QuRepair: Diagnosis-Guided LLM Repair for Multi-Framework Quantum Software

Anonymous authors
Paper under double-blind review

Abstract

Large language models (LLMs) have shown strong performance in classical program repair, but quantum programs introduce challenges arising from quantum-specific operations, semantics, and programming frameworks. We introduce QuRepair, a structured quantum-aware repair framework in which static linter diagnostics, API evidence, and quantum-semantic rules inform a structured typed diagnosis produced by a decision model (Jev). A separate repair LLM uses this diagnosis and a textual circuit rendering to iteratively repair the program. For controlled evaluation, we also introduce a multi-framework fault-injection methodology spanning Qiskit, Cirq, PennyLane, and OpenQASM. Faults are defined by their intended effect and aligned across frameworks through canonical-source mutation and validated translation, with API faults realized through framework-specific mutations. Real-world Bugs4Q-Robust faults provide external validation. Evaluating five general-purpose and quantum-specific baselines, we find that existing methods vary substantially across frameworks and struggle on several challenging quantum-semantic faults, including relative-phase and operation-order errors. With GPT-4o, QuRepair achieves 62.83% overall repair success, 15.83 percentage points above the strongest baseline at 47.0%, with the highest success on three frameworks and a tie with the strongest baseline on OpenQASM. It also leads on real-world Bugs4Q-Robust faults (72.4% vs. 62.1% for the next-best method), and its advantage over direct repair persists across four underlying LLMs. These results show that structured quantum-specific evidence improves LLM-based quantum program repair across frameworks.

1 Introduction

Large language models (LLMs) have become capable code-generation and program-repair systems, (Hou et al., 2024) with recent debugging methods using execution feedback, program analysis, intermediate reasoning, and iterative refinement. These approaches are primarily designed for classical programs, where faults are often exposed through exceptions, failing tests, or deterministic execution traces.

Quantum programs introduce a different repair setting. Syntactically valid and executable programs may still implement an incorrect quantum computation, and empirical studies identify recurring faults involving operations, qubits, measurements, and mathematical logic (Luo et al., 2022; Paltenghi & Pradel, 2022). Moreover, many quantum programs expose behavior through repeated measurements and probability distributions rather than deterministic outputs (García de la Barrera et al., 2023). Successful execution therefore does not establish semantic correctness and may provide little information about faults involving operation order, parameters, qubit interactions, or measurements.

Cross-framework evaluation introduces a methodological bottleneck. Bugs4Q provides real, manually validated Qiskit bugs, but its executable benchmark is centered on Qiskit and does not provide matched framework-independent fault classes (Zhao et al., 2023). Bugs4Q-Robust (Ohto et al., 2026) adapts the original Bugs4Q artifacts for reproducibility under newer Qiskit versions but keeps the same Qiskit-only scope. Existing quantum mutation tools are likewise commonly framework-specific and emphasize gate- and measurement-level mutations (Fortunato et al., 2024; 2022). Neither therefore yields matched local, API, and quantum-semantic faults across Qiskit, Cirq, Penny-

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.

Lane, and OpenQASM; without such control, framework effects are confounded with fault type and difficulty.

We address these challenges with *QuRepair*, a quantum-aware repair framework that combines a static linter, API evidence, and quantum semantic rules through Jev (TypeSafe AI, 2026) a typed decision model that returns structured probabilistic decisions rather than free-form text, and passes the resulting diagnosis, together with a textual circuit rendering, to a repair LLM. For evaluation, non-API faults are injected into a canonical Qiskit source and retained only when the mutation is preserved across generated Qiskit, Cirq, PennyLane, and OpenQASM targets; API faults use aligned framework-specific mutations. The underlying programs are derived from MQT Bench (Quetschlich et al., 2023). We compare against general-purpose and quantum-specific debugging methods using output-distribution fidelity rather than execution success alone.

Research Questions. The study addresses four questions:

- **RQ1: Repair Difficulty.** How does the effectiveness of existing LLM debugging methods vary across quantum-program fault types and target frameworks?
- **RQ2: Quantum-Specific Debugging.** Does quantum-specific debugging information improve LLM repair compared with existing debugging methods?
- **RQ3: Generalization.** Do the conclusions remain valid on external bugs and under alternative underlying language models?
- **RQ4: Benchmark Design and Cross-Framework Control.** Why are existing real-bug benchmarks and quantum mutation techniques insufficient for controlled multi-framework repair evaluation, and what properties are required to compare frameworks without confounding framework choice with fault type?

Contributions. We make three contributions:

- QuRepair, a quantum-aware repair framework combining static, API, and quantum-semantic evidence into a Jev-typed diagnosis (TypeSafe AI, 2026) that guides LLM repair together with a textual circuit rendering. With GPT-4o, it repairs 62.83% of faulty programs versus 47.0% for the strongest baseline.
- A controlled comparison of LLM debugging methods, including adapted QBugLM (Pham et al., 2026), across frameworks and fault types using semantic quantum behavior as the repair criterion.
- A framework-agnostic injection methodology with matched local, API, and quantum-semantic faults across four frameworks, complemented by Bugs4Q-Robust (Ohto et al., 2026) and cross-model evaluation for external generalization.

2 Related Work

LLM debugging and program repair. Recent repair methods augment generation with execution feedback, reasoning, localization, and iterative refinement. Self-Debugging, LDB, RING, and NL-Debugging respectively emphasize explanation, runtime-state reasoning, candidate ranking, and natural-language intermediate representations (Chen et al., 2024; Zhong et al., 2024; Joshi et al., 2023; Zhang et al., 2025); we compare them to test whether quantum-specific evidence adds value.

Quantum software debugging and testing. Prior work studies quantum bug patterns, mutation testing, static analysis, and semantic checking (Ramalho et al., 2025; Luo et al., 2022; Zhao et al., 2021; Fortunato et al., 2022; Paltenghi & Pradel, 2024; Long & Zhao, 2024). Bugs4Q provides manually validated real Qiskit bugs (Zhao et al., 2023), with Bugs4Q-Robust adapting these artifacts for newer Qiskit versions (Ohto et al., 2026), whereas mutation systems provide reproducible synthetic faults but are generally framework- or representation-specific (Fortunato et al., 2024; 2022). Our benchmark uses mutation as controlled infrastructure: non-API faults are injected canonically and validated after cross-framework generation, while API faults use aligned framework-specific edits. Bugs4Q-Robust is reserved for external validation.

LLMs for quantum programming. Quantum-code benchmarks study functional correctness and simulator-guided generation (Feng et al., 2026; Mikuriya et al., 2025; Basit et al., 2025); QuanBench+ aligns generation tasks across Qiskit, Cirq, and PennyLane (Slim et al., 2026). These differ

from repairing a supplied faulty program. QBugLM is the closest debugging approach, combining localization, repair, and simulation-based checking (Pham et al., 2026). QuRepair differs by grounding its diagnosis in deterministic framework-aware analyses and quantum-semantic rules rather than in the source text alone, and unlike the OpenQASM centered original QBugLM, by targeting four frameworks natively.

3 DEBUGGING METHODS

All methods receive the same faulty source, task, target framework, interface, underlying LLM, and final evaluator; the correct implementation is never exposed. Between attempts, the iterative methods (LDB, NL-Debugging, QBugLM, and QuRepair) observe at most the evaluator's pass/fail verdict and coarse failure information; none observes reference distributions or divergence values. Exact prompts and details appear in Appendix B.

Self-Debugging. We adapt its explanation-only "rubber-duck" variant (Chen et al., 2024): each turn explains the program, assesses it against the task, and either accepts it or generates a replacement. No evaluator feedback is exposed; we use five turns.

LDB. We replace CFG-derived basic blocks with ordered quantum-operation groups and execute circuit prefixes for intermediate behavioral summaries before complete regeneration (Zhong et al., 2024). We use five iterations; partitioning details appear in Appendix B.2.

RING. We preserve RING's few-shot generation and likelihood ranking (Joshi et al., 2023), using framework-matched retrieved examples when diagnostics exist and fixed framework examples otherwise. Target-task examples and hidden mutation/evaluator metadata are excluded. We generate 10 candidates and evaluate Top@3.

NL-Debugging. We backtranslate the current program to a natural-language sketch, analyze the failure, refine the sketch, and regenerate (Zhang et al., 2025). Only coarse evaluation feedback is exposed to avoid revealing the semantic oracle.

Quantum-Specific Baseline: QBugLM. We preserve QBugFind localization and QBugFix repair, omit QBugGen because faults are pre-generated, extend the workflow to all four frameworks, and replace QBugCheck with our shared hidden evaluator (Pham et al., 2026). Hidden references and evaluator details are never exposed; we use five attempts.

QuRepair (ours). QuRepair combines multiple quantum-specific evidence channels before repair: static quantum-code diagnostics, API evidence, circuit-level representations, and quantum-semantic reasoning. (Figure 1; details in Appendices A and G).

STAGE 1: STATIC LINTER. A framework-aware source-level analyzer reports suspicious behavior related to operations, measurements, circuit construction, and framework usage. The resulting diagnostics identify potentially relevant source regions but do not directly modify the program.

STAGE 2: API EVIDENCE. Framework API usage is inspected against the configured interface. The evidence layer reports suspicious or unavailable API calls and provides availability or signature information when inspectable.

STAGE 3: CIRCUIT RENDERING. When circuit construction succeeds, QuRepair produces a deterministic textual representation of the implemented quantum computation. This representation exposes operation order, operands, parameters, and measurements while reducing dependence on surrounding host-language details. To conserve Jev's request-size budget, we do not include the rendered circuit in its diagnosis input; instead, we provide it directly to the repair LLM.

STAGE 4: QUANTUM SEMANTIC LAYER. Source-derived quantum-semantic diagnostics are generated as part of the framework-aware analysis pipeline.

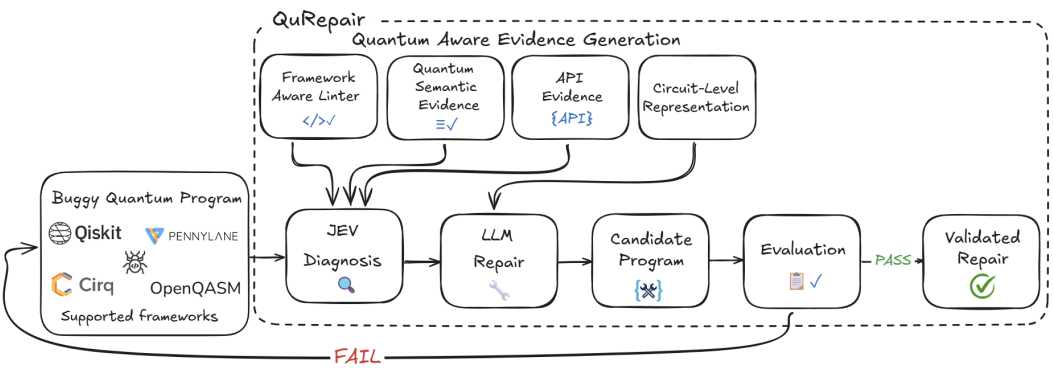


Figure 1: Overview of QuRepair (dashed frame). Jev combines linter, quantum-semantic, and API evidence into a typed diagnosis, the repair LLM uses it with the textual circuit rendering to generate a complete candidate. The shared evaluator returns only pass/fail and runtime errors, on failure, the candidate becomes the program under repair and all evidence is recomputed (at most five iterations).

STAGE 5: JEV TYPED DIAGNOSIS. Jev (TypeSafe AI, 2026) evaluates a fixed catalogue of quantum-semantic rules together with the source-derived diagnostics and available evidence to produce a structured typed diagnosis of the most likely fault.

STAGE 6: REPAIR. The repair LLM receives the task specification, current source program, Jev diagnosis, execution contract, available circuit representation, API evidence, runtime feedback, and retry history. It then generates a complete same-framework repair. After failure, the repaired candidate becomes the source for the next iteration, and all evidence channels are recomputed from the updated candidate. The process continues for up to five iterations.

4 EXPERIMENTAL SETUP

Our experiments isolate the effect of the debugging method while keeping the program, model, and semantic evaluator fixed whenever methods are directly compared.

4.1 PROGRAMS AND FRAMEWORKS

The controlled benchmark is derived from MQT Bench (Quetschlich et al., 2023). We retain computations that can be instantiated and reliably evaluated across Qiskit, Cirq, PennyLane, and OpenQASM while covering multiple algorithm families. Framework-specific implementations may differ syntactically but must represent the same intended computation under the common evaluation contract.

