When Grounding Hurts: Retrieval and Validation for LLM Repair of Quantum Programs

Xuan-Bach Le[1,2(✉)], Phat T. Tran-Truong[1,2], and Duy Nguyen[3]

[1] Faculty of Computer Science and Engineering, Ho Chi Minh City University of Technology (HCMUT), Vietnam
[2] Vietnam National University Ho Chi Minh City (VNU-HCM), Vietnam
[3] University of Economics Ho Chi Minh City (UEH), Vietnam
lexuanbach@hcmut.edu.vn, phatttt@hcmut.edu.vn, duynguyen.726202321440@st.ueh.edu.vn

Abstract. Changes to application programming interfaces (APIs) can stop quantum programs from running, while incorrect gates or qubit ordering can produce wrong results without raising errors. We study how grounding large language models (LLMs) affects the detection and repair of such defects. Our system, LintRepair-RAG-Q, combines static analysis, a retrieval-supported LLM detector, and a repair loop that receives analyser findings and retrieved defect-knowledge entries and checks each patch by executing it. We evaluate it on Q-Defects40, 40 Qiskit programs across eight defect families, using eighteen models, including open-weight models with 3 to 72 billion parameters. Combining the static and LLM detectors raises recall from 0.72 to 0.81 with no false positives on 40 negative examples. The grounded repair system improves pooled accuracy by 16 percentage points ($p < 0.001$), with an effect that varies by defect type: a 50-point gain on API-migration defects, no net change on structural ones, and a 9-point decline on semantic defects that run but return incorrect output. Within the migration-free semantic group, models with higher baseline executability, the fraction of unguided patches that execute, show larger losses after adjusting for baseline accuracy (partial correlation -0.57, 95% interval $[-0.85, -0.17]$). We release the benchmark, system, and scripts.

Keywords: Quantum software engineering · Program repair · Static analysis · Large language models · Retrieval-augmented generation · Competence-dependent evaluation.

1 Introduction

Developers build quantum software on software development kits (SDKs) such as Qiskit, which change quickly. The resulting failures take two forms. API incompatibilities cause execution failures. Qiskit 1.0 removed the top-level `execute()`

Artifact, extended version: https://github.com/lexuanbach/lintrepair-ragq

```
qc.x(0)                    # qubit 0 -> |1>, qubit 1 stays |0>
qc.measure([0, 1], [0, 1])
counts = run(qc)           # Qiskit returns {'01': 1000}
exp = str(1) + str(0)      # BUG: builds '10' (qubit 0 written first)
# FIX: exp = str(0) + str(1) -> '01'  (Qiskit puts qubit 0 last)
hits = counts.get(exp, 0)  # 0 instead of 1000, with no error raised
```

Fig. 1. An illustrative simplification of the endianness defect `G2-label-order` (benchmark family G). Here `run` is a helper, while the figure's 1000 shots stand in for the benchmark's 8192. Nothing crashes and our static detector reports no issue. Asked to fix this program alone, Sonnet corrects the key order. Given the same program with an analyser report of "no structural defects" and retrieved knowledge-base entries, it leaves the bug in place.

helper and moved the `Aer` simulator into a separate package. Programs using those interfaces need migration. Semantic defects instead produce incorrect outputs without raising an exception.

Figure 1 shows a semantic defect. The circuit puts qubit 0 into the computational-basis state $|1\rangle$ and leaves qubit 1 in $|0\rangle$. Qiskit encodes the measured bits in a counts key ordered little-endian, with qubit 0 rightmost, and returns `'01'`. The post-processing line builds its key in the opposite order and looks up `'10'`. Every API call is valid, the circuit is correct, and our static detector reports no issue. The program returns zero hits instead of a thousand.

This example separates information that helps a program execute from information that helps recover its intended behaviour. We ask whether one grounded repair pipeline affects these two tasks differently. Empirical studies document compatibility and correctness challenges of both kinds [2,3,18].

