Mitigating Noise in Quantum Software Testing Using Machine Learning

Asmar Muqeet, Tao Yue, Shaukat Ali, *Senior Member, IEEE*, and Paolo Arcaini, *Member, IEEE*

***Abstract*—Quantum Computing (QC) promises computational speedup over classic computing. However, noise exists in near-term quantum computers. Quantum software testing (for gaining confidence in quantum software's correctness) is inevitably impacted by noise, i.e., it is impossible to know if a test case failed due to noise or real faults. Existing testing techniques test quantum programs without considering noise, i.e., by executing tests on ideal quantum computer simulators. Consequently, they are not directly applicable to test quantum software on real quantum computers or noisy simulators. Thus, we propose a noise-aware approach (named $QOIN$) to alleviate the noise effect on test results of quantum programs. $QOIN$ employs machine learning techniques (e.g., transfer learning) to learn the noise effect of a quantum computer and filter it from a program's outputs. Such filtered outputs are then used as the input to perform test case assessments (determining the passing or failing of a test case execution against a test oracle). We evaluated $QOIN$ on IBM's 23 noise models, Google's two available noise models, and Rigetti's Quantum Virtual Machine, with six real-world and 800 artificial programs. We also generated faulty versions of these programs to check if a failing test case execution can be determined under noise. Results show that $QOIN$ can reduce the noise effect by more than 80% on most noise models. We used an existing test oracle to evaluate $QOIN$'s effectiveness in quantum software testing. The results showed that $QOIN$ attained scores of 99%, 75%, and 86% for precision, recall, and F1-score, respectively, for the test oracle across six real-world programs. For artificial programs, $QOIN$ achieved scores of 93%, 79%, and 86% for precision, recall, and F1-score respectively. This highlights $QOIN$'s effectiveness in learning noise patterns for noise-aware quantum software testing.**

Received 11 December 2023; revised 30 August 2024; accepted 13 September 2024. Date of publication 18 September 2024; date of current version 14 November 2024. This work was supported by the Qu-Test Project (Project #299827) funded by the Research Council of Norway. The work of Shaukat Ali was also supported by Oslo Metropolitan University's Quantum Hub and Simula's internal strategic project on quantum software engineering. The work of Paolo Arcaini was supported in part by ERATO HASUO Metamathematics for Systems Design Project under Grant JPMJER1603, JST, Funding Reference: 10.13039/501100009024 ERATO; and in part by Engineerable AI Techniques for Practical Applications of High-Quality Machine Learning-based Systems Project under Grant JPMJMI20B8, JST-Mirai. Recommended for acceptance by M. Pradel. *(Corresponding author: Asmar Muqeet.)*

Asmar Muqeet is with the Simula Research Laboratory, Oslo, 0164 Norway, and also with the University of Oslo, 0313 Oslo, Norway (e-mail: asmar@simula.no).

Tao Yue is with the Simula Research Laboratory, 0164 Oslo, Norway (e-mail: taoyue@gmail.com).

Shaukat Ali is with the Simula Research Laboratory, 0164 Oslo, Norway, and also with Oslo Metropolitan University, 0130 Oslo, Norway (e-mail: shaukat@simula.no).

Paolo Arcaini is with the National Institute of Informatics, Tokyo, 101-8430 Japan (e-mail: arcaini@nii.ac.jp).

Digital Object Identifier 10.1109/TSE.2024.3462974

***Index Terms*—Software testing and debugging, computing methodologies, quantum computing, and machine learning.**

I. Introduction

RECENTLY there has been an increased interest in software engineering techniques for building quantum computing (QC) applications [1]. In addition, quantum computers and their respective simulators (IBM, Google, Rigetti) [2] are becoming available. However, quantum computers face hardware noise due to immature hardware and environmental factors (e.g., magnetic fields, radiations) [3]. Noise affects the accuracy of calculations a quantum computer performs, thus resulting in incorrect program outputs.

Quantum software testing aims to cost-effectively find quantum software bugs to achieve a certain level of confidence in their correctness [1]. Testing quantum software is challenging due to the inherent quantum mechanics' features, such as superposition and entanglement [4]. Further, noise brings another layer of complexity to the challenge. For example, when checking test results, it becomes difficult to conclude whether a test case failed due to a fault or noise. Existing quantum software testing approaches mainly focus on adopting classic software testing techniques, such as combinatorial testing [5], coverage criteria [6], [7], [8], search-based testing [9], [10], mutation testing [10], [11], [12], metamorphic testing [13], [14], property-based testing [15], and fuzzing [16]. However, these works perform testing on ideal (i.e., noise-free) quantum computer simulators, so test results cannot be trusted when testing on quantum computers with noise or noisy simulators.

Testing quantum programs with simulators is limited by their high computational cost [17]. Thus, we need proper ways to filter the noise to support the testing of programs directly on real computers. However, this is not easy, since each quantum computer exhibits a different noise effect on the outputs of the same program [18], and so the program's outputs (due to noise) vary on each computer.

In short, quantum noise prevents us from applying existing quantum software testing methods on quantum computers. Thus, we propose an approach to make *Quantum sOftware testIng Noise-aware* ($QOIN$). $QOIN$ trains a machine learning model to learn general noise patterns of a quantum computer. Next, since the noise effect on the program output is also specific to each program, $QOIN$ uses a transfer learning method to learn program-specific noise to predict the correct output of a program from a noisy output. The predicted output

can then be assessed with a given test oracle. It is important to note that quantum noise cannot be completely eliminated from the program output without having a non-noisy quantum computer. However, its effects can be mitigated by utilizing machine learning.

Our key contributions are: 1) Applying machine learning (ML) to the challenging QC domain to address the issue of testing quantum programs on noisy quantum computers. 2) An ML-based approach to reduce noise's effect on the test results of a quantum program, where classical noise reduction techniques are ineffective. 3) Empirical evaluation with IBM's 23 quantum computer noise models, Google's two noise models, and Rigetti's Quantum virtual machine (QVM), with six open-source real-world quantum programs and 800 diverse quantum programs generated with Qiskit; and 4) Empirical evaluation of $QOIN$ with a published quantum test oracle to show its effectiveness in aiding test assessment on noisy quantum computers and simulators. 5) An empirical evaluation of $QOIN$, compared with a classical non-ML noise reduction method, demonstrating the effectiveness of the ML-based approach to noise reduction.

Our results showed that $QOIN$ can effectively learn and reduce the noise effects more than 80% on most backends. We used an existing quantum software test oracle [5], [9] with $QOIN$. The results show that $QOIN$ attained scores of 99%, 75%, and 85% for precision, recall, and F1-score, respectively across six real-world programs. For 800 diverse programs, $QOIN$ achieved scores of 93%, 79%, and 85% for precision, recall, and F1-score, respectively.

Open science. Our implementation and all experimental results are being made freely available [19].

II. Background

A. QC Basics and Example

Classic computing uses bits that can only be in 0 or 1 state. QC's instead use quantum bit or *qubit*, which can be in a *superposition* of $|0\rangle$ and $|1\rangle$ with associated amplitudes (α). Amplitude α is a complex number having a *magnitude* and a *phase* in its polar form. We represent a qubit in the Dirac notation [20] as: $|\psi\rangle = \alpha_0|0\rangle + \alpha_1|1\rangle$, where α_0 and α_1 are the amplitudes associated with states $|0\rangle$ and $|1\rangle$, respectively. The probabilities of a qubit being in $|0\rangle$ or $|1\rangle$ are given by the square of the magnitude of α_0 and α_1, with the sum of all squared magnitudes being equal to 1: $|\alpha_0|^2 + |\alpha_1|^2 = 1$. For example, the probability of being in state $|0\rangle$ is $|\alpha_0|^2$, and the probability of being in state $|1\rangle$ is $|\alpha_1|^2$.

Quantum gates (unitary operators) manipulate qubits, i.e., changes a qubit's state based on a unitary matrix [21]. For example, the *Hadamard* gate puts a qubit into superposition. Currently, we program gate-based quantum computers as quantum circuits in which the logic is implemented as a sequence of quantum gates applied on qubits. For improved readability, we will use the term 'circuit' to refer to a quantum program throughout the paper.

Fig. 1 shows a GHZ (Greenberger-Horne-Zeilinger) state, defined with a three-qubit *entanglement* circuit in Qiskit [22].

The circuit puts three qubits in *entanglement*, i.e., when read, they have the same value ($|0\rangle$ or $|1\rangle$). In lines 1-7, we initialize

```
1. qc = QuantumCircuit()
2. qubit_1 = QuantumRegister(1,'qubit_1')
3. qubit_2 = QuantumRegister(1,'qubit_2')
4. qubit_3 = QuantumRegister(1,'qubit_3')
5. qubit_1c = ClassicalRegister(1, 'qubit_1c')
6. qubit_2c = ClassicalRegister(1, 'qubit_2c')
7. qubit_3c = ClassicalRegister(1, 'qubit_3c')
8. qc.h(qubit_1)
9. qc.cx(qubit_1,qubit_2)
10. qc.cx(qubit_2,qubit_3)
11. qc.measure(qubit_1,qubit_1c)
12. qc.measure(qubit_2,qubit_2c)
13. qc.measure(qubit_3,qubit_3c)
```

Fig. 1. A three-qubit GHZ state quantum circuit written in Qiskit.

TABLE I
THE IDEAL AND NOISY OUTPUTS OF A THREE-QUBIT GHZ CIRCUIT (SEE FIG. 1) AFTER EXECUTING 1024 TIMES ON AN IDEAL AND A NOISY SIMULATOR. COLUMN *PROBABILITY* SHOWS THE PROBABILITY OF HAVING A SPECIFIC OUTPUT

	Probability							
State	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
Ideal	0.5	-	-	-	-	-	-	0.5
Noisy	0.476	0.013	0.007	0.016	0.008	0.019	0.020	0.443

the circuit with three qubits (*qubit_1*, *qubit_2*, *qubit_3*), and three classical registers (*qubit_1c*, *qubit_2c*, *qubit_3c*) for storing the qubits' final states. Line 8 applies a *Hadamard* gate on *qubit_1* to put it in superposition of $|0\rangle$ and $|1\rangle$. At line 9, *qubit_1* and *qubit_2* are entangled via the controlled-not gate (*cx*). Since *qubit_1* is already in superposition, after *cx*, the circuit is in a superposition of $|00\rangle$ and $|11\rangle$. At line 10, the *qubit_2* and *qubit_3* are entangled via another *cx*. Then, the circuit has all the qubits entangled: reading them will result in either $|000\rangle$ or $|111\rangle$ with 50% probability. Lines 11-13 apply three *measure* operations to read the qubits and store results in classic registers *qubit_1c* to *qubit_3c*.

B. Quantum Noise

Noise originates from various sources. First, environmental characteristics (e.g., magnetic fields, radiations) affect computations [3]. Qubits interactions with environments can cause disturbances and information loss in quantum states, i.e., *decoherence* [4]. Second, unwanted qubits' interactions among themselves, even when perfectly isolated from the environment (*crosstalk noise* [23]), lead to unwanted quantum states, thus affecting the computations. Third, noise is caused by imprecise quantum gate calibrations, which are required to optimize gate parameters to reduce errors and improve their fidelity. Minor errors in such calibration often result in minor phase changes, amplitude, etc., in qubits, which may not directly destroy a quantum state but could lead to undesired states after a sequence of gate operations [23]. Note that any qubit in a circuit can be affected by noise at any stage, resulting in an accumulated noise effect on the circuit's output.

