Quantum Testing in the Wild: A Case Study with Qiskit Algorithms

Neilson C. L. Ramalho*, Erico A. da Silva*, Higor A. de Souza†, Marcos Lordello Chaim*

*School of Arts, Sciences, and Humanities – University of São Paulo, São Paulo, SP, Brazil
†Department of Computing – São Paulo State University, Bauru, SP, Brazil
neilson@usp.br, augusto.ericosilva@usp.br, higor.amario@unesp.br, chaim@usp.br

Abstract—**Although classical computing has excelled in a wide range of applications, there remain problems that push the limits of its capabilities, especially in fields like cryptography, optimization, and materials science. Quantum computing introduces a new computational paradigm, based on principles of superposition and entanglement to explore solutions beyond the capabilities of classical computation. With the increasing interest in the field, there are challenges and opportunities for academics and practitioners in terms of software engineering practices, particularly in testing quantum programs. This paper presents an empirical study of testing patterns in quantum algorithms. We analyzed all the tests handling quantum aspects of the implementations in the Qiskit Algorithms library and identified seven distinct patterns that make use of (1) fixed seeds for algorithms based on random elements; (2) deterministic oracles; (3) precise and approximate assertions; (4) Data-Driven Testing (DDT); (5) functional testing; (6) testing for intermediate parts of the algorithms being tested; and (7) equivalence checking for quantum circuits. Our results show a prevalence of classical testing techniques to test the quantum-related elements of the library, while recent advances from the research community have yet to achieve wide adoption among practitioners.**

Index Terms—**Quantum Software Engineering, Quantum Software Testing, Test Patterns, Empirical Study**

I. INTRODUCTION

Quantum computing has attracted the attention of both industry and academia in recent years, especially due to its capabilities to solve problems previously considered unmanageable for classical computers in areas like molecular simulations, cybersecurity, finance, and logistics. The main concept behind a quantum computer is to leverage the specific properties described by quantum mechanics to perform computation [1].

The intersection between quantum mechanics and computation brings new challenges to classical software engineering practices, especially in handling concepts like superposition, entanglement, interference, and the probabilistic nature of quantum computers. While classical software engineering has built a robust theoretical and practical basis over recent decades, such practices for quantum computing are still emerging. Ensuring the correct functionality of quantum applications requires creating testing frameworks that support their development from theoretical and textbook algorithms to practical solutions addressing real-world problems. The research community has been actively adapting classical strategies and developing new testing techniques for quantum programs [2]. In parallel, practitioners have started using quantum-specific frameworks and programming languages to implement quantum programs (QPs) to solve real-world problems. These QPs are inherently hybrid, with classical elements acting as integral parts of broader solutions rather than as isolated entities.

In this paper, we analyzed how practitioners apply testing techniques to the implementations of quantum algorithms. We manually examined the tests created for Qiskit Algorithms, a library of quantum algorithms built on Qiskit, designed to run on near-term quantum devices with shallow-depth circuits [3]. We went through the code of every single test method and classified it as either classical (basic validation and classical mathematical supporting functions, for instance) or quantum (tests for quantum algorithms or their properties). With this classification, we filtered out those that were purely classical tests and focused our analysis on the tests that address quantum-related concepts. The experimental dataset with the results and scripts for file pre-processing are available at GitHub [1].

Our analysis uncovers testing patterns that employ classical software testing techniques such as black-box testing, gray-box testing with intermediate result checks, exact and approximate assertions, controlled randomness with fixed seeds, and data-driven testing. We also observe that the tests primarily run on simulators and do not account for transpilation in any of the cases.

These results suggest that practitioners have not yet adopted recent testing techniques such as those documented in recent studies [2, 4, 5, 6, 7]. Possible reasons for the lack of usage of techniques developed by the research community might include the difficulty of applying the proposed techniques in practice or lack of awareness of the quantum software developers of testing techniques.

In what follows, Section II presents Qiskit Algorithms and the testing concepts addressed in this work. The experiment setup is detailed in Section III, followed by the results in Section IV and our discussion in Section V. Section VI shows the threats to the validity. We briefly describe the related work in Section VII and conclude the paper with potential research directions in Section VIII.