The retained corpus contains 34 configured benchmark families and 136 clean framework-specific program instances. The frozen non-API run contains 541 accepted mutation groups, yielding 2,164 framework-specific faulty artifacts; API-misuse faults are generated separately from a locked cross-framework catalog. The primary controlled experiments use a fixed 152-task subset selected independently of debugging outcomes. The subset-selection procedure, complete program inventory, and task composition are reported in Appendix C.

4.2 CONTROLLED BUG INJECTION

We use fault injection because the primary benchmark must support controlled comparison across both framework and fault type. Comparing independently collected bug sets across frameworks would confound framework effects with differences in fault composition and difficulty.

For non-API faults, Qiskit serves as the canonical source representation. We deterministically mutate the canonical source, generate the corresponding Qiskit, Cirq, PennyLane, and OpenQASM artifacts, and retain a group only when the intended change is preserved and the category-specific

validity checks pass. Each accepted non-API group therefore contains four aligned faulty programs, although their source syntax need not be identical.

API misuse is handled separately because framework APIs have no common source-level representation. A locked catalog defines 51 aligned logical groups: 35 wrong-arity, 11 wrong-name, and 5 conditional-API groups. Each group provides one framework-specific mutation for Qiskit, Cirq, PennyLane, and OpenQASM and is accepted only when the clean target succeeds and the mutant fails for the intended API-related reason.

The retained non-API corpus covers conventional local operation and measurement faults and structure-aware semantic faults involving QFT composition, qubit ordering, basis and phase behavior, reset/reuse, measurement permutation, relative phase, and noncommuting operation ordering. Mutation metadata, locations, and clean references are benchmark-only information and are never exposed to debugging methods. All operators and their counts appear in Appendix D.

4.3 Mutation Validation

Every generated mutant is validated before inclusion (Appendix E). Non-API semantic faults intended to remain executable must preserve the intended transformation, satisfy the required parse/construction/execution checks, and be behaviorally distinguishable from the clean reference. API groups instead require the clean target to succeed and the mutated target to fail for the intended attributable API reason. Ineffective, inconsistent, or unreliably evaluable mutants are discarded.

4.4 External Bugs

Controlled injection and real-world bugs answer different validity questions. We use the injected benchmark as the primary evaluation because it supplies matched fault classes across Qiskit, Cirq, PennyLane, and OpenQASM. We then use Bugs4Q-Robust (Ohto et al., 2026) as an independent external benchmark because its faults were collected from naturally occurring Qiskit programs rather than generated by our mutation procedure. Bugs4Q-Robust is therefore not a replacement for the controlled benchmark: it cannot provide the same cross-framework experimental design, but it tests whether improvements observed under controlled injection transfer to pre-existing real-world faults. We retain the 29 Bugs4Q-Robust tasks whose buggy program fails and corrected program passes the official test. Each generated repair is evaluated against that test without modifying the underlying faults or tests. Selection criteria and execution setup details appear in Appendix F.

4.5 Models and Experimental Controls

The controlled comparison uses GPT-4o (OpenAI, 2024) for all debugging methods, holding model capability fixed so differences primarily reflect debugging information and repair procedure. Prompt templates, generation settings, iteration limits, and method-specific stopping conditions are fixed before evaluation.

For cross-model generalization, we evaluate QuRepair with Gemini 3.8 Flash (Google DeepMind, 2026), Claude Sonnet 4 (Anthropic, 2025), and Qwen3-8B (Qwen Team, 2025) as the repair LLM. Model details appear in Appendix I.

Because the methods use different inference protocols, we retain their configured attempt or candidate limits rather than equalizing model-call or token budgets. We report observed pipeline time and model-token usage in Appendix J.

4.6 Semantic Evaluation

Every instance defines a fixed interface and reference quantum behavior. A repair must (i) parse and construct the required quantum program in the target framework, (ii) execute successfully under the configured simulator, and (iii) reproduce the expected quantum behavior. Measurement distributions are normalized to a common logical bit ordering and compared with the reference using KL divergence. Count vectors are converted directly to relative frequencies without additive pseudo-count smoothing; zero probabilities are clipped after normalization using $\epsilon = 10^{-12}$. A repair is accepted when KL divergence is strictly below 0.05.

All candidates use 16,384 simulator shots, random seed 12,345, the same normalization procedure, and the same threshold. Invalid outputs, including programs constructing no meaningful quantum computation, are failures. Full evaluator details appear in Appendix H.

5 RESULTS

We organize results around four research questions. Unless stated otherwise, repair success means passing the criterion of Section 4.6.

5.1 RQ1: WHERE DO EXISTING DEBUGGING METHODS STRUGGLE?

RQ1 examines Self-Debugging, LDB, RING, NL-Debugging, and QBugLM without QuRepair, aggregating outcomes by framework and fault type.

Framework difficulty. Qiskit has the highest average aggregate repair success across the five existing methods at 45.92%, followed by Cirq at 35.28% and PennyLane at 28.3%. OpenQASM is the most difficult framework at 19.86%. This pattern is consistent across most methods: OpenQASM achieves the lowest repair success for LDB, RING, NL-Debugging, and QBugLM, while Self-Debugging is the only exception, where PennyLane has lower success than OpenQASM.

Fault-type difficulty. Table 1 reports pooled repair success by fault family.

Table 1: Repair success (%) of the five existing methods, pooled over frameworks, by implementation fault family on the selected 152-task set.

Local / API faults		**Quantum-semantic faults**	
Fault	Success (%)	Fault	Success (%)
Operand swap[†]	62.5	QFT-region corruption	35.0
API wrong arity	50.0	Basis-conversion error	35.0
Duplicate operation	46.1	Reset removal[‡]	30.0
Qubit replacement	44.0	Phase-rotation error[‡]	20.0
API wrong name	40.0	Qubit-ordering error	15.0
Measurement deletion	35.0	Measurement permutation error[‡]	15.0
Gate replacement	34.1	Relative-phase error	9.6
X-gate error	33.6	Noncommuting operation-order error	4.2
Gate insertion	31.7		
Gate deletion	27.2		
Parameter mutation	26.9		
API wrong conditional	23.3		
Measurement insertion	10.0		

[†]Two tasks; [‡]one task. Interpret these rows cautiously.

Local and API-related faults generally have higher pooled repair success than quantum-semantic faults, although their success rates overlap. Among the semantic faults, noncommuting operation-order and relative-phase errors have the lowest rates, followed by qubit-ordering and measurement-permutation errors (the latter from a single task). Operand swap is grouped with local faults under the injector's taxonomy.

Answer to RQ1. Existing methods vary substantially across frameworks and fault types: OpenQASM is the hardest target, and several quantum-semantic fault families have particularly low repair success.

5.2 RQ2: DOES QUANTUM-SPECIFIC INFORMATION IMPROVE REPAIR?

As Table 2 shows, QuRepair achieves 62.83% overall repair success versus 47.0% for QBugLM, the strongest baseline, a 15.83-point gain (McNemar test over the 608 paired jobs, $p < 0.001$). It is

Table 2: Repair success (%) by framework under the primary model on the selected 152-task subset.

Method	Qiskit	Cirq	PennyLane	OpenQASM	Overall
NL-Debugging	9.2	23.7	11.2	7.2	12.8
Self-Debugging	26.3	23.7	15.8	18.4	21.1
LDB	53.3	40.8	27.0	19.1	35.0
RING	77.0	48.7	40.1	17.1	45.7
QBugLM	63.8	39.5	47.4	37.5	47.0
QuRepair (ours)	**77.63**	**70.4**	**65.79**	**37.5**	**62.83**

best on Qiskit, Cirq, and PennyLane and ties QBugLM on OpenQASM; the largest gain is on Cirq (+21.7 points over RING), while on Qiskit it leads RING by a single task.

QuRepair is also efficient. Successful runs use 1.33 repair iterations on average, and QuRepair has the lowest mean pipeline time per job (35.9 s) and per successful repair (57.1 s, i.e., total time divided by successes; Appendix J). Including both repair-model and Jev Decisions API calls, it uses 68,821 input-plus-output tokens per credited repair (21,628 from repair-model calls alone), 53.2% fewer than the four non-RING baselines pooled; RING uses fewer tokens per Top@3 success.

Answer to RQ2. QuRepair achieves 62.83% overall repair success, 15.83 points above the strongest baseline; it is best on three of four frameworks, ties on OpenQASM, and is the fastest method.

5.3 RQ3: Do the Conclusions Generalize?

We evaluate generalization along two dimensions: transfer to real-world faults and robustness across different underlying language models.

Generalization Dataset. To evaluate generalization beyond our controlled injection benchmark, we apply all debugging pipelines to the selected Bugs4Q-Robust (Ohto et al., 2026)instances using the same evaluation protocol used in this work. Our pipeline achieves a repair success rate of 72.4%, compared with 62.1% for the strongest baseline pipeline under the same setting. These results show that the proposed approach remains effective when the fault source changes from matched, synthetically injected mutations to independently collected real-world Qiskit bugs. The Bugs4Q-Robust evaluation complements rather than replaces the controlled benchmark: the latter enables systematic comparisons across frameworks and fault classes, while Bugs4Q-Robust evaluates whether the observed repair behavior extends beyond the assumptions of our injection procedure. Details about the Bugs4Q-Robust subset, pipeline adaptations, and evaluation configuration are provided in Appendix F.

Different language models. To evaluate whether QuRepair depends on a particular language model, we repeat the evaluation on the same 152-task benchmark using Qwen 3 8B (Qwen Team, 2025), Claude Sonnet 4 (Anthropic, 2025), and Gemini 3.8 Flash (Google DeepMind, 2026). We compare the complete pipeline against a direct-repair baseline, where the model receives only the debugging task and the faulty source code without any additional quantum-specific evidence. The results in Figure 2 show repair success across models for both settings.

QuRepair reaches 48.19% with Qwen3-8B, 62.83% with GPT-4o, 70.26% with Claude Sonnet 4, and 91.61% with Gemini 3.8 Flash, and improves over direct repair for every model, so its benefit is not tied to one repair model. Because direct repair also lacks QuRepair's iterations and feedback, this gap reflects the full procedure rather than the evidence alone.

Ordered from lower to higher overall capability, repair success follows a broadly increasing and approximately linear trend, suggesting that QuRepair benefits from stronger repair models, with Jev held fixed

Answer to RQ3. QuRepair remains effective on real-world Bugs4Q-Robust faults and across all evaluated models. Across the evaluated model ordering, repair success also follows an approximately linear increasing trend with underlying model capability.

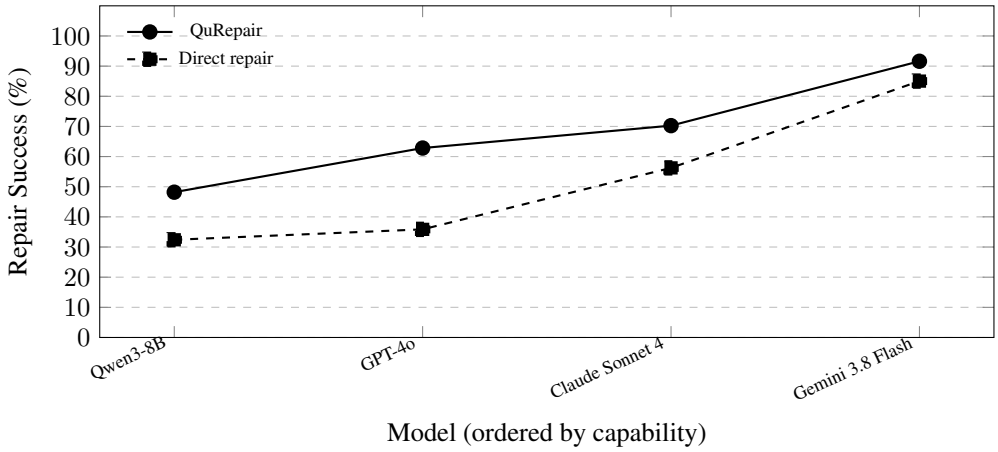


Figure 2: Repair success of QuRepair and direct repair across underlying language models on the 152-task benchmark. Models are ordered by overall capability.

5.4 RQ4: Benchmark Design and Cross-Framework Control

The available alternatives do not provide the combination of realism, fault-type control, and cross-framework comparability required here. Bugs4Q and Bugs4Q-Robust provides real, manually validated Qiskit bugs (Zhao et al., 2023), but is not designed to provide matched framework-independent groups of local, API, and quantum-semantic faults. This does not imply that Bugs4Q-Robust lacks semantic bugs; rather, its naturally collected faults do not provide the controlled cross-framework composition required by our study.

Existing mutation techniques provide reproducibility but are generally tied to a particular framework or representation and often transform individual gates or measurements (Fortunato et al., 2024; 2022). Identical textual operators are neither always expressible nor necessarily equivalent across Qiskit, Cirq, PennyLane, and OpenQASM.