We show an example in Fig. 1. The ideal circuit output is to give almost an equal probability of having state $|000\rangle$ or $|111\rangle$ after 1024 executions (see Table I). However, when running on a quantum computer, the circuit has eight output

states of various probabilities (see Table I). Thus, due to noise, a circuit can produce wrong output states or correct output states with wrong probabilities. Furthermore, each quantum computer exhibits a unique noise effect [18]; thus, outputs of the same circuit on different computers can be different. Moreover, noise in real computers is also specific to each circuit. This requires any noise learning strategy to be computer and circuit-specific, making it harder to generalize across computers and circuits.

A quantum computer's ***noise model*** is a mathematical model that encapsulates the error probabilities of all qubits. It is characterized by parameters, e.g., qubit bit-flip errors, relaxation times (T1), dephasing times (T2), and qubit cross-talk probabilities. In short, the noise model is a general probabilistic representation of potential errors that may occur when performing computations. However, this model does not encapsulate information about specific error instances in a given circuit and is unsuitable for implementing fine-grained noise filtering mechanisms for individual circuits. In our case, we define ***noise pattern*** to refer to the specific behavior of quantum noise for a particular circuit and a backend. It describes how noise manifests based on the combination of qubits and the gate operations performed on a given circuit.

III. Related Work

Noise in quantum computers. A study on quantum noise [18] concluded it is distinguishable across quantum computers with ML since each computer has its own learnable noise footprint. Specifically, a circuit is executed on several quantum computers, followed by using the execution data to train an ML model to classify which execution belongs to which computer. The model had an accuracy of 99%, showing that each quantum computer has a unique noise fingerprint. In contrast to [18], we use ML to minimize the noise effect on circuit outputs. Another study [24] compares executions of a circuit across different computers by providing a holistic picture of the circuit's fidelity on different computers to identify quantum computers with minimal noise.

Quantum software testing. Several studies have been done in quantum software testing. Various strategies such as statistical assertions [25] and projection-based assertions [26] have been proposed to verify the correctness of circuits. Different coverage criteria, including input-output coverage [6], [27] and equivalence class partitioning [7], have been introduced to assess test case quality. Tools like Muskit [12] and QMutpy [28] facilitate mutation testing, evaluating the effectiveness of test suites. Some studies have focused on defining test oracles for circuits, leveraging techniques like statistical tests [6] and property-based testing [15]. Advanced testing methods, such as search-based approaches [9], [10], combinatorial testing [5], and fuzzing [16], have also been applied to circuits. Moreover, bug identification and classification frameworks [29], [30], [31] have been proposed, with Bugs4Q [32] serving as a benchmark for real-world bugs in Qiskit-based circuits, accompanied by test cases aimed at capturing similar issues.

To ensure dependable support for quantum software platforms (QSP) such as Qiskit, approaches like QDiff [33] and MorphQ [14] have been proposed. Testing QSP focuses on checking the platform's implementation, while circuits serve as benchmarks for various QSPs. Among QSP testing works, only QDiff [33] considers quantum noise. QDiff tests QSP by randomly generating logically equivalent circuits. It uses static measures, e.g., gate error rates and T1 relaxation time, to identify and exclude circuits that might be significantly impacted by noise from its test suite. The generated circuits serve as tools to test QSP. Note that QDiff does not filter noise from the circuit output as we do in our work.

Classical noise reduction. Noise also manifests in classical domains like IoT and cyber-physical systems [34]. For these domains, non-ML noise reduction methods are employed, such as noise-free signal processing, adaptive noise filtering, and Bayesian inference, e.g., the Kalman filter and the Extended Kalman filter [35]. However, applying these methods to QC faces challenges due to the principles of no-cloning and state collapse in quantum mechanics [4]. In contrast, classical information can be copied precisely; hence, classical methods rely on error information and redundancy to detect errors caused by noise [36]. Some classical methods, such as error correction codes derived from information theory, have applications in QC [37]. However, quantum noise disrupts quantum states, in ways that fundamentally differ from how classical noise affects classical states. In classical systems, noise typically introduces random fluctuations that can be described using well-understood probability distributions like Gaussian or Poisson distributions [36]. In contrast, quantum noise affects quantum states by causing decoherence, which leads to the loss of quantum properties (e.g., entanglement and superposition) and leads to complex changes in quantum states that cannot be easily described by Gaussian or Poisson distributions.

Another challenge in using non-ML classical noise reduction methods for quantum noise is quantum systems' nonlinear and probabilistic nature. Classical noise reduction techniques, usually assume noise can be approximated as a linear process with a real-valued distribution, such as Gaussian. However, quantum systems operate according to the Schrödinger equation [4], which involves state transitions based on complex-valued distributions and is inherently non-linear and probabilistic. To address this in classical noise reduction approaches, new quantum-specific methods are needed to handle complex-valued distributions effectively [38]. On the other hand, ML-based noise reduction methods do not require explicit noise modeling with real-valued distributions and can learn noise representations from execution data. Thus, ML-based noise reduction methods can be applied in both classical and quantum contexts, whereas effective quantum noise handling by non-ML methods requires quantum-specific techniques.

In classical systems, ML-based noise reduction techniques utilize a range of methods, from neural network training to reinforcement learning [34]. However, the key challenge when applying classical ML methods to quantum systems lies in data representation. Specifically to identify the features that can effectively help ML models reduce noise in quantum circuits. Unlike classical data, a quantum circuit cannot be directly fed

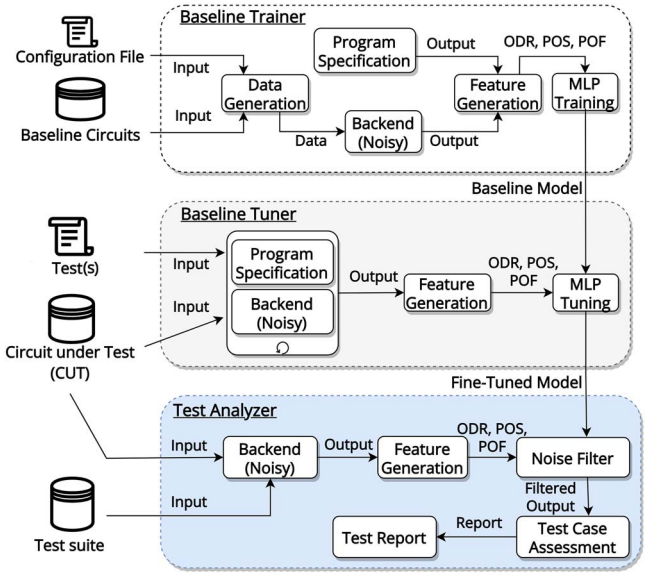

Fig. 2. *QOIN* (ODR: odds ratio for each output; POS: probability of success for each output; POF: probability of failure for each output).

into ML models, as these models require a descriptive data representation to extract relevant features during training [39]. To address this, we introduce three descriptive features derived from quantum circuit outputs, which enable classical ML noise reduction methods to be adapted for quantum systems.

IV. Approach

As shown in Fig. 2, *QOIN* has three modules: *Baseline Trainer*, *Baseline Tuner*, and *Test Analyzer*, detailed in Sects. IV-A–IV-C. For automated testing, a test oracle is typically required for test assessment, usually done by comparing an observed test case output against a *Program Specification*. For training, *QOIN* also relies on the program specification to guide ML for learning the noise pattern of a quantum computer. *QOIN* also uses a *Noisy Backend*, which is either a real computer or a simulator with a noise model of a real computer for noisy circuit execution.

A. Baseline Trainer

This module learns a general noise pattern of a given noisy backend, i.e., one dedicated ML model per noisy backend. *Baseline Trainer* has three components: *Data Generation*, *Feature Generation*, and *MLP Training*, described below.

1) Data Generation: This component takes the *Configuration File* and *Baseline Circuits* as input and generates the input data for circuits to execute on the noisy backend. *Baseline Circuits* are circuits required to produce training and testing data for the ML model. *Configuration File* specifies the input parameters of each circuit, which are used for guiding the generation and controlling the input data required for circuit executions. A circuit takes input parameters in three formats: integer, binary, and string expression. In the configuration file,

```
ID: 3
FORMAT: expression
START: 3
END: 6
PERCENTAGE: 0.2
REGEX: \([a,b,c][\&, \+, \!][a,b,c]\)[\&, \+, \!][a,b,c]
```

Fig. 3. A snippet of *configuration file*. *ID* identifies a given circuit in *baseline circuits*; *FORMAT* specifies the input data format for the circuit. For integers, *START* and *END* define the minimum and maximum values. For binary, *START* and *END* is the bit range. In expression format, only the *START* value matters. *PERCENTAGE* defines the percentage of input space explored during data generation. *REGEX* specifies the required regular expression for generating input data.

one can specify each parameter's range (e.g., the minimum and maximum of an integer). One can also specify the percentage of input space to be explored since executing a circuit with all possible inputs can be computationally expensive. For instance, as shown in the example in Fig. 1, before superposition, the input to the circuit will be binary, i.e., a three-bit string ranging from 000 to 111. In the configuration file, we can specify the percentage of these inputs we want to cover (e.g., 50%). We can also define a possible minimum and maximum value (e.g., from 001 to 101), where the generated input will start from 001 and end at 101, depending on the percentage value. Fig. 3 shows a snippet of a *Configuration File* for a circuit.

Data Generation generates input data for executing a circuit on the noisy backend (see Fig. 2). The program specification is used as ground truth for ML training. The outputs from the noisy backend and the program specification are given to the *Feature Generation* component as input.

2) Feature Generation: A circuit cannot be directly used as input for ML training because ML models require a descriptive representation of data to extract features during training [39]. Thus, the *Feature Generation* component generates three descriptive features (i.e., *Odds Ratio*, *Probability of Success*, and *Probability of Failure*) for each circuit in *Baseline Circuits*. These features have been used to solve several ML problems [40], [41]. All these features are calculated using the outputs from the program specification and the noisy backend. With these three features, we map the noise learning problem as a supervised regression problem, which can be addressed with a trained ML model. After the feature calculation, the three features are used as input for *MLP Training*. There can be multiple states as the outcome of an execution of a circuit. For example, the GHZ example (Fig. 1) has two output states: 000 and 111. We call them *target states*. Each feature is calculated for each *target state*.

Probability of Success (POS) of a target state t is the probability of the state being observed after the execution of a circuit on a noisy backend, which is calculated by dividing the frequency of t by the sum of frequencies of all output states: $POS_t = \frac{frequency_t}{\sum_{i=1}^{n} frequency_i}$.

Odds Ratio (ODR) defines the odds of one event in the presence of another one [42]. Odds ratio is calculated as the probability of one target state divided by the sum of the probabilities of the others: $ODR_t = \frac{POS_t}{1-POS_t}$, where POS_t defines the probability of a target state t observed after the circuit

execution on the noisy backend, and $1 - POS_t$ is the sum of the probabilities of all the other observed states.

Probability of Failure (POF) of a target state t is the likelihood of no occurrence, i.e., the complement of POS: POF_t = 1 - POS_t. ODR and POF are derived from POS. Using these three features allows to capture two different scenarios in which noise can affect the program output.