II. BACKGROUND

Qiskit Algorithms is a Python library that provides quantum algorithms for use on simulators and near-term quantum

---

[1]https://github.com/saeg/saner2025/

devices with shallow circuits. It also includes key building blocks like quantum circuit gradients and state fidelities, which are commonly used in applications such as variational optimization, time evolution, and quantum machine learning [3]. It contains the categories of amplitude amplifiers, amplitude estimators, eigensolvers, gradients, minimum eigensolvers, optimizers, phase estimators, state fidelities, and time evolvers, according to the tasks they perform.

Qiskit Algorithms was the framework chosen for this study because (1) it includes tests for each algorithm, unlike most QP libraries; (2) it has an active GitHub community with 70 contributors; (3) it is built on Qiskit, one of the most widely used quantum development frameworks today; and (4) the algorithms are not limited to circuits only, but offer a basic framework that enables their application to real-world problems. As of October 2024, the Qiskit Algorithms testing suite contains 587 tests, which use different classical testing techniques such as precise and approximate assertions, functional testing, and DDT.

Precise assertions are those that compare objects with the requirement of strict matching, i.e., with no room for tolerance, rounding, or approximation. Examples of such assertions in the Python framework unittest [8] are `assertEqual(a, b)` (checks if a and b are strictly equal) and `assertTrue(x)` (checks if x is true). Approximate assertions consider a tolerance when comparing the expected result with the one returned from the module being tested, as for instance, unittest's assertion `assertAlmostEqual`. NumPy[2] assertions (such as `numpy.testing.assert_allclose`) are also used in the tests, as they can do checks on arrays.

Another important concept discussed in this work is Data-Driven Testing (DDT), which consists of having a test method being executed multiple times with varying inputs and expected outputs [9]. This strategy promotes maintainability and helps reduce errors, as the multiple executions for the different parameters can be consolidated in a single test method, thus avoiding multiple tests with different names.

These concepts are used throughout the Qiskit Algorithms code in combination with general testing techniques such as functional and gray-box testing. Functional testing, or black-box testing, is concerned with testing the program under the perspective of the requirements, looking at it as a black box without looking into its structure. The term gray-box testing is used for test generation approaches that use both source code internals as well as external descriptions of the software [10].

III. EXPERIMENTATION SETUP

The experiment consisted of analyzing the code from the Qiskit Algorithms framework to investigate which software testing techniques were applied to the test of QPs. The steps are summarized as follows.

(1) Separation of classical- and quantum-related test methods. Qiskit Algorithms contains not only a collection of quantum algorithms but also the necessary infrastructure to apply these algorithms in practical scenarios. Each algorithm is encapsulated in classes that contain constructors, getters, setters, and utility and optimization-related methods. This step consisted of removing the classical-related testing methods and analyzing the quantum-related ones to identify which testing techniques were used. The quantum testing techniques considered in our analysis are based on recent studies [2, 6, 7]. The analysis was performed manually and independently by the first and second authors of this work. The cases in which the classification for the tests differed were brought for discussion by all authors, and finally a consensus was reached. Hybrid tests (i.e., those covering both classical and quantum aspects of the code) were classified as quantum tests.

(2) Assertions analysis. Based on the list produced by the previous step, we analyzed each test method to identify which assertions were being used. To achieve this, we executed a Python script to go through every file and count the occurrences of any method call that starts with "assert", covering assertions from both unittest and `numpy.testing` packages.

IV. RESULTS

The code analysis performed in Qiskit Algorithms uncover seven recurring patterns across all the tests for quantum-specific parts of the algorithms. These patterns use (1) fixed seeds for the algorithms based on random elements; (2) deterministic oracles; (3) precise and approximate assertions; DDT; (5) functional testing; (6) gray-box testing for intermediate steps; and (7) equivalence checking for quantum circuits. These patterns can be illustrated by a few examples taken from the tests created for the Variational Quantum Eigensolver (VQE) algorithm and detailed in the next subsections. The VQE is a hybrid quantum-classical algorithm that uses a variational technique to find the minimum eigenvalue of a given Hamiltonian operator H [1].

