Towards Quantum Program Bug Datasets and Benchmarking

Tingting Li*
Zhejiang University
Hangzhou, China
litt2020@zju.edu.cn

Jiongchi Yu*
Singapore Management University
Singapore, Singapore
jcyu.2022@phdcs.smu.edu.sg

Zhikang Fan
Zhejiang University
Hangzhou, China
15030857023@163.com

Junqing Wang
Zhejiang University
Hangzhou, China
wangjunqing388@gmail.com

Yao Zhai
Zhejiang University
Hangzhou, China
zhaiyao@zju.edu.cn

Shenglong Zhang
Zhejiang University
Hangzhou, China
yuluxin911@gmail.com

Yu Peng
Zhejiang University
Hangzhou, China
pengyuy@zju.edu.cn

Yuhan Chen
Zhejiang University
Hangzhou, China
yuhanchen26@163.com

Junyu Chen
Zhejiang University
Hangzhou, China
junyu.chen.quantum@zju.edu.cn

Fanqi Kong
Zhejiang University
Hangzhou, China
fanqikong@st.gxu.edu.cn

Zhaoxuan Li
Institute of Information Engineering
CAS, Beijing, China
lizhaoxuan@iie.ac.cn

Xiaofei Yue
Beijing Institute of Technology
Beijing, China
xfyue@bit.edu.cn

Ziming Zhao†
Zhejiang University
Hangzhou, China
zhaoziming@zju.edu.cn

Jianwei Yin†
Zhejiang University
Hangzhou, China
zjuyjw@cs.zju.edu.cn

Abstract

The growing maturity of quantum computing has led to increasingly complex software ecosystems spanning frameworks, compilers, simulators, and application-level programs. Despite this rapid growth, the community still lacks a realistic benchmark for evaluating automated quantum bug fixing under quantum-specific correctness requirements, such as stochastic outputs, tolerance-based equivalence, and environment-sensitive execution. To bridge this gap, we present *QBench*, an end-to-end benchmark for evaluating automated repair of real-world quantum software bugs. QBench consists of two complementary tracks: *RepoTrack*, which targets defects in quantum software infrastructure and repositories, and *ScriptTrack*, which targets bugs in user-facing quantum programs and experimental workflows. Each benchmark instance provides a reproducible bug-fixing task together with the execution environment, correctness oracle, and evaluation protocol required to validate candidate repairs, enabling standardized assessment of repair systems under realistic development settings. QBench supports deterministic, tolerance-aware, and probabilistic correctness oracles, enabling reliable evaluation under realistic quantum execution semantics. Using QBench, we conduct a large-scale empirical study of automated quantum software repair and analyze the challenges faced by representative repair approaches. Our results show that stochastic correctness criteria, environment fragility, and cross-layer defects remain major obstacles to successful repair, highlighting the unique difficulties of maintaining and repairing quantum software.

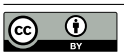
CCS Concepts

• **Software and its engineering → Software maintenance tools**; **Software verification and validation**.

Keywords

Quantum Software Engineering, Automated Program Repair, Benchmark, Large Language Models

*Equal contribution.
†Corresponding authors.



KDD '26, Jeju Island, Republic of Korea

ACM ISBN 979-8-4007-2259-2/2026/08
https://doi.org/10.1145/3770855.3817560

ACM Reference Format:
Tingting Li, Jiongchi Yu, Zhikang Fan, Junqing Wang, Yao Zhai, Shenglong Zhang, Yu Peng, Yuhan Chen, Junyu Chen, Fanqi Kong, Zhaoxuan Li, Xiaofei Yue, Ziming Zhao, Jianwei Yin. 2026. Towards Quantum Program Bug Datasets and Benchmarking. In *Proceedings of the 32nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2 (KDD '26), August 09–13, 2026, Jeju Island, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3770855.3817560

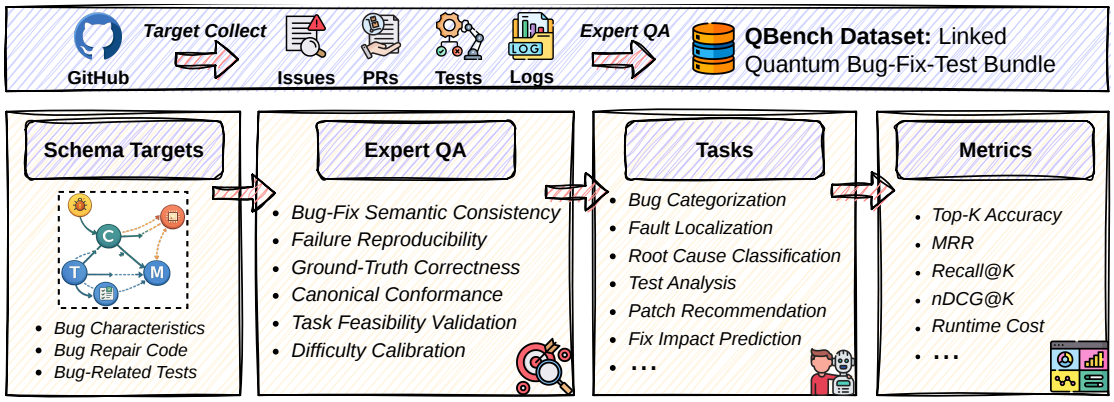


Figure 1: The overall workflow of *QBench*.

1 Introduction

The growing maturity of quantum computing [14, 15] has led to increasingly complex software ecosystems spanning programming frameworks [45], compilation and transpilation toolchains [41], simulators, differentiable programming systems [18], and application-level programs [3, 25]. As quantum software becomes a critical component of the computing stack, ensuring its reliability is increasingly important. Software defects in quantum systems can silently distort scientific conclusions, invalidate benchmarking results, and propagate errors across multi-stage compilation and execution pipelines [27].

Meanwhile, the software engineering community has increasingly adopted end-to-end benchmarks for evaluating automated bug fixing, from traditional program repair techniques to recent LLM-based repair agents [9, 10, 36]. These benchmarks formulate realistic bug-fixing tasks together with executable evaluation protocols, enabling reproducible assessment of repair effectiveness and accelerating progress in automated software maintenance [9, 17]. However, quantum software remains largely absent from this evaluation landscape. Existing research on quantum software reliability has primarily focused on testing and debugging [7, 28, 34, 43], while resources for evaluating automated quantum bug repair remain limited, fragmented, and difficult to compare reproducibly [2]. Therefore, there is currently no standardized benchmark that enables systematic evaluation of automated repair techniques on real-world quantum software bugs.

