Reproducibility of Bugs4Q under Framework Evolution: Toward Supporting Quantum Program Migration

Haruto OHTO†, Yuta ISHIMOTO†, Shinsuke MATSUMOTO†, and Shinji KUSUMOTO†

† The University of Osaka, Osaka, Japan

E-mail: †{hr-ohto,ishimoto,shinsuke,kusumoto}@ist.osaka-u.ac.jp

Abstract The reproducibility of software defect datasets is essential for obtaining reliable and comparable research results. However, software defect datasets can suffer from reproduction failures, where reported bugs become non-reproducible over time. We studied the reproducibility of Bugs4Q, a representative defect dataset of real-world bugs in Qiskit programs, from both quantitative and qualitative perspectives. The study revealed that most reproduction failures stem not from quantum-specific characteristics but from framework version upgrades and dependency changes. Guided by these causes, we manually migrated Bugs4Q to the latest Qiskit version and substantially improved its reproducibility, releasing the result as Bugs4Q-Robust. This suggests that such failures can be addressed systematically, and that automatically migrating a defect dataset to newer framework versions may keep it reproducible in a scalable manner. However, evidence drawn solely from Qiskit limits the external validity of this approach. As a next step toward scalable dataset migration, we present a plan to study framework evolution in other quantum software frameworks and examine how the migration approach generalizes beyond Qiskit.

Key words quantum software, automated migration, software reproducibility, software defects

1. Introduction

Software defect datasets [1], which consist of buggy programs and their corresponding fixed versions, have served as a fundamental resource for research on software testing, fault localization, and automated program repair. Reproducible defect datasets enable systematic evaluation and fair comparison of techniques. This indicates that the reproducibility of defect datasets is essential for obtaining reliable and comparable research results.

Unfortunately, software defect datasets are not immune to *reproduction failures* (i.e., reported bugs become non-reproducible) after their creation. Such failures can occur even when the buggy and fixed code remains unchanged, because of changes in execution environments and dependencies [2]. Even if a study is valid as long as the bugs were reproducible at the time of evaluation, subsequent reproduction failures hinder follow-up studies that seek to verify or extend its results.

While reproduction failures have been studied in defect datasets for classical programs, it remains unclear whether similar failures occur in *quantum programs*. Unlike classical programs, quantum programs consist of quantum gates that manipulate qubits, the basic units of quantum computation [3]. Driven by the growing need for quantum software engineering, researchers have constructed several defect datasets for quantum programs [4, 5]. Prior studies have reported quantum-specific categories of code-quality issues in quantum programs [6, 7]. These quantum-specific characteristics suggest that findings on classical defect dataset reproducibility may not directly generalize to their quantum counterparts.

In this study, we investigate the reproducibility of Bugs4Q [4], a defect dataset widely used in research on testing and debugging quantum programs [8–10]. Our study includes 46,620 quantum program executions of 37 Bugs4Q artifacts across 21 Qiskit core-library versions spanning three major Qiskit series released over a three-year period. We adopt a reproducibility criterion adapted from a prior study on classical defect datasets [2], which checks whether the underlying cause of buggy behavior is consistent with the original source. After this quantitative evaluation, we manually classify the root causes of reproduction failures.

We further outline a research agenda that investigates whether other quantum software frameworks introduce similar breaking changes, and explores how to migrate the affected programs automatically. As a concrete first step, we construct Bugs4Q-Robust, a manually patched version of Bugs4Q guided by the root causes, and we frame this manual patching process as an initial example of the migration we ultimately aim to automate.

2. Experimental Design

This section describes our experimental design, including the RQs, dataset, reproducibility criterion, and experimental procedure for our longitudinal analysis.

2.1 Research Questions

We address the following RQs, which are aligned with those of the prior work [2]:

RQ1 *How does the reproducibility of Bugs4Q change?*
RQ2 *What are root causes for the reproduction failures?*

In RQ1, we quantitatively investigate the changes in reproducibility of Bugs4Q as the Qiskit ecosystem evolves, because subsequent studies can rely on it as a benchmark only if its artifacts continue to exhibit the originally reported behaviors. In RQ2, we qualitatively investigate why the reproduction failures occur. While RQ1 reveals changes in reproducibility over time, it does not explain the underlying causes of reproduction failures; identifying these causes clarifies the maintenance required to keep Bugs4Q usable.