Prior work covers complementary settings. Differential testing compares quantum software stacks [16], LintQ checks a fixed rule catalogue [11], and HornBro synthesises repairs against explicit behavioural specifications [14]. Our repair system instead receives only the source program and contextual findings, leaving the reference to the evaluator alone. LLM repair systems such as InferFix [6] supplement the model with static-analysis findings and retrieved examples and validate patches by execution, a pattern the static-analysis agent literature now surveys [5]. We examine how the effect of such a combined pipeline varies with defect type and with the model's baseline ability.

We evaluate eighteen proprietary and open-weight LLMs, the open-weight ones spanning 3 to 72 billion (B) parameters. The effect of grounding depends on the kind of defect and on how many of a model's unguided patches already execute, which we call its baseline executability. On API-migration defects, grounding supplies migration information and repair improves sharply. On structural defects that crash, grounding changes almost nothing. On silent semantic defects, which our static rules miss, the grounded configuration produces fewer oracle-correct repairs, with larger losses on the backbones of higher baseline executability. Distraction by irrelevant context [13] and capability-dependent self-repair [10] are documented separately, leaving open how the two combine inside one grounded repair pipeline, which is what we measure. A pooled average over all defect types reports a positive effect that hides a sign change across our model

range. Neither a generic warning that the analyser may be incomplete nor a variant naming the planted families improves accuracy across all repair backbones, where the named variant beats the generic one only through benchmark-specific hints.

Approach and contributions. We build LintRepair-RAG-Q, which combines a static linter, a retrieval-grounded LLM detector, and a repair loop that validates every patch by executing it. We evaluate it on Q-Defects40, 40 Qiskit 2.x programs over eight defect families, each defective program carrying a human fix and an execution-based oracle (§3). Across eighteen models (§4) we ask three research questions (RQs): how static and LLM detection complement each other (RQ1), how the effect of grounded repair varies (RQ2), and whether prompt debiasing undoes the losses (RQ3). Our evidence comes from the Qiskit setting. The formal model, the validator-loop analysis, and the full per-model tables appear in the extended version linked on the first page.

2 Background and Related Work

Defects in quantum software. A quantum program is a classical host program, here Python, that builds and runs circuits through an SDK. Two properties make such programs hard to debug. First, their outputs are probabilistic. A wrong circuit can still return a plausible-looking distribution. Second, the SDKs move fast. Qiskit 1.0 removed top-level `execute()` and `qiskit.Aer`, and removed `bind_parameters` in favour of `assign_parameters`. Programs written against those interfaces need migration. Empirical studies document compatibility and correctness challenges in quantum code [2,3], and Bugs4Q catalogues real Qiskit faults [18].

Testing, analysis, and repair. Differential testing [16] and related dynamic techniques find defects without repairing them, in an area where weak oracles remain an open problem. On the static side, LintQ detects a fixed catalogue of structural anti-patterns at high precision [11]. By construction, an analyser of this kind cannot see a defect that leaves no syntactic trace. A formal line of work reasons about quantum programs deductively, where quantum separation logic supplies a program logic for local reasoning about quantum code [7]. These methods establish properties relative to supplied specifications, whereas we target automated repair checked by execution. HornBro synthesises patches against explicit behavioural specifications [14], an input our source-level setting does not provide. Our own linter (§3) is a LintQ-style rule layer that targets Qiskit-2.x migration and index or size errors, complementing LintQ's measurement, composition, and transpilation rules.

LLMs, grounding, and repair. Several repair approaches supplement the model with analysis results or retrieved examples. Retrieval-augmented generation (RAG) injects outside knowledge [8], static-analysis findings steer the model's attention, and execution feedback checks the output. In classical settings, InferFix supplies bug annotations and retrieved similar fixes to the model [6]. Within quantum software, comparative studies report that LLM-based linting

can match or beat rule-based LintQ, lifting F1, the harmonic mean of precision and recall, from 0.41 to 0.68–0.70 with chain-of-thought and RAG variants [1]. Those results motivate evaluating whether rules and LLM detectors complement one another. Related work also reports gains in quantum program repair under different prompting and evaluation protocols. ChatGPT fixed 29/38 Bugs4Q bugs within five candidate patches, given detailed bug descriptions and feedback [4], while mutation-grounded prompting lifted GPT-5 repair on a Bugs4Q subset [17]. Each reports a gain from a particular prompting intervention, which leaves open whether such benefits hold uniformly across models and repair settings.