More importantly, evaluating repair for quantum software cannot be reduced to directly applying existing benchmark designs. Many assumptions underlying classical repair evaluation do not hold in quantum settings [16, 19, 44]. (i) Correctness is often probabilistic rather than deterministic, since many quantum programs produce measurement distributions rather than fixed outputs [7, 26, 34]. (ii) Correctness frequently relies on tolerance-based equivalence, such as expectation values, numerical stability, and floating-point computations [40]. (iii) Quantum software stacks are highly sensitive to dependency versions, simulator backends, and execution environments, making reproducibility substantially more challenging than in conventional software systems [12, 25]. (iv) Many defects arise from interactions across multiple layers of the software stack, including compilers, simulators, and application-level programs [27, 34]. These characteristics require evaluation protocols that go beyond the deterministic assumptions adopted by most existing repair benchmarks.

To address these challenges, we present *QBench* (as shown in Figure 1), a benchmark for evaluating automated repair of real-world quantum software bugs. *QBench* comprises two complementary tracks. *RepoTrack* focuses on defects in quantum software infrastructure, including compilers, simulators, and supporting libraries, while *ScriptTrack* focuses on bugs in user-facing quantum programs and experimental workflows. Each benchmark instance is formulated as an executable bug-fixing task with a reproducible execution environment, a correctness oracle, and standardized validation tests, enabling end-to-end assessment of automated repair systems.

A key aspect of *QBench* is its support for both deterministic and quantum-specific probabilistic correctness oracles. Beyond conventional test-based validation, *QBench* supports tolerance-aware and probabilistic correctness criteria through repeated execution and statistically robust evaluation procedures. This design enables realistic evaluation under quantum execution semantics while maintaining reproducibility across benchmark instances.

We use *QBench* to conduct a comprehensive study of automated quantum software repair. We evaluate representative repair approaches, including modern LLM-based repair agents and conventional baselines, and analyze the factors that most strongly affect repair performance. Our results show that stochastic correctness criteria, environment fragility, and cross-layer defects remain major obstacles for existing repair systems, highlighting the unique challenges of automated quantum software repair and motivating future research on quantum-aware repair techniques.

We summarize the main contributions of this paper as follows.

- We present *QBench*, an end-to-end benchmark for automated quantum software repair. *QBench* contains 520 real-world bug-fixing tasks collected from 129 quantum software repositories, covering both repository-level infrastructure defects (*RepoTrack*) and script-level application bugs (*ScriptTrack*). We publicly release the benchmark and evaluation framework to support reproducible research[1].

[1] https://github.com/Secbrain/QBench

- We develop a quantum-aware evaluation framework that supports deterministic, tolerance-aware, and probabilistic correctness oracles, enabling reliable assessment of bug fixes under realistic quantum execution semantics.
- We conduct a comprehensive empirical study of automated quantum software repair, benchmarking representative repair approaches and analyzing the impact of oracle semantics, environment sensitivity, and bug characteristics on repair performance.

2 Related Work

2.1 Program Repair Benchmarks

Benchmark datasets have played a central role in the development and evaluation of automated program repair techniques. Widely used benchmarks such as Defects4J [10], Bugs.jar [32], Bears [24], and QuixBugs [20] have enabled systematic evaluation of search-based and learning-based repair approaches [13, 21, 22]. More recent benchmarks, including SWE-bench [9], further increase realism by evaluating repair in repository-level settings with build systems, dependencies, and executable tests. However, existing repair benchmarks are primarily designed for classical software and typically assume deterministic correctness criteria. In contrast, *QBench* extends benchmarking to quantum software, where correctness may be probabilistic, tolerance-based, and sensitive to execution environments.

2.2 LLM-Based Automated Repair

Recent advances in large language models (LLMs) have significantly improved automated program repair through code generation, iterative refinement, and tool-augmented reasoning [38, 39, 42]. These methods have demonstrated promising performance on both traditional repair benchmarks and repository-level tasks [8, 9, 31]. Rather than introducing a new repair approach, *QBench* provides a benchmark specifically designed to evaluate repair systems in quantum software settings and to characterize the challenges unique to quantum bug fixing.

2.3 Quantum Software Testing and Debugging

A growing body of research has investigated techniques for improving the reliability of quantum software, including property-based testing [6, 26, 29], differential testing [34], fuzzing, metamorphic testing, and automated debugging [7, 28, 30, 35]. These studies primarily focus on detecting, localizing, and reproducing defects. By contrast, *QBench* targets the downstream repair problem, providing standardized tasks and evaluation protocols for assessing whether automated methods can successfully generate correct bug fixes.

2.4 Quantum Bug Datasets

Several studies have analyzed bug characteristics, defect taxonomies, and reliability challenges in quantum software ecosystems [2, 3, 27, 44]. These efforts provide valuable insights into the nature of quantum software defects and have helped establish the foundations of Quantum Software Engineering. However, they are not designed as executable repair benchmarks and therefore do not support end-to-end evaluation of automated repair systems. *QBench* builds upon these empirical findings by transforming real-world quantum bugs into reproducible repair tasks with standardized evaluation protocols, enabling systematic comparison of repair approaches.

3 Dataset Construction

QBench is constructed from real-world quantum software bugs collected from public quantum software repositories. Our objective is not only to curate representative bug-fixing tasks, but also to ensure that each task can be evaluated reproducibly under the correctness semantics of quantum software. To this end, we adopt a multi-stage construction pipeline that transforms bug-fixing events into executable repair tasks while enforcing environment reproducibility, oracle reliability, and contamination control. Although RepoTrack and ScriptTrack target different classes of defects, both tracks follow the same construction and evaluation workflow. Figure 2 provides an overview of the pipeline.

3.1 Mining Real-World Quantum Bugs

We begin by curating a pool of open-source quantum software repositories spanning the major layers of the ecosystem, including programming frameworks, compilation and transpilation toolchains, simulators, differentiable programming systems, and application-level projects. To ensure both representativeness and executability, repositories are required to satisfy three criteria: (i) public availability with accessible version-control history, (ii) evidence of active software engineering practices such as issue tracking, pull requests, or testing infrastructure, and (iii) runnable artifacts that enable bug reproduction under controlled environments.

From this repository pool, we identify bug-fixing events using multiple complementary signals, including fix-related commit messages and pull requests, linked issue reports, CI outcomes, and documented runtime failures. Each candidate is normalized into a bug-fix pair $\langle c_{\text{bug}}, c_{\text{fix}} \rangle$, where c_{bug} denotes the buggy revision and c_{fix} denotes the corresponding fixing revision. For multi-commit fixes, we preserve the minimal bug-fixing change while maintaining links to the original development history for traceability.