In the first scenario, where the circuit produces correct output states but with incorrect probabilities under noise (see Sect. II-B), POS and POF can help the ML model capture the differences between the ideal probabilities and the probabilities under noise. For example, consider the circuit output in Table I. Assume the target state is 000 with an ideal probability of 0.5 and an observed noisy probability (POS) of 0.47. If the ideal value is known, correcting the observed probability involves taking the difference between the ideal and observed probabilities, in this case, $(0.5 - 0.47)$, resulting in 0.03 as the correcting factor. However, without ground truth, the correcting factor cannot be directly calculated with only POS. Instead, POF can be used as an approximate ground truth. As POF represents the cumulative sum of all other observed probabilities, it can be utilized to determine the correcting factor. In the example for the 000 target state, where POS is 0.47 and POF is 0.53 (essentially ideal probability plus some error, i.e., $0.5 + error$), the correcting factor can be calculated as (POF − POS), which is $(ideal + error - POS)$. Substituting the values $(0.5 + error - 0.47)$ yields $0.03 + error$. This $error$ term varies for different circuit outputs depending on noise pattern and can be learned by ML models with sufficient data. Therefore, the first probability correction scenario can be handled with POS and POF for each target state. Note that explicitly including POF has a benefit, since it simplifies the task for the ML model to learn the correction term needed to mitigate noise. Moreover, using both POS and POF, the model is less likely to become overconfident in its predictions since these features help it balance the information. For example, suppose the model predicts a high POS for a certain outcome, but the POF is also high; this inconsistency indicates that the prediction exceeds the total sum, which should be one. This prompts the model to be more cautious and adjust its confidence level accordingly.

In the second scenario, where a circuit can produce incorrect output states (see Sect. II-B), ODR can be used as an input feature for ML to learn to distinguish noise-induced states from other states. ODR may exhibit different behavior for noise-induced states compared to other states. Consider the circuit output in Table I. For a target state specified in the program specification, e.g., 000, the ODR value is 0.9 ($\frac{0.47}{0.53}$), which is closer to 1, whereas, for all noise-induced states such as 001, 010, etc., the ODR value is very close to the POS value. For example, for 001, the ODR is 0.0137, whereas POS for 001 is 0.013. This is an example of the behavior of ODR, which holds for circuits with numerous noise-induced states but lower probabilities, as shown in Table I. However, depending on the ideal output distribution of a circuit and the effect of noise, ODR might not exhibit the same behavior for other circuits. This different behavior of ODR for noise-induced states for different circuits can help the ML model distinguish between noise-induced states and other states.

3) MLP Training: We chose a commonly used, fully connected, multi-layer neural network called Multilayer Perceptron (MLP) [39] to train a neural network. We split the dataset into training and test sets with a ratio of 80 to 20, following a common practice [43], [44]. We use the Ktrain library [45] for automatic neural network architecture and hyperparameter selection. The library provides standard functions that use different algorithms, such as the cyclical learning rate policy in [46], to simulate neural network training for several iterations to find optimal parameters. We selected Mean Absolute Error (MAE) as the loss function for neural network training [47], commonly used for training regression models.

B. Baseline Tuner

Noise also depends on the arrangement of qubits in a circuit and gate operations performed with specific inputs (named *circuit-specific noise*) in addition to each specific computer. For instance, circuits usually have a sequence of conditional gates, such as one qubit controlled-Not and multi-qubit controlled-Not, each of which acts on a target qubit depending on one or more control qubits. Executing a circuit with conditional gates leads to different circuit paths for each input. Consider a circuit with three qubits, initialized with the inputs 000 and 010. Any conditional gate that has qubit number 2 as the control qubit is only executed for the input 010. This creates two distinct execution paths within the same circuit. The cumulative impact of noise varies depending on the number of quantum gates that are active for each input. Thus, each input has a different noise effect depending upon whether the conditional gates make different execution paths in a circuit for each input. As a result, for a given noisy backend, it is not guaranteed that the baseline MLP model learned for the backend will still be reasonably accurate when applied for all possible inputs to be executed on the same backend. To overcome this challenge, one potential solution is to train the ML model directly on circuit-specific data. However, this approach requires collecting significant circuit-specific data for each backend. Since simulating a single input can be resource-intensive, generating large datasets for circuits becomes challenging [17]. Thus, we propose *Baseline Tuner* to fine-tune the baseline MLP model trained in the *Baseline Trainer*.

Specifically, *Baseline Tuner* incorporates circuit-specific noise into the baseline MLP model for each noisy backend. Using a pre-trained ML model that has learned the noise pattern of a noisy backend as the initial point for fine-tuning, a circuit-specific ML model would require substantially less data than training an ML model from scratch. This fine-tuning approach, known as transfer learning, is commonly employed in ML to mitigate the data requirements and reduce costs [48]. Fig. 2 shows a Circuit Under Test (CUT), and a set of predefined non-failing tests of the CUT are used as the input to *Baseline Tuner*. The predefined tests refer to non-failing tests used as sanity checks during program implementation. *Baseline Tuner* executes the CUT on the noisy backend with these tests to

execute the CUT on different combinations of circuit paths so that the baseline MLP model can learn the input-specific and circuit-specific noise effects on the CUT. The *Feature Generation* component in the *Baseline Tuner* uses the same three features (POS, ODR, POF) as the *Baseline Trainer*. It utilizes outputs from the noisy backend and the program specification to calculate the feature dataset for the *MLP Tuning* component (see Sect. IV-A2). The key difference between the *Feature Generation* components in *Baseline Trainer* and *Baseline Tuner* lies in the data that the features represent. In *Baseline Trainer*, features are computed for all *target states* across all *Baseline Circuits*. Instead, in *Baseline Tuner*, features are computed only for the *target states* of a single CUT. Each predefined test is executed multiple times (e.g., 100 times), and the three features are calculated for each target state. This repeated execution for each test helps capture the variations in probability distribution caused by inherent QC uncertainties and the effects of noise.

The *MLP Tuning* component uses transfer learning on the baseline model from *Baseline Trainer*. The use of a pre-trained network (i.e., *Baseline Model*), allows *Fine-Tuned Model* to learn noise patterns with fewer data. The hyper-parameters for the transfer learning were set automatically according to the specifications in [45].

C. Test Analyzer

This component adds a filter layer between the circuit output and the input to the test assertion module of a given quantum software testing strategy to enable the strategy to work on a noisy backend. Namely, *Test Analyzer* takes the *CUT* and a *Test suite* as input. *Test suite* is a set of test cases that has been generated by a testing method (e.g., QuCAT [5], QuSBT [9]). The CUT and *Test suite* are executed on the noisy backend. *Feature Generation* of *Test Analyzer* processes the output from the noisy backend and calculates the three features (see Sect. IV-A2) required by *Noise Filter*. *Noise Filter* uses the three calculated features and the *Fine-Tuned Model* from *Baseline Tuner* to filter the noise out from the CUT output. The filtered output is used by the test assertion module of a testing method to determine the passing or failing of a test case against a test specification.

$QOIN$ does not have its own test assessment module. Thus, to demonstrate its effectiveness, we have integrated $QOIN$ with test assessment modules from two previously published works [5], [9]. The integrated module provides two distinct test oracles: **Unexpected Output Failure (UOF):** This oracle examines the circuit's output to identify any unexpected output states not included in the program specification. **Wrong Output Distribution Failure (WODF):** This oracle evaluates the circuit's output distribution and compares it to the program specification using statistical tests. By incorporating these oracles, $QOIN$ can assess the correctness of circuits and detect discrepancies between circuit outputs and their intended specifications.

TABLE II
CHARACTERISTICS OF THE REAL-WORLD BENCHMARKS. CIRCUITS WITH n IN COLUMN *#QUBITS* DENOTE THAT, THEORETICALLY, THEY CAN BE PROGRAMMED WITH ANY NUMBER OF QUBITS GIVEN ENOUGH HARDWARE RESOURCES

Real-World Benchmarks	#Qubits	#Gates	Depth
Addition	7∼n	11	17
Simon	6∼n	6	14
Greenberger–Horne–Zeilinger state (GHZ)	3∼n	5	7
Binary Similarity	7	6	9
N-CNOT	7	6	9
Permutations	7	3	12
Phase Estimation	4	15	21
Quantum Fourier Transform (QFT)	5	4	10
Expression Evaluation	3	3	7

V. EXPERIMENT DESIGN

We aim to answer two research questions: compactitem

RQ1 How effective is $QOIN$ in reducing the noise effect on circuits' outputs?

RQ2 How can $QOIN$ help testing methods improve their test assessment?

Below, we present benchmarks in Sect. V-A, the experiment settings in Sect. V-B, and evaluation metrics in Sect. V-C.

A. Benchmarks

Real-world Benchmarks. We selected nine circuits from two repositories [49], [50] based on these criteria: 1) written in Qiskit and publicly available, 2) input qubits greater or equal to three to ensure that the *Data Generation* component has sufficient input data, and 3) executable on noisy backends from IBM, Google and, Rigetti. Table II shows the characteristics of these benchmarks.

Artificial Benchmarks. We generated 1000 diverse circuits with Qiskit's random circuit generator [22]. Given the same input, two circuits are considered diverse if their outputs differ when measured with Jensen Shannon Distance (JSD), which has been used in the literature to calculate circuit diversity based on circuit outputs [51]. It ranges from 0 to 1, where 0 means the outputs of two circuits are exactly the same, and 1 means completely different. We calculated the average pairwise JSD value for each circuit as its diversity score. Most benchmarks have an average diversity score of more than 0.5, meaning that on average, each generated circuit differs more than 50% from all other circuits. We also created three faulty circuit versions in both benchmarks using conditional quantum gates such as *cx* and *ccx*. Conditional gates can be configured to activate only on a specific test, which is convenient for seeding faults in circuits in a controllable manner. To assess the effectiveness of $QOIN$, we use the original and the faulty versions of the circuits to answer the RQs.

B. Experiment Settings

We use the Qiskit, Cirq, and pyQuil frameworks for circuit execution [2]. To automatically obtain the program specifications for the benchmarks, we use Qiskit's AER, Cirq's Qsim

simulator, and Rigetti's QVM without any noise model to obtain the correct circuit outputs for *MLP Training*. For noisy circuit execution, we use noise models provided by IBM, Google, and Rigetti. For IBM, we use Qiskit's AER simulator integrated with IBM-provided noise models as a noisy backend. Qiskit provides 47 noise models for IBM quantum computers. Out of these 47, we selected 23 based on the criterion that all available noise models must have at least seven qubits for execution since some selected circuits require at least seven qubits for their execution. For Google, we use Cirq's Qsim simulator integrated with Google's provided noise models (rainbow and weber) as noisy backend. For Rigetti, we use Rigetti's Quantum Virtual Machine (QVM) with the provided 9 qubit noise model (9q-square) as a noisy backend. We split the circuits into baseline circuits and CUTs. For the real-world benchmarks, we split them using a 70:30 ratio, i.e., 9 programs split into 3 baseline circuits and 6 CUTs. Although an 80:20 split ratio is more common, we opt for 70:30 for the real-world benchmarks to have at least three baseline circuits to generate enough data for training the baseline models. Note that the 80:20 split ratio cannot be achieved if we want to have three baseline circuits. The three baseline circuits were then inputted to the *Data Generation* component of *Baseline Trainer*. We split the artificial benchmarks using an 80:20 split ratio, commonly used in ML pipelines [43], [44], [52], i.e., 80% for CUTs and 20% as baseline circuits for the *Baseline Trainer* component.