We therefore evaluate the combined system by defect type and model. Context distraction [9,13] and deference to assertions placed in the prompt [12] are related hypotheses for the regression we observe on stronger models. Because those studies do not diagnose our repair runs, we add prompt controls to examine possible explanations.

3 Approach and Benchmark

For detection we combine a high-precision static analyser with a retrieval-grounded LLM detector, because the two see different classes of defect. For repair we supply analyser findings, retrieved knowledge, and execution feedback, while varying how the findings are presented.

The LintRepair-RAG-Q pipeline. (1) A static detector. We wrote an abstract-syntax-tree (AST) analyser of roughly 200 lines in the spirit of LintQ [11]. It flags removed or renamed APIs (`execute`, `Aer`, `bind_parameters`), out-of-range qubit indices, register-size mismatches, and counts requested without measurement. It stays silent on any defect without a syntactic signature. **(2) A retrieval-grounded LLM detector.** We built a knowledge base (KB) of 15 *general* defect-family descriptions and remedies, indexed by a TF–IDF retriever (term frequency weighted by inverse document frequency) returning the top k=4 documents. No KB entry encodes the answer to a benchmark instance. **(3) A validator-backed repair loop.** The model receives the program, the union of findings, and the retrieved documents, then proposes a complete patch. The patch must parse, import, and run to completion while binding `RESULT`, the variable that holds the program's observable output. On failure we feed the runtime error back, allowing up to m=3 total attempts. Because this execution-only validator never sees the human-written fix, it rescues crashes but cannot certify that a silent semantic bug is gone. A separate reference-based oracle, described below, measures whether a patch's output is correct.

Detector scope and repair configurations. We prompt the LLM detector with a *closed* list of the eight benchmark families, which means the recall in §4 is measured under a closed-world taxonomy matched to the benchmark. We fix the detector to Sonnet throughout. All grounded configurations reuse its precomputed findings, which controls detector variation across repair backbones, while the repair results remain conditional on its outputs and taxonomy. Each configuration uses one of the eighteen models as its repair backbone. **B0 (plain)**

is a single-shot "fix this code" prompt with no findings, no retrieval, and no validator. **B1 (RAG)** adds the detector union and the retrieved documents in a single shot. **B2 (naive)** adds the validator loop. When nothing is flagged, B2 tells the model that "static analysis reported no structural defects," the phrasing we later find accompanies lower accuracy on capable models. **B3** and **B4** change the findings preamble. Both add a header warning that the analyser is incomplete and, when nothing is flagged, replace the cleanliness sentence with a request to inspect the logic. B3's header also names silent modes the analyser might miss: a wrong or forgotten gate, a swapped control and target, an angle in degrees, and an endianness error. B4 gives the generic warning only and serves as our generic-warning control. B2 minus B0 is a combined-system difference: B2 adds findings, retrieved documents, and up to three attempts against B0's single generation.

The Q-Defects40 benchmark. Q-Defects40 holds 40 self-contained Qiskit 2.x programs, 32 defective and 8 clean, over eight families with four instances each, drawn from documented real-world faults: the Qiskit 1.0 API removals, structural defects in the LintQ and Smelly-Eight tradition [2,11], and silent semantic faults in the spirit of Bugs4Q [18]. Every defective program carries a minimal human-written fix. Having fixed the families and the four-instances-each budget before evaluating any model, we never tuned the taxonomy to our results. The 8 clean programs are correct reference idioms, which together with the 32 human fixes form a 40-program negative set for measuring false positives.

Because grounding behaves differently on different defects, we partition the 32 defects into three groups throughout. *API-migration* (12) covers the Qiskit 1.0 removals of `execute()`, `qiskit.Aer`, and `bind_parameters`. All of them crash, every one within our analyser's reach. *Structural/runtime* (11) covers out-of-range qubit indices, register-size mismatches, and counts requested without measurement. These also crash. The analyser reaches them fully or in part. *Quantum-semantic* (9) covers silent errors in the quantum computation or its classical result processing: a wrong gate or parameter such as `cz` (controlled-Z) for `cx` (controlled-X), a little-endian misreading of the counts key, and one missing measurement that fails silently. These are the *silent* defects, which run to completion, return the wrong distribution, and escape our static rules. They are the class our headline result concerns, as well as the hardest to author with a deterministic oracle, which is why they form our smallest group (§5).