3.2 Repair Task Construction

Each bug-fix pair is transformed into an executable repair task rather than being released as a raw patch. A *QBench* instance packages all artifacts required for bug reproduction and repair validation, including the buggy revision, the corresponding fixing patch, a correctness oracle, a reproducible execution environment, and optional contextual information such as issue reports, stack traces, execution logs, and reproduction instructions. Formally, a repair instance is represented as $I = \langle \mathcal{R}, c_{\text{bug}}, C, O, \mathcal{E}, p^{\star} \rangle$, where $\mathcal{R}$ denotes the repository, c_{bug} the buggy revision, C the contextual information, O the correctness oracle, $\mathcal{E}$ the execution environment, and $p^{\star}$ the developer bug-fixing patch.

To support reproducible evaluation, every instance follows a standardized packaging specification. The package records provenance information, execution commands, oracle configurations, environment dependencies, and evaluation constraints, allowing different repair systems to be evaluated under identical conditions. Listing 1 shows the canonical organization of *QBench*, including benchmark releases, repair instances, and evaluation infrastructure.

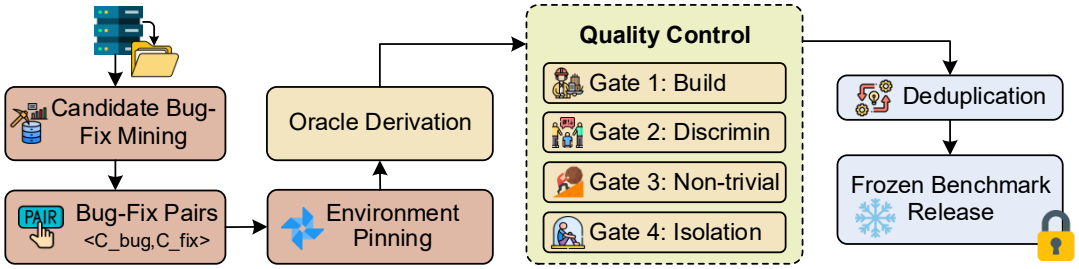


Figure 2: Construction pipeline of *QBench*.

Table 1: Statistics of *QBench*.

Statistic	RepoTrack	ScriptTrack
# Instances	264	256
# Repositories Covered	55	74
Median Files Changed	3	1
Median Lines Changed	34	12
Mean Lines Changed	58.7	19.4
Cross-Module Patches	0.47	0.18
Deterministic-Oracle Fraction	0.56	0.61
Probabilistic/Tolerance Fraction	0.44	0.39
Median Oracle Repetitions	5	3
Estimated Flakiness Rate	0.21	0.11

Each repair instance is further described through a machine-readable manifest that specifies provenance information, execution commands, oracle parameters, and environment configurations. Listing 2 illustrates an example manifest used by *QBench*.

3.3 Oracle and Environment

A key challenge in benchmarking quantum software repair is that correctness is often influenced by both execution environments and quantum-specific program semantics. Small changes in simulator versions, compiler toolchains, numerical libraries, or dependency stacks can alter program behavior and compromise reproducibility. To mitigate these effects, each *QBench* instance includes a fully specified execution environment with pinned dependencies and reproducible environment specifications.

Beyond environment control, *QBench* supports heterogeneous correctness semantics through the following evaluation oracles.

- **Deterministic oracles**, including unit tests, exception checks, API contracts, and exact-output validation.
- **Tolerance-aware oracles**, which assess approximate numerical equivalence using predefined acceptance thresholds.
- **Probabilistic oracles**, which evaluate correctness based on repeated executions and statistical properties of quantum measurement outcomes.

To ensure reliable evaluation, probabilistic and tolerance-aware oracles undergo a stabilization process involving repeated execution, robust acceptance criteria, and flakiness auditing. Instances that cannot reliably distinguish buggy and fixed behavior are excluded from the benchmark.

3.4 Quality Assurance

Every candidate instance must satisfy a series of quality checks before inclusion in *QBench*.

Build Validity. The buggy revision must build, install, and execute successfully under the specified environment.

Oracle Validity. The correctness oracle must reliably distinguish buggy and fixed behavior: $Eval(c_{\text{bug}}) = fail \wedge Eval(c_{\text{bug}} + p^{\star}) = pass$. Instances that do not satisfy this criterion are excluded.

Task Validity. We exclude documentation-only changes, pure refactorings, and other modifications that do not correspond to meaningful bug-fixing behavior.

Execution Isolation. Instances are validated to prevent state leakage across runs. Sources of nondeterminism must be explicitly captured and stabilized by the oracle specification.

We also ensure that repair systems are not allowed to modify benchmark infrastructure, evaluation harnesses, or oracle artifacts unless explicitly permitted by a separate evaluation setting.

3.5 Deduplication and Contamination

To prevent inflated evaluation results, we perform deduplication at multiple levels, including provenance-level duplicates, near-identical patches, and semantically equivalent repair instances. We further provide contamination controls through temporal splits, repository-disjoint splits, and provenance metadata that record commit identifiers, pull requests, and fix dates.

Listing 1: Canonical organization of *QBench*.

```
qbench/
  releases/
    v1.0.0/ # frozen benchmark release
    instances/
    RepoTrack/
      QB-000123/
        manifest.yaml
        context/
        oracle/
        env/
    ScriptTrack/
      ...
    framework/
      oracles/
      runners/
    outputs/
```

Each release of *QBench* is associated with immutable repository revisions, oracles and environment specifications, ensuring

reproducibility and comparability across studies. A continuous validation pipeline periodically re-executes benchmark instances to detect environment drift and reproducibility regressions.

Table 1 summarizes the evaluated release used in this paper. The release contains 520 repair tasks collected from 129 quantum software repositories, including 264 RepoTrack instances and 256 ScriptTrack instances. RepoTrack generally contains more cross-module fixes, while ScriptTrack primarily consists of localized application-level defects. Notably, both tracks contain a substantial fraction of probabilistic and tolerance-aware oracles, highlighting the importance of quantum-aware evaluation protocols.

Listing 2: Example *manifest.yaml* for a repair instance.

```yaml
id: QB-000123
track: RepoTrack
repo:
  name: qiskit
  url: https://github.com/Qiskit/qiskit
revision:
  bug: <sha_bug>
  fix: <sha_fix>
provenance:
  pr_url: <optional>
  issue_url: <optional>
  fix_date: YYYY-MM-DD
patch:
  path: patch.diff
  files_touched: 3
  lines_changed: 31
oracle:
  type: probabilistic   # {deterministic,
      tolerance, probabilistic}
  command: "bash reproduce/run.sh"
  params:
    repeats: 5          # K
    shots: 4096         # S (if applicable)
    metric: tvd         # {tvd, hellinger, jsd}
    delta: 0.05
    quorum: null        # or a voting threshold
        rho
env:
  kind: docker
  dockerfile: env/Dockerfile
  python: "3.10"
  timeout_sec: 1200
harness:
  interface: [setup, apply_patch, evaluate]
  no_test_modification: true
```

4 *QBench* Benchmark Tasks

4.1 Task Formulation

QBench evaluates end-to-end automated repair of real-world quantum software bugs under realistic development settings [9]. Similar to modern software repair benchmarks [5, 10, 37], each benchmark instance represents an executable bug-fixing task that can be evaluated reproducibly.