2.2 Dataset

We use *Bugs4Q* [4], a real-world defect dataset for quantum programs written in Qiskit. Qiskit is a Python framework developed by IBM and the open-source community, with 7.3k stars on GitHub.Bugs4Q consists of 42 buggy Qiskit programs collected

from three major platforms: GitHub, Stack Overflow, and Stack Exchange. For each buggy program, Bugs4Q provides a corresponding fixed version and a test code that reproduces the bug.

To the best of our knowledge, Bugs4Q is the only benchmark that meets all three of the following criteria, which motivates our choice.

- The buggy programs are written by developers; that is, the dataset contains real-world bugs.
- Both buggy and fixed programs are provided.
- Test code for reproducing the bugs is included.

Our criteria are aligned with those of Java defect datasets used in prior work [2], such as Defects4J [1]. The first criterion is necessary because we are interested in problems actually faced by programmers. We therefore excluded synthetic datasets such as those based on quantum circuit mutations [5]. The fixed programs are necessary to verify the expected behavior. Test code is necessary to confirm that bugs in the buggy programs remain reproducible and that the expected behavior of the fixed programs is preserved over time.

We excluded five artifacts that lack test code, leaving 37 artifacts for our study. Following the bug categories in the original paper [4], these 37 artifacts are classified into one of the three categories. *Wrong Output (WO, 17 artifacts)* refers to artifacts where the output of the buggy program does not match the expected output specified by the test code. *Throw Exception (TE, 18 artifacts)* refers to artifacts where the buggy program raises an exception during execution. *Simulation Failure (SF, 2 artifacts)* refers to artifacts where the simulation fails for the buggy program. Specifically, the simulation outcome is inspected by the `success` attribute of the object representing the simulation result.

2.3 Criterion for Reproducibility

We regard a Bugs4Q artifact as reproducible when the fixed version succeeds and the buggy version exhibits runtime evidence consistent with the cause described in the original source of the artifact. This criterion focuses on whether the originally reported bug is reproduced, rather than merely checking whether the buggy version fails for any reason.

For artifacts categorized as TE, we check whether the error message in the traceback is consistent with the cause described in the source. For artifacts categorized as WO, we check whether the incorrect output observed at runtime agrees with the incorrect output described in the source. For example, if the source describes biased measurement counts toward a specific state, we regard the output as consistent when the reproduced buggy version exhibits the same bias. For artifacts categorized as SF, we check whether the simulator failure messageis consistent with the cause described in the source. Because semantic consistency in this criterion is difficult to check automatically, the first author manually validated it.

2.4 Longitudinal Analysis

We study reproducibility across 21 snapshots from May 16, 2022, the release date of Bugs4Q, to April 1, 2026. We use the release date as the starting point because we are interested in how reproducibility changes after the dataset became available to the research community. Each snapshot simulates the environment at a given point in time by using the latest stable version of Qiskit available on PyPI by that date. The snapshots cover three major Qiskit version series, namely 0.x, 1.x, and 2.x, with seven snapshots for each series.

Qiskit Versions and Snapshot Periods. For the 0.x series, the first snapshot is taken on the release date of Bugs4Q. For the 1.x and 2.x series, the first snapshot is taken on the release date of the initial version of each series, because we aim to evaluate reproducibility after each major version upgrade. In all series, subsequent snapshots are taken every two months over the following year, yielding seven snapshots per series. We use Python 3.9.25 for the 0.x and 1.x series, and Python 3.10.20 for the 2.x series. In total, these 21 snapshots

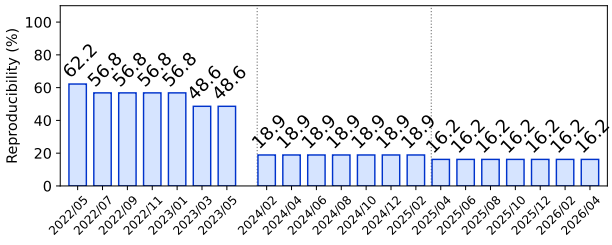

Fig. 1: Reproducibility for each snapshot. The value above each bar indicates the percentage of reproducible artifacts at the corresponding snapshot. Dashed vertical lines separate the major versions of Qiskit: 0.x, 1.x, and 2.x series.

cover a three-year observation period. The exact Qiskit versions and snapshot dates are available in our replication package.

Environment at Each Snapshot. For each snapshot, we build a separate Docker image to isolate the corresponding execution environment. We install the target Qiskit version and adjust the surrounding Qiskit-related packages for compatibility, following the official release notes and documentation. When a surrounding package has no version compatible with the target Qiskit version, we remove it from the environment; otherwise, we pin it to a compatible version.