Our benchmark therefore aligns logical fault groups rather than textual edits. For non-API faults, we mutate the canonical Qiskit source and retain the group only when the mutation survives generation and validation in all four targets. API faults instead use aligned framework-specific source mutations because the framework APIs have no common source representation. This design controls fault composition across frameworks, while Bugs4Q-Robust in RQ3 separately tests external validity. This control makes the RQ1 framework gaps interpretable: every framework receives the same fault groups, so the 26-point Qiskit–OpenQASM spread cannot stem from fault composition (Section 6).

Answer to RQ4. Neither existing real-bug benchmarks nor existing mutation techniques alone provide the control required for equivalent multi-framework repair evaluation. We align non-API faults through canonical-source mutation and validated cross-framework generation, and API faults through matched framework-specific edits. Bugs4Q-Robust is used separately to test whether conclusions extend beyond the injected benchmark.

6 Discussion and Future Work

Interpretation. RQ1 shows that repair difficulty depends strongly on both fault type and target framework. Local structural and API-related faults are generally repaired more often than several deeper quantum-semantic faults, while Qiskit has the highest aggregate baseline success and OpenQASM the lowest. These framework differences should not be interpreted as intrinsic rankings of the languages themselves, because source representation, available diagnostics, and baseline adaptations also differ.

Existing methods are most informative when faults produce clear syntax, API, construction, or framework errors. When a circuit remains legal and structurally plausible but semantically incor-

rect, conventional execution feedback provides little localization evidence. QuRepair supplies this missing evidence: linter and semantic-rule diagnostics flag suspicious operation patterns, API evidence exposes invalid calls, and the circuit rendering exposes operation order, operands, and parameters. Consistent with this, QuRepair retains most of its Qiskit performance on Cirq and PennyLane (70.4% and 65.8% vs. 77.6%), where the best baselines fall to 48.7% and 47.4%. OpenQASM has the lowest overall repair success, possibly because its lower-level representation obscures the algorithmic cues needed for semantic repair.

Self-Debugging (21.1%) and NL-Debugging (12.8%) fall well below direct repair with the same model (35.9%; Figure 2), and LDB (35.0%) roughly matches it. The first two repair from the model's own account of the program: Self-Debugging stops once the model declares the program correct, which is likely when a fault leaves the circuit legal and plausible, and NL-Debugging regenerates the whole program from a natural-language sketch, which can drop exact angles, qubit indices, and operation order.

Regarding controls, all methods generate code with the same LLM, and iterative methods observe at most a pass/fail verdict between attempts; QuRepair also queries Jev, yet is the fastest method per job and per repair (Appendix J). The closest comparison is QBugLM, which shares the localize-then-repair structure, five-attempt limit, and binary feedback; QuRepair outperforms it by 15.83 points.

The cross-model results additionally suggest that QuRepair benefits from stronger underlying models. Repair success rises from 48.19% with Qwen3-8B to 91.61% with Gemini 3.8 Flash and follows an approximately linear increasing trend across the evaluated model ordering. This indicates that QuRepair is not tied to a single model family, although absolute repair performance remains strongly dependent on model capability.

Limitations. The controlled benchmark should not be interpreted as an estimate of the natural frequency of quantum-software bug types. Fault injection intentionally trades ecological realism for experimental control: it lets us control and align local, API, and quantum-semantic fault classes across four frameworks, but no mutation scheme can reproduce the full distribution of mistakes made by developers. We therefore evaluate separately on Bugs4Q-Robust to reduce dependence on our own injection procedure. This external check is itself limited because Bugs4Q-Robust is centered on Qiskit and does not provide an equivalent real-bug corpus for Cirq, PennyLane, and OpenQASM (Zhao et al., 2023). The MQT-based programs also emphasize algorithmic circuits rather than large hybrid quantum-classical applications.

Cross-framework equivalence introduces another source of variation because equivalent computations and equivalent fault effects may require substantially different source structures in Qiskit, Cirq, PennyLane, and OpenQASM. Consequently, the framework differences reported in RQ1 reflect the complete experimental setting rather than an intrinsic measure of language difficulty.

Finite-shot execution also introduces statistical uncertainty near the semantic threshold. QuRepair also depends on an external typed decision model (Jev). Beyond the direct-repair comparison, which removes all evidence and the diagnosis stage simultaneously, isolating the individual contribution of each evidence channel and of Jev-based diagnosis is left to future work. Finally, several classical debugging pipelines require nontrivial adaptation because their original runtime assumptions do not directly transfer to quantum programs. We preserve their defining mechanisms and document material deviations, but the adapted implementations cannot be assumed identical to the original systems.

Future work. Larger collections of naturally occurring quantum bugs would strengthen generalization evaluation and could guide more realistic mutation operators. QuRepair could also extend to hybrid classical–quantum programs, dynamic circuits, hardware- and noise-related failures, and richer quantum representations. Further directions include multi-framework static analysis, framework-neutral semantic representations, improved mutation design, automatic classification of quantum-programming misconceptions, and more interactive or agentic debugging systems.

AI USE STATEMENT

Generative AI tools were used to assist with software implementation and manuscript drafting and editing, including improving readability. Generative AI was not used for ideation, experiment design, or interpretation of experimental results. The conceptual design of the proposed debugging method, the experimental methodology, and the interpretation of the results were developed and verified by the authors. Generative AI was not used to generate the synthetic faulty artifacts or to design, select, or validate the fault-injection instances used in the benchmark. The synthetic benchmark was constructed through the authors' deterministic fault-injection methodology and independently validated before inclusion. All AI-assisted code, citations, and manuscript content were reviewed and verified by the authors. Experimental implementations were tested against the evaluation procedures described in the paper. The authors take responsibility for the final content of this work, including all text, claims, code, and artifacts produced with the assistance of generative AI. Separately, LLMs are the object of study in this work; their use as repair models is described in Sections 3 and 4.

REPRODUCIBILITY STATEMENT

All implementations, evaluation scripts, benchmark construction code, and configuration files will be released upon publication. The submission provides sufficient detail to reproduce the setup: QuRepair and the baseline adaptations, with key prompts (Appendices A–B), benchmark construction, mutation procedures, and validation (Appendices C–E), evaluation criteria (Appendix H), and model configurations (Appendix I). Jev, like the other models, is publicly accessible through OpenRouter.

REFERENCES

Anthropic. System card: Claude opus 4 & claude sonnet 4, May 2025. URL https://www-cdn.anthropic.com/4263b940cabb546aa0e3283f35b686f4f3b2ff47.pdf.

Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. QL: Object-oriented queries on relational data. In *Proceedings of the 30th European Conference on Object-Oriented Programming (ECOOP)*, volume 56 of *Leibniz International Proceedings in Informatics*, pp. 2:1–2:25. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016. doi: 10.4230/LIPIcs.ECOOP.2016.2. URL https://doi.org/10.4230/LIPIcs.ECOOP.2016.2.

Abdul Basit, Minghao Shao, Muhammad Haider Asif, Nouhaila Innan, Muhammad Kashif, Alberto Marchisio, and Muhammad Shafique. QHackBench: Benchmarking large language models for quantum code generation using PennyLane hackathon challenges. In *2025 IEEE International Conference on Quantum Artificial Intelligence (QAI)*, pp. 316–322, 2025.

Xinyun Chen, Maxwell Lin, Nathanael Schaerli, and Denny Zhou. Teaching large language models to self-debug. In *International Conference on Learning Representations*, 2024.

Yunhe Feng, Xiaokun Yang, Kewei Sha, and Junhua Ding. Benchmarking the benchmarks for quantum code generation with llms. In *2026 IEEE 19th Dallas Circuits and Systems Conference (DCAS)*, 2026.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1536–1547. Association for Computational Linguistics, 2020. doi: 10.18653/v1/2020.findings-emnlp.139.

Alvise Fortunato, Fabiano Pecorelli, Fabio Palomba, and Andrea De Lucia. Mutq: Mutation testing for quantum programs. In *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 567–571, 2024.

Daniel Fortunato, José Campos, and Rui Abreu. Mutation testing of quantum programs: A case study with qiskit. *IEEE Transactions on Quantum Engineering*, 3:1–17, 2022.

Antonio García de la Barrera et al. Quantum software testing: State of the art. *Journal of Software: Evolution and Process*, 2023.

Google DeepMind. Gemini 3.8 flash. Model Card, September 2026. URL https://deepmind.google/models/model-cards/gemini-3-8-flash/.

Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33:1–79, 2024.

Harshit Joshi, José Cambronero Sanchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radiček. Repair is nearly generation: Multilingual program repair with llms. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pp. 5131–5140, 2023.

Peixun Long and Jianjun Zhao. Equivalence, identity, and unitarity checking in black-box testing of quantum programs. *Journal of Systems and Software*, 211:112000, 2024.

Junjie Luo, Pengzhan Zhao, Zhongtao Miao, Shuhan Lan, and Jianjun Zhao. A comprehensive study of bug fixes in quantum programs. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 1239–1246, 2022.

Taku Mikuriya, Tatsuya Ishigaki, Masayuki Kawarada, Shunya Minami, Tadashi Kadowaki, Yohichi Suzuki, Soshun Naito, Shunya Takada, Takumi Kato, Tamotsu Basseda, Reo Yamada, and Hiroya Takamura. QCoder Benchmark: Bridging language generation and quantum hardware through simulator-based feedback. In *Proceedings of the 18th International Natural Language Generation Conference*, pp. 743–752. Association for Computational Linguistics, 2025.

Yuki Ohto et al. On the reproducibility of quantum software defect datasets: A case study of bugs4q. *arXiv preprint arXiv:2606.27124*, 2026.

OpenAI. Gpt-4o system card. System Card, August 2024. URL https://openai.com/index/gpt-4o-system-card/.

Matteo Paltenghi and Michael Pradel. Bugs in quantum computing platforms: An empirical study. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1):1–27, 2022.

Matteo Paltenghi and Michael Pradel. Analyzing quantum programs with LintQ: A static analysis framework for Qiskit. *Proceedings of the ACM on Software Engineering*, 1(FSE):2144–2166, 2024.

An B. B. Pham, Hoa T. Nguyen, and Muhammad Usman. QBugLM: An agentic benchmarking framework for llm-based quantum software debugging. In *2026 IEEE International Conference on Quantum Software (QSW)*, pp. 185–191, 2026.

Nils Quetschlich, Lukas Burgholzer, and Robert Wille. MQT Bench: Benchmarking Software and Design Automation Tools for Quantum Computing. *Quantum*, 7:1062, 2023. doi: 10.22331/q-2023-07-20-1062. MQT Bench is available at https://mqt-bench.app/.

Qwen Team. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Neilson Carlos Leite Ramalho, Higor Amario de Souza, and Marcos Lordello Chaim. Testing and debugging quantum programs: The road to 2030. *ACM Transactions on Software Engineering and Methodology*, 34(5):1–46, 2025.

Ali Slim, Haydar Hamieh, Jawad Kotaich, Yehya Ghosn, Mahdi Chehimi, Ammar Mohanna, Hasan Abed Al Kader Hammoud, and Bernard Ghanem. QuanBench+: A unified multi-framework benchmark for LLM-based quantum code generation. In *ICLR 2026 Workshop on I Can't Believe It's Not Better*, 2026.

TypeSafe AI. Jev: A non-autoregressive typed decision model. https://typesafe.ai, 2026.

Weiming Zhang, Qingyao Li, Xinyi Dai, Jizheng Chen, Kounianhua Du, Weiwen Liu, Yasheng Wang, Ruiming Tang, Yong Yu, and Weinan Zhang. NL-debugging: Exploiting natural language as an intermediate representation for code debugging. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 1533–1549. Association for Computational Linguistics, 2025.

Pengzhan Zhao, Jianjun Zhao, and Lei Ma. Identifying bug patterns in quantum programs. In *2021 IEEE/ACM 2nd International Workshop on Quantum Software Engineering (Q-SE)*, pp. 16–21, 2021.

Pengzhan Zhao, Zhongtao Miao, Shuhan Lan, and Jianjun Zhao. Bugs4Q: A benchmark of existing bugs to enable controlled testing and debugging studies for quantum programs. *Journal of Systems and Software*, 205:111805, 2023.

Li Zhong, Zilong Wang, and Jingbo Shang. Debug like a human: A large language model debugger via verifying runtime execution step by step. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 851–870. Association for Computational Linguistics, 2024.

A QuRepair: Full Specification

Overview. QuRepair is our proposed quantum-program repair framework. Unlike previously published debugging pipelines, it is designed around a quantum-specific evidence generation layer combined with a Jev-based diagnosis stage and an LLM repair stage. Each iteration operates on the current candidate program. Evidence is regenerated from the current source, Jev identifies likely faults from the available evidence, and a repair model generates a complete same-framework candidate that is evaluated by the shared quantum evaluator.