A repair instance in *QBench* is represented as

$$\mathcal{I} = \langle \mathcal{R}, c_{\text{bug}}, C, O, \mathcal{E}, p^{\star} \rangle \tag{1}$$

where $\mathcal{R}$ denotes the repository or project, c_{bug} the buggy revision, C the available problem context, O the correctness oracle, $\mathcal{E}$ the execution environment, and $p^{\star}$ the developer-authored bug-fixing patch [5, 10].

The problem context C may include issue reports, pull-request descriptions, execution logs, stack traces, reproduction instructions, and related discussions when available [9]. Given an instance $\mathcal{I}$, an automated repair system generates a candidate patch $\hat{p}$, which is applied to the buggy revision and evaluated through the harness.

A repair attempt is considered successful if the resulting program satisfies the associated correctness oracle:

$$Eval(c_{\text{bug}} + \hat{p}) = pass \tag{2}$$

The evaluation function *Eval* executes the repair under the specified environment and determines whether the generated patch resolves the bug according to the oracle specification [9, 10].

4.2 Benchmark Tracks

QBench consists of two complementary tracks that capture distinct quantum software repair scenarios. Figure 3 illustrates the scope and characteristics of each track.

Repository-Level Track (RepoTrack). RepoTrack focuses on defects in quantum software infrastructure, including libraries, SDKs, compilers, transpilers, simulators, and other supporting components. These instances frequently require repository-level reasoning, cross-module modifications, and interaction with complex build and testing workflows [5, 9].

Script-Level Track (ScriptTrack). ScriptTrack focuses on user-facing quantum programs and experimental workflows, including algorithm implementations, benchmark scripts, variational training pipelines, and application-level analyses. These instances are typically more localized and emphasize domain-specific correctness, API usage, and experimental assumptions [43].

4.3 Oracle Semantics

Correctness in quantum software often extends beyond deterministic pass/fail behavior. To accommodate the diverse semantics of quantum programs, *QBench* supports multiple classes of correctness criteria [1].

Depending on the benchmark instance, correctness may be defined through deterministic tests and assertions [5, 36], numerical tolerance checks [43], or probabilistic validation of quantum measurement outcomes [28, 34]. These oracle semantics capture the heterogeneous correctness requirements encountered in real-world quantum software and form the basis of benchmark evaluation.

5 Evaluation

This section describes the evaluation protocol, compared systems, model coverage, and fairness controls used to benchmark automated repair on *QBench*. Following prior large-scale repair benchmarks [9, 38], our goal is not only to report aggregate success rates, but to enable a *diagnostic* and *reproducible* comparison across repair paradigms under realistic constraints. To disentangle genuine repair from oracle manipulation, we evaluate controlled variants including allowing test edits, removing issue context, and limiting oracle feedback [9, 33, 39].

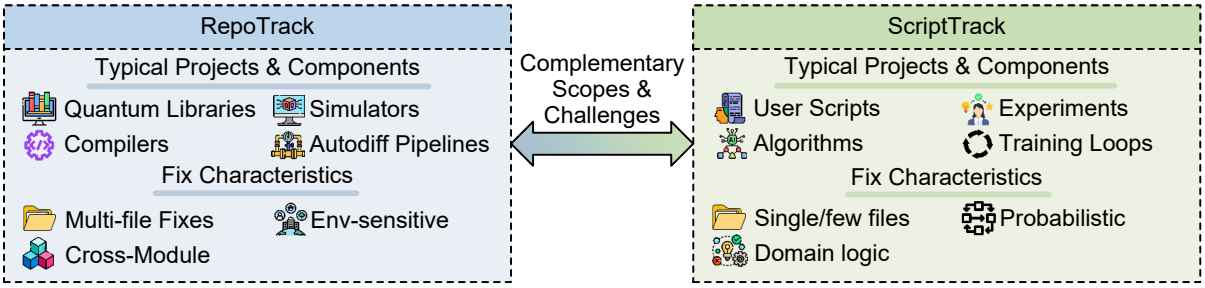


Figure 3: Intuitive explanation of RepoTrack and ScriptTrack.

5.1 Evaluation Protocol

Each benchmark instance is defined as $\mathcal{I} = \langle \mathcal{R}, c_{\text{bug}}, C, O, \mathcal{E}, p^\star \rangle$, where $\mathcal{R}$ is the repository, c_{bug} the buggy snapshot, C the task context, O the correctness oracle, $\mathcal{E}$ the pinned execution environment, and $p^\star$ the developer fix.

Given an instance $\mathcal{I}$ and a fixed resource budget, a repair system generates a candidate patch $\hat{p}$. The benchmark harness evaluates the patch by executing the associated oracle under the specified environment:

$$Eval(\mathcal{R}, c_{\text{bug}}, \hat{p}, O, \mathcal{E}) \in \{pass, fail\} \tag{3}$$

Results are reported separately for RepoTrack and ScriptTrack, as well as in aggregate, following common practice in software repair benchmarking [5, 9, 10].

Repetitions and Randomness. For deterministic instances, candidate patches are evaluated once unless environment-induced nondeterminism is observed [23]. For probabilistic and tolerance-aware instances, evaluation follows the instance-specific repetition and sampling budgets (K, S), and robust success criteria are applied to mitigate stochastic variability [4].

Timeouts and Resource Limits. We enforce per-instance limits on wall-clock time, CPU and memory consumption, and GPU resources when applicable. All evaluations are conducted within containerized environments to ensure comparability across systems [9].

No-Test-Modification Setting. Unless otherwise specified, repair systems may modify only project source code and configuration files. Modifications to benchmark infrastructure, harnesses, or oracle artifacts are disallowed to prevent overfitting and oracle manipulation [9, 33].

5.2 Compared Systems

We evaluate conventional program repair baselines and LLM-based repair agents. The former serve as sanity checks, while the latter constitute the primary focus of our study.

5.2.1 Conventional Baselines. To contextualize LLM-based performance, we include representative conventional baselines. (i) *Edit localization with heuristic templates* [11], which applies common fix patterns (*e.g.*, missing imports or argument mismatches) guided by failure localization. (ii) *Search-based automated program repair* [36], which explores candidate patches via mutation operators under oracle feedback with bounded search. These baselines are not intended to represent state-of-the-art performance, but to identify instances solvable by shallow or non-semantic fixes.