What counts as a defect. We call a program defective when its observable output, the value bound to `RESULT`, differs from a reference we fixed at authoring time from the intended computation. The comparison with that reference alone decides the label. For sampling programs we use total-variation distance (TVD) between measurement distributions, thresholded at 0.08 over 8192 shots under fixed seeds, where a shot is one execution of the circuit yielding one sampled outcome. For measurement-free circuits we use state fidelity, a similarity score between quantum states, thresholded at 0.99. Every fix matches its reference exactly, the nine buggy programs that run sit at TVD$\geq$0.23, and the other 23 crash

without producing a distribution (§5 reports threshold and repeatability checks). A circuit applying `cz` where `cx` was intended is valid Python and valid Qiskit. It is a defect only relative to the intended Bell state, the equal superposition of 00 and 11, which our reference encodes. The reference serves evaluation only and is never shown to LintRepair-RAG-Q during repair. The repair prompt asks only that the model preserve the author's evident intent. Where a semantic instance's prompt does not state the intended computation, the outcome also depends on the model inferring that intent (§5). We release the full construction log.

Implementation, models, and statistics. LintRepair-RAG-Q runs on Qiskit 2.4.2 with Qiskit-Aer 0.17.2, validating each candidate patch in an isolated subprocess under a 60 s timeout. We evaluate eighteen repair backbones that span proprietary APIs and open weights from 3B to 72B parameters,[4] listed in Table 3. For every model, the same backbone serves as both the unguided baseline and the grounded system, with findings identical no matter which backbone repairs. With greedy decoding and every call cached, the reported analysis reproduces from frozen outputs, though identical regeneration from the providers is not guaranteed. We measure detection recall per program, as the fraction of the 32 defective programs whose true family the detector flags. Repair accuracy, written pass@1, is the oracle-correct fraction of the final patch from one repair invocation. B0 permits a single generation, whereas B2–B4 permit up to three attempts inside the validator loop. This use of repeated samples is related to distributional program analysis [15]. We summarise each backbone's unguided ability by its baseline executability $\hat{\rho}$, the fraction of its 32 B0 patches that execute, and use "competence" as shorthand for this in-setting proxy. We report Wilson 95% confidence intervals (CIs) and compare configurations on the *same* instances using a paired bootstrap with the exact McNemar test, with resample counts recorded alongside each result. With n=32 we treat any single-model difference as descriptive, reserving significance claims for the pooled tests.

4 Evaluation

The released scripts reproduce every reported analysis from recorded executions. We claim significance only where paired bootstrap intervals and exact McNemar tests agree.

RQ1: How do static and LLM detection complement each other? Table 1 gives full confusion matrices. The 32 human fixes are structurally identical to the defects apart from the repair, making them hard negatives. Our static analyser reaches recall 0.719, the LLM detector 0.781, and their union 0.812, all at precision 1.000, though 40 negatives cannot establish zero false positives beyond Q-Defects40. On the silent-semantic families our static analyser is blind, scoring 0/4 on endianness and 0/4 on wrong-gate, while the LLM recovers 2/4

[4] The artifact records requested model identifiers, Ollama tags, and cached responses, but not call dates or hosted-model revisions.

Table 1. RQ1 Detection on Q-DEFECTS40: true positives (TP), false negatives (FN), false positives (FP), and true negatives (TN) over 32 defective programs and N^- negatives, the number each detector was scored on. Ours use all 40 matched negatives (8 clean idioms and 32 human fixes), LintQ only the 8 clean programs, so the precision columns are not a common-set comparison. "n/a" means no positives emitted.