Execution at Each Snapshot. We execute the buggy and fixed versions of each artifact through the bundled test case on the corresponding Docker container. Because quantum programs may produce non-deterministic outputs [11], we run both versions 30 times, following empirical guidelines [12]. For each execution, we record the test outcome and save the runtime log, including stdout and stderr. We then check whether reproducibility criterion defined in Section 2.3 is satisfied.

2.5 Hardware

All our experiments are performed on a classical computer using the simulator functionality provided by Qiskit. Although programs written in Qiskit can be executed on actual quantum computers, we use the simulator due to the limited availability of real quantum hardware.

3. Evaluation

3.1 RQ1: Bugs4Q Reproducibility Change

Targets. At each snapshot defined in Section 2.4, we execute the buggy and fixed quantum programs of each Bugs4Q artifact. Our analysis covers 46,620 quantum program executions in total, corresponding to the full combination of 21 snapshots, 37 artifacts, 30 repeated runs, and buggy/fixed programs.

Evaluation Metric. We consider artifact p reproducible at snapshot t if the reproducibility criterion defined in Section 2.3 is satisfied in all 30 trials. This definition treats flaky artifacts [11,13] (i.e., those that intermittently succeed and fail) as non-reproducible.

We define $\mathrm{Repro}(t, p)$ as an indicator variable that takes 1 when this condition holds and 0 otherwise. The number of reproducible artifacts at snapshot t is then:

$$R(t) = \sum_{p} \mathrm{Repro}(t, p). \tag{1}$$

We report $R(t)$ as the percentage of reproducible artifacts at each snapshot to examine how reproducibility changes.

Results. Figure 1 plots $R(t)$ for each snapshot t. **As shown in Figure 1, Bugs4Q reproducibility dropped sharply across major Qiskit versions: from 48.6%–62.2% in the 0.x series to 16.2%–18.9% in the 1.x and 2.x series, indicating that it retains a meaningful level of reproducibility only under legacy versions.** This

sharp drop is consistent with the breaking changes introduced at each major Qiskit release [14, 15]. Notably, even at the earliest snapshot, v0.20.1, 62.2% of artifacts were reproducible. This is lower than Defects4J, the dataset with the highest reproducibility (96.9%) reported by Zhu et al. [2]. This imperfection is plausibly due to the non-determinism of quantum measurements as well as inconsistencies between the environment used during Bugs4Q construction and our experimental setup, since Bugs4Q does not record the exact library versions used at collection time. We further investigate the root causes of these reproduction failures in Section 3.2.

> **RQ1:** Bugs4Q reproducibility dropped sharply (from 48.6% to 18.9%) once major version upgrades of Qiskit occur after its construction. Over the three-year observation period, 31 of the 37 Bugs4Q artifacts (83.8%) experienced reproduction failures at least once.

3.2 RQ2: Root Causes and Patch Strategies

Targets. We analyze the set of snapshot–artifact pairs (t, p) for which $\mathrm{Repro}(t, p) = 0$, totaling 543 pairs. These pairs cover all reproduction failures observed in Section 3.1.

Classification of Reproduction Failures. The reproduction failure categories proposed by Zhu et al. [2] are too specific to fit our setting, as they depend on particular Java build tools such as Maven and Gradle. Therefore, we adopt an open coding approach to classify reproduction failures in quantum software defect datasets, since no existing taxonomy is available for this purpose. For each target (snapshot, artifact) pair, two of the authors independently classify the root cause of each of the 543 instances by inspecting the runtime logs and the Bugs4Q source (e.g., the linked GitHub issues from which the artifact was collected), assigning a short descriptive code. After the independent coding is completed, we compute Fleiss's kappa [16] to assess the inter-rater reliability. The resulting value is 0.67, indicating *substantial agreement*. The two annotators then jointly resolve disagreements through discussion to determine the final root cause of each instance, and refine the categories by merging semantically equivalent codes. In addition, for each resulting root-cause category, we describe a patch strategy to restore reproducibility, which the annotators came up with during the coding process while inspecting the raw data such as the runtime logs.