1) Fixed Seeds

The tests for the VQE algorithm are located in the `TestVQE` class, within the `test/minimum_eigensolvers/test_vqe.py` file. Listing 1 presents the `setUp` method for this class. In line 3, a random seed is set using `self.seed = 50` to ensure reproducibility in the parts of the algorithm that contain randomness, such as the parameters initialization and the classical optimizer.

```
1  def setUp(self):
2      super().setUp()
3      self.seed = 50
4      algorithm_globals.random_seed = self.seed
5      self.h2_op = SparsePauliOp(
6          ["II", "IZ", "ZI", "ZZ", "XX"],
7          coeffs=[-1.052373245772859,0.39793742484318045,
8              -0.39793742484318045,-0.01128010425623538,
9              0.18093119978423156])
10     self.h2_energy = -1.85727503
```

Listing 1. Setup method for the test class TestVQE

Setting a fixed seed reduces flakiness in tests, ensuring consistent behavior across multiple executions [11]. The seed is not directly used in the tests but applied to the global constant `algorithm_globals.random_seed` from the

---

[2] https://numpy.org/

package `qiskit_algorithms.utils`, which is then used to build the global constant object `algorithm_globals` (instance of `QiskitAlgorithmGlobals()`). This class contains a property called `random` that is derived from the `np.random.Generator` class and is used to create different probabilistic distributions (e.g., `algorithm_globals.random.uniform` and `algorithm_globals.random.normal`). This object is used across some of the classical optimizers in the `qiskit_algorithms.optimizers` package and in quantum-related routines (e.g., the `validate_initial_point` method in the `utils` package).

2) Deterministic Oracles

From lines 5 to 9, the Hamiltonian operator `self.h2_op` for the hydrogen molecule (H_2) is defined using Pauli strings with corresponding coefficients. This operator represents the electronic structure of H_2 mapped to a two-qubit system:

- *Pauli Strings:* Each string, such as II, IZ, ZI, ZZ, and XX, specifies the Pauli operators applied to the two qubits. For example, II applies the Identity operator to both qubits, while IZ applies the Identity to the second qubit and the Pauli-Z operator to the first qubit.
- *Coefficients:* The coefficients represent the contribution of each term to the total energy, with values like -1.052 and 0.397, as shown in the Listing 1.

The variable `self.h2_energy = -1.85727503` contains the known ground-state energy of the H_2 molecule. The VQE algorithm uses `self.h2_op` to compute the minimum eigenvalue, which should be equal (given a certain tolerance) to this value (-1.85727503). By comparing the computed energy with `self.h2_energy`, one can validate that the VQE algorithm worked as expected (see line 8 in Listing 2).

```
1  @data(CG(), L_BFGS_B(), P_BFGS(), SLSQP(), TNC())
2  def test_with_gradient(self, optimizer):
3      estimator = Estimator()
4      vqe = VQE(estimator, self.ry_wavefunction, optimizer,
5          gradient=ParamShiftEstimatorGradient(estimator),
6      )
7      result = vqe.compute_minimum_eigenvalue(operator=self
        .h2_op)
8      self.assertAlmostEqual(result.eigenvalue.real, self.
        h2_energy, places=5)
```

Listing 2. Testing the VQE algorithm using gradient primitive.

3) Functional Testing

The test validates the output of the VQE algorithm (e.g., the minimum eigenvalue) without inspecting the internal parts of the quantum circuits or the optimization process. This tests the correctness of the algorithm as a whole, treating it as a "black box." In line 7 of Listing 2, the main method of the VQE algorithm is called with the operator for the H_2 molecule (`h2_op`) defined in the `setUp` method (Listing 1). The outcome of the method call is saved into the variable `result` and compared to the expected, pre-calculated value (`h2_energy` variable) in the `assertAlmostEqual` method, with a tolerance of 10^5 (places = 5).

4) Gray-box Testing

Listing 3 presents the `test_gradient_calculation` method for the `TestAdaptVQE` class. The test directly interacts with the internal gradient calculation process, checking the correct computation of commutators between operators. This low-level test is a form of gray-box testing as it verifies a specific internal method (`_compute_gradients`), providing visibility into the quantum-related operations before the full AdaptVQE algorithm is executed. The method itself is tested as a black box by comparing the computed gradient result with a manually calculated reference value. The test then checks the correctness of this intermediate result, which is needed for further iterations of the algorithm.

```
1      def test_gradient_calculation(self):
2          solver = VQE(Estimator(), QuantumCircuit(1), self
        .optimizer)
3          calc = AdaptVQE(solver)
4          calc._excitation_pool = [SparsePauliOp("X")]
5          res = calc._compute_gradients(operator=
        SparsePauliOp("Y"), theta=[])
6          # compare with manually computed reference value
7          self.assertAlmostEqual(res[0][0], 2.0)
```

Listing 3. Test gradient calculation method in AdaptVQE.

