features in prompt design, 2024. arXiv:2310.11324.
- 20 Ze Sheng, Zhicheng Chen, Shuning Gu, Heqing Huang, Guofei Gu, and Jeff Huang. LLMs in software security: A survey of vulnerability detection techniques and insights. *ACM Comput. Surv.*, 58(5), November 2025. doi:10.1145/3769082.
- 21 Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse Coskun, and Gianluca Stringhini. LLMs cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 862–880. IEEE, 2024. doi:10.1109/SP54263.2024.00210.
- 22 Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023. arXiv:2201.11903.
- 23 Quanjun Zhang, Chunrong Fang, Bowen Yu, Weisong Sun, Tongke Zhang, and Zhenyu Chen. Pre-trained model-based automated software vulnerability repair: How far are we? *IEEE Transactions on Dependable and Secure Computing*, 21(4):2507–2525, 2024. doi:10.1109/TDSC.2023.3308897.