5.2.2 LLM-Based Repair Agents. Our primary evaluation focuses on LLM-based repair agents. Given repository context C and code at $\mathcal{R}@c_{\text{bug}}$, an agent proposes a candidate patch and may iteratively refine it using oracle feedback [9, 38, 39].

Model Coverage. We evaluate a broad and representative set of contemporary LLMs spanning proprietary and open-weight families, general-purpose and code-specialized variants, and multiple model scales. Models are grouped into four categories: general frontier models, code-specialized models, efficiency-oriented models, and open-weight models. The exact model list is frozen per evaluated release and reported in Table 2, while the harness supports adding new models under identical settings [9].

Agent Scaffolding. All models are evaluated under a unified agent scaffolding that standardizes prompts, repository context formatting, tool access, iteration limits, and stopping criteria. This design isolates model capability from orchestration effects and enables fair comparison across heterogeneous models [39].

Interaction Modes. We evaluate both one-shot patching, where a single patch is proposed without oracle execution, and iterative repair, where the agent may execute the oracle up to B times and refine patches using observed feedback [39]. The iterative setting reflects practical repair workflows and is particularly relevant for repository-level instances with complex dependencies [9].

5.3 Context and Tool Permissions

To mitigate confounds arising from uncontrolled context or tool usage, we enforce a shared fairness envelope across all LLM-based systems [9].

Context Budget. Agents are provided with issue or pull-request descriptions when available, failing commands and summarized logs, repository structure metadata (excluding ground-truth patch information), and optional retrieval access for repository search. Total context length is capped uniformly across models using identical truncation and summarization rules [9, 39].

Tool Access and Sandboxing. Agents may invoke a restricted set of deterministic tools, including *run_oracle*, *run_tests*, *grep/search*, and *apply_patch*. All execution occurs in a sandboxed environment without network access to ensure reproducibility and prevent data leakage [9]. Tool outputs are normalized to reduce spurious nondeterminism.

Table 2: Evaluated model suite for the frozen *QBench* release.

Category	Models	Notes
General frontier LLMs	GPT-4o; Claude-3.5-Sonnet; Gemini-2.5-Pro	State-of-the-art general-purpose models, representing the upper bound of LLM capability
Code-specialized LLMs	DeepSeek-Coder; Qwen2.5-Coder; Codestral-22B	Models optimized for code understanding and repository-level reasoning
Efficient / smaller models	GPT-4o-mini; DeepSeek-Coder-1.3B	Efficiency-oriented models used to study performance-cost trade-offs
Open-weight models	LLaMA-3.1-Instruct; DeepSeek-R1	Open-weight models evaluated under identical settings for transparency

Budget Constraints. Each instance is evaluated under fixed budgets on oracle executions B, wall-clock time, and (when applicable) model token usage. Budgets are track-specific to reflect differing computational demands, and results are reported at multiple budget levels to characterize scaling behavior [9, 39].

5.4 Experimental Metrics

We report multiple metrics to characterize repair effectiveness.

Primary Metric. Pass@1 success rate measures the fraction of instances solved within the given budget using a single final patch submission [5, 9, 36].

Anytime Performance. For iterative agents, we report success rates under different oracle-call budgets. An instance is considered solved under budget B if the agent produces a patch that passes the oracle within at most B oracle calls. This metric captures convergence speed under oracle-guided repair [39].

Robust Success under Probabilistic Oracles. For probabilistic instances, a patch is considered successful only if it satisfies the robust oracle definition. We additionally report empirical pass probabilities under repeated evaluation [4, 5].

5.5 Benchmark Packaging and Model List

To support comparable evaluation across systems and over time, each instance includes an explicit *reproducibility environment.* (i) *Pinned environment.* A concrete specification of interpreter/runtime version, dependencies, and OS/container image [9, 10, 37]. (ii) *Deterministic controls.* Standardized seeds and configuration for pseudo-randomness where applicable [4, 23]. (iii) *Executable protocol.* Machine-readable commands for setup, execution, and oracle evaluation [5, 9, 10]. (iv) *Isolation.* Execution in a sandboxed workspace to avoid leaking state across instances [9]. Each instance is distributed as a self-contained bundle with a standardized directory layout and a machine-readable manifest [5, 10, 37]. Conceptually, the manifest exposes identifiers and metadata (project, revision, track, tags); setup commands (environment creation/build); reproduction and evaluation commands; oracle parameters (type, repetitions, thresholds, timeouts); optional context payloads (issue text, stack traces, failing logs). We evaluate a fixed set of ten representative large language models that together span current capability frontiers, code-specialized variants, efficiency-oriented models, and open-weight alternatives. As shown in Table 2, this selection balances coverage and clarity, enabling meaningful comparison without excessive redundancy. For each evaluated release, the model list is frozen and all provenance information (provider version identifiers, decoding parameters, agent configuration, and tool-call budgets) is recorded. All models are evaluated under identical agent scaffolding and pinned container environments, and intermediate traces are retained to support auditing and re-evaluation.

Table 3: Pass@1 success rates for all evaluated repair systems.

System	RepoTrack	ScriptTrack	Overall
Conventional baselines			
Heuristic templates	7.3%	14.6%	10.9%
Search-based APR	11.8%	22.4%	17.1%
Efficient / smaller LLMs			
GPT-4o-mini	19.4%	33.1%	26.3%
DeepSeek-Coder-1.3B	17.6%	29.8%	23.7%
Code-specialized LLMs			
DeepSeek-Coder	25.1%	43.7%	34.4%
Qwen2.5-Coder	26.4%	44.2%	35.3%
Codestral-22B	27.9%	45.1%	36.5%
General frontier LLMs			
Gemini-2.5-Pro	34.1%	58.2%	46.2%
Claude-3.5-Sonnet	36.8%	60.9%	48.8%
GPT-4o	**38.6%**	**61.7%**	**50.1%**
Open-weight LLMs			
LLaMA-3.1-Instruct	21.9%	36.4%	29.1%
DeepSeek-R1	29.6%	49.8%	39.7%

6 Evaluation Results

This section presents the experimental results on *QBench*. Specifically, we first report quantitative results under the evaluation protocol defined in § 5, and then analyze the observed trends to surface quantum-specific challenges and implications. All results correspond to the frozen benchmark release used for evaluation.

6.1 Program Repair Effectiveness

We first report pass@1 success rates for all evaluated repair systems, including conventional baselines and individual LLM variants. Table 3 summarizes results across RepoTrack and ScriptTrack. Three key findings emerge. First, all LLM-based systems consistently outperform conventional baselines, indicating that most quantum bugs in *QBench* require semantic reasoning beyond shallow heuristics. Second, even the strongest frontier models solve fewer than 40% of repository-level instances, highlighting the intrinsic difficulty of repository-scale quantum software repair. Third, the performance gap between RepoTrack and ScriptTrack is stable across systems, reflecting the additional challenges introduced by integration, environment configuration, and cross-module dependencies. Overall, these results establish a clear capability hierarchy while revealing substantial room for improvement.