Results. Table 1 summarizes each root cause category, its corresponding patch strategy, and its share of instances. Our taxonomy consists of four main categories and five subcategories, yielding seven leaf categories in total. The rest of this section discusses each category, presenting its symptoms (see ⚠ icon), code examples when available (</>), and potential patch strategies (🛠).

a) Import incompatibility

This is the most common root cause, accounting for 261 failures (48.1%). ⚠ These failures are caused by import-path mismatches: the artifact source code remains unchanged, but the referenced modules or packages are no longer available at their original paths in the target Qiskit version. </> The following code excerpt from an artifact (ID:19 in Bugs4Q) provides an example of this root cause.

```
from qiskit.extensions import HGate, CXGate
```

This import statement fails in v1.0.0 and later because the legacy `qiskit.extensions` module is no longer available at that import path. 🛠 The patch strategy updates such outdated import paths to their replacements documented in the Qiskit migration guides [14, 15]. In the code excerpt above, the legacy `qiskit.extensions` import is migrated to `qiskit.circuit.library`.

b) API invocation incompatibility

This root cause accounts for 184 failures (33.9%). ⚠ These failures occur when the artifact source code invokes Qiskit APIs that have been removed or whose call forms have changed in the target Qiskit version. </> The following code excerpt from an artifact (ID:39 in Bugs4Q) provides an example of this root cause.

```
job = execute(
    qc,
    backend = Aer.get_backend('qasm_simulator'),
    shots=1024
)
```

This code submits a circuit to a backend through the legacy `execute` API function. The failure surfaces only when `execute` is called. 🛠 The patch strategy replaces legacy API invocations with equivalents supported by the target version.

c) Library–library incompatibility

This root cause accounts for 25 failures (4.6%). ⚠ These failures occur when Qiskit-related packages lose runtime compatibility with each other, even though they can still be installed together. 🛠 A patch strategy is to adjust the versions of the surrounding Qiskit-related packages so that they remain mutually compatible with the target Qiskit version. This is the only root cause whose patch strategy operates entirely at the environment level (e.g., version pinning), without modifying artifact source code or test oracles.

d) Library-internal behavior change

This root cause accounts for 33 failures (6.1%). ⚠ These failures occur when the Qiskit interfaces used by the artifact remain valid, but changes in the internal behavior cause the reproduction failures. For instance, an operation that originally triggered the bug may now be handled internally by Qiskit, causing the original bug-triggering condition to disappear. In such cases, no general patch strategy is applicable.

e) Output format change

This root cause accounts for five failures (0.9%). ⚠ These failures occur when Qiskit changes the textual representation of outputs while the program behavior remains unchanged. For example, a qubit label printed in the output changes from `q_0` to `q`, causing a string-based assertion to fail. 🛠 A patch strategy is to update the test oracle to match the output format of the target Qiskit version. This modification is confined to the test oracle and does not alter the program logic.

f) Quantum-specific reproduction failures

This root cause accounts for 28 failures (5.1%). ⚠ These failures stem not from API or dependency changes, but from the probabilistic nature of quantum programs: the observed output may occasionally deviate from the expected distribution due to non-determinism of quantum programs, even when the source code is unchanged. </> The following excerpt from an artifact (ID:26 in Bugs4Q) provides an example of this root cause.

```
circuit = QuantumCircuit(2)
circuit.h(0)
circuit.x(1)
circuit.cx(0, 1)
circuit.measure_all()
backend = QasmSimulator()
job = backend.run(..., shots=1024)
counts = job.result().get_counts(circuit)
```

The excerpt executes a quantum circuit and returns sampled counts for each state from measurement. The ideal output distribution assigns a probability of 0.5 to each of the `01` and `10` states. In our setting of 1,024 shots, the expected count for each state is 512. Thus, the corresponding test applies a statistical assertion [10, 17] to check whether the sampled counts deviate from the ideal distribution. In the observed run, the sampled counts were `{'01': 476, '10': 548}`. Although the result appeared close to the expected distribution, the test oracle rejected it and caused the fixed version to fail. 🛠 A patch strategy is to tune the shot count and oracle threshold so that

Table 1: Root causes and possible patch strategies. Gray-shaded rows indicate the main categories. N/A indicates that no general patch strategy is applicable. Repro. denotes reproduction.