The evaluated configuration permits at most five repair iterations. Let C_0 denote the original faulty program. Iteration i analyzes C_{i-1} and produces candidate C_i. If C_i fails evaluation, it becomes the source for iteration $i + 1$. Source-derived diagnostics, API evidence, circuit rendering when available, and semantic diagnostics are recomputed for the updated candidate rather than remaining anchored to the original faulty program.

Stage 1: Static quantum-code diagnostics. The current program is analyzed by a framework-aware source-level analyzer. The analyzer identifies suspicious behavior that can be detected from the program source, including parser issues, framework misuse, dataflow problems, circuit-construction issues, operation-level inconsistencies, and measurement related problems. The resulting report is diagnostic only: it highlights potentially relevant source regions and conditions but does not directly modify the program.

The normalized report records the framework, diagnostic status, diagnostic records, available source locations, and analysis errors when applicable. Diagnostics distinguish strong failures from weaker structural clues. The analyzer receives neither mutation metadata nor the clean reference program.

Stage 2: API evidence. The current source is additionally inspected for framework API usage. This layer reports API calls that are unavailable or suspicious according to the configured framework interface. When inspectable, it provides the API name, availability status, and signature information. The API layer does not provide full framework documentation and does not directly generate repairs.

API evidence remains useful when circuit construction fails because repair can continue using source-level diagnostics, API information, semantic rules, task information, and runtime feedback.

Stage 3: Circuit-level representation. When circuit construction succeeds, QuRepair produces a deterministic textual representation of the quantum computation generated by the current source. Qiskit circuits are rendered through the configured Qiskit representation, Cirq circuits through their textual representation, PennyLane circuits through the configured drawing mechanism, and OpenQASM programs through the configured adapter.

The circuit representation exposes operation order, operands, parameters, and measurements while reducing dependence on host-language details. It is derived from the current candidate program and is therefore treated as evidence about the implementation rather than correctness information. If construction or rendering fails, the circuit representation is marked unavailable and the iteration continues using the remaining evidence channels.

Stage 4: Quantum-semantic evidence. Source-derived quantum-semantic diagnostics are generated as part of the framework-aware analysis pipeline. These diagnostics capture suspicious quantum-program behaviors such as repeated operations, operations after measurement, measurement inconsistencies, gate-pattern anomalies, and unusual qubit participation. In addition, a fixed catalogue of quantum-semantic rules defines higher-level semantic conditions considered during diagnosis. This semantic evidence is derived only from the current candidate and does not contain mutation metadata, clean reference behavior, or expected repairs.

Stage 5: Jev diagnosis. Jev (TypeSafe AI, 2026) receives the current source program together with the framework-aware diagnostics, API evidence, quantum-semantic evidence, and available execution feedback. It evaluates the applicable semantic rules and other evidence to produce a structured typed diagnosis identifying the most likely fault. The circuit-level representation is not supplied to Jev and is reserved for the subsequent repair stage.

Stage 6: Repair. The repair LLM receives the task specification, current source program, structured Jev diagnosis, execution contract, available circuit representation, runtime feedback, and retry history. It generates a complete same-framework repair. After failure, the repaired candidate becomes the source for the next iteration, and all applicable evidence channels are recomputed from the updated candidate. The process continues for up to five iterations.

Generation configuration. The repair model uses temperature 0.0 and a maximum completion cap of 16,384 tokens. Jev requests are typed Decisions API calls and do not use completion-token limits. Each iteration normally produces one repair completion, while the number of Jev requests may vary depending on batching and follow-up requirements.

The dedicated quantum-aware launcher applies a 300-second repair-model timeout. The pipeline timeout is configured to 1,800 seconds and is checked at iteration boundaries rather than acting as an external hard interrupt.

No retrieval, example bank, token-log-probability ranking, or multi-candidate repair generation is used.

Candidate extraction. The repair model is instructed to call the repair submission tool exactly once and provide the complete repaired source as the submitted program. Markdown fence extraction is only a compatibility fallback. The primary expected output is therefore the submitted complete source program rather than a required single Markdown block.

If fallback extraction is required, compatible complete-program blocks may be selected according to the extraction rules implemented by the pipeline. The presence of multiple code fences does not automatically imply failure if a valid complete-program block can be identified.

Evaluation feedback and failed-attempt history. Each extracted candidate is evaluated once using the shared hidden evaluator. The clean program, reference distribution, mutation metadata, candidate distribution, and KL divergence values are never exposed to the repair process.

Runtime failures are converted into feedback available to subsequent iterations. When a candidate executes but fails the semantic comparison, the model receives only the available failure information rather than the hidden reference behavior. Previous failed attempts are summarized and provided as negative evidence.

Iteration and evidence recomputation. QuRepair follows candidate chaining. The original faulty source is used only for the first iteration. After each failed repair, the successfully extracted candidate becomes the new source program. Before the next iteration, source diagnostics, API evidence, circuit rendering when possible, and semantic diagnostics are recomputed from the latest candidate.

Therefore, the debugging trajectory is a sequence of program revisions:

$$C_0 \rightarrow C_1 \rightarrow C_2 \rightarrow \cdots$$

rather than independent repair attempts over a fixed faulty program.

Stopping and failure handling. A candidate satisfying the shared semantic evaluator terminates the pipeline immediately. Otherwise, iterations continue until the five-iteration limit is reached. If all attempts fail, the latest successfully extracted candidate is returned.

Parse errors and circuit-construction failures do not automatically terminate repair. Instead, they are represented as available evidence. Invalid configuration, missing dependencies, unrecoverable model failures, extraction failures, and malformed evaluator responses are handled as pipeline failures.

Design rationale. QuRepair separates deterministic evidence generation from model-based reasoning. Static diagnostics expose source-level issues, API evidence captures framework usage problems, circuit representations expose the implemented quantum computation when available, and Jev provides structured reasoning over quantum-specific evidence. Candidate chaining ensures that

every subsequent iteration reasons about the actual program being repaired rather than stale information from the original faulty source.

B Complete Debugging Method Specifications

All debugging methods operate on the same task contract: a faulty program, a natural-language specification, a target framework, and the callable or program interface expected by the evaluator. The clean reference implementation, injected mutation identity, and mutation location are never exposed to the models. Provider retries caused by transient API failures are handled by the shared LLM wrapper and are not counted as debugging iterations.

For every model call, we store the rendered prompt, raw response, extracted artifact, model identifier, decoding parameters, input and output token counts, and finish reason. Code-producing calls are processed by the same framework-aware extraction utility. Extraction removes presentation formatting but does not rewrite generated code. Every executable candidate is evaluated by the common evaluator described in Appendix H.

B.1 Self-Debugging

Our implementation adapts the explanation-only, or "rubber-duck," variant of Self-Debugging (Chen et al., 2024) to the repair of pre-existing faulty quantum programs. The original generation stage is not reproduced; debugging begins directly from the faulty program supplied by the common experimental interface. No clean implementation, reference distribution, mutation type or location, simulator output, execution error, or evaluator result is exposed to the model during debugging.

Interaction sequence. The initial conversation contains the task specification and faulty source program. Each debugging turn then consists of two model calls. The first asks the model to explain the current program:

```
Explain the <framework> code above line by line,
including every quantum gate and its effect on the
circuit behavior.
```

The second call asks the model to assess the program against the task specification using the generated explanation. The model must either declare the current program correct or return a complete replacement. The accepted response protocol is:

```
The code above is correct.
```

or

```
The code above is wrong.  Please fix the code.
<complete replacement program>
```

The full conversation history is retained between turns, so a replacement generated in one turn becomes the current program visible in the following turn.

Candidate extraction and stopping. A "wrong" verdict must contain one extractable complete program. For Qiskit, Cirq, and PennyLane, the implementation accepts a Python-labelled or unlabelled fenced code block. For OpenQASM, it accepts an OpenQASM-compatible label or an unlabelled fenced block. Failure to satisfy the response protocol terminates the method with a pipeline error.

The loop terminates when the model declares the current program correct, when the response protocol is violated, or when the configured debugging-turn limit is reached. Each completed turn therefore incurs exactly two model calls. The experimental configuration uses a maximum of five debugging turns.

Evaluator interaction. Self-Debugging receives no execution feedback during the repair process. After the method terminates, only its final candidate is passed to the common hidden evaluator described in Appendix H. A model declaration that the program is correct therefore represents internal convergence only and does not determine experimental repair success. Conversely, a candidate produced at the turn limit may still pass the final evaluator even if the model never explicitly declared it correct.

Adaptation from the original method. The adaptation preserves the explanation-driven self-debugging mechanism but changes the surrounding task setting. The original Self-Debugging experiments begin from model-generated candidate solutions and consider multiple feedback settings, including execution-based feedback. Our implementation instead begins from externally supplied faulty quantum programs and consistently uses the explanation-only setting without debugging-time execution feedback. Final correctness is determined exclusively by the shared quantum behavioral evaluator rather than by the model's self-assessment.

B.2 LDB

Adaptation. We implement a quantum adaptation of LDB rather than a literal reproduction. The original method partitions executed classical programs into CFG-derived basic blocks and exposes the values of in-scope variables before and after each block. Our adaptation preserves LDB's central structure of intermediate block-level judgment followed by complete-program regeneration, but replaces classical basic blocks and variable traces with partitions of quantum operations and circuit-prefix behavioral summaries.

A direct use of host-language basic blocks was unsuitable for quantum programs. Classical control-flow boundaries need not correspond to meaningful changes in the constructed quantum computation, and a single host-language function call may expand into a large sequence of quantum operations. In preliminary implementations, retaining this decomposition also produced excessively large debugging contexts. We therefore perform decomposition at the circuit-operation level.

Quantum block construction. Operations are retained in circuit order and deterministically partitioned into blocks. A block terminates after a barrier, reset or initialization operation, classically controlled operation, or a contiguous terminal run of measurements. Otherwise, a block terminates after ten operations. Every operation belongs to exactly one block.

These blocks are not CFG basic blocks: the partitioning does not follow host-language branches, loops, or function boundaries. The ten-operation rule provides a fallback boundary for unitary regions that contain no explicit quantum-semantic boundary.

When more than ten blocks are available, the first five and last five are retained for judgment, following LDB's endpoint-selection strategy. If the resulting prompt still exceeds the configured context budget, the selected set is reduced further while preserving a deterministic first/last subset. Omitted block identifiers are reported to the model.

Circuit-prefix traces. For each selected block, the circuit prefix ending at that block is executed to construct intermediate behavioral evidence. The model receives a compressed representation rather than the complete stored trace. For each block, the representation contains its operation-index range and operation kinds, the most probable exit basis state and probability, omitted probability mass, the largest probability change across the boundary with an optional phase change, and at most one statevector amplitude when statevector extraction is available.

The serialized trace begins with the following header:

```
Block operation_range is inclusive; operation text is in the source
program. Snapshot keys: l=operation limit; p=top [basis,probability];
po=omitted probability mass; sv=top [basis,real,imag]. Change entries
are [basis, probability delta, phase delta degrees].
```

Each selected block is then represented as a compact JSON object. For example:

```
{"block_id":"BLOCK-000","block_index":0,
```

```
"exit_snapshot":{"l":1,"p":[["00",1.0]],"po":0.0,
"sv":[["00",1.0,0.0]]},
"largest_probability_changes":[["00",0.0,null]],
"operation_kinds":["unitary"],"operation_range":[0,0],
"trace_status":"complete"}
```

The JSON serialization uses sorted keys and compact separators. The trace ends with either

```
No blocks were omitted.
```

or a list of omitted middle blocks. In the evaluated implementation, this omission note is also inserted separately into the judgment template and is therefore present twice in the final judgment prompt.

Judgment call. The initial conversation contains a system instruction identifying the model as an expert quantum programming assistant, followed by the task specification, framework requirements, and faulty source program. The judgment call appends the binary result of the private evaluator and the selected circuit-prefix trace. Its user prompt is:

```
The quantum program above fails evaluation against the specification.
Inspect every supplied quantum block in order and decide whether its
sampled transition is consistent with the intended behavior.

Evaluation feedback:
{final_evaluation_feedback}

Selected block trace:
{selected_block_trace}

{trace_limit_note}

Quantum snapshot caveat:
{quantum_snapshot_caveat}

Examples:
{"block":"BLOCK-000","correct":true,
"explanation":"The sampled transition agrees with the intended
operation behavior."}
{"block":"BLOCK-001","correct":false,
"explanation":"The sampled transition contradicts the intended
behavior; this block should be inspected for the faulty operation."}

Output only the verdict JSON--do not include analysis, Markdown, or
program code. Return one JSON object per supplied block, in supplied
order. Each object must contain `block`, boolean `correct`, and a
short non-empty `explanation`. An array of these objects is also
accepted.
```

The evaluation-feedback field contains only one of:

```
The code is correct.
```

or

```
The code is incorrect.
```

An initially correct candidate bypasses the debugging loop, so ordinary LDB judgment iterations receive the latter.