6.2 Model Capacity and Specialization

To analyze the impact of model capacity and specialization, we aggregate results by model family. Figure 4 reports average pass@1 success rates across RepoTrack and ScriptTrack. Two trends are evident. First, model capacity strongly correlates with repair performance. General frontier LLMs achieve the highest success rates (36.5% on RepoTrack and 60.3% on ScriptTrack), with gains particularly pronounced on RepoTrack, where long-horizon reasoning,

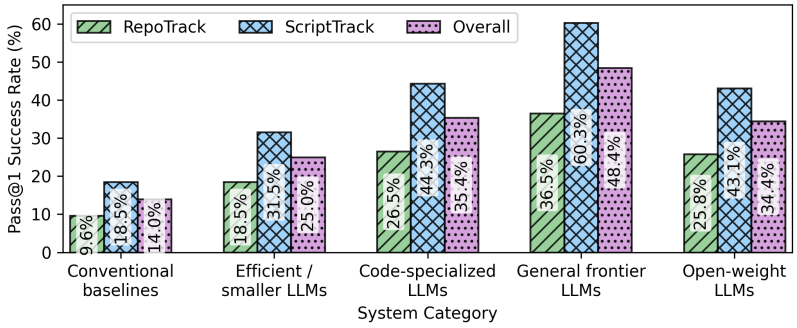


Figure 4: Average pass@1 success rates by system category.

Table 4: Robustness under probabilistic oracles.

System category	$\hat{\pi}(\hat{p})$ (mean)	Std. dev.
Search-based APR	0.81	0.14
Efficient LLMs	0.84	0.11
Code-specialized LLMs	0.87	0.09
General frontier LLMs	**0.93**	**0.05**
Open-weight LLMs	0.86	0.10

cross-file context integration, and environment handling are critical. Second, code specialization provides a clear advantage over smaller or efficiency-oriented models on both tracks, especially on ScriptTrack. However, specialization alone does not bridge the gap to frontier models on RepoTrack, indicating that it complements but does not replace overall model capacity. Open-weight models achieve intermediate performance, outperforming efficient models but remaining behind proprietary frontier systems, particularly for repository-level bugs. Overall, these results show that both capacity and specialization matter for quantum program repair, but their relative importance differs by track.

6.3 Oracle Feedback and Interaction

This research question studies the effect of oracle feedback and iterative interaction on repair performance. We evaluate representative LLMs spanning different capability levels: an efficient model (GPT-4o-mini), a code-specialized model (Codestral-22B), and a frontier model (GPT-4o). Figure 5 compares one-shot patching with iterative repair under increasing oracle budgets. Across all models, oracle feedback consistently improves performance, with substantially larger gains on RepoTrack than on ScriptTrack. For the efficient model, iteration nearly doubles RepoTrack success rates, indicating that feedback helps resolve build errors, missing dependencies, and partial fixes, though absolute performance remains limited. The code-specialized model shows moderate gains that saturate earlier, while the frontier model achieves the largest absolute improvement, with over a 17-point increase on RepoTrack from one-shot to B=10. In contrast, ScriptTrack exhibits smaller marginal gains across models, suggesting that many script-level bugs can be resolved without extensive interaction.

Overall, oracle feedback amplifies model capability but cannot fully compensate for limited reasoning capacity, and its benefits are most pronounced for repository-level quantum bugs.

6.4 Difficulty with Oracle Semantics

This research question examines how oracle semantics influence the difficulty of automated quantum bug repair. Unlike classical benchmarks that rely primarily on deterministic test oracles, quantum software frequently employs tolerance-based and probabilistic correctness criteria. Figure 6 reports pass@1 success rates stratified by oracle type, both for a frontier model (GPT-4o) and aggregated by model category. Across all models, deterministic oracles are consistently the easiest to satisfy, while probabilistic oracles are the most challenging, with gaps exceeding 20 percentage points for frontier models and widening for smaller ones. Tolerance-based oracles fall in between, as they relax strict equality but still require reasoning about acceptable numerical ranges, which many systems implicitly treat as deterministic conditions with noise. Probabilistic oracles pose a fundamentally different challenge: correctness is defined over distributions rather than single executions, yet current repair systems optimize against limited point estimates from oracle feedback. This mismatch produces brittle patches that fail under repeated evaluation, particularly in RepoTrack, where statistical uncertainty compounds integration and environment complexity. Overall, these results show that oracle semantics are a first-order factor in quantum software repair difficulty and cannot be reduced to noisy variants of deterministic testing.

6.5 Robustness and Probabilistic Evaluation

Passing a probabilistic oracle once does not guarantee a stable or semantically correct fix. To assess robustness, we re-evaluate patches deemed successful under the main evaluation and estimate their empirical pass probabilities. Table 4 reports the mean $\hat{\pi}(\hat{p})$ and standard deviation, aggregated by system category. Robustness varies substantially across systems. Search-based APR exhibits the lowest robustness and highest variance, indicating strong sensitivity to stochastic execution. LLM-based systems produce more stable patches on average, with frontier models achieving the highest pass probabilities and lowest variance. However, even frontier models fail in approximately 7% of repeated evaluations, showing that single-run success is an unreliable correctness signal for probabilistic quantum programs. Lower robustness is often associated with patches that adjust thresholds, sampling counts, or aggregation logic without resolving underlying distributional mismatches. These results highlight a critical distinction between passing an oracle and being correct under probabilistic semantics. By explicitly

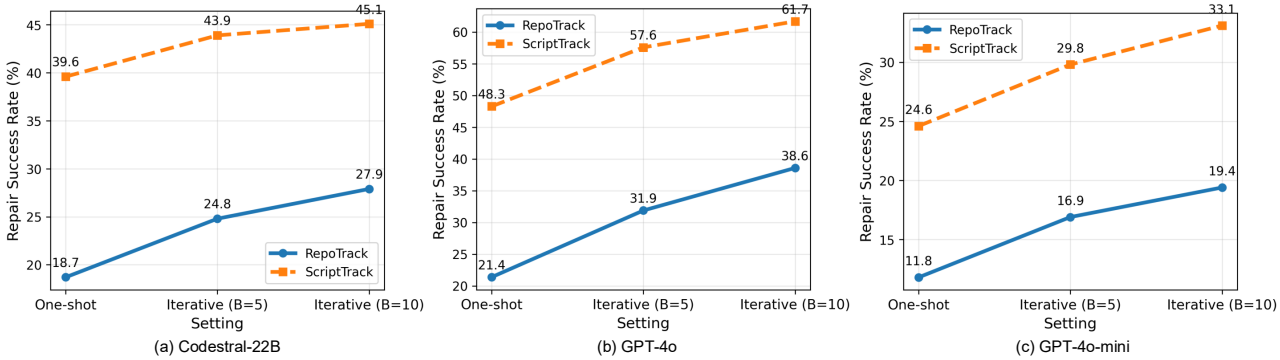


Figure 5: Impact of oracle-guided iteration.