Root cause	Percentage	Definition of main category	Patch strategy
Source-related repro. failures	**82.0%**	Failures caused by artifact source code using Qiskit interfaces that are no longer valid in the target Qiskit version.	Update obsolete import paths or replace incompatible API invocations with calls supported by the target Qiskit version.
Import incompatibility	48.1%		
API invocation incompatibility	33.9%		
Library-related repro. failures	**11.6%**	Failures caused by changes in Qiskit-related libraries, even though artifact source code still uses valid Qiskit interfaces.	Adjust dependency versions or update test oracles when applicable; otherwise N/A.
Library–library incompatibility	4.6%		
Library-internal behavior change	6.1%		
Output format change	0.9%		
Quantum-specific repro. failures	**5.1%**	Failures caused by probabilistic variation in quantum measurement or simulation results.	Tune the shot count and relax the oracle threshold.
Original test issues	**1.3%**	Failures where the original test does not adequately validate program behavior, making the correctness of the fixed version unclear.	N/A

statistically plausible sampling variation is not rejected as a reproduction failure, while preserving the oracle's ability to distinguish buggy and fixed behavior.

g) Original test issues

This root cause accounts for the remaining seven failures (1.3%), all originating from a single artifact. ⚠ These failures stem from a problem in the original test rather than Qiskit evolution: the test does not seem to correctly check the target behavior. We could not confirm whether the fixed program produced the expected output when Bugs4Q was constructed. Given this uncertainty about the correctness of the fixed program, we consider this case not patchable.

We further observe that some artifacts contribute multiple reproduction failures classified under different root causes across Qiskit versions. For example, an artifact (ID:20 in Bugs4Q) fails under *Library-internal behavior change* in v0.20.1 – v0.22.3, shifts to *API invocation incompatibility* in v0.23.2 – v0.24.0, and finally fails under *Import incompatibility* from v1.0.0 and later. This indicates that addressing one root cause for a given artifact does not guarantee its long-term reproducibility, because the same artifact may later experience failures caused by different root causes.

RQ2: We identified seven root causes across 543 reproduction failures. Among these failures, 93.6% are dependency-related (Source-related and Library-related). The most frequent leaf category is *Import incompatibility* (48.1%), where import statements reference Qiskit modules that have become incompatible. Only 4.6% of failures have a patch strategy that does not require modifying artifact source code.

4. Case Study: Manual Migration of Bugs4Q

As an initial example of quantum program migration, we manually migrate Bugs4Q to the latest Qiskit version (v2.3.1 as of April 1, 2026) and release the result as *Bugs4Q-Robust*. This case study illustrates how the patch strategies identified in Section 3.2 can be applied in practice, and surfaces the categories of reproduction failures that remain difficult to recover even with manual effort. We use this experience to motivate the future research plans.

4.1 Construction

We construct Bugs4Q-Robust by manually creating concrete patches guided by the patch strategies identified in Section 3.2. Patches are applied only to root causes that have an applicable patch strategy; *Library-internal behavior change* and *Original test issues* are left unpatched. For each artifact, we restrict modifications to the observed root cause and keep the patch minimal, preserving the original program logic and test intent.

As the only environment-level patch in Bugs4Q-Robust, we update the dependency specification by selecting the latest versions of the surrounding Qiskit-related packages that remain mutually compatible with Qiskit v2.3.1. This patch addresses *Library–library incompatibility*.

The remaining patches modify artifact source code. For *Import incompatibility* and *API invocation incompatibility*, we migrate the affected import statements and API invocations following the Qiskit migration guides [14, 15]. One common patch is the migration from the legacy `execute` API function to the recommended `transpile`–`backend.run`.We applied this migration to 12 of the 37 artifacts.

4.2 Reproducibility After Manual Migration

Under v2.3.1, the latest version as of April 1, 2026, the reproducibility of Bugs4Q-Robust improved from 16.2% to 59.5% over the original Bugs4Q, demonstrating the effectiveness of the patch strategies identified in Section 3.2. This +43.3 percentage point gain corresponds to recovering 16 additional artifacts under the latest framework version, indicating that a substantial portion of the reproduction failures observed in Section 3.1 are addressable through manual patching.

To investigate the causes of artifacts that remain non-reproducible in Bugs4Q-Robust, the first author labeled them by reusing the categories established in Section 3.2. Table 2 summarizes the classification results of the 15 artifacts that remained non-reproducible after manual migration. **Table 2 shows that most failures categorized as *Source-related reproduction failures* have been resolved in Bugs4Q-Robust.** The remaining four cases of *Import incompatibility* involve Qiskit-related packages or modules that have been removed without direct replacements. Overall, the dominant remaining cause is *Library-internal behavior change*, which accounts for 10 of the 15 remaining artifacts. This indicates that Bugs4Q-Robust effectively resolves reproduction failures caused by source-level interface mismatches, whereas library-internal behavior changes remain difficult to recover through our applied patches.

