The Smelly Eight: An Empirical Study on the Prevalence of Code Smells in Quantum Computing

Qihong Chen[1], Rúben Câmara[3], José Campos[2,3], André Souto[3,4], Iftekhar Ahmed[1]
[1]University of California, Irvine, USA
[2]Faculty of Engineering, University of Porto, Porto, Portugal
[3]LASIGE, Faculdade de Ciências, Universidade de Lisboa, Lisboa, Portugal
[4]Instituto de Telecomunicações, Lisboa, Portugal
chenqh@uci.edu, rcamara@lasige.di.fc.ul.pt, jcmc@fe.up.pt, ansouto@fc.ul.pt, iftekha@uci.edu

***Abstract*—Quantum Computing (QC) is a fast-growing field that has enhanced the emergence of new programming languages and frameworks. Furthermore, the increased availability of computational resources has also contributed to an influx in the development of quantum programs. Given that classical and QC are significantly different due to the intrinsic nature of quantum programs, several aspects of QC (e.g., performance, bugs) have been investigated, and novel approaches have been proposed. However, from a purely quantum perspective, maintenance, one of the major steps in a software development life-cycle, has not been considered by researchers yet. In this paper, we fill this gap and investigate the prevalence of code smells in quantum programs as an indicator of maintenance issues.**

We defined eight quantum-specific smells and validated them through a survey with 35 quantum developers. Since no tool specifically aims to detect quantum smells, we developed a tool called QSmell that supports the proposed quantum-specific smells. Finally, we conducted an empirical investigation to analyze the prevalence of quantum-specific smells in 15 open-source quantum programs. Our results showed that 11 programs (73.33%) contain at least one smell and, on average, a program has three smells. Furthermore, the *long circuit* is the most prevalent smell present in 53.33% of the programs.

***Index Terms*—Quantum computing, Quantum software engineering, Empirical study, Quantum-specific code smell**

I. Introduction

Quantum Computing (QC) is a fast developing field that has the potential to solve computational problems in areas such as cryptography, computational physics, and machine learning [1, 2, 3] that were deemed intractable by classical computing [4]. Due to such potential, private companies, universities, and government labs worldwide have heavily invested in QC. As a result, the field of QC has seen a sequence of rapid scientific and engineering advancements and an influx in the development of new programming languages and frameworks [4, 5].

For instance, languages such as Q# by Microsoft and frameworks such as Qiskit by IBM or Cirq by Google have allowed practitioners to quickly develop more complex Quantum Programs (QPs). Since there is an increasing availability of universal quantum devices via Quantum-as-a-Service (QaaS) [6, 7], continuous monitoring in the form of performance improvement, mandatory upgrades, testing & debugging, and fixing bugs is necessary to ensure the quality of any QP [8, 9, 10, 11, 12, 13, 14, 15, 16].

Along with the aforementioned activities, maintenance of QPs is required to ensure their continuous availability. Research in classical software has shown that maintenance can be hindered by poor design and implementation choices due to developers' lack of experience and skills [17, 5]. Also, the programming practices in an emerging field like QC are evolving, and the related libraries and frameworks are undergoing significant revisions, making that QPs are mostly consisting of code that patches together the program. This led us to believe that it is high time for the software engineering research community to investigate techniques that can proactively help maintain QPs before these problems become widespread. To the best of our knowledge, the software engineering research community has not investigated the maintenance of QPs from a purely quantum perspective.

Prior research investigated indicators of maintenance issues (in particular, code smells) and methods to identify and quantify their impact on classical software [18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31]. Openja et al. [32] have recently studied the presence of generic smells in QC projects. However, none of these works focus on quantum-specific smells and only provide a partial picture of code smells in QC.

Classical and quantum computing are significantly different due to the intrinsic nature of QPs and their ability to have multiple possible states in superposition at the same time or entangled [33, 34]. Moreover, quantum logic is more complex than classical logic, which makes the synthesis of quantum circuits more challenging both in the algorithmic design process and during implementation [35]. Hence, we posit that defining code smells specific to QPs is essential. To the best of our knowledge, no one has yet investigated the prevalence of quantum-specific smells in QPs, prompting us to ask:

RQ1: How do practitioners perceive quantum-specific code smells?

RQ2: What is the prevalence of quantum-specific code smells in quantum programs?

To answer RQ1, we first identified eight quantum-specific smells extracted from the best practice in QC [36]. Then, and to shed some light on the perceptions of real-world developers' on the impact of these smells, we surveyed developers that have contributed to quantum open-source projects to assess their opinion on the set of quantum-specific smells. To answer RQ2, we built QSMELL, a tool to detect the eight quantum-specific smells in QPs. Finally, using the tool, we performed an empirical study on the prevalence of code smells in 15 QPs. This paper makes the following contributions:

- A catalog of eight novel smells tailored for quantum programs.
- A novel tool (coined QSMELL[1]) that automatically identifies whether a quantum-specific smell occurs in a given quantum program written in the Qiskit's full-stack library.
- An empirical evaluation of the prevalence of quantum-specific smells in 15 quantum programs.
- A detailed discussion on the implications of our findings in future research on quantum computing.
- A replication package of our study for others to use available at https://doi.org/10.5281/zenodo.7556360.