Another example of gray-box testing in Qiskit Algorithms is the test of the callback functions of the iterative, hybrid algorithms. For instance, the method `test_callback` in the `TestVQD` class (file `test/eigensolvers/test_vqd.py`) tests a callback mechanism that allows for the observation of intermediate states during optimization, which gives visibility into the system's internal state during execution and could support further metamorphic testing approaches if conditions or outputs were cross-checked.

For many of the analyzed test methods categorized as gray-box, Qiskit Algorithm's developers created complex input datasets to thoroughly exercise the important paths and conditions of the algorithms being tested. This need to understand the algorithm's internal functioning to generate appropriate test data was the decision criterion for classifying these tests as gray-box rather than black-box.

Of the 309 analyzed tests, 63 are classified as purely black-box (i.e., the test consists of calling a quantum-related routine and asserting the results against a pre-calculated value), and 80 are classified as gray-box (in which the test case and input data are prepared considering prior knowledge of the algorithm's internal structure). The remaining 166 tests are classical and do not involve quantum-related concepts.

5) Classical Assertions

Assertions such as `assertAlmostEqual` from the unittest Python framework or `np.testing.assert_allclose` from NumPy are used extensively throughout the tests for the Qiskit Algorithm framework. As exemplified in line 8 of Listing 2, the result from the `compute_minimum_eigenvalue` method might not strictly match the pre-calculated value expected as the response. However, with the `places` parameter, the developer can control the tolerance for the assertion and guarantee that deviations due to the randomness of the

quantum algorithm or possible floating point issues can be properly handled. The results with the top five most frequent assertions are summarized in Table I.

TABLE I
SUMMARY OF ASSERTION TYPES AND OCCURRENCES

Assertion Name	Occurrences	Percentage
assertEqual	135	27.95%
assertAlmostEqual	128	26.50%
assert_allclose	59	12.22%
assertIsInstance	38	7.87%
assertTrue	34	7.04%

6) Data-Driven Approach

As the VQE uses a classical optimizer, the test in Listing 1 runs once for each optimizer passed in the `@data` parameter (`CG()`, `L_BFGS_B()`, `P_BFGS()`, `SLSQP()`, `TNC()`). This shows that the test explores the hybrid behavior of the VQE algorithm, as the optimizers are classical routines. The use of DDT in this case is convenient, as the developer needs to implement a single test method that runs once for each passed parameter. This makes testing code more maintainable, for instance, when a new classical optimization routine is added to the project. Instead of creating a separate method, the developer needs to add the name of the new optimization routine in the `@data` parameter and the test will run for it as well. The DDT-related annotations (`@data` and `@idata`) are used in 133 test methods, accounting for 43% of the total 309 tests in the suite.

7) Equivalence checking for quantum circuits

There are tests in which the developer compares the unitary matrix from two circuits to determine whether they are equivalent. For instance, the `TestBernoulli.test_qae_circuit` method (`test/test_amplitude_estimators.py` file) is designed to test the correctness of the circuit generated for the Amplitude Estimation algorithm, which uses a quantum circuit that is built and optimized to low qubit usage and reduced gate depth. This test manually constructs the amplitude estimation circuit and compares the resulting unitary matrix with the unitary matrix generated by the optimized version of the circuit that is implemented in the `AmplitudeEstimation` class. The goal of the test is to verify that the optimization preserves the correctness of the circuit. This test verifies the circuit creation process for circuits with 2 to 5 qubits. Since it compares unitary matrices to determine equivalence, there will be scalability issues in case the number of qubits increases.

V. DISCUSSION

Testing quantum code is a complex task. The developer needs to have a good understanding of the algorithm being tested and be able to identify preconditions, post-conditions, intermediate results, and edge cases.