When statevector information is available, the judgment prompt additionally states that complex amplitudes and relative phases may be used to reason about probability magnitudes, phase relationships, and entanglement. When only sampled probabilities are available, it warns that phase errors may remain unobservable until a later interference operation.

The model must return one verdict for every supplied block. Each verdict contains the block identifier, a Boolean correctness judgment, and a non-empty natural-language explanation.

Repair call. The complete raw judgment response is retained as an assistant message, so the repair call receives both the block-level correctness decisions and their explanations. The subsequent repair instruction is:

```
Please fix the {framework_name} code. Return the complete program in
one Markdown code block, using `python` for Python frameworks or
`openqasm3` for OpenQASM.
```

Thus, the repair stage regenerates the complete program rather than editing individual blocks.

Iteration, history, and stopping. The evaluated implementation permits five debugging iterations. Each normal iteration consists of one block-judgment call followed by one complete-program repair call. Only the immediately preceding completed iteration is retained in ordinary debugging history; earlier judgment and repair exchanges are removed.

After each repaired candidate is produced, the shared private evaluator determines whether another debugging iteration is required. The model does not receive the clean reference distribution, the numerical divergence value, or the candidate distribution. It receives only the binary correct/incorrect feedback described above.

Malformed or incomplete judgment output terminates the method; no additional format-repair call is performed. If the iteration limit is reached without a passing candidate, the last evaluable repaired program is returned rather than the candidate with the lowest divergence.

Profiling failure. If quantum block profiling or trace construction fails, the implementation falls back to direct whole-program repair. This fallback is an engineering extension and is not part of the original LDB procedure. The following message is appended:

```
Runtime block profiling was unavailable for the current candidate
({error_type}): {error_message}. Repair the complete program directly
from the task specification.
```

It is immediately followed by the standard complete-program repair instruction shown above. No block judgment is performed in this fallback path.

Relation to the original LDB method. Table 3 summarizes the material differences between the original LDB procedure and our quantum adaptation.

Table 3: Principal differences between LDB and its quantum adaptation.

Original LDB	Quantum adaptation
CFG-derived executed basic blocks	Ordered partitions of constructed quantum operations.
Concrete values of in-scope classical variables	Circuit-prefix measurement summaries and compressed state information.
Failed visible unit test provides debugging stimulus	A private behavioral evaluator supplies only a binary correctness verdict.
Visible tests determine continuation	The shared quantum evaluator determines continuation after each repair.
Python program execution	Qiskit, Cirq, PennyLane, and OpenQASM quantum programs.
Up to ten debugging iterations	Five debugging iterations in the evaluated implementation.

B.3 RING ADAPTATION

RING (Joshi et al., 2023) is included as a representative example-based program repair baseline. The original method was designed for general-purpose programming languages and performs repair through three main components: diagnostic-based localization, few-shot repair generation, and candidate ranking. We implement a quantum adaptation of RING that preserves these core mechanisms while replacing the original programming domains with quantum programs.

The adaptation should therefore be viewed as a controlled transfer of RING to quantum debugging rather than an exact reproduction of the original system. The goal is to evaluate how a general-purpose repair strategy performs when applied to quantum programs without providing it with quantum-specific reasoning capabilities.

Unlike QuRepair, the adapted RING baseline does not receive quantum semantic evidence. In particular, it does not use circuit representations, framework-specific semantic rules, API documentation evidence, execution feedback, mutation metadata, mutation locations, or reference behavior during generation. Its input consists only of the faulty source program, available language-tool diagnostics, and retrieved buggy/fixed repair examples.

Repair Workflow. The adapted RING method follows the original one-shot repair formulation. Given a faulty quantum program, it performs the following steps:

1. obtain available parser or importer diagnostics;
2. retrieve similar buggy/fixed quantum repair examples;
3. generate multiple repair candidates using few-shot prompting;
4. rank candidates using model likelihood;
5. evaluate the three highest-ranked candidates using the shared quantum behavioral evaluator.

As in the original RING formulation, evaluation results are not returned to the model and no iterative repair-feedback loop is performed. Each repair attempt is therefore independent of the evaluator outcome.

Diagnostic-Based Localization. RING does not perform a separate learned fault-localization stage. Instead, localization information is obtained from available language tooling and provided directly to the repair prompt.

For Python-based quantum frameworks (Qiskit, Cirq, and PennyLane), the source program is first processed using Python's abstract syntax tree parser. When parsing fails, the resulting diagnostic message and source location are included:

```
Error: <parser message>.
Error in line: <line>, span starts <start> and ends <end>.
```

For OpenQASM programs, the implementation uses the Qiskit OpenQASM importer. Importer failures are converted into diagnostics supplied to the model.

When no parser or importer diagnostic is produced, RING receives no explicit localization information. Consequently, API misuse, runtime failures, and quantum-semantic faults are not localized by this stage. The baseline is not provided with mutation locations, execution traces, failing assertions, or reference probability distributions.

Few-Shot Example Retrieval. Following RING, repair generation is guided by buggy/fixed demonstrations. The adapted example database contains framework-specific quantum repair examples for Qiskit, Cirq, PennyLane, and OpenQASM. Each example contains a faulty program, its diagnostic when available, and the corresponding repaired program.

When a diagnostic is available, examples are selected using diagnostic-message similarity. Diagnostic texts are embedded using the pretrained `microsoft/codebert-base` (Feng et al., 2020). The final hidden representations are mean-pooled over non-padding tokens and L2-normalized. Candidate examples are then ranked using cosine similarity between the target diagnostic and each example diagnostic.

The evaluated configuration uses three few-shot demonstrations. Examples corresponding to the same task identifier as the target, as well as identical faulty/fixed pairs, are excluded during selection.

If the target program has no diagnostic, diagnostic-based retrieval cannot be performed. In this case, the implementation falls back to three fixed framework-specific demonstrations.

Prompt Construction. The repair prompt follows RING's completion-style formulation. Each retrieved demonstration is represented as:

```
### Buggy <framework>
<buggy program>
<diagnostic>

### Fixed <framework>
<fixed program>
```

The faulty target program is appended after the demonstrations. The prompt does not include the clean reference program, mutation type, mutation location, execution result, expected probability distribution, or evaluator feedback.

Therefore, unlike QuRepair, RING must infer repairs primarily from the faulty source code, available diagnostics, and retrieved examples.

Candidate Generation and Ranking. The adapted implementation generates ten candidate repairs from a single prompt. Sampling uses temperature 0.7 and $\texttt{top_p} = 1.0$. Candidate outputs are ranked using the average log-probability of generated tokens, following the ranking mechanism used by RING.

For a generated candidate containing tokens $y_1, \ldots, y_n$, the ranking score is:

$$s(y) = \frac{1}{n} \sum_{i=1}^{n} \log p(y_i \mid y_{<i}, x),$$

where x denotes the repair prompt. Candidates with higher scores are ranked first. Duplicate candidates and candidates identical to the faulty program are removed before ranking.

The original RING evaluation considers multiple ranked candidates rather than only the highest-ranked completion. Following this evaluation philosophy, our implementation evaluates the three highest-ranked unique candidates after generation. The ranking mechanism is therefore preserved, while correctness is determined using the quantum behavioral evaluator described below.

Evaluation. RING does not use execution feedback during generation. After ranking, the three highest-ranked unique candidates are evaluated independently using the shared quantum behavioral evaluator described in Appendix H.

Repair success is measured using a Top@3 criterion: an instance is considered successfully repaired if at least one of the three evaluated candidates passes the evaluator. Successful lower-ranked candidates are not used to update the prompt or influence subsequent generations.

The original RING evaluation relied on language-specific correctness criteria. In contrast, our evaluation measures quantum behavioral equivalence. A candidate repair is successful only if it executes successfully and produces a probability distribution whose KL divergence from the reference behavior is below the predefined threshold.

Differences from Original RING

Table 4 summarizes the main changes introduced by the quantum adaptation.

B.4 NL-DEBUGGING

Adaptation. We implement a quantum adaptation of NL-Debugging (Zhang et al., 2025). The central idea of NL-Debugging is to perform repair through a natural-language intermediate representation rather than modifying the faulty program directly. The method first backtranslates the

Table 4: Principal implementation differences between RING and the quantum adaptation.

Original RING	Quantum adaptation
Compiler, parser, static analyzer, and linter diagnostics	Parser/importer diagnostics only when available; no quantum semantic diagnostics
Diagnostic-driven smart example selection evaluated through benchmark leave-one-out construction	CodeBERT diagnostic retrieval when diagnostics exist, otherwise fixed demonstrations
Generation evaluated using language-specific correctness criteria and edit constraints	Repair correctness determined by a quantum behavioral evaluator without an edit-distance constraint
Candidate evaluation reported across larger Top@k settings	Ten generated candidates with Top@3 evaluation
Post-generation localization analysis of repaired edits	No localization accuracy measurement
Completion-style Codex generation	Chat-based generation requiring code extraction from model responses

current program into a high-level natural-language sketch, analyzes the observed failure, refines the sketch in natural-language space, and finally regenerates executable code from the corrected sketch.

Our implementation preserves this backtranslation–refinement–regeneration structure while adapting the prompts, execution feedback, and regeneration requirements to quantum programs. The method is iterative: a regenerated candidate is evaluated and, if it remains incorrect, becomes the current program for the following debugging iteration.

Interaction sequence. Before debugging begins, the current program is evaluated using the common evaluator described in Appendix H. If the program already passes evaluation, it is returned immediately. Otherwise, each debugging iteration consists of four model calls:

1. backtranslation of the current program into a natural-language sketch
2. analysis of the failure and its likely root cause
3. refinement of the natural-language sketch using that analysis
4. regeneration of a complete executable program from the refined sketch.

The regenerated program is then evaluated. If it passes, debugging terminates. Otherwise, it becomes the current candidate for the next iteration.

The evaluated configuration uses `continuous=False`. Consequently, the natural-language representation is not simply carried forward as the sole representation of the program. Instead, after an unsuccessful repair, the newly generated source program is backtranslated again at the beginning of the following iteration. This maintains the code-to-natural-language-to-code cycle used by NL-Debugging.

Backtranslation. The first model call converts the current quantum program into a high-level natural-language representation. The implementation uses the *Sketch* representation from NL-Debugging rather than the alternative pseudocode or key-point representations investigated in the original paper.

The principal instruction is:

```
Translate the supplied quantum program into a precise
natural-language sketch of its intended algorithmic
operations.
```

The model receives the task specification together with the current source program. The purpose of this stage is to represent the program's algorithmic and quantum behavior in natural language, allowing subsequent debugging to operate on its higher-level logic rather than directly on source-code syntax.

For a quantum program, the sketch may therefore describe, for example, state preparation, gate applications, control and target relationships, parameterized rotations, entangling operations, measurements, and the intended flow of the constructed circuit.

The backtranslation stage does not itself modify the source program.

Execution feedback. After evaluation, the implementation constructs a compact description of the observed outcome for use during natural-language debugging. The feedback interface supports the following failure classes:

```
syntax_error
invalid_output
execution_error
distribution_mismatch
unknown_failure
```

The available evaluator information is converted into this representation before being supplied to the analysis and refinement stages. The clean reference implementation is never exposed to the model.

This differs in representation from the original NL-Debugging setting. The original method executes the current program against visible test cases and constructs feedback from the test input, expected output, and observed execution output. Our adaptation instead communicates evaluator-compatible feedback suitable for quantum programs, for which correctness may depend on program execution, circuit behavior, or output distributions rather than a single deterministic classical output.

Failure analysis. The second model call performs explicit root-cause analysis. Its inputs include the task specification, current source program, generated natural-language sketch, and available execution feedback.

The model is instructed to provide a short explanation of why the current program failed. The response is restricted to approximately two or three sentences and must identify the root cause without providing code, a corrected program, or explicit repair instructions.

Conceptually, the analysis stage computes

$$A_t = f_{\mathrm{LLM}}\left(S,\, C_t,\, N_t,\, F_t\right),$$

where S is the task specification, C_t is the program at iteration t, N_t is its backtranslated natural-language sketch, and F_t denotes the available execution feedback.

This stage provides an explicit reasoning step between observing a failure and modifying the natural-language representation. It is therefore not a formal statistical or spectrum-based fault-localization procedure. Instead, localization is implicit: the model infers the likely erroneous component or assumption from the program, its natural-language representation, the task specification, and the available execution evidence.