II. Related Work

Code smells were introduced by Fowler [19] to describe the design and implementation flaws in source code. These flaws do not necessarily make the software behave incorrectly or crash but make it harder to understand and maintain [28].

Generic Code Smells: Researchers have deeply investigated the impact of generic code smells in classical programs, e.g., how code smells impact fault-proneness and change-proneness [27, 28, 37], code smells' impact on maintainability [38, 39, 29, 30, 31], when and why code smells are introduced [21], and how code smells evolve over time [22, 23, 24, 25]. Openja et al. [32] have recently studied the presence of technical debts, i.e., generic smells and coding errors, in QC code. Their results showed that 80% of the technical debts are related to code smells, and more than half of technical debts are classified with major severity, implying that those debts significantly impact developers' productivity.

Domain-specific Code Smells: Besides studies on generic smells, researchers have also been developing and studying domain-specific code smells. For example, code smells for deep learning systems [20, 40], SQL [41], and security code smells for infrastructure as code scripts [42]. Due to the unique characteristics of QC code vs. classical software, we foresee that a domain-specific set of code smells for QC is required (as it has been for other specific domains) as a proxy to assess, e.g., long-term maintenance issues on QPs. This paper aims to fill the gap by proposing eight novel quantum-specific code smells. Furthermore, this paper is also the first to conduct an empirical analysis regarding the prevalence of quantum-specific code smells in QPs.

Code Smells Tools: Several techniques that include metric-based, machine learning-based, history-based, textual-based, search-based, and static analysis (the most popular) have been developed to detect generic code smells [38, 39, 43, 44, 45, 46, 47, 48, 49]. Since our work focuses on identifying code smells in QPs written in Python, we briefly discuss the code smell detection tools developed for Python. Omari et al. [50] and Bafatakis et al. [51] used Pylint to analyze Python code smells. Chen et al. [52] investigated the detection and prevalence of code smells using a tool named PySmell, which has been validated by others [20]. To the best of our knowledge, no tool aims specifically at identifying quantum-specific code smells in QPs. Our tool, QSMELL addresses this gap.

[1]QSMELL is officially available at https://github.com/jose/qsmell and also in the replication package in `tools/qsmell`.

III. Quantum Computing Background

Quantum computing is the generic term to identify the field of computer science that uses quantum mechanics principles to perform computations [53, 54].

Being governed by different physics laws, quantum information and computation are different from classical ones. In classical computation, the basic unit of information is the bit, which has two possible values, 0 or 1. In quantum computation, the basic information unit is the qubit, and it is generically represented by a superposition $|\phi\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle$ of two orthogonal physical states $|0\rangle$ and $|1\rangle$, where α_0 and α_1 are complex numbers. Note that a qubit can be, simultaneously, in both states $|0\rangle$ and $|1\rangle$, but one cannot directly observe such superposition.

To manipulate quantum information, one has to use quantum gates, i.e., unitary transformations with complex coefficients. Simple examples of common unitary gates are the `Not` gate (i.e., `Pauli-`X operator [55]) and the `Hadamard` gate [56] One can apply individual operations to single qubits, but one can also apply more general operators involving several qubits simultaneously.

Any quantum programming languages store classical and quantum information in classical and quantum registers. In practice, quantum computation is made through quantum circuits, a generalization of classical circuits. Therefore, a quantum program is expressed using unitary gates (describing unitary operations) applied in a specific order. In a quantum circuit, quantum gates play the same role as logic gates in classical circuits. The major advantage of representing quantum programs by quantum circuits is that there is no need to know the intrinsic details of the programming languages used to implement them, and it is independent of the programming language used. The execution of a quantum circuit is made through real quantum devices or simulators and entails the creation of basis states and applying the operations, in order, represented in the circuit. In the end, the output is measured and sorted in classical registers. We say the circuit has a fault if the probability distribution obtained by running the circuit multiple times does not correspond to the expected one.

IV. Quantum-specific Code Smells

In this section, we first describe the methodology to derive quantum-specific code smells for QC (Section IV-A), followed by the definition and a motivational example of each code smell (Sections IV-B to IV-D), and we finally answer **RQ1** on how practitioners perceive code smells in Section IV-E.

Table I: Brief summary of the proposed quantum-specific smells. We opt to rename the best practices to better fit the proper description of the (quantum) smell obtained from that practice.