To handle the inherent complexity, our analysis of the test methods in Qiskit Algorithms shows that developers often opt for simpler, classical testing techniques when testing quantum algorithms, as summarized in Table II. The hybrid nature of these algorithms is evident from the distribution of testing techniques within the test suite. As expected, classical components still play a significant role, accounting for 54.69% of the total tests. This predominance is explained by the fact that quantum algorithms require classical infrastructure to interface with the classical world (through input and output encoding, communication with classical optimizers, and structural code such as getters, setters, constructors, helper functions, and mathematical routines). For quantum-related parts, developers typically use either black-box testing (20.39%) or gray-box testing (24.92%), where knowledge of certain internal conditions and elements of the algorithms' structure is used to design test data that exercise these specific components.

TABLE II
SUMMARY OF TESTING TECHNIQUES AND OCCURRENCES

Testing Technique	Number of Tests	Percentage
Black-box Testing	63	20.39%
Classical	169	54.69%
Gray-box Testing	77	24.92%

The use of functional testing combined with fixed seeds in Pseudo-Random Number Generators (PRNGs) and precise and approximate assertions is a predominant pattern in the test cases for the quantum-related parts of the library. The use of fixed seeds, for instance, has been listed by previous works [11] as a solution for flakiness in tests of QPs.

As for the usage of assertions, the data indicates that among all quantum-related test methods, `assertEquals` is the most frequently used assertion, accounting for nearly 30% of all assertions. This result shows that although the probabilistic nature of QP is a concern for the research community, in practice it is handled by using known answers for certain parameters when executing the quantum algorithms and fixed seeds. For the cases in which floating point precision might be an issue, the solution is to use an approximate assertion with a certain precision, which explains `assertAlmostEqual` as the second most used assertion, also accounting for almost 30% of the total number of assertions in quantum-related tests.

DDT is a classical testing technique that is extensively used in the Qiskit Algorithms test methods. Our analysis shows that the use of DDT reduces test complexity, as it helps separate test logic from data and makes it easier to manage several combinations of input parameters and expected outputs.

In terms of gray-box testing, the approaches adopted in the tests are more related to the test of intermediate routines in the algorithms. There are no tests for branches, conditions, or individual statements. These intermediate routines are then tested using functional testing with a pre-calculated expected value using precise and approximate assertions. Some quantum algorithms allow for callback functions to be passed as arguments in the iterative part of the algorithm (VQE, for instance). For these algorithms, the tests check whether intermediate results (e.g., evaluation count, parameters, mean values) can

be stored during the optimization process using a callback function. This testing approach explores the interface between the quantum and the classical parts of the algorithm.

The tests in Qiskit Algorithms with quantum-related elements are executed using either the statevector simulator or the Qasm simulator (shot-based). There are no interactions with real quantum computers, as access to quantum hardware is still limited. However, to better understand how these techniques perform in real-world scenarios, it would be valuable to run the tests on real quantum hardware. Approaches to make testing more realistic could include using recordings, similar to those employed in Azure Quantum tests [12]. In the Azure Quantum Python project, the testing infrastructure uses Python VCR [13] to record HTTP calls against a live service. These recordings (or cassettes) are used to playback the responses that work as a mock of the live service. Transpilation is another topic not covered in the analyzed test suite, as no specific transpilation setup is defined in tests that depend on circuit simulation. This can be problematic as transpilation can not only adapt the circuit to the target architecture but it also optimizes such a circuit. While optimization often reduces gate count, it can also increase circuit depth and width, potentially adding more gates and even introducing issues like the Long Circuit smell in previously clean circuits [14].

From the tests classified as having quantum features, we found no evidence of the testing techniques for QPs presented in recent works [2, 4, 5, 6, 7]. This gap needs to be explored further, as it may indicate that practitioners are either unaware of recent advancements in QP testing or that these techniques are difficult to apply in practice. Another possibility is that academic research may face practical limitations in implementation, either due to a lack of ongoing maintenance or an inability to scale effectively for real-world applications.

Mutation testing of quantum programs is well-studied in research, offering frameworks for mutating quantum circuits [15, 16, 17]. However, these techniques are currently not used to evaluate test suite quality for Qiskit Algorithms. This may occur because mutation operators focus on the circuit itself, while Qiskit Algorithm artifacts are designed to be components of larger applications where circuits are just part of the solution. For algorithms based on parameterized circuits, the circuit serves as a scaffold for parameter optimization [1]. Mutating these circuits during training may not reflect bugs that a developer would actually introduce.