Natural-language refinement. The third model call modifies the natural-language representation rather than the program itself. The model receives the original sketch, source program, task specification, execution feedback, and the root-cause analysis generated in the preceding stage.

It is instructed to produce a corrected natural-language sketch that addresses the identified failure. No executable code is requested at this stage. Thus, if the backtranslation is denoted by N_t and the analysis by A_t, the refinement stage produces

$$N'_t = f_{\mathrm{LLM}}\left(S,\, C_t,\, N_t,\, F_t,\, A_t\right).$$

The intent is to move the repair operation into natural-language space. Rather than asking the model to locally edit individual source-code tokens, the model can revise the higher-level description of how the program should operate. This may involve changing an incorrect algorithmic assumption, reordering operations, correcting a control/target relationship, revising a measurement strategy, or otherwise altering the description from which the next implementation will be generated.

The original NL-Debugging method similarly separates analysis from refinement: the model first reasons about the cause of the observed failure and then uses that analysis to construct a corrected natural-language sketch.

Code regeneration. The final model call translates the refined sketch back into executable quantum code. The regeneration prompt contains the task specification, refined natural-language representation, target framework, and the execution requirements imposed by the common experimental interface.

The model must return one complete program in a single Markdown code block. Python-based frameworks use a `python` fence, whereas OpenQASM uses an OpenQASM-compatible fence. Explanatory prose and multiple alternative implementations are not accepted.

The resulting candidate can be represented as

$$C_{t+1} = f_{\mathrm{LLM}}\left(S,\, N'_t,\, R\right),$$

where R denotes the framework and execution-interface requirements.

The regeneration stage therefore produces the complete program rather than a localized patch. This is consistent with the original NL-Debugging formulation, in which executable code is regenerated from the corrected natural-language representation.

Candidate extraction. The regeneration response must contain an extractable complete program. For Qiskit, Cirq, and PennyLane, the implementation expects a Python-compatible fenced code block. For OpenQASM, it expects an OpenQASM-compatible fenced block.

The extracted program is normalized and passed to the common evaluator. Failure to obtain a valid program from the regeneration response terminates the method with a pipeline error; no additional format-repair call is made.

Unlike Self-Debugging and LDB, the four stages of an NL-Debugging iteration are issued as separate model requests rather than being maintained as a single growing conversation. Information from an earlier stage is explicitly inserted into the prompt for the subsequent stage.

Iteration and stopping. The evaluated implementation permits a maximum of five debugging iterations. Each complete unsuccessful iteration normally requires four model calls: one for backtranslation, one for root-cause analysis, one for natural-language refinement, and one for code regeneration. A task that exhausts all five iterations can therefore require up to twenty debugging-time model calls.

The generation temperature is fixed to 0, and one output is requested from each stage.

After every regeneration, the resulting program is evaluated. The loop terminates when the candidate passes evaluation, when one of the model stages fails to satisfy its output protocol, or when the five-iteration limit is reached. Evaluator feedback therefore determines whether another backtranslation–refinement–regeneration cycle is required.

Natural-language representation. The original NL-Debugging study investigates several possible intermediate representations, including pseudocode, key points, and natural-language sketches. Our implementation uses only the Sketch representation.

The sketch representation is intentionally more descriptive than a compact list of key points while remaining more abstract than executable code. Its purpose is to expose the program's overall reasoning and algorithmic structure so that a logical error can be revised without constraining the model to a small source-level modification.

No explicit search over alternative natural-language representations is performed during our experiments. Similarly, each stage generates a single representation rather than sampling multiple natural-language candidates and selecting among them.

Evaluator interaction. Execution feedback is used during the iterative debugging process, but the model does not receive the clean reference implementation or direct access to the evaluator itself. Only the feedback representation constructed by the method is inserted into subsequent analysis and refinement prompts.

The evaluator is invoked after the original candidate and after regenerated candidates to determine whether debugging should continue. A candidate is considered experimentally repaired only according to the common evaluator; the model's natural-language analysis or refined sketch does not independently determine correctness.

Relation to the original NL-Debugging method. The adaptation preserves the defining mechanism of NL-Debugging (Zhang et al., 2025): transforming faulty code into natural language, analyzing execution feedback in conjunction with that representation, correcting the program logic in natural-language space, and regenerating executable code from the refined representation. It also retains the Sketch representation used as the default configuration in the original method, temperature 0, iterative execution-guided repair, and a maximum of five debugging iterations.

The principal differences arise from adapting the method from classical algorithmic programming tasks to quantum programs. The original experiments obtain detailed visible-test feedback containing the input, expected output, and current program output. Our implementation converts the feedback available from the common quantum evaluator into a smaller set of execution-status categories.

The original study also evaluates multiple intermediate natural-language representations and analyzes their effects on debugging performance. Our implementation fixes the representation to Sketch and does not reproduce the paper's pseudocode, key-point, natural-language sampling, diversity, or long-chain-of-thought experiments.

B.5 QBugLM Adaptation

Adaptation. We implement a quantum debugging method adapted from QBugLM rather than a direct reproduction of the original system. The adaptation preserves the Finder–Fixer decomposition and simulation-based validation workflow while modifying the surrounding evaluation setting to enable controlled comparison with other debugging methods.

The original QBugLM workflow consists of four conceptual stages:

1. **QBugGen** generates or injects faulty quantum programs.
2. **QBugFind** localizes and classifies the suspected fault.
3. **QBugFix** generates a corrected candidate program.
4. **QBugCheck** validates the candidate and determines whether further debugging is required.

In our evaluation, QBugGen is not executed because faulty programs are generated before debugging through the shared benchmark construction procedure. Instead, each method receives the same pregenerated faulty instance, framework-specific execution interface, and hidden behavioral oracle. QBugFind and QBugFix are retained as the localization and repair stages. The validation role of QBugCheck is kept but implemented by the shared evaluator. No stage receives access to the clean program, mutation metadata, reference distribution, numerical evaluator scores, or other method-specific information unavailable to competing approaches.

Unlike the original OpenQASM-oriented setting, our implementation extends QBugLM to Qiskit, Cirq, PennyLane, and OpenQASM. QBugCheck is also replaced with the shared behavioral evaluator used throughout the experiments, which determines correctness using the output-distribution criterion described in Appendix H.

QBugFind. QBugFind is implemented as a tool-using language-model agent that performs fault localization and classification. It receives the task specification, current faulty program, framework requirements, attempt number, and abbreviated information from previous failed attempts. It does not receive simulator execution traces, runtime exceptions, the reference distribution, KL-divergence values, clean source code, or mutation metadata.

The Finder is instructed to produce a structured report containing:

```
BUG_LINE: <line number(s)>
BUG_TYPE:
MUTATED:
REASON:
```

The returned bug categories are normalized into four groups:

```
deprecated_syntax
structural_error
gate_overuse_misuse
semantic_deviation
```

Alternative model-generated labels are mapped into these categories before being passed to the repair stage. Malformed reports are handled according to the failure procedure described below.

QBugFix. QBugFix receives the Finder report, task specification, framework requirements, current faulty program, previous attempt history, and restricted file tools. It is instructed to produce a minimal repair while preserving the intended algorithmic behavior. The Fixer writes exactly one complete candidate program to the designated output file; code returned only in the conversational response is not considered an executable candidate.

QBugCheck. Candidates that satisfy syntax and framework-shape validation are evaluated by the shared quantum evaluator. The evaluator executes the candidate and compares its output distribution with the reference distribution using the KL-divergence criterion described in Appendix H. The model receives only a binary correctness outcome and never observes numerical evaluation feedback.

Iteration and configuration. The repair loop allows up to five attempts. Each attempt follows the sequence: QBugFind, QBugFix, structural validation, and behavioral evaluation. Previous attempts are summarized through retry history, while detailed evaluator information remains private. The evaluated configuration uses `openai/gpt-4o` with temperature 0.0 and the same model configuration used for the other baseline methods.

Relation to the original QBugLM method. The original QBugLM framework combines QBugGen, QBugFind, QBugFix, and QBugCheck in an OpenQASM-centered debugging environment. Our implementation retains the core Finder–Fixer decomposition but changes the experimental setting in three ways: faulty programs are generated externally rather than by QBugGen, QBugCheck is replaced by the shared distributional behavioral criterion used for all methods, and the workflow is extended from OpenQASM to all four frameworks. These modifications ensure that QBugLM is compared under the same benchmark, oracle, and correctness standard as the remaining debugging methods while preserving its original localization and repair structure.

C Program Selection

The controlled benchmark is built from parameterized quantum-program factories rather than from isolated hand-written snippets. The source suite contains 36 configured MQT-derived program families before final retention. After applying the framework-specific construction and validation requirements described in the main paper, the retained corpus contains 34 benchmark families and 136 clean framework-specific program instances.

The frozen *non-API* mutation run records 541 accepted mutation groups. Each accepted group has one retained faulty artifact for Qiskit, Cirq, PennyLane, and OpenQASM, yielding 2,164 framework-specific non-API faulty artifacts. API misuse is generated and validated through a separate locked cross-framework catalog described in Appendix D; API groups are therefore not included in the 541/2,164 non-API counts.

Primary evaluation subset selection. The primary controlled evaluation uses a fixed subset of 152 benchmark tasks selected from the validated benchmark pool before running any of the debugging methods; repair outcomes or model performance therefore did not influence task inclusion. From the validated mutants, we selected tasks to provide coverage across the available algorithm families and fault families while keeping the evaluation computationally tractable and limiting the substantial inference and execution costs associated with evaluating multiple debugging methods across four target frameworks. Selection was performed at the logical task level rather than independently for each target framework, so each selected task was evaluated using its corresponding Qiskit, Cirq, PennyLane, and OpenQASM implementation. This preserves a matched cross-framework design in which the same task composition is used for every framework, avoiding confounding framework comparisons with differences in the evaluated tasks. The resulting subset contains 105 local, 25 quantum-semantic, and 22 API tasks, its distribution across algorithm families is reported below.

Algorithm	Local	Semantic	API	Total
ae	6	0	0	6
bmw quark cardinality	9	1	0	10
bmw quark copula	10	2	2	14
bv	10	3	0	13
dj	7	0	0	7
draper qft adder	1	0	1	2
dynamic qft	3	1	1	5
ghz	6	0	2	8
ghz dynamic	3	0	0	3
grover	6	0	0	6
hhl	2	0	2	4
iqpe	5	1	3	9
qaoa	6	0	0	6
qft	0	1	0	1
qftentangled	6	0	2	8
qpeexact	4	1	2	7
qpeinexact	3	2	2	7
qwalk	6	0	1	7
rg qft multiplier	1	0	0	1
seven qubit steane code	4	12	3	19
wstate	7	1	1	9
Total	105	25	22	152

Table 5: Distribution of the 152 mutant tasks by algorithm and source.

Core algorithmic programs. The 36 configured families are Amplitude Estimation (AE), BMW Quark cardinality and copula variants, Bernstein–Vazirani, Deutsch–Jozsa, GHZ, Dynamic GHZ, Graph State, Grover, HHL, iterative QPE, QAOA, QFT, Dynamic QFT, entangled QFT, QNN, exact and inexact QPE, Quantum Walk, W-State, CDKM ripple-carry adder, Draper QFT adder, full adder, half adder, modular adder, multiplier, HRS cumulative multiplier, RG QFT multiplier, VBE ripple-carry adder, the seven-qubit Steane code, Shor's nine-qubit code, Shor's algorithm, VQE with RealAmplitudes, VQE with EfficientSU2, VQE with TwoLocal, and Random Circuit.

Retained program inventory. The frozen run retains all configured families except Random Circuit and Shor's algorithm, yielding 34 retained families and 136 clean framework-specific program instances. The frozen non-API mutation run contains 541 accepted mutation groups and 2,164 framework-specific faulty artifacts, while the primary controlled experiments use the selected 152-task subset.

Several factories are parameterized, so assigning one universal qubit count or circuit size to an algorithm name would be misleading. The final benchmark manifest records these values per instantiated program rather than in a hand-maintained paper table.

C.1 Cross-Framework Construction

Qiskit is used as the canonical source representation for benchmark generation. Cirq and PennyLane targets are generated as native standalone programs and do not import Qiskit at runtime. OpenQASM targets are likewise emitted as standalone OpenQASM programs under the same intended computation and observable contract. The translation process preserves natural source structure where it can be represented faithfully, including function arguments, loops, formulas, and algorithm-local helper functions for the Python frameworks. Compatibility helpers that only reimplement Qiskit behavior are removed when a native target operation exists; when an exact high-level analogue does not exist, the configured Qiskit construct is lowered during generation and emitted as native target operations.

The current translation environment pins Qiskit 2.3.0, Cirq 1.7.0, and PennyLane 0.45.1. Clean targets must parse or import, construct the expected quantum object, execute under the target framework, and satisfy semantic equivalence checks before corresponding non-API mutants are admitted to the benchmark.