Best Practice [36]	Smell Name	Acronym	Summary	Section
		Running a circuit in hardware		
"Getting a circuit to run on hardware"	Use of Customized Gates	CG	Any customized gate is decomposable into built-in operators of the framework. This decomposition requires a substantial higher number of operators when compared to the equivalent solution made exclusively of built-in operators.	IV-B1
"Using CircuitOperation to reduce circuit size"	Repeated set of Operations on Circuit	ROC	Due to technological and physical limitations, the number of operations one can pass to a quantum computer is limited, therefore the circuit implementing the whole algorithm should be prepared in such a way that the number of sequential repeated set of operations to be performed is the least possible.	IV-B2
		Inefficient circuits		
"Use sweeps when possible"	Non-parameterized Circuit	NC	Real devices work in a shared policy. To reduce communication payloads and avoid queuing for different initial values, the circuit should be designed parametrically to allow the different initial values to be provided at once.	IV-C1
		Erroneous circuits		
"Short gate depth"	Long Circuit	LC	Unitary gates and measurements are prone to errors (specially due to quantum noise). The higher the depth of the circuit and/or wider the circuit, the higher is the probability of affecting a quantum circuit's intended behavior.	IV-D1
"Terminal Measurements"	Intermediate Measurements	IM	Measurements affect the state of the entire system, making it prone to more errors. Therefore, measurements should be postponed to be the very last operation on the circuit to avoid error propagation.	IV-D2
"Keep qubits busy"	Idle Qubits	IdQ	With current technology it is only possible to ensure the correctness of a state for very short periods of time. Having idle qubits for too long enhance the loss of quantum information that may jeopardize the results of a quantum circuit.	IV-D3
"Delay initialization of qubits"	Initialization of Qubits differently from $\|0\rangle$	IQ	Keeping the coherence of a quantum excited state is technologically difficult. Hence, initially one should keep it in its ground state (i.e., in state $\|0\rangle$) as long as possible.	IV-D4
"Qubit picking"	No-alignment between the Logical and Physical Qubits	LPQ	The topology of real qubits impacts the circuit behavior, i.e., the results obtained from the circuit can change according to the physical qubits configuration. Therefore, not aligning the logical qubits to the proper physical qubits may lead to less accurate results.	IV-D5

A. Mapping Best Practices to Code Smells

To derive the set of quantum-specific smells in QC, we qualitatively analyzed the best practices proposed by Google Cirq's team [36] and categorized them into: *Running a circuit in hardware*, *Inefficient circuits*, and *Erroneous circuits*. Then, three authors of the paper independently identified a code smell that violates a corresponding best practice. For example, one of the best practices mentioned in the sub-section "Using Circuit Operation to reduce circuit size" in [36]:

> *Particularly large batches (or sweeps) of circuits may encounter errors when sent to Quantum Engine due to an upper limit on request size. If the circuits in question have a repetitive structure, cirq.CircuitOperations can be used to reduce the request size and avoid this limit.*

The recommendation is not to use a "repetitive structure" in the quantum circuit to avoid hitting the request size limit. The code smell is therefore, the use of repetitive structure (e.g., operations or sets of operations). Table I shows the mapping of each smell and the corresponding best practice. To ensure the reliability of the mapping, each smell underwent an agreement process, and in case of uncertainty and disagreement, we discussed it until we reached a consensus. We finally ended up with a set of eight code smells. We call the reader's attention to the fact that the smells we propose are generic best practices and therefore can be considered and implemented for any quantum framework including, for example, Cirq, Qiskit, or Forest. In this paper, we, however, focus mainly on programs written in Qiskit as it is one of the most frequently used frameworks in research [13, 15, 57, 58, 59, 60] and the most popular quantum framework [61]. The following subsections explain each quantum-specific smell in detail.

In order to have a uniform drawing representation of each quantum circuit used in the remaining of the paper, we opt to present the *transpile* [62] version of each circuit. *Transpilation* is the process of converting a given quantum circuit into instructions supported by current quantum devices.

B. Running a circuit in hardware

1) Use of Customized Gates (CG): Quantum frameworks are very versatile and allow one to define and run customized gates unavailable in the framework. It has been, however, recommended that developers should only use built-in gates for two reasons. First, before running the quantum circuit, any customized gate must be converted into a sequence of built-in gates that often require more operators than equivalent solutions consisting exclusively of built-in operators. Second, customized gates (not optimized by design as built-in gates) increase the probability of circuit errors. Thus, the recommendation is to avoid the use of customized gates.

Consider the example presented in Listing 1[2] illustrating a possible customized implementation of a `Toffoli` gate which leads to the circuit represented in Figure 1 and its corresponding transpiled version in Figure 2a.

[2]Note that the purpose of each *listing* is to highlight one specific smell. It is, however, conceivable that more than one smell may be present in a listing. We tried to keep the number of smells per listing to one to avoid confusion, but it was not always possible. To further help the reader to identify the main smell represented in each listing, we highlighted (in red and green) the lines that correspond to the main smell.