***Library-internal behavior change* can eliminate the original bug-triggering conditions.** For example, one artifact (ID:14 in

Table 2: Remaining root causes of reproduction failures in Bugs4Q-Robust.

Root cause	# Artifacts	Percentage
Import incompatibility	4	26.7%
Library-internal behavior change	10	66.7%
Original test issues	1	6.7%

Bugs4Q), which could not be reproduced even in Bugs4Q-Robust, reported a bug caused by omitting the diagonalization step required when using `PauliExpectation`. In the successor `Estimator` API, however, diagonalization is handled implicitly, so the same user-level omission no longer triggers the original bug. This represents a fundamental limitation of bug reproduction: when the new API automatically handles what users previously had to do manually, the original bug can no longer be triggered, regardless of patching.

5. Future Research Plan

We outline two directions for future research that build on the findings of this study. The first generalizes our investigation beyond Qiskit to other quantum programming frameworks (Section 5.1), and the second explores automated migration support for quantum programs affected by framework evolution (Section 5.2).

5.1 Generalizing Compatibility Analysis Beyond Qiskit

In the previous sections, we showed that the evolution of a quantum programming framework can break the quantum programs that depend on it through breaking changes such as removed APIs or modified behavior, and that these failures can in turn compromise the reproducibility of a defect dataset built from such programs. Concretely, we observed this phenomenon for Qiskit and the Bugs4Q dataset, where updates to the framework rendered previously reproducible defects no longer reproducible.

These findings, however, are specific to Qiskit and Bugs4Q. The quantum software ecosystem comprises multiple independently developed frameworks, such as Amazon Braket [18], Cirq [19], and PennyLane [20]. Each framework may evolve in its own way, with breaking changes differing in both frequency and impact. Thus, observing only Qiskit does not show whether the trend we found generalizes across the ecosystem, nor how it varies from one framework to another. We therefore plan to extend our investigation to other quantum programming frameworks and the quantum programs that depend on them, with the goal of characterizing ecosystem-wide trends in how framework evolution affects quantum programs.

We examine framework evolution from two perspectives to characterize these trends. The first concerns the *breaking changes* that quantum programming frameworks introduce during their evolution, such as removed or renamed APIs, changed default parameters, or modified call patterns. The second concerns their *practical impact* on real-world quantum programs, that is, the extent to which these changes actually break existing code or alter its behavior. We study these through the following future research questions:

FRQ1 *How frequently do breaking changes occur in quantum programming frameworks?*

FRQ2 *What is the practical impact of framework evolution on real-world quantum programs?*

We first select the target frameworks. Following prior empirical work on quantum software [7, 21], we use the project list curated by the Quantum Open Source Foundation (QOSF) as our starting point. We select from this list the frameworks distributed as Python libraries, since Python is the most widely used language for quantum software development [21]. We then apply two further filters. First, following the QOSF survey [22], we keep frameworks that are not archived and have received at least 20 commits in the last year. Second, following a large-scale study of API breaking changes in classical software [23], we retain those above the first quartile in both repository age and number of releases.

To answer FRQ1, we identify breaking changes introduced across consecutive releases of the selected frameworks. Although tools such as AexPy [24] detect API breaking changes in Python, we are not aware of a tool that covers the broader range of breaking changes considered in this study. We therefore adapt Xia et al.'s approach [25] to Python frameworks, as it addresses a broader range of breaking changes in JavaScript, another dynamically typed language. Following Xia et al. [25], we collect breaking changes from documentation sources such as release notes, changelogs, and pull requests. We then classify the collected changes and analyze their frequency across frameworks and releases.

To answer FRQ2, we collect real-world quantum programs that depend on the selected frameworks and run them across framework versions. Following Upadhyay et al. [21], we identify repositories that depend on a target framework and retain executable ones. We then vary the framework version, re-execute each program, and check whether it still runs or changes its observable behavior. Finally, we relate the observed failures to the breaking changes identified in FRQ1 and quantify the impact of framework evolution on real-world quantum programs.

5.2 Toward Automated Code Migration for Quantum Programs

A natural next challenge is to automatically migrate quantum programs affected by framework evolution to newer framework versions. The findings for Qiskit demonstrate that framework evolution can affect quantum programs in practice, and Section 5.1 aims to determine whether similar effects extend across the broader ecosystem. So far, however, such migration has been carried out manually, both in our own case (Section 4) and in a recent experience report [26]. Ideally, it would instead be handled automatically rather than by manual effort from developers.