Detector	N^-	TP	FN	FP	TN	Precision	Recall	F1
LintQ, high-prec. (default) [11]	8	0	32	0	8	n/a	0.000	n/a
LintQ, all checkers [11]	8	11	21	3	5	0.786	0.344	0.478
Static (rule-based)	40	23	9	0	40	1.000	0.719	0.836
RAG-LLM	40	25	7	0	40	1.000	0.781	0.877
Union (LINTREPAIR-RAG-Q)	**40**	**26**	**6**	**0**	**40**	**1.000**	**0.812**	**0.897**

and 1/4. The union retains every static detection. Because recall is measured under the closed taxonomy of §3, it reflects benchmark alignment more than open-world coverage.

LintQ [11] ran under CodeQL 2.11.2 and modelled our circuits, which makes its non-detections a matter of rule coverage, since parsing succeeded. Its all-checkers hits are incidental, since two structural checkers fire on buggy and correct endianness programs alike.

RQ2: How does grounded repair affect accuracy? Pooled over all eighteen models on matched instances (576 model-program pairs), the grounded repair system (B2) beats unguided repair (B0) by +0.160 (CI $[0.109, 0.214]$, McNemar $p < 0.001$, 168 wins against 76 losses). To account for the same 32 programs recurring across models, a program-clustered bootstrap over 5,000 resamples gives +0.160 with CI $[0.038, 0.276]$. A mixed-effects logistic model with program and model random intercepts agrees (odds ratio 2.69, approximate 95% interval $[2.2, 3.3]$). Because the positive pooled estimate combines different defect types, we next examine which groups account for it.

Repair effects by defect type. Table 2 splits the gain across the three defect groups of §3: API-migration gains +0.495, structural/runtime defects remain at 0.000 (with 41 wins and 41 losses), and quantum-semantic defects have a gain of −0.093. Compared with B0, the combined grounded system improves API-migration repair but reduces accuracy on the semantic subset. Stratified tests agree. Grounding helps the 23 crash defects by +0.258 (CI $[0.196, 0.321]$, $p < 0.001$, n=414) and hurts the 9 silent-semantic ones by −0.093 (CI $[-0.173, -0.012]$, $p = 0.036$, n=162). Of the 168 wins, 153 are crash defects and 15 silent-semantic, while the latter account for 30 of the 76 losses. A deterministic Qiskit-1.0 migration script without an LLM fixes only 10/32 defects, all in API-removal families.

Repair gains are associated with baseline executability. Across models, the B2−B0 gain falls as $\hat{\rho}$ rises. We measure Pearson $r = -0.629$ and Spearman rank correlation −0.538 over n=18 models. A model-level regression of gain on $\hat{\rho}$ has slope −0.47 (bootstrap CI $[-0.68, -0.19]$, explaining R^2=0.40 of the variation) and crosses zero near $\hat{\rho} \approx 0.84$. Table 3 gives the per-model picture.

Table 2. RQ2 Grounding's effect by defect type. Instances counts programs, and each row pools over the 18 backbones and therefore rests on Instances × 18 pairs. The last column is the Pearson correlation between $\hat{\rho}$ (baseline executability, §3) and the B2−B0 gain over those backbones.

Defect group	Instances	B0	B2	B2−B0	corr($\hat{\rho}$, gain)
API-migration	12	0.231	0.727	+0.495	−0.302
Structural/runtime	11	0.460	0.460	0.000	−0.465
Quantum-semantic	9	0.352	0.259	−0.093	**−0.864**
Pooled	32	0.344	0.503	+0.160	−0.629

Table 3. RQ2–RQ3 Repair pass@1 by model, sorted by $\hat{\rho}$, the fraction of B0 patches that execute. B0 plain, B2 grounded, B3 family-specific warning, B4 generic warning, Δ = B2 − B0. Pooled B2−B0: +0.160 [.109,.214], p<0.001, 168/76 wins/losses. Model key: C is Coder, Cmd-R is Command R, and Gemini Pro, Gem Flash, and Gem F-Lite are Gemini 2.5 Pro, Flash, and Flash-Lite.