We use Ktrain [45] to build the ML training and testing pipeline with mean absolute error as the loss function [47]. The hyper-parameters were automatically selected based on the guideline in [45]. After training, *Baseline Trainer* produces 26 baseline MLP models, each of which was fine-tuned in *Baseline Tuner* for each CUT. For fine-tuning the baseline MLP models, we generated data using a maximum of four non-faulty tests, depending on the CUT's input space. Although a 7-qubit CUT could have up to 128 tests, we intentionally limited it to four, demonstrating that fine-tuning does not require many tests. The *Baseline Tuner* results in 156 (6 CUTs, 26 backends) fine-tuned MLP models, which were used in *Noise Filter*. For the artificial benchmarks, with the 80:20 ratio, we obtain 20800 (800 CUTs, 26 backends) fine-tuned models for *Noise Filter*.

We also compare $QOIN$ with optimized Kalman filter (OKF) [53] –a variant of the Kalman filter [38], a classical noise reduction technique– for its application in mitigating quantum noise. When applying classical noise reduction methods to the quantum domain, we need to manually determine various parameters to adapt the classical method appropriately. The advantage of the optimized Kalman filter over other methods is its ability to compute the suitable parameters automatically—in this case, the process noise matrix Q and the measurement uncertainty matrix R based on past data. We optimized the OKF for each circuit-backend pair using the same non-faulty tests that were used in the Baseline Tuner module of $QOIN$.

We experimented on a machine with Intel Core i9-12900K processor (20 CPUs), 32 GB of RAM, and an Nvidia RTX-3080ti graphics card.

C. Metrics and Statistical Tests

In RQ1, we used the Hellinger Distance, which is commonly used to compare independent execution results of a circuit on a noisy computer [54]. For each circuit, the Hellinger Distance between the program specification (P) and the noisy output (P_N), or between P and the filtered output (P_F), is calculated with Eq. 1:

$$H_{Ideal}X = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{\sqrt{2}}|\sqrt{P}-\sqrt{X}| \tag{1}$$

where n is the number of tests of a circuit, and X is either P_N or P_F. Hellinger Distance ranges between 0 and 1, where 0 means no difference, and 1 means no similarity.

RQ1 analyzes results from two aspects: from the backend aspect, we average the Hellinger distances of all CUTs for each noisy backend, while from the circuit aspect, we average the Hellinger Distances of all noisy backends for each CUT. Moreover, to test statistical significance, based on the guideline [55], we used the Kruskal-Wallis test for both the circuit and backend aspects. We use it to check whether $QOIN$ has significantly different performance in terms of the Hellinger Distance across different backends and circuits. Similar performance of most noisy backends indicates that $QOIN$ can effectively learn and filter noise from different noisy backends. In case of differences in performance, we also use Epsilon Squared effect size and Dunn's test as recommended posthoc tests for further analyses [56]. For Epsilon Squared effect size, we interpret the values according to [56]: an effect size in $[0.01, 0.08)$ is *Small*, in $[0.08, 0.26)$ is *Medium*, and in $[0.26, 1]$ is *Large*.

RQ2 uses an existing test oracle for quantum software testing [5], [9]. We compared the test assessment results for the original and faulty circuits with and without $QOIN$ across all noisy backends. We also used the Mann-Whitney statistical test and Vargha Delaney $\hat{A}_{12}$ effect size as recommended in [55], [56] for studying the statistical significance of the results. We use it to compare the ideal test assessment with $QOIN$ integrated test assessment. To evaluate the magnitude of differences between these two groups, $\hat{A}_{12}$ is interpreted according to [55]: $\hat{A}_{12}$ in $(0.34, 0.44]$ or $[0.56, 0.64)$ is *Small*, in $(0.29, 0.34]$ or $[0.64, 0.71)$ is *Medium*, and in $[0, 0.29]$ or $[0.71, 1]$ is *Large*.

We perform test assessments on all possible tests of the CUTs with the existing test oracle. Each CUT is assigned a score at the end of the test assessments: *Score%*, defined as the average percentage (across all noisy backends) of tests for which the CUT failed to pass the test oracle. Test assessment results for each test are classified as True Positive (TP), False Negative (FN), False Positive (FP), or True Negative (TN), as follows: **TP** - Both the program specification and test assessment say "faulty"; **FP** - Test assessment says "faulty", but the program specification says "not faulty"; **FN** - test assessment says not faulty, but the program specification says "faulty"; and **TN** - Both the program specification and test assessment say "not faulty". We calculate F1-score using the standard formula (combining precision and recall) [57] for the program specification of all the tests and compare them with both noisy test assessment results and filtered test assessment results.

TABLE III
RQ1 (BACKEND ASPECT) – AVERAGE HELLINGER DISTANCE FOR EACH BACKEND ACROSS ALL CUTS. COLUMNS H_n,H_f AND H_c SHOW THE AVERAGE HELLINGER DISTANCE BETWEEN THE IDEAL OUTPUTS OF THE PROGRAM SPECIFICATION AND NOISY, $QOIN$ FILTERED, AND OPTIMIZED KALMAN FILTERED CIRCUIT OUTPUTS, RESPECTIVELY. COLUMN $I_f\%$ IS THE PERC. CHANGE BETWEEN H_n AND H_f I.E., $\frac{(H_n - H_f)}{H_n} * 100$. COLUMN $I_c\%$ IS THE PERC. CHANGE FOR THE OPTIMIZED KALMAN FILTER

backend	Real-world Benchmarks					Artificial Benchmarks				
	H_n	H_f	H_c	$I_f\%$	$I_c\%$	H_n	H_f	H_c	$I_f\%$	$I_c\%$
Almaden	0.47	0.08	0.62	**83.85**	-31.51	0.22	0.04	0.31	**81.01**	-41.16
Boeblingen	0.45	0.06	0.56	**86.82**	-25.19	0.21	0.04	0.3	**81.70**	-42.37
Brooklyn	0.40	0.05	0.49	**86.80**	-24.10	0.17	0.03	0.26	**81.15**	-56.27
Cairo	0.33	0.04	0.43	**88.91**	-29.27	0.15	0.03	0.25	**81.29**	-66.96
Cambridge	0.54	0.25	0.54	53.33	0.240	0.31	0.08	0.40	73.59	-27.99
CambridgeV2	0.54	0.26	0.60	52.02	-9.680	0.31	0.08	0.40	73.71	-27.98
Casablanca	0.42	0.05	0.48	**87.19**	-15.41	0.17	0.03	0.26	**82.55**	-55.22
Guadalupe	0.39	0.06	0.51	**84.08**	-30.23	0.16	0.03	0.26	**80.90**	-60.43
Hanoi	0.29	0.03	0.36	**89.94**	-24.20	0.14	0.03	0.24	**80.48**	-69.29
Jakarta	0.41	0.06	0.49	**86.18**	-18.73	0.17	0.03	0.26	**82.42**	-55.33
Johannesburg	0.49	0.10	0.63	78.80	-28.64	0.26	0.04	0.35	**84.77**	-32.66
Kolkata	0.32	0.03	0.40	**91.14**	-24.28	0.13	0.03	0.23	77.99	-80.72
Lagos	0.32	0.04	0.38	**88.96**	-18.57	0.12	0.03	0.22	78.23	-87.59
Manhattan	0.44	0.06	0.51	**87.38**	-15.73	0.27	0.10	0.36	62.00	-35.49
Montreal	0.34	0.05	0.42	**84.84**	-23.92	0.15	0.03	0.25	**80.02**	-66.51
Mumbai	0.35	0.04	0.41	**88.78**	-18.57	0.17	0.03	0.27	**83.38**	-54.01
Nairobi	0.38	0.05	0.46	**87.99**	-21.17	0.17	0.03	0.26	**82.45**	-56.00
Paris	0.37	0.05	0.47	**86.72**	-25.39	0.17	0.03	0.27	**80.70**	-54.87
Rochester	0.59	0.38	0.66	35.14	-10.99	0.40	0.13	0.49	67.11	-21.24
Singapore	0.45	0.07	0.51	**85.20**	-13.04	0.24	0.04	0.33	**82.56**	-35.51
Sydney	0.36	0.05	0.45	**86.92**	-26.60	0.18	0.03	0.28	**82.30**	-51.26
Toronto	0.57	0.42	0.64	26.49	-13.04	0.30	0.09	0.39	69.41	-29.82
Washington	0.34	0.04	0.44	**89.23**	-29.09	0.16	0.03	0.26	**81.77**	-60.03
9q-square	0.66	0.39	0.59	40.40	10.64	0.54	0.28	0.64	48.08	-17.65
Rainbow	0.68	0.30	0.61	56.24	9.400	0.58	0.31	0.68	46.78	-16.71
Weber	0.70	0.49	0.67	29.87	4.610	0.59	0.31	0.69	46.96	-16.34

The F1 score ranges in [0, 1], where 0 means all the test assessments are inconsistent, and 1 means all are consistent.

VI. RESULTS AND DISCUSSION

A. RQ1 – Noise Effect Reduction

We assess the accuracy and quality of the trained ML model in reducing the noise effect from a noisy backend.

1) Results From the Backend Aspect: Table III shows the average results of the Hellinger Distances from the backend aspect.

Column H_n shows the average difference between the results of the ideal circuit outputs from the program specification and noisy circuit outputs for each noisy backend across all CUTs. Columns H_f and H_c show the differences between the results of the ideal circuit outputs in the program specification and the filtered circuit outputs by $QOIN$ and OKF, respectively for each noisy backend across all CUTs. The columns $I_f\%$ and $I_c\%$ show the percentage improvement from H_n to H_f and H_c, respectively. The table shows that the OKF method performed even worse than the actual noisy output. This could be because classical noise reduction methods like OKF approximate noise errors using a linear model. However, quantum noise is inherently non-linear, leading to poor noise approximation. Moreover, classical methods such as OKF lack a mechanism to distinguish between actual and noise-induced states, thus attempting to correct the probability values of all states, which results in a noisier circuit output. This demonstrates that classical noise reduction methods cannot be directly used for quantum systems without incorporating quantum-specific modifications in the methods. For $QOIN$, we see that for the real-world benchmarks, on 18 out of 26 backends, $QOIN$ achieved an improvement of over 80%, whereas on backend *Cambridge*, *CambridgeV2*[1], *Rochester*, *9q-square*, *Rainbow*, *Weber*, and *Toronto*, $QOIN$ achieved the least improvement. For the artificial benchmarks, on 16 backends, $QOIN$ achieved an improvement of more than 80% whereas, on backends *Cambridge*, *CambridgeV2*, *Rochester*, *9q-square*, *Rainbow*, *Weber*, *Toronto* and *Manhattan*, $QOIN$ performed the worst. For both benchmarks, on backends *Cambridge*, *CambridgeV2*, *Rochester*, *9q-square*, *Rainbow*, *Weber*, and *Toronto*, $QOIN$ achieved the least improvement, indicating that these backends have more considerable noise effect that is hard to be filtered out by $QOIN$ than the others for the selected CUTs. Considering all backends, the overall average improvement for real-world circuits is 74.7%, and for artificial ones, it is 75.1%; this shows that $QOIN$ effectively filters out the noise effect from the circuit outputs produced by most backends.

To test whether there is a significant difference in the performance of $QOIN$ across backends, we present the results of the Kruskal-Wallis test and the Epsilon-Squared effect size. For the real-world benchmarks, from the backend aspect, the p-value is close to **0**, less than **0.05** with a large range effect size **0.32**, indicating that there is a significant difference among the backends considering the Hellinger Distance. To evaluate which backend pairs have significant differences, we performed the Dunn's test for post-hoc analysis. Results show that the *Weber* backend from Google has significant differences with at least four other backends. For the artificial benchmarks, the p-value of the Kruskal-Wallis test is also close to **0**, less than the **0.05** with a large range effect size of **0.44**. This shows that in terms of Hellinger Distance, significant differences exist among the noisy backends for the artificial benchmarks. Dunn's test results show that each of the seven noisy backends (i.e., Rochester, Cambridge, CambridgeV2, Toronto, 9q-Square, Rainbow and Weber) had significant differences with all other backends, which can also be seen in Table III where they have the least improvement. Due to space limitations, the complete results are provided in the online repository [19].