C.2 Measurement Contract and Exclusion

Each program carries an explicit observable mapping that defines the logical classical output used by the evaluator. Terminal all-qubit measurements, partial measurement, reordered classical destinations, and supported dynamic measurement behavior are normalized through this mapping rather than inferred from raw count-string order.

A configured program is excluded when it cannot be represented faithfully in a target framework, cannot be executed under the common interface, or lacks a reliable semantic oracle. Unsupported translation is recorded explicitly as its own status rather than merged into generic execution failure. The final retained-program list and all exclusions are generated from the frozen benchmark manifest.

D Bug Taxonomy and Injection Procedure

The mutation system is deterministic. For non-API faults, candidate source sites are enumerated from the canonical Qiskit program and each mutation applies a fixed source transformation. The method records the source program hash, operator name, selected location, variant, and operator-specific metadata. The global experiment seed is used where sampled validation requires randomness; it is not used to randomly choose the source mutation itself. An attempted mutation returns an explicit outcome such as success, not applicable, unsupported, or error.

The selected fault families are informed by empirical studies of quantum software faults (Zhao et al., 2021). The implementation uses two distinct injection paths: canonical-source injection for non-API local and quantum-semantic faults, and a locked target-level catalog for API misuse.

D.1 Local Source and Operation Mutations

The local family contains conventional source mutations that perturb one operation, operand, parameter, or measurement without requiring algorithm-level structural recognition:

GATE_DELETE	Remove one applicable quantum-operation statement.
GATE_INSERT	Insert a deterministic compatible quantum operation at an applicable source location.
GATE_REPLACE	Replace one supported operation by a different gate family according to the fixed replacement rules.
GATE_DUPLICATE	Duplicate one selected quantum-operation statement.
OPERAND_SWAP	Exchange operands of an applicable multi-qubit operation.
QUBIT_REPLACE	Replace one operand by another valid qubit or wire expression.
PARAMETER_MUTATE	For an ordinary numeric gate parameter, add $\pi/4$. For recognized library-circuit construction, mutate a saved configuration parameter by replacing

	an explicit `entanglement` setting with `"linear"` when available, or otherwise increasing the configured `reps`/depth parameter by one.
MEASURE_DELETE	Remove one measurement statement.
MEASURE_INSERT	Insert before an eligible operation an additional measurement using an already existing measurement's quantum and classical targets in the same source scope.
MEASURE_TARGET	Corrupt the qubit-to-classical-destination mapping of one measurement.
XGATE	Replace a programmer-owned gate call by a fixed, distinct wrong gate-family mapping.

The implementation classifies `OPERAND_SWAP`, `PARAMETER_MUTATE`, and `MEASURE_INSERT` as local mutations.

D.2 API Misuse Mutations

API misuse is generated separately because Qiskit, Cirq, PennyLane, and OpenQASM do not share one source-level API representation. The locked catalog contains 51 logical API-misuse groups: 35 `API_WRONG_ARITY` groups, 11 `API_WRONG_NAME` groups, and 5 `API_WRONG_CONDITIONAL` groups. Each logical group contains one framework-specific mutation specification for each of the four target frameworks.

API_WRONG_ARITY	Change the operand or argument count of a selected framework/gate call so that the mutated target fails with an attributable arity, operand, wire, or qubit error.
API_WRONG_NAME	Replace a selected operation/API name with a catalogued nonexistent or invalid name.
API_WRONG_CONDITIONAL	Replace or misuse a framework-specific conditional-control API so that the failure is attributable to the injected conditional API operation.

An API group is accepted only when the corresponding clean target succeeds and the mutated target fails for the intended API-related reason. The validator rejects failures at unrelated stages and checks compatibility between the observed failure category and the catalogued mutation.

D.3 Quantum-Semantic Mutations

The structure-aware family encodes recurring quantum-programming failure mechanisms. These mutations are applied only when their structural precondition is detected.

QORDER	Reverse one explicit operation interface, such as a qubit list supplied to `append` or `compose`, so that qubit ordering is coherently wrong at that logical boundary.
PERM	Corrupt an observable or measurement permutation while leaving the preceding computation unchanged.
RESET	Remove a reset only from a detected measurement–reset–reuse sequence.
XBASIS	Remove the basis conversion immediately preceding a supported non-Z-basis measurement pattern.
VRZ	Remove or corrupt a phase rotation whose effect becomes observable after a later basis transformation and measurement.
QFT	Corrupt a recognizable QFT/inverse-QFT use, for example by toggling supported QFT configuration such as inverse/swap behavior or by inverting the call at a recognized composition site.
UNCOMP	Remove or disrupt a recognizable inverse cleanup or uncomputation region when the required pattern is present.

DECOMP	Swap adjacent source operations that share a qubit and are shown by the bounded validation check to be noncommuting. The stored identifier is retained for compatibility, but the implemented fault is an operation-order corruption rather than replacement of a composite gate's internal decomposition.
SP	Misuse an explicitly recognizable state-preparation operation at its call site when the source pattern is present.
VQEM	Remove an explicitly identifiable mitigation, twirling, folding, or related operation when present.
GPHASE	Insert a phase of $\pi/4$ on the control operand immediately after a supported controlled operation, making the resulting relative phase observable under later interference/control.
MAPINFO	Remove explicit mapping metadata at a supported adapter or transpilation call when such metadata is source-owned.
MATRIX	Conjugate an explicitly supplied custom-unitary matrix when the source directly implements such a matrix.

Not every catalogued semantic operator is applicable to the retained source suite. In the frozen 541-group non-API run, no accepted instances were retained for `UNCOMP`, `SP`, `VQEM`, `MAPINFO`, or `MATRIX`. Their presence above describes supported operator families and does not imply nonzero representation in the retained run.

D.4 Retained Non-API Mutation Counts

The frozen non-API run contains 541 accepted mutation groups, corresponding to 2,164 framework-specific faulty artifacts. The implementation classifies 481 of these groups as local source/operation mutations and 60 as structure-aware quantum-semantic mutations.

Table 6: Accepted mutation groups in the frozen non-API run.

Local mutation	**Groups**	**Quantum-semantic mutation**	**Groups**
Gate deletion	88	Relative-phase (`GPHASE`)	28
Gate replacement	83	Qubit ordering (`QORDER`)	13
Gate insertion	73	Noncommuting order (`DECOMP`)	9
X-gate family error	71	QFT corruption	4
Gate duplication	59	Basis conversion (`XBASIS`)	3
Qubit replacement	53	Reset/reuse	1
Operand swap	24	Measurement permutation	1
Parameter mutation	18	Phase rotation (`VRZ`)	1
Measurement deletion	6		
Measurement insertion	5		
Measurement target	1		
Local subtotal	**481**	**Semantic subtotal**	**60**

D.5 Cross-Framework Injection Alignment

For non-API faults, local and structure-aware mutations are source transformations of the canonical Qiskit program. They are not independently invented after translation. The mutated canonical program is processed through the same translation/emission machinery used to generate the framework targets. For translated Cirq and PennyLane artifacts, provenance metadata links the changed canonical source region to emitted target operations where available. OpenQASM is emitted under the same intended computation and observable contract. A retained group is admitted only when the intended change survives the cross-framework construction and the target remains evaluable under the category's validity requirements.

API faults follow the separate target-level catalog described above. This is necessary because conversion to another representation can remove the original framework-API evidence. Source-framework API calls are therefore not approximated by unrelated textual edits after conversion.

E MUTATION VALIDATION

Mutation validation answers three separate questions: whether the intended edit was produced, whether the resulting artifact satisfies the validity requirements of its fault category, and, for non-API cross-framework faults, whether the intended mutation survives translation into the target representation. These checks are deliberately separated so that a runnable target is not mistaken for a faithful mutant.

E.1 APPLICABILITY AND SOURCE MODIFICATION

Each non-API operator first checks its structural preconditions. An absent precondition returns a "not applicable" result; an unsupported language/representation or unavailable transformation returns a separate "unsupported" result. For a successful mutation, the manifest records the changed source location or operation indices, original and mutated fragments where available, the mutation variant, and operator-specific details. The method verifies that the output differs from the clean canonical source in the intended region.

API groups use catalogued selectors and expected source snippets to establish the intended target-level site. The API validator verifies that the mutated source contains the intended catalogued edit before execution.

E.2 PARSE, CONSTRUCTION, AND EXECUTION REQUIREMENTS

Validity requirements depend on the category. Non-API semantic mutations intended to model executable quantum mistakes must parse, construct, and execute under the required interface. A semantic mutant that fails for an unrelated parse, construction, or execution reason is rejected rather than relabeled as a simpler syntax or API bug.

API misuse intentionally follows a different criterion. For every framework-specific API variant, the corresponding clean target must succeed under the API validator, while the mutated target must fail at the expected API/build/import stage for an attributable reason. A mutated target that succeeds, or fails for an unrelated reason, is rejected.

E.3 SEMANTIC DIFFERENCE

For non-API categories intended to alter observable behavior, the mutant is compared with the clean reference under the validation oracle. Small supported circuits can be checked with exact or dense simulation within the configured validation bound; sampled or measurement-sensitive cases use the same normalized observable mapping as the experiment. A behaviorally equivalent mutant is rejected from categories that require semantic difference.

This check is distinct from final debugging evaluation: it validates the benchmark instance before any LLM sees it.

E.4 TRANSLATION PRESERVATION

For non-API mutations, cross-framework validation checks both the clean translation and preservation of the injected change. Provenance checks compare the changed canonical source region with emitted target operations where supported, while semantic-equivalence checks verify the corresponding clean translations. Observable-mapping checks ensure that target measurements are interpreted using the same logical output ordering as the canonical source.

A target that executes but silently changes the clean algorithm or loses the intended mutation is rejected. Unsupported exact translation is recorded explicitly rather than replaced by an approximate runnable program.

E.5 RECORDED REJECTION REASONS

The manifests distinguish at least: not applicable; unsupported operator; source modification not produced; parse or import failure when execution was required; construction failure; execution fail-

ure; behaviorally equivalent mutant; clean-target translation mismatch; mutation not preserved in translation; invalid observable mapping; evaluator failure; clean API target failure; mutated API target unexpectedly succeeding; and API failure category not matching the intended injected misuse.

F BUGS4Q-ROBUST EVALUATION

We use Bugs4Q-Robust (Ohto et al., 2026) as an external source of real Qiskit defects for RQ3. These programs are not processed by our mutation operators. Accordingly, this experiment does not test whether the controlled mutation categories are realistic; it tests whether the conclusions obtained on the controlled corpus remain valid for independently collected faults that may differ in context, complexity, and composition.

F.1 SELECTION AND EVALUATION

Bugs4Q-Robust contains 37 executable artifacts with buggy programs, corrected programs, and associated test code (Ohto et al., 2026). We retain the 29 cases for which the buggy program fails and the corresponding corrected program passes the official benchmark test in the pinned execution environment. This criterion follows the fail-to-pass notion of artifact existence used in the Bugs4Q-Robust reproducibility study, while the original Bugs4Q benchmark provides tests intended to distinguish buggy and fixed versions (Ohto et al., 2026; Zhao et al., 2023). The resulting evaluation set contains 29 Qiskit tasks:

$$\{1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 16, 17, 18, 19, 20, 24, 25, 26, 27, 28, 29, 31, 33, 37, 38, 39, 42\}.$$

We exclude IDs 9, 30, 35, and 40 because the buggy program fails but the corresponding corrected program also fails in the pinned environment and therefore does not establish a valid repaired state. We exclude IDs 14, 15, 36, and 41 because both the buggy and corrected programs pass, providing no observable failing bug to repair. Thus, the retained set consists only of tasks with a validated failing-to-passing buggy/corrected pair.

For every retained task, the buggy program fails the official test and the corrected program passes that test. A generated repair is counted as successful only if it passes the same official test. Most tasks use one deterministic test execution. For the three tasks with statistical chi-square oracles (IDs 12, 26, and 39), the candidate is evaluated five times and must pass at least four executions.

The model receives the buggy program and a concise behavioral specification derived from the publicly reported defect (Zhao et al., 2023). The corrected program and official test remain hidden from the model. Thus, Bugs4Q success is defined by the benchmark's task-specific behavioral oracle rather than by the KL-divergence criterion used for the controlled mutation corpus.

Compatibility preprocessing is restricted to visualization-only behavior. Plot generation is disabled for ID 2, and unrelated circuit-drawing or visualization operations are removed from IDs 6, 8, 19, 24, 25, 31, 39, and 42. These changes are applied symmetrically to the buggy and corrected programs and do not alter the fault-relevant computation.