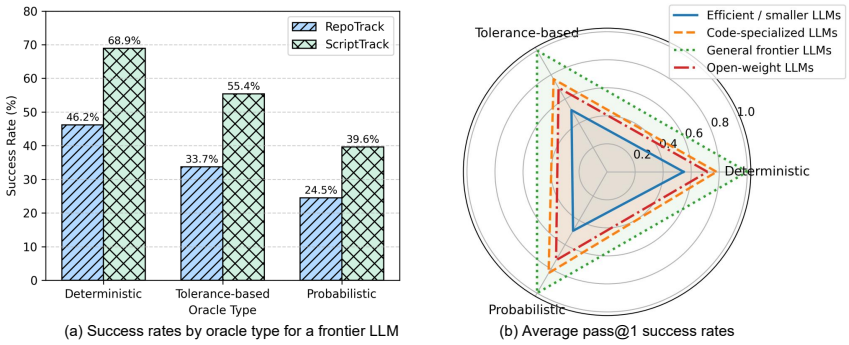


Figure 6: Effect of oracle semantics on repair difficulty.

Table 5: Comparison of bug characteristics by track.

Characteristic	RepoTrack	ScriptTrack
Cross-file fixes	68%	24%
Probabilistic oracles	41%	36%
Domain-level logic errors	22%	49%
Environment sensitivity	57%	29%

measuring robustness, *QBench* exposes brittle solutions that deterministic or single-run evaluations would overlook and emphasizes the need for statistically grounded repair methods.

6.6 Repository-Level and Script-Level Bugs

This research question analyzes the structural differences between repository-level and script-level quantum bugs and their implications for automated repair. Table 5 summarizes key characteristics across the two tracks. Repository-level bugs are dominated by non-local failure modes, with most fixes spanning multiple files and involving interactions among compilation passes, simulator backends, numerical libraries, and environment configuration. Over half of RepoTrack instances are sensitive to dependency versions or runtime environments, requiring integration reasoning beyond localized code edits. In contrast, script-level bugs more often stem from domain-level logic errors, such as incorrect parameter initialization, misinterpretation of measurement results, or improper handling of probabilistic outputs. These bugs are typically localized but demand semantic understanding of quantum mechanics and probabilistic reasoning. Probabilistic oracles are prevalent in both tracks, but play different roles: in RepoTrack they compound integration complexity and environment nondeterminism, while in ScriptTrack they more directly reflect domain-level semantic errors. Overall, these results reveal a clear bimodality in quantum software bugs. Repository-level bugs primarily stress infrastructure robustness and cross-module reasoning, whereas script-level bugs stress domain knowledge and probabilistic semantics. This distinction motivates the two-track design of *QBench* and suggests that effective automated repair requires track-aware strategies rather than a single unified approach.

7 Conclusion

In this work, we present *QBench*, an end-to-end benchmark for automated repair of real-world quantum software bugs. By supporting both deterministic and quantum-specific correctness semantics, *QBench* enables realistic and reproducible evaluation of repair systems. Our empirical study shows that stochastic correctness criteria, environment sensitivity, and cross-layer defects remain major challenges for existing repair approaches. By establishing a standardized evaluation framework, *QBench* provides a foundation for future research on automated quantum software repair and dependable quantum software engineering.

Acknowledgments

This research was supported in part by the Ningbo Yongjiang Talent Programme and in part by the CPS-Yangtze Delta Region Industrial Innovation Center of Quantum and Information Technology-MindSpore Quantum Open Fund.

References

[1] Earl T. Barr, Mark Harman, Phil McMinn, Muhammad Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering (TSE)* (2015). doi:10.1109/TSE.2014.2372785

[2] José Campos and André Souto. 2021. QBugs: A Collection of Reproducible Bugs in Quantum Algorithms and a Supporting Infrastructure to Enable Controlled Quantum Software Testing and Debugging Experiments. In *Proceedings of the 2021 IEEE/ACM 2nd International Workshop on Quantum Software Engineering (Q-SE)*. 28–32. doi:10.1109/Q-SE52541.2021.00013

[3] Manuel De Stefano, Fabiano Pecorelli, Dario Di Nucci, Fabio Palomba, and Andrea De Lucia. 2024. The quantum frontier of software engineering: A systematic mapping study. *Information and Software Technology* 175 (2024), 107525.

[4] Saikat Dutta, August Shi, Rutvik Choudhary, Zhekun Zhang, Aryaman Jain, and Sasa Misailovic. 2020. Detecting Flaky Tests in Probabilistic and Machine Learning Applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 211–224. doi:10.1145/3395363.3397366

[5] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Transactions on Software Engineering (TSE)* (2015). doi:10.1109/TSE.2015.2454513

[6] Hossein Honarvar et al. 2020. Property-Based Testing of Quantum Programs in Q#. In *Proceedings of the International Workshop on Quantum Software Engineering (Q-SE@ICSE)*. doi:10.1145/3387940.3391459

[7] Tianmin Hu, Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Huanting Wang, Meng Li, and Zheng Wang. 2024. Upbeat: Test input checks of q# quantum libraries. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 186–198.

[8] Kai Huang, Xiangxin Meng, Jian Zhang, Yang Liu, Wenjie Wang, Shuhao Li, Yuqing Zhang, et al. 2023. An Empirical Study on Fine-Tuning Large Language Models of Code for Automated Program Repair. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE '23)*. IEEE/ACM, 1162–1174. doi:10.1109/ASE56229.2023.00181

[9] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=VTF8yNQM66

[10] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA)*. 437–440. doi:10.1145/2610384.2628055

[11] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic Patch Generation Learned from Human-Written Patches. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. IEEE, 802–811. doi:10.1109/ICSE.2013.6606626

[12] Linsey J. Kitt and Myra B. Cohen. 2024. MorphQ++: A Reproducibility Study of Metamorphic Testing on Quantum Compilers. In *Proceedings of the 2024 ASE Workshop on Replications and Negative Results (RENE)*. 15–21. doi:10.1145/3695750.3695823

[13] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering (TSE)* 38, 1 (2012), 54–72. doi:10.1109/TSE.2011.104

[14] Tingting Li, Liqiang Lu, Ziming Zhao, Ziqi Tan, Siwei Tan, and Jianwei Yin. 2024. Qust: Optimizing quantum neural network against spatial and temporal noise biases. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 44, 4 (2024), 1434–1447.