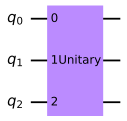


Figure 1: CG smell circuit.

If, instead the built-in `Toffoli` gate is used (i.e., a `CCNot` gate corresponding to the `ccx` gate in Listing 1 on line 13), which is directly implemented in the platform using `CNot` and rotations operations, we would have the transpiled circuit represented in Figure 2b. Compared with the former, this has lower depth and fewer gates and therefore less error prone (see Section IV-D1 for details).

Listing 1: Implementation of a `Toffoli` gate by writing its describing unitary matrix highlighted with lines in red. The alternative smell-free lines are colored in green and uses a (optimized) built-in implementation of the same gate. (See Figure 2a for a translation of the code into operations to be run in a real device.)

```
1    from qiskit import QuantumCircuit, transpile
2    qc = QuantumCircuit(3)
3  - qc.unitary([
4  - [1, 0, 0, 0, 0, 0, 0, 0],
5  - [0, 1, 0, 0, 0, 0, 0, 0],
6  - [0, 0, 1, 0, 0, 0, 0, 0],
7  - [0, 0, 0, 1, 0, 0, 0, 0],
8  - [0, 0, 0, 0, 1, 0, 0, 0],
9  - [0, 0, 0, 0, 0, 1, 0, 0],
10 - [0, 0, 0, 0, 0, 0, 0, 1],
11 - [0, 0, 0, 0, 0, 0, 1, 0]
12 - ], [0, 1, 2])
13 + qc.ccx(0, 1, 2)
```

Metric: Number of occurrences of the `UnitaryGate` [63], `HamiltonianGate` [64], or `SingleQubitUnitary` [65] gates invoked with a matrix as input.

Detection: A program is said to have this smell if its CG metric value is greater than or equal to 1.

2) Repeated set of Operations on Circuit (ROC): Due to hardware limitations, the number of gates that one can send to real devices is limited [66]. Hence, when preparing a circuit to run on a quantum computer, one should minimize the set of instructions using built-in operators such as `repeat` [67]. This operator allows to reduce the circuit size and preserve its structure when serialized (e.g., when it is sent to a backend to be executed on a real quantum computer). Consider the example in Listing 2, which leads to the circuit represented in Figure 3, where the for loop (lines 7-12) contains a set of operations sent to the device in each iteration. This drastically increases the circuit's size (18 operations in total). On the other hand, the use of the `repeat` [67] operator avoids the repetitions of the operations in the circuit and produces an equivalent circuit three times smaller.

Listing 2: Example of a circuit with a repeated set of operations (highlighted in red) and its alternative (i.e., smell-free highlighted in green) version which uses a built-in operation to avoid such repetitions.

```
1    from qiskit import QuantumCircuit
2    qc = QuantumCircuit(3, 3) # 3 Quantum and 3 Classical
   registers
3    hadamard = QuantumCircuit(1, name='H')
4    hadamard.h(0)
5    measureQubit = QuantumCircuit(1, 1, name='M')
6    measureQubit.measure(0, 0)
7  - for i in range(3):
8        for j in range(3):
9            qc.append(hadamard, [j])
10       for j in range(3):
11           qc.append(measureQubit, [j], [j])
12       qc.barrier()
13 + qc.repeat(3)
```

Metric: Number of sequential repeated sets of operations.

Detection: A program is said to have this code smell if its ROC metric value is greater than or equal to 1.

C. Inefficient execution of a circuit

1) Non-parameterized Circuit (NC): In QP, it is typical to run a circuit several times, each time with different parameters. Therefore, it is recommended to minimize the number of times instructions and results are communicated between the backend and the client. This can be achieved by binding the values to the circuit as parameters and run the circuit, containing all values only once in the backend. Doing so reduces the payload associated with the number of communications between the backend and the client, which makes the process more efficient. Furthermore, a positive side-effect of avoiding this smell is reducing compilation time [68].

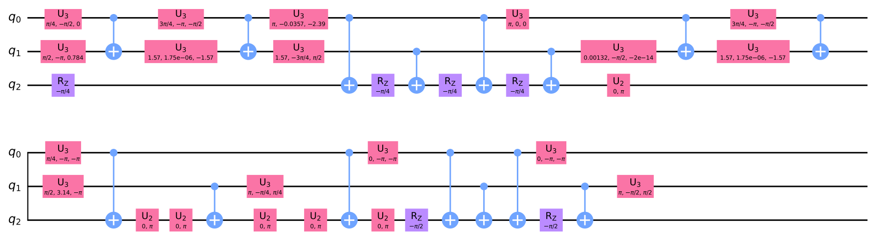
Listing 3 (adapted from [69]) shows an example of optimizing the execution of circuits with real quantum devices through a parameterized circuit. Instead of initializing the circuit with different values of theta and sending each circuit to the backend, we can bind theta values to the circuit as parameters and run this circuit, containing all values only once in the server backend. For QPs written in Qiskit, for example, to evaluate the existence of this smell, one can identify the number of methods that specifically run the code in a real quantum device, e.g., `run` [70] or `execute` [71].