Validation uses Python 3.13, Qiskit 2.3.0, Qiskit Aer 0.17.2, Qiskit Machine Learning 0.9.0, Qiskit Experiments 0.13.0, and Qiskit IBM Runtime 0.45.0.

F.2 RESULTS

Method	Passed
QBugLM	17/29
LDB	18/29
Self-Debugging	12/29
NL-Debugging	7/29
RING	9/29
QuRepair (ours)	**21/29**

QuRepair repaired 21 of the 29 external Bugs4Q tasks (72.4%). LDB repaired 18 tasks (62.1%). Thus, QuRepair achieved three additional successful repairs over the strongest baseline on the ex-

ternal evaluation. The result indicates that QuRepair remains effective on independently collected Qiskit defects evaluated using task-specific official tests.

The external set includes defects whose semantic structure is not captured by a single local mutation. For example, inverse-QFT failures in phase estimation can involve missing swaps, incorrect ordering of Hadamard and controlled-phase operations, and incorrect phase angles. Such cases are valuable because they test whether QuRepair can reason about interacting quantum-program errors rather than only isolated mutation patterns.

G FRAMEWORK-AWARE LINTERS

QuRepair performs deterministic, source-only static analysis before LLM repair. The active linter receives only the candidate source code, the framework identifier, and the selected ruleset; it does not receive the clean program, mutation type, expected patch, or execution oracle. A diagnostic is therefore repair evidence rather than a repair prescription.

The active linters were informed by the rule concepts and quantum-specific diagnostic goals of LintQ (Paltenghi & Pradel, 2024), an earlier static analyzer built on CodeQL (Avgustinov et al., 2016), but were reimplemented for the framework versions used in this evaluation. Rather than executing the earlier CodeQL backends, the active method uses framework-aware source extraction followed by a shared conservative semantic analyzer.

G.1 QISKIT

For Qiskit, the extractor recognizes supported circuit method invocations, selected `append` calls, and in-place circuit composition, and converts them into an ordered representation of operations and qubit use. The shared analyzer reports structural evidence such as repeated or cancelling operations, repeated measurement, operations following measurement, anomalous parameter or control–target patterns, and missing or extra corresponding operations.

Qiskit additionally performs definite source-level checks for insufficient or out-of-range classical measurement targets and recognized classical conditions that read a classical register before a definite measurement write. These checks are intentionally conservative: the linter does not attempt to validate every Qiskit API call, infer arbitrary dynamic Python behavior, or prove that an unrecognized operation is safe.

G.2 CIRQ

The Cirq backend uses the same source-only architecture, with Cirq-specific extraction for supported gate constructions and operations inserted into circuits, including selected `Circuit` construction, append, composition, and controlled-operation patterns. The resulting operation sequence is analyzed using the shared structural rules.

The active implementation does not claim a complete model of all Cirq concepts, including arbitrary `Qid` identities, moments, measurement keys, `FrozenCircuit`s, nested `CircuitOperation`s, or all classically-controlled operations. Consequently, unsupported or highly dynamic constructs may produce no linter evidence; absence of a diagnostic is not a proof of correctness.

G.3 PENNYLANE

For PennyLane, the linter statically recognizes supported `qml` operation calls, wire arguments, and selected controlled-operation wrappers. It then reports the same conservative structural evidence used for the other Python frameworks, including repeated or cancelling operations, repeated measurement, operations following measurement, and anomalous operation patterns.

The analysis does not attempt to fully reconstruct QNodes, tapes, terminal return semantics, or arbitrary dynamic Python behavior. Unsupported constructs may therefore yield no diagnostic; an empty diagnostic set is not evidence that the program is correct.

G.4 OpenQASM

OpenQASM analysis combines the shared source-level semantic analysis with the native `LintOpenQASM` parser and AST analyzer. The default native ruleset reports parse and version failures, include failures, duplicate declarations, undefined symbols, invalid indices, invalid gate arity, duplicate qubit operands, uninitialized classical reads, and repeated measurement.

An expanded ruleset additionally reports operations after measurement, constant classical bits caused by reset–measurement patterns, and discarded measurement results. OpenQASM source does not contain Python framework calls, so framework-API checks are intentionally outside the scope of this backend.

G.5 Diagnostic Normalization

All backends return a shared report containing the framework, analysis status, diagnostic count, runtime, and normalized diagnostics. Each diagnostic includes a rule identifier, severity, concise message and explanation, source location and region when available, affected operation and qubits when available, and structured evidence.

The repair prompt receives a bounded representation of this normalized report. If analysis produces no findings, the method records an empty diagnostic set rather than inventing an explanation. Parse failures are preserved as explicit diagnostics, while linter infrastructure failures halt the attempt instead of being silently interpreted as an empty result.

H Evaluation Details

The evaluator is shared by all debugging methods and is the sole authority for final repair success. Model-facing method stages provide repair assistance but do not determine semantic correctness.

H.1 Evaluation Procedure and Settings

For each candidate, the evaluator parses or imports the generated source, verifies the task-specific execution interface, and constructs the quantum computation. Python tasks must expose the required entrypoint and produce the required circuit or QNode type; OpenQASM tasks must be valid complete programs. Syntax or construction failures, missing entrypoints, invalid return types, zero-qubit outputs, missing required measurement output, timeouts, and simulator errors cause evaluation failure.

The evaluator supports Qiskit, Cirq, PennyLane, and OpenQASM 3 through framework-specific execution adapters. Qiskit circuits are executed with Qiskit Aer, Cirq circuits with `cirq.Simulator`, PennyLane tasks through their returned QNodes, and OpenQASM 3 programs through Qiskit's OpenQASM importer and Qiskit Aer. For QAOA tasks, whose circuits are intentionally unmeasured, the evaluator instruments a copy of the circuit to measure all qubits. Execution success alone does not imply semantic success.

The primary controlled experiments use 16,384 shots and seed 12,345 for every candidate. The same settings are supplied to all debugging methods within a comparison. The experiment environment uses Qiskit 2.3.0, Qiskit Aer 0.17.2, Cirq 1.7.0, PennyLane 0.45.1, and `openqasm3` 1.0.1. Provider-side model randomness is controlled by the method-specific generation settings described in Appendix I.

H.2 Semantic Comparison and Reference Isolation

Framework-native outputs are converted to canonical binary-string count distributions. Framework-specific output conventions are handled by the adapters, and counts are converted to relative frequencies without additive pseudo-count smoothing. The evaluator rejects outputs whose bit width differs from that of the reference.

Let p be the normalized reference distribution and q the normalized candidate distribution over the resulting output strings. The primary discrepancy measure is

$$D_{\mathrm{KL}}(p \parallel q) = \sum_x p(x) \log \frac{p(x)}{q(x)}.$$

After normalization, zero probabilities used in the numerical computation are clipped with $\epsilon = 10^{-12}$: terms are evaluated using $\max(p(x), \epsilon)$ and $\max(q(x), \epsilon)$. A candidate is semantically successful only if execution succeeds, its output width matches the reference, and

$$D_{\mathrm{KL}}(P_{\mathrm{reference}} \parallel P_{\mathrm{candidate}}) < 0.05.$$

The inequality is strict; a value equal to 0.05 is a failure.

Reference behavior is precomputed for each prepared task, so every method is judged against the same target. The clean reference program, exact reference and candidate distributions, and numerical KL-divergence value are not provided to the debugging model. A method may receive only sanitized derived feedback, such as a pass/fail verdict or coarse failure class.

I Model Configuration

All provider calls are routed through a shared model interface that records the model identifier, decoding parameters, token usage, and generation metadata.

The controlled method comparison uses GPT-4o as the underlying LLM for Self-Debugging, LDB, RING, NL-Debugging, QBugLM, and QuRepair. Holding the model fixed ensures that the primary comparison isolates differences in debugging procedures and available evidence rather than differences in base-model capability. QuRepair additionally issues typed Jev Decisions API calls for its diagnosis (Appendix A), and all code is generated by the repair LLM.

For RQ3, QuRepair is additionally evaluated using Qwen3-8B (Qwen Team, 2025), Claude Sonnet 4 (Anthropic, 2025), and Gemini 3.8 Flash (Google DeepMind, 2026) to study robustness across different model families. For QuRepair, the repair model is varied across experiments, while Jev remains fixed across all model configurations. Specifically, every repair model uses the same Jev model (`typesafe/jev-1.13`) and the same Jev endpoint and configuration. Direct repair does not use Jev. The direct-repair baseline uses a single model generation per task with temperature 0.0. The model receives the behavioral specification and the buggy program and is instructed to return one complete repaired program. The resulting candidate is then evaluated once using the private task oracle. This baseline is non-iterative: it receives no execution or evaluator feedback and performs no feedback-guided refinement. Retries caused by transient provider or network failures are request-level retries only and do not constitute additional repair attempts.

Most methods use deterministic decoding with temperature 0. RING is the exception and follows its multi-candidate generation protocol with temperature 0.7 and $\texttt{top_p} = 1.0$, generating ten candidates and evaluating the three highest-ranked candidates. All remaining generation parameters are fixed globally and recorded in the experiment metadata.

J Complete Experimental Results

We report method-runtime measurements on the matched 152-task benchmark. Each method was evaluated on 152 tasks in each of four frameworks, yielding 608 task–framework jobs per method. All methods used GPT-4o as the repair model; our pipeline additionally invoked Jev through the Decisions API. Recorded token totals for our pipeline include both services unless explicitly labeled repair-model-only.

Analysis. QuRepair has the lowest overall mean reported pipeline time per job (35.9 s), narrowly below Self-Debugging (37.4 s). It has the lowest per-job mean for Qiskit, PennyLane, and OpenQASM, while Self-Debugging is lower for Cirq (38.6 s versus 60.7 s).

Qurepair also has the lowest amortized reported pipeline time per successful repair in every framework, with an overall value of 57.1 s, compared with 177.5 s for Self-Debugging, the next-lowest method.

Table 7: Runtime efficiency on the matched 152-task benchmark. Panel A reports mean pipeline time per job. Panel B reports amortized pipeline time per successful repair, computed as total recorded pipeline time divided by the number of successful repairs. All values are in seconds; lower is better.

Method	Qiskit	Cirq	PennyLane	OpenQASM	Overall
Panel A: Mean pipeline time per job					
Self-Debugging	24.9	**38.6**	36.8	49.2	37.4
LDB	413.8	462.2	436.0	1,275.8	647.0
RING	105.3	143.4	117.0	92.5	114.5
NL-Debugging	113.0	160.5	231.2	95.5	150.0
QBugLM	119.3	279.0	218.1	242.6	214.8
QuRepair (ours)	**17.8**	60.7	**28.5**	**36.5**	**35.9**
Panel B: Amortized pipeline time per repair					
Self-Debugging	94.5	162.8	233.4	267.3	177.5
LDB	776.6	1,133.1	1,616.5	6,687.0	1,846.8
RING	136.8	294.6	291.6	540.6	250.5
NL-Debugging	1,227.1	677.6	2,067.1	1,319.2	1,169.6
QBugLM	187.0	706.8	460.5	647.0	456.6
QuRepair (ours)	**23.0**	**86.2**	**43.4**	**97.2**	**57.1**

The amortized measure assigns the time spent on unsuccessful attempts to the successful outcomes. It therefore represents aggregate recorded pipeline time per successful repair rather than the duration of an individual successful job.

Table 8: Recorded model usage on the matched 152-task benchmark (608 jobs per method). Token totals include input and output tokens from all attempted jobs. Per-success values divide those totals by the credited successes; RING uses credited Top@3 successes.

Method	Successes	Calls	Total tokens	Tokens/success	Calls/success
Self-Debugging	128	2,344	10,073,211	78,697	18.31
LDB	213	2,747	8,365,438	39,274	12.90
RING	278	608	1,510,405	5,433	2.19
NL-Debugging	78	10,492	13,782,283	176,696	134.51
QBugLM	286	12,267	71,363,522	249,523	42.89
QuRepair (ours)	**382**	**3,745**	**26,289,610**	**68,821**	**9.80**
Four non-RING baselines pooled	705	27,850	103,584,454	146,928	39.50

Model-token accounting. The pooled row combines Self-Debugging, LDB, NL-Debugging, and QBugLM: $103{,}584{,}454/705 = 146{,}928$ tokens per credited repair. For QuRepair, the repair model used 8,261,924 tokens in 1,650 recorded calls, while Jev Decisions API calls used 18,027,686 tokens in 2,095 recorded calls. Thus, including both services gives $26{,}289{,}610/382 = 68{,}821$ recorded tokens per credited repair, or 53.2% fewer than the pooled non-RING baselines. Repair-model usage alone is $8{,}261{,}924/382 = 21{,}628$ tokens per credited repair. These are amortized quantities: tokens consumed by unsuccessful attempts remain in the numerator.