[15] Tingting Li and Ziming Zhao. 2024. Moirai: Optimizing quantum serverless function orchestration via device allocation and circuit deployment. In *2024 IEEE International Conference on Web Services (ICWS)*. IEEE, 707–717.

[16] Tingting Li, Ziming Zhao, Liqiang Lu, Siwei Tan, and Jianwei Yin. 2025. Empowering quantum error traceability with moe for automatic calibration. In *2025 Design, Automation & Test in Europe Conference (DATE)*. IEEE, 1–7.

[17] Tingting Li, Ziming Zhao, and Jianwei Yin. 2025. AutoFid: Adaptive and Noise-Aware Fidelity Measurement for Quantum Programs via Circuit Graph Analysis. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2631–2643.

[18] Tingting Li, Ziming Zhao, and Jianwei Yin. 2025. Fortuna: Towards efficient selection of high-fidelity link for quantum network in the wild. In *IEEE INFOCOM 2025-IEEE Conference on Computer Communications*. IEEE, 1–10.

[19] Tingting Li, Ziming Zhao, and Jianwei Yin. 2025. Task-Driven Device Fingerprinting for Quantum Cloud Platforms via Modeling QNN Outcomes under Noise. *IEEE Transactions on Information Forensics and Security* (2025).

[20] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. QuixBugs: A Multi-Lingual Program Repair Benchmark Set Based on the Quixey Challenge. In *Proceedings of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion '17)*. ACM, 55–56. doi:10.1145/3135932.3135941

[21] Fan Long and Martin Rinard. 2015. Staged Program Repair with Condition Synthesis. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. 166–178. doi:10.1145/2786805.2786811

[22] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. doi:10.1145/2837614.2837617

[23] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An Empirical Analysis of Flaky Tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*. 643–653. doi:10.1145/2635868.2635920

[24] Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. 2019. Bears: An extensible java bug benchmark for automatic program repair studies. In *2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 468–478.

[25] Juan M. Murillo, Jose Garcia-Alonso, Enrique Moguel, Johanna Barzen, Frank Leymann, Shaukat Ali, Tao Yue, Paolo Arcaini, Ricardo Pérez-Castillo, Ignacio García Rodríguez de Guzmán, Mario Piattini, Antonio Ruiz-Cortés, Antonio Brogi, Jianjun Zhao, Andriy Miranskyy, and Manuel Wimmer. 2025. Quantum Software Engineering: Roadmap and Challenges Ahead. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 34, 5, Article 154 (2025). doi:10.1145/3712002

[26] Noah H Oldfield, Christoph Laaber, Tao Yue, and Shaukat Ali. 2025. Faster and better quantum software testing through specification reduction and projective measurements. *ACM Transactions on Software Engineering and Methodology* 34, 7 (2025), 1–39.

[27] Matteo Paltenghi and Michael Pradel. 2022. Bugs in Quantum Computing Platforms: An Empirical Study. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1, Article 86 (2022). doi:10.1145/3527330

[28] Matteo Paltenghi and Michael Pradel. 2023. MorphQ: Metamorphic Testing of the Qiskit Quantum Computing Platform. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE)*. 2413–2424. doi:10.1109/ICSE48619.2023.00202

[29] Gabriel Pontolillo and Mohammad Reza Mousavi. 2022. A multi-lingual benchmark for property-based testing of quantum programs. In *Proceedings of the 3rd international workshop on quantum software engineering*. 1–7.

[30] Leite Ramalho et al. 2025. Testing and debugging quantum programs: The road to 2030. *ACM Transactions on Software Engineering and Methodology* 34, 5 (2025), 1–46.

[31] Francisco Ribeiro, Rui Abreu, and João Saraiva. 2022. Framing program repair as code completion. In *Proceedings of the Third International Workshop on Automated Program Repair*. 38–45.

[32] Ripon K. Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R. Prasad. 2018. Bugs.jar: A Large-Scale, Diverse Dataset of Real-World Java Bugs. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR '18)*. ACM, 10–13. doi:10.1145/3196398.3196473

[33] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the Cure Worse Than the Disease? Overfitting in Automated Program Repair. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '15)*. ACM, 532–543. doi:10.1145/2786805.2786825

[34] Jiyuan Wang, Qian Zhang, Guoqing Harry Xu, and Miryung Kim. 2021. QDiff: Differential Testing of Quantum Software Stacks. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 692–704. doi:10.1109/ASE51524.2021.9678792

[35] Xinyi Wang, Paolo Arcaini, Tao Yue, and Shaukat Ali. 2022. QuSBT: Search-based testing of quantum programs. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 173–177.

[36] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically Finding Patches Using Genetic Programming. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*. 364–374. doi:10.1109/ICSE.2009.5070536

[37] Ratnadira Widyasari et al. 2020. BugsInPy: A Database of Existing Bugs in Python Programs to Enable Controlled Testing and Debugging Studies. In *ESEC/FSE*. 5 pages. doi:10.1145/3368089.3417943

[38] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-trained Language Models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*. IEEE/ACM, 1482–1494. doi:10.1109/ICSE48619.2023.00129

[39] Chunqiu Steven Xia and Lingming Zhang. 2024. Automated Program Repair via Conversation: Fixing 162 out of 337 Bugs for $0.42 Each using ChatGPT. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24)*. ACM. doi:10.1145/3650212.3680323

[40] Amanda Xu, Abtin Molavi, Swamit Tannu, and Aws Albarghouthi. 2025. Optimizing Quantum Circuits, Fast and Slow. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 777–793. doi:10.1145/3669940.3707240

[41] Xusheng Xu et al. 2024. MindSpore Quantum: a user-friendly, high-performance, and AI-compatible quantum computing framework. *arXiv:2406.17248* (2024).

[42] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems* 37 (2024), 50528–50652.

[43] Jiaming Ye, Shangzhou Xia, Fuyuan Zhang, Paolo Arcaini, Lei Ma, Jianjun Zhao, and Fuyuki Ishikawa. 2023. QuraTest: Integrating Quantum Specific Features in Quantum Program Testing. In *ASE*. IEEE, 1149–1161.

[44] Pengzhan Zhao et al. 2021. Bugs4Q: A Benchmark of Real Bugs for Quantum Programs. In *Proceedings of the 36th IEEE/ACM International Conference on ASE*. 1373–1376. doi:10.1109/ASE51524.2021.9678908

[45] Ziming Zhao, Tingting Li, Zhaoxuan Li, and Jianwei Yin. 2026. Relational Verification for Cost-Aware Quantum Program Optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 40. 14414–14422.