Listing 3: Example of a program to compute results of a circuit depending on a parameter value theta. Smelly lines are highlighted in red and alternative smell-free lines are highlighted in green.

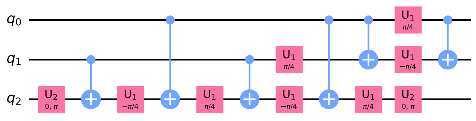
```
  from qiskit import QuantumCircuit, Aer
+ from qiskit import transpile
+ from qiskit.circuit import Parameter
+ theta = Parameter('0')
  def init_circuit(theta):
      qc = QuantumCircuit(5, 1)
      qc.h(0)
      for i in range(4):
          qc.cx(i, i+1)
    qc.barrier()
    qc.rz(theta, range(5))
    qc.barrier()
    for i in reversed(range(4)):
        qc.cx(i, i+1)
    qc.h(0)
    qc.measure(0, 0)
    return qc

  theta_range = [0.00, 0.25, 0.50, 0.75, 1.00]

- for theta_val in theta_range:
-     qc = init_circuit(theta_val)
-     backend = Aer.get_backend('qasm_simulator')
-     job = backend.run(qc)
-     job.result().get_counts()
+ qc = init_circuit(theta)
+ circuits = [qc.bind_parameters({theta: theta_val})
+             for theta_val in theta_range]
+ backend = Aer.get_backend('qasm_simulator')
+ job = backend.run(transpile(circuits, backend))
+ job.result().get_counts()
```

Metric: Number of calls to execution methods such as `run` [70] or `execute` [71] minus the number of calls to the `bind_parameters` [72] method.

Detection: A program is said to have this code smell if its NC metric value is greater than or equal to 1. In other words, if the number of calls to execution methods such as `run` or `execute` is higher the number of



(a) Smelly *transpiled* circuit shown in Figure 1.



(b) Smell-free circuit.

Figure 2: Example of a customized Tofolli gate implementation in Qiskit with the automatic transpile to one- and two-qubit native gates corresponding to Listing 1.

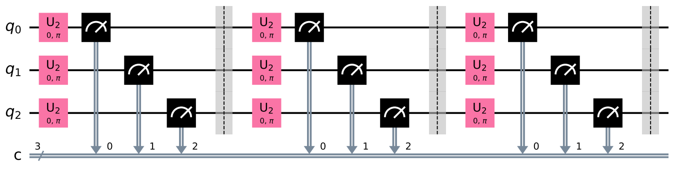


Figure 3: Example of a quantum circuit with the repeated set of operations on circuit smell. The initial part of the circuit, i.e. up to the barrier ⁞ is the set of operations repeated twice afterwards.

calls to the `bind_parameters` method. For example, a program that calls twice the `run` method and once the `bind_parameters` method is said to have this smell.

D. Erroneous circuits

1) Long Circuit (LC): Unitary gates in quantum circuits and measurement devices are prone to errors due to imperfections in the devices and especially due to noise [73]. Although quantum hardware is becoming more accurate, it still has significant error rates [66]. Therefore, if the circuit has high depth (i.e., a high number of gates applied to the same qubit) and/or high width (i.e., gates applied simultaneously to a high number of qubits), then the result of the execution of such circuit might be incorrect as the total amount of error may be too high.

Listing 4 illustrates a very well-known identity example within the quantum community related to Pauli operators, $HZH = X$, which is applying a *not* operation to a qubit that can be used to simplify the circuit (see [53] for more details). The sequence of gates `Hadamard` [56], `Pauli-Z` [74], and `Hadamard` [56] (lines 3-5) can be described solely by a `Pauli-X` [55] gate (line 6). This example illustrates the possibility of reducing the number of circuit operations.

Listing 4: Example of a long (i.e., more erroneous) circuit highlighted in red and its equivalent and short (i.e., less erroneous and therefore smell-free) version highlighted in green.

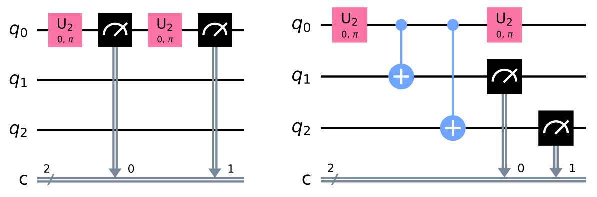
```
1    from qiskit import QuantumCircuit
2    qc = QuantumCircuit(1)
3  - qc.h(0)
4  - qc.z(0)
5  - qc.h(0)
6  + qc.x(0)
```

Metric: Likelihood of a circuit not having any error as $(1 - \text{error})^{l \cdot c}$, whereas error is the maximum error of any active gate of the real device that is used to run the circuit and l is the maximum number of operations in any qubit and c is the maximum number of parallel operations in the circuit. For example, in August 2022, the maximum error value on IBM's Kolkota quantum machine was 0.03512.