2) Results From the Circuit Aspect: Table IV shows the average results of Hellinger Distance difference from the circuit aspect for the real-world benchmarks. For OKF, we see similarly bad results as for the backends, meaning that the classical methods cannot handle quantum noise directly. For $QOIN$, we observe that, for 5 out of the 6 CUTs, $QOIN$ achieved more than 60% improvement. For *Phase Estimation*, $QOIN$, however, only improved 21.6%. For the 800 artificial benchmarks, due to space limits, complete results are reported online [19], and we only summarize key findings. For 155 out of the 800

[1]Qiskit has two Cambridge backends: Cambridge, and CambridgeV2, having the same hardware configuration, but different measurement basis.

TABLE IV
RQ1 (CIRCUIT ASPECT) – AVERAGE HELLINGER DISTANCE FOR EACH CUT ACROSS ALL NOISY BACKENDS FOR THE REAL-WORLD BENCHMARKS. COLUMNS H_n, H_f, AND H_c DENOTE THE AVERAGE HELLINGER DISTANCE BETWEEN THE IDEAL OUTPUTS SPECIFIED IN THE PROGRAM SPECIFICATION AND NOISY, *QOIN* FILTERED, AND OPTIMIZED KALMAN FILTERED OUTPUTS, RESPECTIVELY; COLUMN $I_f\%$ IS THE PERC. CHANGE BETWEEN THE H_n AND H_f, I.E., $\frac{(H_n - H_f)}{H_n} * 100$. COLUMN $I_c\%$ IS THE PERC. CHANGE FOR THE OPTIMIZED KALMAN FILTER

CUT	H_n	H_f	H_c	$I_f\%$	$I_c\%$
GHZ	0.299	0.002	0.373	**99.2**	-24.67
Simon	0.152	0.055	0.249	**63.9**	-63.12
QFT	0.664	0.142	0.768	**78.5**	-15.66
Addition	0.573	0.138	0.593	**75.8**	-3.440
Binary Similarity	0.672	0.219	0.710	**67.4**	-5.630
Phase Estimation	0.313	0.245	0.379	21.6	-21.27

TABLE V
RQ1 (CIRCUIT ASPECT) – DUNN'S TEST RESULTS FOR THE REAL-WORLD BENCHMARKS. A HIGHER VALUE MEANS THAT THE MAGNITUDE OF NOISE EFFECT REDUCTION IN HELLINGER DISTANCE IS MORE SIMILAR BETWEEN A PAIR OF CIRCUITS

	Addition	*GHZ*	*Phase*	*QFT*	*Similarity*	*Simon*
Addition	**1.0**	0.31	<0.05	**1.0**	<0.05	**1.0**
GHZ	0.31	**1.0**	<0.05	0.3	<0.05	<0.05
Phase	<0.05	<0.05	**1.0**	<0.05	**1.0**	<0.05
QFT	**1.0**	0.3	<0.05	**1.0**	<0.05	**1.0**
Similarity	<0.05	<0.05	**1.0**	<0.05	**1.0**	**1.0**
Simon	**1.0**	<0.05	<0.05	**1.0**	**1.0**	**1.0**

TABLE VI
RQ2 – AVERAGE PAIRWISE DIFFERENCES IN HELLINGER DISTANCE (COLUMN *AVG. PAIRWISE HL*) FOR THE CIRCUITS WITH A SIMILAR DIVERSITY SCORE. THE FIRST COLUMN SHOWS THE AVERAGE DIVERSITY SCORE (IN *JSD*) GROUPS, AND THE SECOND COLUMN IS THE NUMBER OF CUTS BELONGING TO EACH GROUP

Avg. Diversity Score Group	*# Circuits*	*Avg. Pairwise HL*
0.5	46	0.025
0.6	485	0.031
0.7	236	0.038
0.8	16	0.028

benchmarks, *QOIN* achieved an average improvement of more than **80%**, 281 circuits have more than **70%**, 203 circuits have more than **50%**, and 142 circuits have more than **10%** average improvement; for the other 19 circuits, *QOIN* achieved a very minor or no improvement.

An explanation for the exceptions of *Phase Estimation* and the 19 artificial circuits is that they all consist of more phase gates than other types. This indicates that the noise effect of different noisy backends on programs, mostly consisting of phase gates, is challenging to distinguish and requires further analysis. A possible reason could be the train-test split of the baseline circuits and CUTs for the *Baseline Training* component. We selected baseline circuits randomly with a specific split ratio (see Sect. V-B). After observing the results for *Phase Estimation* and the 19 artificial circuits, we checked the baseline circuits used for training in *Baseline Trainer* and noted that the selected baseline circuits have fewer phase gates than other types of quantum gates. This could lead to a bias in training, as the training dataset captures the accumulating effect of noise and not individual gate noise. Thus, the imbalance between the number of phase gates and other types of gates could affect the performance of *QOIN* on circuits with many phase gates. In future studies, we will also focus on better circuits for MLP training.

The results of the Kruskal-Wallis test and Epsilon-Squared effect size, for the real-world benchmarks, the p-value is $\mathbf{1.27e^{-13}}$ which is less than **0.05** with an effect size of **0.44**, indicating that there is at least one circuit for which *QOIN* performed significantly different than for the others across all noisy backends. We further checked the results of Dunn's test (Fig. 5) for the real-world benchmarks. Fig. 5 shows that *Phase Estimation* clearly differs from the other circuits, which is consistent with what we observed in Table IV. We also see that *Similarity* is similar to *Simon* and *Phase Estimation*, whereas *Addition*, *GHZ*, and *QFT* are similar to each other. A reason is that the circuit structure (in terms of phase gates) of *Similarity* is more similar to *Phase Estimation* and *Simon* than to other circuits. Similarly, circuits *Addition*, *GHZ*, and *QFT* (that have no phase gates) are similar to each other. After *Phase Estimation*, *Similarity* has the highest value for H_if because it has more phase gates than the other circuits, which supports the hypothesis that noise introduced by phase gates is more difficult to distinguish than for other quantum gates.

Results of the Kruskal-Wallis test and Epsilon-Squared effect size for the artificial benchmarks show that the p-value is close to **0** (less than **0.05**) with an effect size of **0.40**, indicating a significant difference regarding *QOIN*'s performance among the artificial benchmarks. Dunn's test results show that 19 out of the 800 artificial benchmarks (consisting of phase gates) have significant differences among themselves and with the others. Detailed results are available in the online repository [19]. This observation asks if *QOIN* can generalize noise learning among similar circuits for all noisy backends. To answer this question, we identify groups of similar artificial benchmarks based on their output diversity scores (*JSD*) (see Sect. V-A) and then quantify the average pairwise difference for H_if (Table IV) for circuits in each group across all noisy backends. To identify circuit groups, we rounded the diversity score to one decimal place and identified four scores ranging from 0.5 to 0.8. Table VI presents the results. As shown in Table VI, 485 out of the 800 CUTs have an average diversity score of 0.6, whereas only 16 CUTs have a diversity score of 0.8. Like Table IV, for each circuit in each average diversity score group, its H_if values across all noisy backends are averaged. For *QOIN* to have a similar magnitude of noise effect reduction for a particular group of circuits, the average pairwise difference in H_if should be as close to zero as possible. Table VI shows that the average pairwise difference in H_if for all groups is greater than 0.02 but less than 0.04, with the maximum value being 0.038. This shows that for any pair of circuits with similar output distributions,

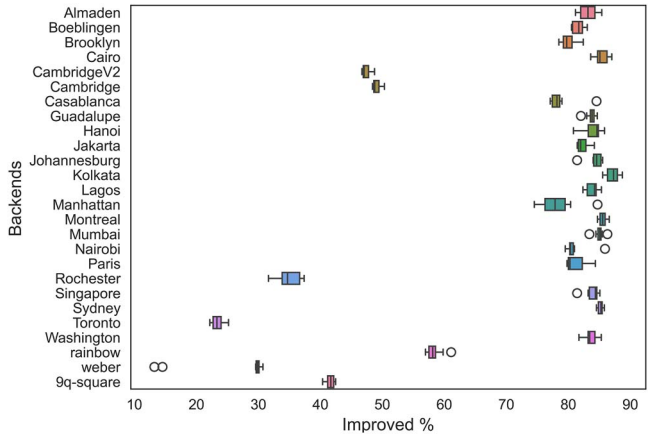

Fig. 4. RQ1 – result of *Improved%* (see Table III) on the real-world benchmark for 10 runs. Each box plot shows the distribution of percentage improvement achieved by $QOIN$ for each backend for 10 runs.

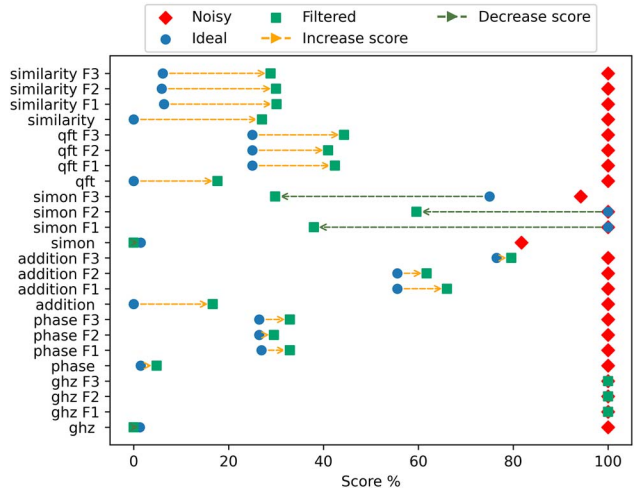

Fig. 5. RQ2 – test assessment for the original and its faulty circuits of the real-world benchmarks across all noisy backends on average. The x-axis (*Score %*) is the average perc. of tests on which a circuit failed the test assessment across all backends; the y-axis labels the original circuits (e.g., *qft*) and its three faulty ones (e.g., *qft F1*, *qft F2*, *qft F3*).

$QOIN$ generalizes the amount of noise effect it filters from the circuits' output.

3) Variance of ML Models: ML models are inherently probabilistic, leading to multiple predicted filtered outputs for multiple noisy circuit outputs. To assess the extent of variation in ML model predictions with changes in noisy circuit output, we computed the metrics detailed in Table III over ten runs, only for the real-world benchmarks because a huge amount of time would be required if computing for all benchmarks. Results are reported in Fig. 4. Fig. 4 demonstrates a consistent alignment of results with the average outcomes presented in Table III for each backend. The observed variance, in general, is minimal for most backends, suggesting that the ML models consistently predicted similar results for different noisy circuit outcomes across various runs.

Answer to RQ1: $QOIN$ effectively reduces the noise effect from the circuit outputs running on most noisy backends by more than 80%. With a pairwise difference of less than 0.04 for a group of similar circuits, $QOIN$ also generalizes the amount of noise effect it filters out from outputs.