Model	$\hat{\rho}$	B0	B2	B3	B4	Δ	Model	$\hat{\rho}$	B0	B2	B3	B4	Δ
GPT-4o-mini	.00	.000	.625	.719	.625	+.625	GPT-4o	.50	.406	.844	.844	.812	+.438
Qwen2.5-14B	.12	.125	.406	.344	.312	+.281	Qwen2.5-C-7B	.53	.250	.250	.219	.156	+.000
Mistral-7B	.16	.031	.156	.156	.031	+.125	Qwen2.5-C-32B	.56	.250	.688	.719	.688	+.438
Llama-3.1-8B	.19	.094	.375	.438	.375	+.281	Llama-3.3-70B	.59	.406	.500	.594	.500	+.094
Gemini Pro	.41	.344	.438	.250	.219	+.094	Gem F-Lite	.69	.500	.656	.688	.562	+.156
Llama-3.2-3B	.44	.156	.062	.094	.000	−.094	Gem Flash	.78	.656	.719	.688	.750	+.062
Gemma2-9B	.47	.156	.375	.344	.375	+.219	Qwen2.5-72B	.84	.656	.625	.750	.719	−.031
Qwen2.5-7B	.47	.188	.500	.375	.406	+.312	Haiku 4.5	.88	.781	.688	.812	.750	−.094
Cmd-R-35B	.47	.219	.281	.312	.281	+.062	Sonnet 4.6	1.00	.969	.875	.938	.969	−.094

GPT-4o-mini gains +0.625 because every unguided patch emits removed Qiskit-1.0 APIs and retrieval lets those patches run, while Sonnet falls from 0.969 to 0.875 and Qwen2.5-72B, the open-weight model with the highest $\hat{\rho}$ in this panel, also turns negative.

SDK familiarity and baseline accuracy. Because our lower-$\hat{\rho}$ models are also older, $\hat{\rho}$ partly measures Qiskit familiarity. Two observations bear on this. First, the association is strongest in the quantum-semantic group (Table 2, Pearson $r = -0.864$ across the 18 backbones), whose modern Qiskit 2.x programs contain no migrated API. Second, it persists after adjusting for headroom, the room left to improve over semantic B0 accuracy. Although $\hat{\rho}$ correlates +0.834 with semantic B0, which in turn correlates −0.822 with the gain, the partial correlation between $\hat{\rho}$ and semantic gain, controlling for semantic B0 accuracy, remains −0.569 (bootstrap 95% CI $[-0.849, -0.170]$, 10^4 model resamples). Part of the coupling is arithmetic, since gain cannot exceed 1 − B0, although a ceiling alone does not explain the strongest models' absolute losses. This reduces one headroom concern without fully separating SDK familiarity from other aspects of repair ability, which is why we read $\hat{\rho}$ as an in-setting proxy for repair competence with an SDK-familiarity component.

RQ3: Can debiasing undo the regression? Because some grounded runs regress, we test whether changing the findings preamble improves repair. The effects vary by model. The named variant B3 warns that the analyser may miss

"a wrong gate, a swapped control, degrees-vs-radians, an endianness error," and gives a small positive difference over naive grounding that is not significant (B3−B2 = +0.012, $p = 0.48$). Because those four modes match our planted families, B3's advantage is consistent with benchmark-aligned hints, although the two prompts also differ in wording. The generic-warning control B4 omits the families and modestly reduces naive grounding (B4−B2 = −0.030, CI $[-0.056, -0.003]$, $p = 0.036$). B3 still edges B4 by +0.042 (CI $[0.019, 0.066]$, $p < 0.001$), a gap consistent with the same hints. At the model level, B4 restores Sonnet from 0.875 to 0.969 and Haiku from 0.688 to 0.750, while eroding the benefit on weaker models, costing Mistral-7B 0.125 and Gemini 2.5 Pro 0.219. No variant is uniformly beneficial.

Controls for possible explanations of regression. Since these warning comparisons do not identify why regression occurs, we examine sampling and context controls. Consider `F1-cz-for-cx`, a Bell preparation applying `qc.cz(0,1)` where `cx` was intended. Writing counts keys as q_1q_0, measurement returns $\{00, 01\}$ rather than the Bell support $\{00, 11\}$. No detector fires, because `cz` is a valid call with no syntactic signature. Under B0, Sonnet restores `cx`. Under the grounded prompt B2, carrying the "no structural defects" report and retrieved entries, the same model leaves `cz` in place.