Detection: A program is said to have this code smell if its LC metric value is lower than a threshold value. In other words, the lower the value, the more likely the quantum circuit's output is to be incorrect. Although in practice (see Section VI-B) we observed that quantum programs have a median LC value of 0.39 and an average of 0.43, the majority of the results are incorrect because they are not producing accurate results. In order to be possible to run the circuit multiple times to get some valid results (for example, using majority voting [75] as a reasonable validation rule), we need to have a threshold of at least 0.50 (i.e., 50%). Hence, any program with a LC metric value lower than the threshold (i.e., 0.50) is considered smelly.

2) Intermediate Measurements (IM): Observing (i.e., performing a measurement) a qubit among several qubits used in a circuit has an impact on the state of the remaining qubits (specially if the operations entangle the measured qubit with other ones). These interactions with the physical hardware containing the quantum information may introduce undesirable errors. In quantum computation, there exists a principle called the *deferred measurement principle* that allows the construction of a circuit that is equivalent to the original circuit but where all the measurements are postponed to the very last operation in the circuit (see Section 4.4 of Reference [53] for a detailed description of this principle). This principle applied in practice means one can postpone the measurements to the last moment of the circuit execution. Therefore, a circuit should avoid intermediate measurements and alternatively should be designed to measure all qubits with a single gate or by adding the measurement gate after all optimizers have run, i.e., as the last operation of the circuit.

Listing 5, which leads to Figure 4, is presented the code to implement two truly random bits. The smelly code contains an intermediate measurement in line 11 and then immediately applies the `Hadamard` [56] operation (line 12) to one of the qubits that was measured (i.e., the first qubit). To avoid having this intermediate measurement, one can, for example, simulate the behavior of the intermediate measurement by storing the result in the second qubit by applying a `C-Not` (line 9) involving qubit 0 and qubit 1 and then undoing the `Hadamard` and then measuring the qubit 1. Notice that one can keep the original state of the first qubit by also simulating



(a) Smelly. (b) Smell-free.

Figure 4: Example of a circuit with the IM smell (left) where one intermediate measurement is performed between the `Hadamard` gates, and the equivalent circuit (right) that does not have the smell.

the final measurement (line 12) by applying a `C-Not` (line 10) involving qubit 0 and qubit 2 and then measuring the qubit 2.

Listing 5: Example of a circuit to compute two random bits with an intermediate measurement (smelly) highlighted in red and its equivalent smell-free version highlighted in green. The smelly version uses qubit 0 to perform twice `Hadamard` and measurement of the qubit to store in two different classical registers the two results, while the smell-free version uses `Hamadard` and two `C-Not`'s from the first to the two last qubits to simulate both intermediate (the smelly one) and final measurement.

```
1   from qiskit import QuantumRegister, ClassicalRegister,
    QuantumCircuit
2
3   qreg_q = QuantumRegister(3, 'q')
4   creg_c = ClassicalRegister(2, 'c')
5   qc = QuantumCircuit(qreg_q, creg_c)
6
7   qc.h(qreg_q[0])
8  - qc.measure(qreg_q[0], creg_c[0])
9  + qc.cnot(qreg_q[0], qreg_q[1])
10 + qc.cnot(qreg_q[0], qreg_q[2])
11  qc.h(qreg_q[0])
12 - qc.measure(qreg_q[0], creg_c[1])
13 + qc.measure(qreg_q[1], creg_c[0])
14 + qc.measure(qreg_q[2], creg_c[1])
```

Metric: Number of non-terminal measurements.

Detection: A program is said to have this code smell if its IM metric value is greater than or equal to 1.

3) Idle Qubits (IdQ): With current technology, keeping the quantum information in its correct state is only possible for a short period [66]. Therefore, the lifetime of a qubit is limited and being idle leads to dephase and decoherence, which leads to a decrease of accuracy of the results. To avoid this problem, if some qubit is idle for a long period of time, one should apply twice an idempotent operator like a Pauli Y-gate to force the qubit to be active and keep its coherence. Notice that if we apply the operator Y twice to a qubit, it is the same as applying the identity operator (idempotent), i.e., the operator that leaves the qubits as they are.