B. RQ2 – Test Assessment Improvement

1) Noise Effect in Test Assessments: Fig. 5 shows the average results of the test assessments for all noisy backends on the original and faulty circuits (with suffixes of F1, F2, F3) of the real-world benchmarks. Fig. 5 shows that four out of six original circuits (four blue dots located at the 0% position on the x-axis) exhibit no fault. The remaining two circuits (*Phase and Simon*) are closer to 0% but not exactly 0% due to false positives for some tests. Assessment results for the noisy backends without $QOIN$ (red diamonds) indicate that most circuits failed on 100% of the tests. The green squares denote the test assessment results after applying $QOIN$, and the dashed arrows visualize the difference between the ideal outputs and the results after applying $QOIN$. We see that $QOIN$ allowed test assessment with the noisy backends much closer to the program specifications, as the yellow dashed lines connecting results of the program specifications (ideal) and noisy backends with $QOIN$ applied are very short. Fig. 5 shows three cases: ***Case-1–no difference*** in the scores of the ideal outputs and results from the noisy backend and filtered with $QOIN$, shown as overlapped blue dots and green squares, i.e., *ghz F1*, *ghz F2* and *ghz F3*; ***Case-2–decrease*** in the scores regarding the noise effect migration from the ideal outputs to the noisy backend, indicating fewer tests failed and more false negatives in the test assessment results with $QOIN$ on the noisy backend (e.g., dashed green arrow on *simon F3*); ***Case-3–increase*** in the scores regarding the noise effect migration from the ideal outputs to the noisy backend (e.g., dashed yellow arrow on *addition*), telling that more tests failed and more false positives observed in filtered test assessment results.

Tests can have different results on different noisy backends; therefore, we use the Mann-Whitney statistical test and Vargha Delaney $\hat{A}_{12}$ effect size [58] to compare the scores achieved by comparing the ideal outputs in the program specification with those from the noisy backends and filtered with $QOIN$ for all noisy backends. Table VII shows that four out of the six real-world benchmarks fall into Case-3, among which only one exhibits a large effect size on the significant difference between the ideal outputs in the program specifications and results from the noisy backends filtered with $QOIN$ across all noisy backends. For the 18 faulty circuits, only 6 have a large effect size, for which the ideal outputs are significantly better than that from the noisy backend and filtered by $QOIN$ across all noisy backends. For the artificial benchmarks, only eleven of the 800 circuits show a large effect size when comparing the ideal outputs and results from the noisy backends filtered with $QOIN$ across all noisy backends, and 21% (505 out of 2400) of the faulty circuits exhibit significant differences.

2) F1-Score of QOIN in Test Assessment: In the real-world and artificial benchmarks, we observed many circuits

TABLE VII
RQ2 – NUMBER OF CIRCUITS FALLING INTO THE THREE NOISE EFFECT MIGRATION CASES: CASE-1–NO DIFFERENCE; CASE-2–DECREASE FROM THE IDEAL TO FILTERED; CASE-3–INCREASE FROM THE IDEAL TO FILTERED. ROW *EFFECT (LARGE)* SHOWS THE NUMBER OF CIRCUITS HAVING A LARGE EFFECT SIZE ($\hat{A}_{12}$)

	Real-world Benchmarks		Artificial Benchmarks	
	Original	*Faulty*	*Original*	*Faulty*
Case-1	0	3	93	79
Case-2	2	3	0	581
Case-3	4	12	707	1740
Effect (Large)	1	6	11	505

TABLE VIII
RQ2 – F1-SCORE, PRECISION, AND RECALL FOR ALL TESTS ACROSS ALL NOISY BACKENDS. A HIGHER F1-SCORE MEANS A LOWER CHANCE OF HAVING FALSE POSITIVES OR NEGATIVES, A HIGHER PRECISION MEANS FEWER FALSE POSITIVES, AND A HIGHER RECALL MEANS FEWER FALSE NEGATIVES. COLUMNS *W/O QOIN* AND *WITH QOIN* SHOW THE RESULTS FOR ALL TESTS ACROSS ALL NOISY BACKENDS WITHOUT *QOIN* AND *WITH QOIN*, RESPECTIVELY

	Real-World Benchmarks		Artificial Benchmarks	
	w/o *QOIN*	**with** *QOIN*	**w/o** *QOIN*	**with** *QOIN*
F1-score	0.02	**0.86**	0.07	**0.86**
Precision	1.0	**0.99**	1.0	**0.93**
Recall	0.01	**0.75**	0.03	**0.79**
False Positives	0	**22**	0	**32,237**
False Negatives	11,689	2,946	553,965	117,485

changed in *Score%* (see Sect. V-C), though only a small portion shows a large effect size on the significance (Table VII). Such changes occurred due to false positives (Case-3) or false negatives (Case-2) produced by *QOIN*. To evaluate the quality of *QOIN* in reducing false positives or false negatives, we calculate the F1-score. A typical test assessment is similar to binary classification in that each assessment results in either a fault or not. We compare the F1-score for the test assessment with *QOIN* and without *QOIN* for all circuits across all noisy backends. Results are summarized in Table VIII.

In Table VIII, a substantial improvement in the F1 score is evident after integrating *QOIN*. For the real-world benchmarks, the F1 score increased from 2% to 86%, and for the artificial benchmarks, it improved from 7% to 0.86%. Generally, many false positives and negatives in test assessments indicate the assessment is unreliable. Table VIII shows a notable reduction in false negatives with *QOIN* (from 11,689 to 2,946 for real-world benchmarks and from 553,965 to 117,485 for artificial ones). However, *QOIN* also introduces false positives, indicating instances where it could not filter out noise from some tests. Considering precision, in both benchmarks, we observe that without *QOIN*, precision is 1.0, indicating no false positives occurred. This is because all tests failed, including the ones meant to fail. Moreover, recall for real-world and artificial benchmarks without the *QOIN* is 0.01 and 0.03, which is very low, indicating a high number of false negatives. *QOIN* shows a significant improvement in recall for both benchmarks, while precision is slightly decreased, indicating that for some tests, *QOIN* could not reduce noise. Overall, the F1 score of *QOIN* is close to the ideal F1 score of 1.0 and demonstrates significantly fewer false negatives. This suggests that *QOIN* can be a valuable tool for enhancing the test assessment of existing methods for noisy backends.

Answer to RQ2: *QOIN* can be integrated effectively with existing testing methods to support testing on noisy backends. It reduces the gap between the program specification and test assessments with precision, recall, and F1-score of 99%, 75%, and 85%, for real-world benchmarks. Similarly, for artificial benchmarks, it shows a precision, recall, and F1-score of 93%, 79%, and 85%.

TABLE IX
ABLATION STUDY – F1-SCORE, PRECISION, AND RECALL FOR ALL TESTS ACROSS ALL NOISY BACKENDS. A HIGHER F1-SCORE MEANS A LOWER CHANCE OF HAVING FALSE POSITIVES OR NEGATIVES; A HIGHER PRECISION MEANS FEWER FALSE POSITIVES; A HIGHER RECALL MEANS FEWER FALSE NEGATIVES. COLUMNS *W/O BASELINE* AND *WITH BASELINE* SHOW THE RESULTS WITHOUT *BASELINE TRAINER* AND *WITH BASELINE TRAINER*, RESPECTIVELY

	Real-World Benchmarks		Artificial Benchmarks	
	w/o *Baseline*	**With** *Baseline*	**w/o** *Baseline*	**With** *Baseline*
F1-score	0.78	**0.86**	0.77	**0.86**
Precision	0.96	**0.99**	0.93	**0.93**
Recall	0.66	**0.75**	0.66	**0.79**
False Positives	305	**22**	25604	32,237
False Negatives	4005	**2946**	193328	**117485**

C. Ablation Study

By leveraging three descriptive features, we adapted classical machine learning to reduce noise in quantum systems. However, relying on three features requires a substantial amount of circuit-specific data. We used pre-training and fine-tuning techniques to overcome this challenge to improve performance in terms of noise reduction while reducing the need for circuit-specific data. In this section, we evaluate the benefits of our two-module method through an ablation study, comparing the standard *QOIN* approach with a variant that excludes the Baseline Trainer.

Both versions use the same amount of circuit-specific data for the Baseline Tuner, but the default *QOIN* fine-tunes from a baseline model, while the other version trains directly on circuit-specific data for each backend from scratch. Table IX compares *QOIN* with and without the Baseline Trainer, evaluating F1-score, precision, and recall for test assessment across all noisy backends. Without the Baseline Trainer, the F1-scores for both benchmarks are 78% and 77%, respectively, indicating that the ML model can directly identify noise patterns from the circuit-specific data. However, the recall for both benchmarks is 66%, indicating a high number of false negatives resulting in less reliable test assessment. Higher false negatives suggest the need for more features or additional data to improve performance. Conversely, with the Baseline Trainer, *QOIN* achieves significantly better F1-scores, recall, and fewer false negatives. This demonstrates that including the Baseline Trainer

can improve the effectiveness with the same amount of circuit-specific data, thereby reducing the overall data requirements. If a large amount of circuit-specific data is available, the Baseline Tuner can be used directly, removing the Baseline Trainer's training costs. If data is limited, including the Baseline Trainer module can enhance effectiveness while incurring additional training costs.

D. Threats to Validity

External validity: We selected a small set of real-world circuits for evaluation, threatening the results' generalizability. More circuits help improve generalizability; however, only a few real-world circuits are available. To mitigate this threat, we generated 1000 artificial circuits by following a common practice [14], [33]. We also selected circuits that operate on a small number of qubits, which may affect generalizability. Indeed, the noise effect depends on the number of qubits, i.e., a higher number of qubits will increase the chances of errors due to noise. However, we cannot choose circuits with a large number of qubits since doing so will restrict the selection of noisy backends to use and make the circuits requiring a large number of qubits impossible to execute on all noisy backends. Out of 47 available noisy backends from IBM, only 23 have a qubit count of at least seven, which we chose. Another threat is the improper optimization of hyperparameters in ML algorithms. To reduce this threat, we followed the guidelines of [45], [46] to construct our ML pipeline for auto-tuning the parameters and monitoring the training loop to avoid over-fitting.

Construct Validity: Concerning the evaluation metrics, we used Hellinger Distance (see Sect. V-C) to measure the diversity of circuit outputs, which has been widely used for this purpose [54]. We also used standard F1-score, precision and recall to measure the quality of $QOIN$ in noise effect reduction [57]. For statistical tests, we followed the guidelines provided in [55], [56].

Conclusion Validity: Not applying $QOIN$ on real computers is one threat; however, available real computers offer limited public access, making it infeasible to evaluate $QOIN$ on them. To mitigate this threat, we used noise models provided by IBM, Google, and Rigetti for their real computers. Each noise model closely approximates the behavior of a quantum computer and is updated frequently.

To demonstrate the effectiveness of $QOIN$, we used the available noise models from IBM, Google, and Rigetti, matching our inclusion criteria at the time of the experiment. Thus, our conclusions apply to these noise models. In the future, we will conduct dedicated experiments to study whether $QOIN$ can be used for other noise models as more noise models become available.

VII. Discussion

A. Generalizability

Generalizability of QOIN's modules. $QOIN$ is designed modularly to incorporate any new backend without a major change. Specifically, $QOIN$ expects a baseline circuit as a QASM file, a common format Qiskit uses. To enhance the modularity, $QOIN$ provides an abstract interface class, which can be used to define the logic for executing a QASM circuit on any noisy backend. Given the baseline circuits, $QOIN$ automatically generates the required dataset for *MLP Training* and uses an automated selection of optimal hyperparameters for the ML models. Regarding *Feature Generation*, each quantum gate is affected by a certain type of noise, and different gate combinations accumulate, leading to different noise patterns, consequently affecting the circuits' outputs. Hence, to generalize for different noise types, we extract training features from the circuit outputs; otherwise, the feature extraction would become gate- and circuit-specific. These features capture the accumulated noise effect without understanding the noise pattern details of specific gates and how they are combined to form circuits. For *Test Analyzer*, since $QOIN$ only requires the outputs to generate features for *Noise Filter*, $QOIN$ can be integrated with any test strategy that assumes the availability of program specifications in the form of circuit outputs by acting as a filter between test execution and test assertion.