Automated code migration has been widely studied in classical software engineering [27, 28]. Prior techniques reduce the manual effort of updating client code broken by evolving libraries. More recently, techniques using Large Language Models (LLMs) have been proposed to automatically update real-world programs affected by such library evolution [25, 29, 30].

In contrast, automated code migration in quantum software remains in its infancy. A recent work has attempted to automate the migration of quantum programs written in Qiskit using LLMs [31]. However, its evaluation is limited to a single framework, a narrow range of Qiskit versions, and controlled code snippets rather than real-world quantum programs. It therefore remains unclear whether such techniques can handle migration cases arising from real-world quantum programs affected by framework evolution. Therefore we study the following future research questions:

FRQ3 *How effective are existing automated code migration techniques for quantum programs?*

FRQ4 *What are causes for the migration failures?*

To answer FRQ3, we apply existing migration techniques to the affected programs identified in FRQ2. We focus on behavior-preserving migration, that is, adapting programs so they keep running under newer framework versions without changing their original behavior. For each affected program, we evaluate whether the migrated code runs under the target framework version and whether it preserves the intended behavior.

To answer FRQ4, we qualitatively analyze the cases in which the techniques evaluated in FRQ3 fail to migrate a program or produce

a behaviorally different one, inspecting each to identify its cause. We focus on whether the cause is quantum-specific or a general limitation of existing migration techniques. If such quantum-specific causes exist, they motivate migration techniques that account for them, rather than directly applying techniques developed for classical software.

6. Conclusions

We investigated the reproducibility of Bugs4Q [4], a representative quantum software defect dataset. Our experiments covered 46,620 executions of 37 artifacts across 21 Qiskit core-library versions spanning three years. The results showed that the reproducibility of Bugs4Q dropped sharply after major Qiskit upgrades, and 83.8% of the artifacts failed to reproduce at least once. Through manual investigation of 543 reproduction failures, we found that 93.6% are dependency-related, consistent with findings on classical software defect datasets [2]. However, unlike classical datasets, only 4.6% of reproduction failures of Bugs4Q can be addressed by dependency updates alone. Guided by these findings, we curated Bugs4Q-Robust, a patched version of Bugs4Q for Qiskit v2.3.1, which raises the reproducibility by 43.3 percentage points. As a next step toward scalable maintenance, we plan to investigate whether similar compatibility issues occur in other quantum programming frameworks and explore automated migration support for affected quantum programs.

7. Data Availability

All data and code used in this study are available at the following repository: `https://github.com/kusumotolab/quantum-defect-reproducibility`

Acknowledgements This research was partially supported by JSPS KAKENHI Japan (Grant Number: JP25K15056, JP25K03102, and JP24H00692)

References

[1] R. Just, D. Jalali, and M.D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," Proceedings of the International Symposium on Software Testing and Analysis, pp.437–440, 2014.

[2] H.N. Zhu and C. Rubio-González, "On the reproducibility of software defect datasets," Proceedings of the IEEE/ACM 45th International Conference on Software Engineering, pp.2324–2335, 2023.

[3] M.A. Nielsen and I.L. Chuang, Quantum computation and quantum information, Cambridge university press, 2010.

[4] P. Zhao, Z. Miao, S. Lan, and J. Zhao, "Bugs4q: A benchmark of existing bugs to enable controlled testing and debugging studies for quantum programs," Journal of Systems and Software, vol.205, no.111805, 2023.

[5] E.M. Usandizaga, T. Laurent, P. Arcaini, and S. Ali, "Qmutbench: A dataset of quantum circuit mutants," arXiv preprint arXiv:2604.15870, 2026.

[6] Q. Chen, R. Câmara, J. Campos, A. Souto, and I. Ahmed, "The smelly eight: An empirical study on the prevalence of code smells in quantum computing," Proceedings of the IEEE/ACM 45th International Conference on Software Engineering, pp.358–370, 2023.

[7] M. Paltenghi and M. Pradel, "Bugs in quantum computing platforms: an empirical study," Proceedings of the ACM on Programming Languages, vol.6, no.OOPSLA1, pp.1–27, 2022.

[8] Y. Li, H. Pei, L. Huang, B. Yin, and K.Y. Cai, "Automatic repair of quantum programs via unitary operation," ACM Transactions on Software Engineering and Methodology, vol.33, no.6, pp.1–43, 2024.