Listing 6 represents a circuit involving several operations (`Hadamard`, `Pauli-Z`, `Phase rotations`) over a few qubits. It first applies the `Hadamard` operator (in Figure 5, this operator is shown by its general representation $U_2(\phi, \lambda)$ [76] with $\phi = 0$ and $\lambda = \pi$ where U_2 is a $X + Z$ axis rotation) to all the qubits (line 6) and then, after all the remaining operations are done, measures all the qubits at the end (line 23). For example, the number of operations between the last unitary operation applied to the 1st qubit (line 6) and the final measurement (line 23) is 8. So, by the time the quantum computer measures the 1st qubit, its state may have been changed. On the other hand, in the smell-free version, the measurement is done right after all the operations involving the 1st qubit. Figure 5 shows the smelly and smell-free circuits after fixing the IdQ smell.

Listing 6: Example of a circuit with idle qubits (smelly) highlighted in red and its equivalent smell-free version highlighted in green.

```
1   from qiskit import QuantumRegister, ClassicalRegister,
    QuantumCircuit
2   from numpy import pi
3   qreg_q = QuantumRegister(3, 'q')
4   creg_c = ClassicalRegister(3, 'c')
5   qc = QuantumCircuit(qreg_q, creg_c)
6  - qc.h(qreg_q)
7  + qc.h(qreg_q[0])
8   qc.p(pi / 2, qreg_q[0])
9   qc.z(qreg_q[0])
10  qc.s(qreg_q[0])
11 + qc.measure(qreg_q[0], creg_c[0])
12  qc.barrier()
13 + qc.h(qreg_q[1])
14  qc.p(pi / 4, qreg_q[1])
15  qc.z(qreg_q[1])
16  qc.s(qreg_q[1])
17 + qc.measure(qreg_q[1], creg_c[1])
18  qc.barrier()
19  qc.h(qreg_q[2])
20  qc.p(pi / 8, qreg_q[2])
21  qc.z(qreg_q[2])
22  qc.s(qreg_q[2])
23 - qc.measure_all(add_bits=False)
24 + qc.measure(qreg_q[2], creg_c[2])
```

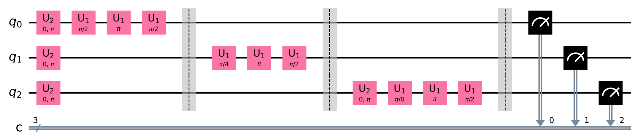
Metric: Maximum number of circuit operations between one operation using a qubit and the subsequent operation where that qubit is used again.

Detection: A program is said to have this code smell if its IdQ metric value is higher than a threshold value. The exact threshold calculation mechanism is explained in Section VI-B.

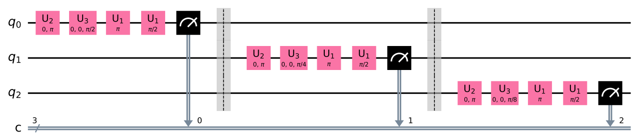
4) Initialization of Qubits (IQ): It is known that qubits in an equilibrium state, usually $|0\rangle$, are more robust than qubits in an excited state, usually $|1\rangle$ or a uniform superposition $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. From a practical point of view, this means that a qubit in its initial state $|0\rangle$ may be accurate for longer than in any other state. Therefore, if any qubit needs to be set to a particular initial state, this should only happen before a gate is applied to it. In other words, if a qubit is initialized at the beginning of the circuit but a gate is only applied to that qubit after several other gates were applied to other qubits, the intended state of the first qubit might no longer be the expected one. The smell IQ differs from the smell IdQ since the former only refers to the difference between the first and the second time a qubit is used, and therefore one can consider it a particular case of IdQ.

We use Listing 6 once more to illustrate the smell. Notice that the Hadamard operation initializes the 3rd qubit in line 6. However, this qubit, is only used again in line 19, which corresponds to 7 operations distance between initialization and actual use. As in the previous smell description, by the time the quantum computer performs the second operation on the 3rd qubit, the state of that qubit may have already changed. On the other hand, in the smell-free version, the first Hadamard is applied right before the operations involving the 3rd qubit are performed.

Metric: Maximum number of operations performed in the circuit between the initialization of any qubit (usually the



(a) Smelly.



(b) Smell-free.

Figure 5: Example of a circuit with the IdQ smell (left) and the equivalent circuit without the smell (right).

first operation applied to the qubit) and the second operation applied to the same qubit.

Detection: A program is said to have this code smell if its IQ metric value is higher than a threshold value.

5) No-alignment between the Logical and Physical Qubits (LPQ): The design of quantum memories with current technology has asymmetries among the qubits. This means that the geometry of physical qubits, their interaction, and their readout error affect the results. As stated in Qiskit's documentation [77] *"the choice of initial layout can mean the difference between getting a result, and getting nothing but noise"*. Therefore, depending on the device that the code is executed on, choosing which physical qubit should be picked for which logical qubit in the code/circuit is an essential operation to achieve more accurate results. For example, in Qiskit, this can be achieved by using the parameter `initial_layout` in the `transpile` method [62].