Generalizability of the machine learning (ML) models of $QOIN$**.**

Developing a (generalized) single model covering all possible quantum computers requires a further understanding of the characteristics of the quantum computers and their quantum noise. For now, it is challenging to achieve because each quantum computer has different noise characteristics directly correlating with the hardware configuration, physical implementation of qubits, and the surrounding environment [3]. Moreover, the same circuit exhibits a completely different noise distribution on different quantum computers based on gate composition and dynamic circuit mapping on physical hardware [3]. Currently, no set of generalized features can accurately define or map the behavior of quantum noise for all quantum computers and circuits. However, new studies show that Bayesian inference might be a possible alternative to characterize noise for different quantum computers and circuits [59], [60], which we will investigate in the future.

B. Applicability and Maintainability

$QOIN$'s maintainability is affected by data drift, which refers to the phenomenon where the statistical properties of the input data change over time, leading to a degradation in the ML model performance [61]. Handling data drift is crucial to maintaining the accuracy and effectiveness of the models. The noise behavior for a quantum computer changes more frequently than any other data over time. To keep the noise model updated, IBM recalibrates its noise models every 24 hours. Currently, we do not handle data drift, $QOIN$ can be integrated with the current best strategies for handling data drifts such as adaptive learning, data drift mitigation, and detection [61].

From the application perspective, a tester shall perform three steps to test a circuit (i.e., c) on a noisy backend (i.e., b) with a test suite (i.e., ts): (1) Obtain the baseline model of b from our repository if available; otherwise, use *Baseline Trainer* for training a baseline model from scratch; (2) Tune the baseline

model for c with a subset of tests from ts using *Baseline Tuner*; and (3) Use *Test Analyzer* to test c against ts, including filtering of results.

For the applicability of $QOIN$ as a whole, one can question why $QOIN$ is needed since we can use noise-free simulators. It is needed because quantum simulators are computationally expensive; even a small circuit can take up to 12 hours to execute one test [17]. Though, in the literature, some strategies (e.g., CutQC [62]) have been proposed to reduce the computational cost, testing is still time-consuming. $QOIN$ solves this problem by allowing testing directly on real quantum computers. *Baseline Trainer* of $QOIN$ requires simple circuits without high execution costs to capture the general noise pattern of a backend (RQ1). The program specification can be generated by simulation or can be provided by developers. *Baseline Tuner* requires only a handful of tests to generate a circuit-specific model. For CUTs, if the program specification is unavailable, it can be generated using strategies like CutQC [62] on noise-free simulators. The time cost of simulating only a handful of tests is way less than doing systematic testing on noise-free simulators. By generating the circuit-specific model, testing can be performed on a real quantum computer, which is much more scalable.

C. Data Requirements

ML algorithms often require a significant amount of data depending on the complexity of the problem [63]. However, generating a large dataset by executing circuits is time-consuming. Additionally, quantum noise is unique to the backend on which a circuit is executed. Training a model for all possible circuit-backend pairs would demand massive data and necessitate larger, more complex ML models, increasing the training and inference time costs. Testing often face budget constraints, and quantum circuits pose a significant challenge due to their vast state space. To sufficiently test a circuit, lots of test executions are required. Given these limitations, employing a complex ML model that demands extensive data and large inference times may not be a practical solution. Having a balance between model complexity and the practical considerations of testing resources is crucial.

To address data-related challenges, we divided $QOIN$ into two modules: Baseline Trainer and Baseline Tuner. Baseline Trainer learns general noise patterns requiring collecting observed noisy quantum output states from multiple circuits, i.e., one-time cost per quantum computer. Baseline Tuner learns circuit-specific noise patterns, which require the noisy output states from a specific circuit. This two stage process reduces the required data and allows using less complex ML models. In our experiments, the amount of generated data depends on the number of selected baseline circuits and the number of output states produced by each circuit-input pair. For a real-world experiment with three baseline circuits, the generated data had approximately 1000 output states for each backend. For the artificial circuit experiment with 200 baseline circuits, the generated data exceeds 5000 output states. For both experiments, we chose an MLP model with only two hidden layers, as increasing the number of layers would result in overfitting due to the limited amount of data available. Our results are satisfactory for most backends. However, it becomes evident that additional training data or a more complex model is needed for certain backends where improvements are limited. Nevertheless, obtaining more training data through simulation would significantly increase training costs. Consequently, the amount of data required highly depends on the particular quantum computer targeted.

D. Time Cost

The adoption of $QOIN$ enhances the reliability of test assessments on quantum computers, but it requires increased time for the overall test assessment of a circuit. Here, we discuss the time cost associated with applying $QOIN$ about its core components. First, the Data Generation component within the Baseline Trainer module generates the necessary data to train a baseline MLP model for a specific noise backend. Its time cost is a one-time expense for each noisy backend and is directly influenced by the number of baseline circuits and the quantum simulator utilized. Our experiments had three baseline circuits for real-world benchmark experiment one and 200 baseline circuits for the artificial benchmark. The average time cost of the Data Generation component for one noisy backend amounted to 11.8 minutes. For a real quantum computer, the time cost would typically be in the order of seconds. Second, the MLP training component of Baseline Trainer also has a one-time cost for a given noisy backend, which is, on average, approximately one minute for our experiments conducted on an Nvidia 3080 GPU. This highlights that although the time cost for the Baseline Trainer module is incurred only once, the quantum simulator occupies a significant amount of time compared to model training due to the current hardware limitations.

There is also a time cost associated with the Baseline Tuner module, which is not a one-time cost as it depends on how frequently noise changes in the noisy backend. Typically, for IBM quantum computers, the noise models are calibrated at least once every 24 hours, implying that the Baseline Tuner module needs to be executed once every 24 hours. In our experiments, the average time cost for the Baseline Tuner module for one circuit-backend pair was approximately 14 minutes. About 12 minutes were required for data generation to fine-tune a baseline MLP model. Note that the data generation cost would significantly decrease when performed on quantum computers. This is due to the fact that the execution of a circuit on a quantum computer typically takes seconds, whereas, on simulators, it takes several minutes. This indicates that the MLP tuning cost is roughly 2 minutes for one circuit-backend pair.

The time cost of test assessment of a single circuit-backend pair would be the time cost of the Baseline Tuner and the inference time required by the Noise Filter component. The inference time cost for the MLP model on a GPU is only a few seconds. Thus, $QOIN$ does not entail a substantial time cost for the test assessment.

E. Limitations

First, $QOIN$ only provides the baseline models of 26 noisy backends (23 from IBM, two from Google, and one from Rigetti). Therefore, a user can use $QOIN$ to test circuits on these 26 noisy backends with any test strategy that uses circuit outputs and their probabilities to check the correctness of circuits. As a result, additional effort is needed to learn baseline models for other backends. This would require having the noisy outputs of the baseline circuits to train a new baseline model for a new noisy backend (see Generalizability of modules in Sect. VII-A).

Second, noisy backends keep changing; therefore, their corresponding baseline models must be updated regularly. Thus, $QOIN$ is expected to benefit from adaptive learning [61]. Third, we need to conduct research on integrating different models as one model, such that it could model the noise of all backends or, at least, a subset of backends with similar characteristics (e.g., number of qubits, qubit layouts). One possible aspect could be the adoption of Bayesian inference models [64] for this purpose. Lastly, we only explored using the program output diversity as the criterion to generate diverse circuits. However, other criteria also deserve attention, such as structural diversity, circuit depth, or combinations of multiple diversity criteria.

VIII. Conclusion and Future Work

To enable quantum software testing techniques to deal with hardware noise—inevitable in near-term quantum computers—, we presented the approach $QOIN$. $QOIN$ applies machine learning to learn quantum noise, followed by filtering noise from circuit outputs, which are then used for testing. $QOIN$ helps quantum software testers determine whether a test case failed due to real faults or noise. We evaluated $QOIN$ using six real circuits and 800 diverse artificial circuits. Moreover, faulty versions of these circuits were generated to determine whether $QOIN$ allows the correct detection of failing test cases, i.e., whether it allows for distinguishing between faults and noise. We used 23 noise models from IBM, two from Google, and one from Rigetti to run experiments. Our results showed that $QOIN$ effectively removes noise. In the future, we will extend our experiments to more diverse circuits. Moreover, we also plan to verify $QOIN$ on real quantum computers and integrate $QOIN$ with other quantum software testing techniques.

Acknowledgment

The authors benefited from the Experimental Infrastructure for Exploration of Exascale Computing (eX3), financially supported by the Research Council of Norway under contract 270053, for conducting the experiments. The authors acknowledge the use of IBM Quantum services for this work. The views expressed are those of the authors, and do not reflect the official policy or position of IBM or the IBM Quantum team.