Controls on the two affected models over our nine silent-semantic instances show that the cleanliness claim is not necessary for the regression. It also appears under a five-sample evaluation, where unguided Sonnet fixes 9/9 silent bugs against grounded's 6/9, a gap that persists over the full 32 at 31/32 against 28/32. A neutral-background control, B0filler, supplies retrieved documents without findings or a cleanliness claim and reproduces almost the entire drop, taking Sonnet from 9 to 6 and Haiku from 7 to 3. It is not exactly length-matched, with a retrieval query that differs from B2's. These runs are consistent with interference from added context [9,13], without isolating document content, prompt length, framing, or validation. The generic caveat B4 improves Sonnet from 6/9 to 8/9 in this comparison.

Evaluation on generated semantic mutants (n=115). To test the defect-type effect beyond nine instances, we mutated the *correct* programs with gate, control, and angle perturbations, validated every mutant by execution, and re-ran B0, B2, and B0filler on all eighteen models over the resulting 115 silent bugs. On this purely semantic, modern-API set the harm is close to universal. Mean correct falls from 13.8 to 2.3 out of 115, with 12 of the 18 models scoring exactly zero under grounding. Only GPT-4o-mini improves, rising from 1 to 13 off the unguided floor. Sonnet falls from 42 to 0, all 42 discordant pairs counting as losses ($p \ll 10^{-3}$). The response to B0filler differs across models: it sits near 0 for open-weight models and equals B0 for Gemini 2.5 Pro. We use this set for the defect-type effect and its scale. With grounded scores pressed against the floor it cannot show a competence trend, which is why our competence claim rests on the migration-free partial correlation above. Because the 115 mutants come from 28 distinct parents, they broaden a synthetic semantic task without adding independent real-world evidence.

5 Discussion and Threats to Validity

Deployment hypothesis. One hypothesis for future evaluation is to vary grounding with $\hat{\rho}$: ground and validate at low $\hat{\rho}$ while withholding the cleanliness verdict at high $\hat{\rho}$, since an incomplete analyser's silence can read as evidence of correctness. Our study validates neither a switching threshold nor a deployment policy.

Construct. Our oracle (§3) uses untuned thresholds of 0.08 TVD and 0.99 fidelity. Labels hold over thresholds 0.02–0.20 and five repeated executions at fixed seeds, showing repeatability, which is weaker than seed robustness. A correct flag does not establish correct reasoning. The oracle measures agreement with an authored reference. Where a semantic instance's repair prompt does not state the intended computation, the outcome also depends on the model's inference of that intent.

Internal. We authored the KB and prompts. Bias controls include a KB free of instance answers, B0 and static-only baselines, and B3 against B4.

External. Q-Defects40 holds 32 authored defects, only 9 quantum-semantic, with a single model as detector. On Bugs4Q [18], only 2/42 run under Qiskit 2.4.2 against 19/42 under 0.45. Of the 12 reproducing deterministically, both parts point the same way on small subgroups: crash bugs rescued from 7 to 12 of 28 pairs (seven bugs, four models), and the silent regression reappearing at 4/10 to 0/10 (five bugs, two models). The pooled estimate over all 48 pairs is imprecise ($+0.021$, CI $[-0.104, 0.146]$) and establishes neither benefit nor harm. A Cirq probe on 10 programs near ceiling did not reproduce the interference. We scope our claims to Qiskit.

Statistical conclusion. We report per-model estimates descriptively, reserving claims for pooled tests needing both a bootstrap CI excluding zero and McNemar agreement. B2−B0 and the B3−B4 gap meet that bar at $p < 0.001$. Per-model tests are not uniformly null: six improvements are individually significant by exact McNemar, three under Bonferroni 0.05/18. The individual-model regressions do not reach significance at n=32 (Sonnet $p = 0.38$, Haiku $p = 0.55$), and two pooled claims near $p \approx 0.04$ do not survive Bonferroni.