VI. Threats to validity

Our study has two main threats to its validity, namely, internal and external validity, since it is largely observational and does not rely on statistical analysis.

Internal validity. The manual analysis of the testing techniques might be a process prone to errors as it depends on human judgment, which can introduce subjectivity and inconsistencies. However, the classification was initially performed by the first two authors and subsequently discussed by all authors. This process reduces the chances of mistakes. The process of looking for testing techniques consisted of analyzing each test case and classifying it as either quantum or classical.

External validity. Qiskit Algorithms may not fully represent how the broader community applies quantum testing techniques. However, it is built on Qiskit, one of the most active and widely used frameworks for quantum programming development. The repository is licensed under Apache 2.0, remains active, and implements several fundamental building blocks of quantum applications, including Quantum Phase Estimation (QPE), Quantum Phase Amplification (QPA), Grover's Algorithm, and Variational Quantum Eigensolvers (VQE).

VII. Related Work

Previous studies have analyzed quantum programs (QPs) to identify bug patterns [18], examining real-world bugs and bug fixes [19, 20], and focusing specifically on bugs within quantum machine learning applications [21]. Other approaches employ static and dynamic analysis to detect bugs in QPs [22, 23] and investigate code smells [24]. To the best of our knowledge, our work is the only study focused on analyzing patterns in testing techniques applied to real QPs.

VIII. Conclusions and Future Work

Quantum computing has become a promising field due to its potential to tackle complex problems. As interest grows in quantum programming languages and tools, it is important to develop and refine techniques for testing quantum programs.

This paper brought an overview of the main testing patterns used by practitioners to test the Qiskit Algorithms framework. We showed that, although the research community has started developing techniques to test different parts of a QP, in practice, developers continue using classical strategies to test quantum algorithms.

Our results highlight the importance of filling the gap between academia and practitioners in terms of the testing strategies for QPs. On one hand, there are multiple active research groups worldwide developing techniques to test QPs, which are not used in practice, to the best of our knowledge. On the other hand, developers might be missing opportunities to apply the state-of-the-art in terms of testing methods for QPs and improve the overall quality of the artifacts they produce.

In future work, we plan to extend this analysis to consider other programming languages such as Q#, frameworks from Qiskit-Community GitHub repository[3], and frameworks such as PennyLane[4] and Cirq[5]. The idea is to check whether the patterns observed in Qiskit Algorithms are also present in these other frameworks and languages as well as identify any new patterns that may emerge. Another possible extension of this work is identifying opportunities in the frameworks' code to apply testing techniques developed by the research community. Future investigations could explore how easily these techniques can be applied in practice and the benefits they offer compared to current classical approaches.

---

[3]https://github.com/qiskit-community
[4]https://pennylane.ai/
[5]https://quantumai.google/cirq

REFERENCES

[1] J. D. Hidary, *Quantum Computing: An Applied Approach*. Springer, 2019.

[2] N. C. L. Ramalho, H. A. de Souza, and M. L. Chaim, "Testing and debugging quantum programs: The road to 2030," 2024. [Online]. Available: https://arxiv.org/abs/2405.09178

[3] A. Javadi-Abhari, M. Treinish, K. Krsulich, C. J. Wood, J. Lishman, J. Gacon, S. Martiel, P. D. Nation, L. S. Bishop, A. W. Cross, B. R. Johnson, and J. M. Gambetta, "Quantum computing with qiskit," 2024. [Online]. Available: https://arxiv.org/abs/2405.08810

[4] A. García de la Barrera, I. García-Rodríguez de Guzmán, M. Polo, and M. Piattini, "Quantum software testing: State of the art," *Journal of Software: Evolution and Process*, vol. 35, no. 4, p. e2419, 2021.

[5] J. Zhao, "Quantum software engineering: Landscapes and horizons," 2020. [Online]. Available: https://arxiv.org/abs/2007.07047

[6] D. Fortunato, L. Jiménez-Navajas, J. Campos, and R. Abreu, "Verification and validation of quantum software," in *Quantum Software: Aspects of Theory and System Design*, I. Exman, R. Pérez-Castillo, M. Piattini, and M. Felderer, Eds. Springer, Cham, 2024, pp. 93–123.