References

[1] J. M. Murillo et al., "Challenges of quantum software engineering for the next decade: The road ahead," 2024, *arXiv:2404.06825*.
[2] J. D. Hidary and J. D. Hidary, "Development libraries for quantum computer programming," in *Quantum Computing: An Applied* Approach, Cham, Switzerland: Springer: 2021, pp. 67–86.
[3] S. Resch and U. R. Karpuzcu, "Benchmarking quantum computers and the impact of quantum noise," *ACM Comput. Surv.*, vol. 54, no. 7, pp. 1–35, Jul. 2021.
[4] R. Alicki, "Decoherence and the appearance of a classical world in quantum theory," *J. Phys. A Math. General*, vol. 37, no. 5, Feb. 2004, Art. no. 1948.
[5] X. Wang, P. Arcaini, T. Yue, and S. Ali, "Application of combinatorial testing to quantum programs," in *Proc. IEEE 21st Int. Conf. Softw. Qual., Rel. Secur. (QRS)*, 2021, pp. 179–188.
[6] S. Ali, P. Arcaini, X. Wang, and T. Yue, "Assessing the effectiveness of input and output coverage criteria for testing quantum programs," in *Proc. IEEE 14th Int. Conf. Softw. Testing, Validation Verification (ICST)*, 2021, pp. 13–23.
[7] P. Long and J. Zhao, "Testing multi-subroutine quantum programs: From unit testing to integration testing," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 6, pp. 1–61, Jun. 2024, doi: 10.1145/3656339.
[8] J. Ye et al., "QuraTest: Integrating quantum specific features in quantum program testing," in *Proc. 38th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, 2023, pp. 1149–1161.
[9] X. Wang, P. Arcaini, T. Yue, and S. Ali, "QuSBT: Search-based testing of quantum programs," in *Proc. ACM/IEEE 44th Int. Conf. Softw. Eng.: Companion Proc.*, New York, NY, USA: ACM, 2022, pp. 173–177.
[10] X. Wang, T. Yu, P. Arcaini, T. Yue, and S. Ali, "Mutation-based test generation for quantum programs with multi-objective search," in *Proc. Genetic Evol. Comput. Conf.*, New York, NY, USA: ACM, 2022, pp. 1345–1353.
[11] D. Fortunato, J. Campos, and R. Abreu, "Mutation testing of quantum programs: A case study with Qiskit," *IEEE Trans. Quantum Eng.*, vol. 3, pp. 1–17, 2022.
[12] E. Mendiluze, S. Ali, P. Arcaini, and T. Yue, "Muskit: A mutation analysis tool for quantum software testing," in *Proc. 36th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, 2021, pp. 1266–1270.
[13] R. Abreu, J. P. Fernandes, L. Llana, and G. Tavares, "Metamorphic testing of oracle quantum programs," in *Proc. 3rd Int. Workshop Quantum Softw. Eng.*, New York, NY, USA: ACM, 2023, pp. 16–23.
[14] M. Paltenghi and M. Pradel, "MorphQ: Metamorphic testing of the Qiskit quantum computing platform," in *Proc. 45th Int. Conf. Softw. Eng.*, Piscataway, NJ, USA: IEEE Press, 2023, pp. 2413–2424.
[15] S. Honarvar, M. R. Mousavi, and R. Nagarajan, "Property-based testing of quantum programs in Q#," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng. Workshops*, New York, NY, USA: ACM, 2020, pp. 430–435.
[16] J. Wang, F. Ma, and Y. Jiang, "Poster: Fuzz testing of quantum program," *Proc. IEEE 14th Int. Conf. Softw. Testing, Verification Validation (ICST)*, 2021, vol. 2, no. 1, pp. 466–469.
[17] E. Younis, K. Sen, K. Yelick, and C. Iancu, "QFAST: Conflating search and numerical optimization for scalable quantum circuit synthesis," in *Proc. IEEE Int. Conf. Quantum Comput. Eng. (QCE)*, 2021, pp. 232–243.
[18] S. Martina, S. Gherardini, L. Buffoni, and F. Caruso, "Noise fingerprints in quantum computers: Machine learning software tools," *Softw. Impacts*, vol. 12, 2022, Art. no. 100260.
[19] A. Muqeet, "QOIN: Public release," Aug. 2024, doi: 10.5281/zenodo.13488933.
[20] P. A. M. Dirac, "A new notation for quantum mechanics," in *Math. Proc. Cambridge Philos. Soc.*, vol. 35, no. 3. Cambridge, U.K.: Cambridge University Press, 1939, pp. 416–418.
[21] D. P. DiVincenzo, "Quantum gates and circuits," *Proc. Roy. Soc. London Ser. A: Math., Phys. Eng. Sci.*, vol. 454, no. 1969, pp. 261–276, 1998.
[22] A. Cross, "The IBM Q experience and QISKit open-source quantum computing software," in *APS March Meeting Abstr.*, vol. 2018, 2018, p. L58-003.
[23] T. Ayral, F.-M. L. Régent, Z. Saleem, Y. Alexeev, and M. Suchara, "Quantum divide and compute: Exploring the effect of different noise sources," *SN Comput. Sci.*, vol. 2, no. 3, 2021, Art. no. 132.
[24] S. Ruan, Y. Wang, W. Jiang, Y. Mao, and Q. Guan, "VACSEN: A visualization approach for noise awareness in quantum computing," 2022, *arXiv:2207.14135*.
[25] Y. Huang and M. Martonosi, "Statistical assertions for validating patterns and finding bugs in quantum programs," in *Proc. 46th Int. Symp. Comput. Archit.*, New York, NY, USA: ACM, 2019, pp. 541–553.

[26] G. Li, L. Zhou, N. Yu, Y. Ding, M. Ying, and Y. Xie, "Projection-based runtime assertions for testing and debugging quantum programs," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020, pp. 1–29.
[27] X. Wang, P. Arcaini, T. Yue, and S. Ali, "Quito: A coverage-guided test generator for quantum programs," in *Proc. 36th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, 2021, pp. 1237–1241.
[28] D. Fortunato, J. Campos, and R. Abreu, "QMutPy: A mutation testing tool for quantum algorithms and applications in Qiskit," in *Proc. 31st ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, New York, NY, USA: ACM, 2022, pp. 797–800.
[29] J. Luo, P. Zhao, Z. Miao, S. Lan, and J. Zhao, "A comprehensive study of bug fixes in quantum programs," in *Proc. IEEE Int. Conf. Softw. Anal., Evolution Reengineering (SANER)*, 2022, pp. 1239–1246.
[30] P. Zhao, J. Zhao, and L. Ma, "Identifying bug patterns in quantum programs," in *Proc. IEEE/ACM 2nd Int. Workshop Quantum Softw. Eng. (Q-SE)*, 2021, pp. 16–21.
[31] A. Miranskyy, L. Zhang, and J. Doliskani, "Is your quantum program bug-free?" in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng.: New Ideas Emerg. Results*, New York, NY, USA: ACM, 2020, pp. 29–32.
[32] P. Zhao, J. Zhao, Z. Miao, and S. Lan, "Bugs4Q: A benchmark of real bugs for quantum programs," in *Proc. 36th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, 2021, pp. 1373–1376.
[33] J. Wang, Q. Zhang, G. H. Xu, and M. Kim, "QDiff: Differential testing of quantum software stacks," in *Proc. 36th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, 2021, pp. 692–704.
[34] Y. Wu, "Robust learning-enabled intelligence for the Internet of Things: A survey from the perspectives of noisy data and adversarial examples," *IEEE Internet Things J.*, vol. 8, no. 12, pp. 9568–9579, Jun. 2021.
[35] S. V. Vaseghi, *Advanced Digital Signal Processing and Noise Reduction*. Hoboken, NJ, USA: Wiley, 2008.
[36] C. Tannous and J. Langlois, "Classical noise, quantum noise and secure communication," *Eur. J. Phys.*, vol. 37, no. 1, 2015, Art. no. 013001.
[37] A. Jayashankar and P. Mandayam, "Quantum error correction: Noise-adapted techniques and applications," *J. Indian Inst. Sci.*, vol. 103, no. 2, pp. 497–512, 2023.
[38] K. Ma, J. Kong, Y. Wang, and X.-M. Lu, "Review of the applications of Kalman filtering in quantum systems," *Symmetry*, vol. 14, no. 12, 2022, Art. no. 2478.
[39] C. López-Martín, "Machine learning techniques for software testing effort prediction," *Softw. Qual. J.*, vol. 30, no. 1, pp. 65–100, 2022.
[40] G. Forman, "An extensive empirical study of feature selection metrics for text classification," *J. Mach. Learn. Res.*, vol. 3, pp. 1289–1305, Mar. 2003.
[41] H. Kaur and V. Kumari, "Predictive modelling and analytics for diabetes using a machine learning approach," *Appl. Comput. Inform.*, vol. 18, no. 1/2, pp. 90–100, 2022.
[42] A. S. Glas, J. G. Lijmer, M. H. Prins, G. J. Bonsel, and P. M. Bossuyt, "The diagnostic odds ratio: A single indicator of test performance," *J. Clin. Epidemiol.*, vol. 56, no. 11, pp. 1129–1135, 2003.
[43] D. Kici, A. Bozanta, M. Cevik, D. Parikh, and A. Başar, "Text classification on software requirements specifications using transformer models," in *Proc. 31st Annu. Int. Conf. Comput. Sci. Softw. Eng.*, USA: IBM Corp., 2021, pp. 163–172.
[44] A. K. Dwivedi, A. Tirkey, and S. K. Rath, "Software design pattern mining using classification-based techniques," *Frontiers Comput. Sci.*, vol. 12, no. 5, pp. 908–922, 2018.
[45] A. S. Maiya, "ktrain: A low-code library for augmented machine learning," *J. Mach. Learn. Res.*, vol. 23, pp. 7070–7075, Oct. 2022.
[46] L. N. Smith, "Cyclical learning rates for training neural networks," in *Proc. IEEE Winter Conf. Appl. Comput. Vis. (WACV)*, 2017, pp. 464–472.
[47] J. Qi, J. Du, S. M. Siniscalchi, X. Ma, and C.-H. Lee, "On mean absolute error for deep neural network based vector-to-vector regression," *IEEE Signal Process. Lett.*, vol. 27, pp. 1485–1489, 2020.
[48] C. Tan, F. Sun, T. Kong, W. Zhang, C. Yang, and C. Liu, "A survey on deep transfer learning," in *Proc. 27th Int. Conf.Artif. Neural Netw. Mach. Learn. (ICANN),* Rhodes, Greece, (Part III 27). Cham, Switzerland: Springer, 2018, pp. 270–279.
[49] Quantum Computing UK. Accessed: Sep. 26, 2024. [Online]. Available: https://quantumcomputinguk.org/code-repository
[50] Quantum Algorithm Zoo. Accessed: Sep. 26, 2024. [Online]. Available: https://quantumalgorithmzoo.org/
[51] E. Wilson, F. Mueller, L. Bassman, and C. Iancu, "Empirical evaluation of circuit approximations on noisy quantum devices," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, New York, NY, USA: ACM, 2021, pp. 1–15.
[52] R. Sanders, "The Pareto principle: Its use and abuse," *J. Services Marketing*, vol. 1, no. 2, pp. 37–40, 1987.
[53] I. Greenberg, N. Yannay, and S. Mannor, "Optimization or architecture: How to hack Kalman filtering," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 36, 2024, pp. 50482–50505.
[54] S. Dasgupta and T. S. Humble, "Characterizing the reproducibility of noisy quantum circuits," *Entropy*, vol. 24, no. 2, pp. 1–20, 2022.
[55] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proc. 33rd Int. Conf. Softw. Eng.*, New York, NY, USA: ACM, 2011, pp. 1–10.
[56] M. Tomczak and E. Tomczak, "The need to report effect size estimates revisited. An overview of some recommended measures of effect size," *Trends Sport Sci.*, vol. 21, no. 1, pp. 19–25, 2014.
[57] K. P. Murphy, "Performance evaluation of binary classifiers," The Univ. British Columbia, Tech. Rep., 2007. Available: https://www.cs.ubc.ca/~murphyk/Teaching/CS340-Fall07/reading/rocHandout.pdf
[58] A. Vargha and H. D. Delaney, "A critique and improvement of the "CL" common language effect size statistics of McGraw and Wong," *J. Educ. Behav. Statist.*, vol. 25, no. 2, pp. 101–132, 2000.
[59] V. Gebhart et al., "Learning quantum systems," *Nature Rev. Phys.*, vol. 5, no. 3, pp. 141–156, 2023.
[60] M. Zheng, A. Li, T. Terlaky, and X. Yang, "A Bayesian approach for characterizing and mitigating gate and measurement errors," *ACM Trans. Quantum Comput.*, vol. 4, no. 2, pp. 1–21, Feb. 2023.
[61] S. Arora, R. Rani, and N. Saxena, "A systematic review on detection and adaptation of concept drift in streaming data using machine learning techniques," *Wiley Interdisciplinary Rev. Data Mining Knowl. Discovery*, vol. 14, no. 4, 2024, Art. no. e1536.
[62] W. Tang, T. Tomesh, M. Suchara, J. Larson, and M. Martonosi, "CutQC: Using small quantum computers for large quantum circuit evaluations," in *Proc. 26th ACM Int. Conf. Archit. Support Program. Lang. Operating Syst.*, New York, NY, USA: ACM, 2021, pp. 473–486.
[63] S. Gollapudi, *Practical Machine Learning*. Packt Publishing Ltd, 2016.
[64] A. G. Wilson and P. Izmailov, "Bayesian deep learning and a probabilistic perspective of generalization," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 33, Red Hook, NY, USA: Curran Associates Inc., 2020, pp. 4697–4708.