[9] S. Tan, L. Lu, D. Xiang, T. Chu, C. Lang, J. Chen, X. Hu, and J. Yin, "Hornbro: Homotopy-like method for automated quantum program repair," Proceedings of the ACM on Software Engineering, vol.2, no.FSE, pp.734–756, 2025.

[10] N. Sato and R. Katsube, "Bug-locating method based on statistical testing for quantum programs," IEEE Transactions on Software Engineering, vol.51, no.10, pp.2804–2829, 2025.

[11] K. Kaur, D. Kim, A. Jamshidi, and L. Zhang, "Identifying flaky tests in quantum code: A machine learning approach," Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering-Companion, pp.158–165, 2025.

[12] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," Proceedings of the 33rd International Conference on Software Engineering, pp.1–10, 2011.

[13] O. Parry, G.M. Kapfhammer, M. Hilton, and P. McMinn, "A survey of flaky tests," ACM Transactions on Software Engineering and Methodology, vol.31, no.1, pp.1–74, 2021.

[14] IBM Quantum, "Qiskit 1.0 migration guide." `https://docs.quantum.ibm.com/migration-guides/qiskit-1.0`, 2024.

[15] IBM Quantum, "Qiskit v2.0 migration guide." `https://quantum.cloud.ibm.com/docs/migration-guides/qiskit-2.0`, 2025.

[16] J.L. Fleiss, "Measuring nominal scale agreement among many raters.," Psychological bulletin, vol.76, no.5, p.378, 1971.

[17] Y. Huang and M. Martonosi, "Statistical assertions for validating patterns and finding bugs in quantum programs," Proceedings of the 46th International Symposium on Computer Architecture, pp.541–553, 2019.

[18] "Amazon Braket Python SDK." `https://github.com/amazon-braket/amazon-braket-sdk-python`. Accessed: 2026-05-29.

[19] C. Developers, "Cirq." `https://zenodo.org/doi/10.5281/zenodo.4062499`, 2025.

[20] V. Bergholm, J. Izaac, M. Schuld, *et al.*, "PennyLane: Automatic differentiation of hybrid quantum-classical computations," arXiv preprint arXiv:1811.04968, 2022.

[21] K. Upadhyay, V. Chhetri, A. Siddique, and U. Farooq, "Analyzing the evolution and maintenance of quantum software repositories," Proceedings of the IEEE International Conference on Quantum Software, pp.173–184, 2025.

[22] M. Fingerhuth, T. Babej, and P. Wittek, "Open source software in quantum computing," PloS one, vol.13, no.12, 2018.

[23] L. Xavier, A. Brito, A. Hora, and M.T. Valente, "Historical and impact analysis of API breaking changes: A large-scale study," Proceedings of the IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, pp.138–147, 2017.

[24] X. Du and J. Ma, "AexPy: Detecting API breaking changes in Python packages," Proceedings of the IEEE 33rd International Symposium on Software Reliability Engineering, pp.470–481, 2022.

[25] Y. XIA, C. LIU, Z. XIE, L. ZHANG, P. LIU, K. LU, Y. LIU, W. WANG, and S. JI, "Break to adapt: Knowledge-based updates of breaking dependencies in JavaScript," 2026.

[26] J. Cardinal, I. Benzarti, C. Pere, *et al.*, "Migrating QAOA from Qiskit 1. x to 2. x: An experience report," arXiv preprint arXiv:2512.08245, 2025.

[27] B. Dagenais and M.P. Robillard, "Recommending adaptive changes for framework evolution," Proceedings of the ACM/IEEE 30th International Conference on Software Engineering, pp.481–490, 2008.

[28] B.B. Nielsen, M.T. Torp, and A. Møller, "Semantic patches for adaptation of javascript programs to evolving libraries," Proceedings of the 43rd International Conference on Software Engineering, pp.74–85, 2021.

[29] F. Reyes, M. Mahmoud, F. Bono, S. Nadi, B. Baudry, and M. Monperrus, "Byam: Fixing breaking dependency updates with Large Language Models," Empirical Software Engineering, vol.31, no.4, 2026.

[30] L. Fruntke and J. Krinke, "Automatically fixing dependency breaking changes," Proceedings of the ACM on Software Engineering, vol.2, no.FSE, pp.2146–2168, 2025.

[31] J.M. Suárez, L.M. Bibbó, J. Bogado, and A. Fernandez, "Automatic Qiskit code refactoring using large language models," arXiv preprint arXiv:2506.14535, 2025.