6 Conclusion

On Q-Defects40, combining static and LLM detection raises recall from 0.72 to 0.81 at precision 1.000. The grounded repair system improves pooled accuracy, with most of the gain on API-migration defects and a negative point estimate of -0.09 on the semantic subset, where within our panel the losses grow with baseline executability (partial correlation -0.57 after adjusting for headroom). Neither warning variant improves accuracy across all backbones. These are system-level effects measured on Qiskit programs. They do not isolate retrieved content, framing, and validation as separate causes, while our small authored semantic set limits how far they generalise.

Acknowledgment. We acknowledge Ho Chi Minh City University of Technology (HCMUT), VNU-HCM for supporting this study.

References

1. Cassieri, P., Scanniello, G., Shin, S.Y., Pastore, F., Bianculli, D.: Beyond rules: LLM-powered linting for quantum programs. arXiv:2605.03943 (2026).
2. Chen, Q., Câmara, R., Campos, J., Souto, A., Ahmed, I.: The smelly eight: An empirical study on the prevalence of code smells in quantum computing. In: Proc. ICSE. pp. 358–370 (2023).
3. De Stefano, M., Pecorelli, F., Di Nucci, D., Palomba, F., De Lucia, A.: Software engineering for quantum programming: How far are we? J. Syst. Softw. **190**, 111326 (2022).
4. Guo, X., Zhao, J., Zhao, P.: On repairing quantum programs using ChatGPT. In: Proc. Q-SE. pp. 9–16 (2024).
5. Hua, H.Q.B., Le, X.B.: Emerging trends and challenges toward LLM agents in static analysis. In: Proc. IEA/AIE. LNCS, vol. 16614, pp. 101–115 (2026).
6. Jin, M., Shahriar, S., Tufano, M., et al.: InferFix: End-to-end program repair with LLMs. In: ESEC/FSE (2023).
7. Le, X.B., Lin, S.W., Sun, J., Sanan, D.: A quantum interpretation of separating conjunction for local reasoning of quantum programs based on separation logic. Proc. ACM Program. Lang. **6**(POPL), 1–27 (2022).
8. Lewis, P., Perez, E., Piktus, A., et al.: Retrieval-augmented generation for knowledge-intensive NLP tasks. In: Proc. NeurIPS. vol. 33, pp. 9459–9474 (2020)
9. Liu, N.F., Lin, K., Hewitt, J., et al.: Lost in the middle: How language models use long contexts. TACL **12**, 157–173 (2024).
10. Olausson, T.X., Inala, J.P., Wang, C., Gao, J., Solar-Lezama, A.: Is self-repair a silver bullet for code generation? In: ICLR (2024)
11. Paltenghi, M., Pradel, M.: Analyzing quantum programs with LintQ: A static analysis framework for Qiskit. PACMSE **1**(FSE), 2144–2166 (2024).
12. Sharma, M., Tong, M., Korbak, T., et al.: Towards understanding sycophancy in language models. In: ICLR (2024).
13. Shi, F., Chen, X., Misra, K., et al.: Large language models can be easily distracted by irrelevant context. In: ICML. PMLR, vol. 202, pp. 31210–31227 (2023)
14. Tan, S., Lu, L., Xiang, D., et al.: HornBro: Homotopy-like method for automated quantum program repair. PACMSE **2**(FSE), 734–756 (2025).
15. Tran-Truong, P.T., Le, X.B.: Distributional program analysis: Treating large language models as program samplers. In: Proc. ASE (NIER) (2026)
16. Wang, J., Zhang, Q., Xu, G.H., Kim, M.: QDiff: Differential testing of quantum software stacks. In: Proc. ASE. pp. 692–704 (2021).
17. Yoshida, C., Ishimoto, Y., Nourry, O., et al.: Leveraging mutation analysis for LLM-based repair of quantum programs. In: Proc. SANER-ERA (2026).
18. Zhao, P., Miao, Z., Lan, S., Zhao, J.: Bugs4Q: A benchmark of existing bugs to enable controlled testing and debugging studies for quantum programs. J. Syst. Softw. **205**, 111805 (2023).