[7] M. Paltenghi and M. Pradel, "A survey on testing and analysis of quantum software," 2024. [Online]. Available: https://arxiv.org/abs/2410.00650

[8] Python Software Foundation, *unittest – Unit testing framework*, 2024, accessed: 2024-10-24. [Online]. Available: https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertAlmostEqual

[9] P. Ammann and J. Offutt, *Introduction to Software Testing*, 2nd ed. Cambridge University Press, 2016.

[10] M. E. Khan and F. Khan, "A comparative study of white box, black box and grey box testing techniques," *International Journal of Advanced Computer Science and Applications*, vol. 3, no. 6, 2012.

[11] L. Zhang, M. Radnejad, and A. Miranskyy, "Identifying flakiness in quantum programs," in *Proceedings of the 2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM 2023. Los Alamitos, CA, USA: IEEE Computer Society, 2023, pp. 1–7. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ESEM56168.2023.10304850

[12] Microsoft, "Unit tests for azure quantum python sdk," https://github.com/microsoft/azure-quantum-python/blob/main/azure-quantum/tests/README.md, 2024, accessed: 2024-10-21.

[13] Kevin Chan, Dustin Lacewell, and contributors, *VCR.py: Automatically Mock Your HTTP Interactions to Simplify and Speed Up Testing*, VCR.py Community, 2023, version 5.0.0. [Online]. Available: https://vcrpy.readthedocs.io/en/latest/

[14] M. D. Stefano, D. D. Nucci, F. Palomba *et al.*, "An empirical study into the effects of transpilation on quantum circuit smells," *Empirical Software Engineering*, vol. 29, no. 61, 2024. [Online]. Available: https://doi.org/10.1007/s10664-024-10461-9

[15] E. Mendiluze, S. Ali, P. Arcaini, and T. Yue, "Muskit: A mutation analysis tool for quantum software testing," in *Proceedings of the 2021 36th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE'21, 2021, pp. 1266–1270.

[16] D. Fortunato, J. Campos, and R. Abreu, "Qmutpy: A mutation testing tool for quantum algorithms and applications in qiskit," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022. New York, NY, USA: ACM, 2022, pp. 797—800. [Online]. Available: https://doi.org/10.1145/3533767.3543296

[17] S. G. Gil, L. L. Díaz, and J. I. R. Jarabo, "QCRMut: Quantum circuit random mutant generator tool," 2024. [Online]. Available: https://arxiv.org/abs/2410.01415

[18] P. Zhao, J. Zhao, and L. Ma, "Identifying bug patterns in quantum programs," in *Proceedings of the 2021 IEEE/ACM 2nd International Workshop on Quantum Software Engineering*, ser. Q-SE 2021, 2021, pp. 16–21.

[19] J. Luo, P. Zhao, Z. Miao, S. Lan, and J. Zhao, "A comprehensive study of bug fixes in quantum programs," in *Proceedings of the 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering*, ser. SANER 2022, 2022, pp. 1239–1246.

[20] M. Paltenghi and M. Pradel, "Bugs in quantum computing platforms: An empirical study," *Proceedings of the ACM on Programming Languages*, vol. 6, no. OOPSLA1, apr 2022. [Online]. Available: https://doi.org/10.1145/3527330

[21] P. Zhao, X. Wu, J. Luo, Z. Li, and J. Zhao, "An empirical study of bugs in quantum machine learning frameworks," in *Proceedings of the 2023 IEEE International Conference on Quantum Software*, ser. QSW 2023, 2023, pp. 68–75.

[22] P. Zhao, X. Wu, Z. Li, and J. Zhao, "Qchecker: Detecting bugs in quantum programs via static analysis," in *Proceedings of the 2023 IEEE/ACM 4th International Workshop on Quantum Software Engineering*, ser. Q-SE 2023, 2023, pp. 50–57.

[23] M. Paltenghi and M. Pradel, "Analyzing quantum programs with lintq: A static analysis framework for qiskit," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 2144–2166, 2024. [Online]. Available: https://doi.org/10.1145/3660802

[24] 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," in *Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering*, ser. ICSE'23, 2023, pp. 358–370.