The smelly version in Listing 7 transpiles the circuit using a default layout. However, one could configure her layout to produce the most efficient results and a smell-free version. To do so, one can pass a list of integers to `qiskit.compiler.transpile()` via the `initial_layout` keyword argument, where the index labels the virtual qubit in the circuit and the corresponding value is the label for the physical qubit to map onto. For example, mapping virtual qubits 0, 1, and 2 to physical qubit 3, 4, and 2 (as described in line 11).

Listing 7: Example of a circuit that does not properly align logical and physical qubits represented with lines colored in red and its equivalent smell-free version represented with lines colored in green where a explicit correspondence between logical and physical qubits is provided as a initial layout.

```
1   from qiskit import QuantumCircuit, transpile
2   from qiskit.providers.fake_provider import FakeVigo
3   backend = FakeVigo()
4   qc = QuantumCircuit(3, 3)
5   qc.h(0)
6   qc.cx(0,range(1,3))
7   qc.barrier()
8   qc.measure(range(3), range(3))
9  - qc = transpile(qc, backend)
10 + qc = transpile(qc, backend,
11 + initial_layout=[3, 4, 2])
```

Metric: Number of calls to the `transpile` method without the parameter `initial_layout` set.

Detection: The `transpile` method is invoked by Qiskit by default. In that case, the choice of mapping depends on the properties of the circuit, the target device, and the chosen optimization level. Thus, we label a program with this code smell if the `transpile` method is explicitly invoked and its LPQ metric value is greater than or equal to 1.

Table II: Survey results per smell.

	CG	ROC	NC	LC	IM	IdQ	IQ	LPQ	*Average*
Agree	25.71%	34.29%	60.00%	71.43%	31.43%	82.86%	42.86%	71.43%	52.50%
Disagree	45.71%	25.71%	28.57%	20.00%	57.14%	11.43%	22.86%	22.86%	29.29%
Do not know	28.57%	40.00%	11.43%	8.57%	11.43%	5.71%	34.29%	5.71%	18.21%

E. RQ1: How do practitioners perceive quantum-specific code smells?

1) Subjects: To validate our defined code smells, we surveyed a larger population of quantum developers. We focused on quantum developers who had contributed to the curated set of 21 active quantum projects (the detailed process of curating this list of projects is explained in Section VI-A). We used the GitHub API [78] to mine contributor emails from these 21 projects. After removing emails of accounts that were no longer active, we were left with 470 developer email addresses.

2) Survey: Our survey comprised 16 questions, a mix of multiple-choice and open-ended questions, which can be found in the replication package in `survey/survey-doc.pdf`. The survey included demographic questions and participant experiences with quantum programming. Then, we showed the participants a quantum code snippet and its circuit draw and asked them to assess whether the snippet was affected by a specific code smell. We asked one question per code smell (a total of eight), and at the end, we asked about participants' perceptions of the severity of these code smells.[3] We conducted three pilot studies with graduate students and professionals with quantum experience. After each pilot study, we collected feedback and refined the survey based on the feedback.

3) Procedure: We used Qualtrics [79] as a distribution platform to deploy our survey. We emailed the survey to 470 developers (following university-approved IRB protocol), and 5 emails bounced (giving 465 valid emails). The survey was open for two weeks, during which we received 35 responses or a response rate of 7.53%. These response rates are consistent with other studies in software engineering [80]. We quantitatively analyzed the closed-ended questions to understand developers' perceptions of the definition and example of the shown code smells.

4) Results: Most of our respondents had 2 to 20 years of programming experience. In addition, the participants had 1 to 5 years of quantum programming experience, with a median of 2 years of quantum programming experience.

Table II reports the percentage of participants that *agreed*, *disagreed*, or *did not know* whether a shown example was affected by a quantum-specific smell. On average, 52.50% of

[3]Note that we did not ask participants for additional smells because that would require additional interviews and surveys to understand and validate the new smells.

Table III: Quantum-specific smell's severity. Severity ranges from 1 (the most severe) to 8 (the least severe). 10 out of 35 participants ranked LC as being the most severe (rank 1) and 17 participants ranked the CG smell as being the least severe (rank 8). Taking the *median* of the rank positions given by all participants to all smells, we have, from the most severe smell to the least, the following order: LC (2.50), NC, IdQ, and LQP (3.00), ROC and IM (4.00), and CG.

	CG	ROC	NC	LC	IM	IdQ	IQ	LPQ
Average	5.82	3.59	4.15	2.62	4.76	3.85	4.44	3.41
Median	7.00	4.00	3.00	2.50	4.00	3.00	5.00	3.00

all participants agreed that each shown example was affected by a quantum-specific smell. Contrary to the authors of this paper that derived the quantum-specific smells, 1/4 of all participants did not agree on CG and nearly 1/3 on IM. One interpretation is that participants perceived the smells as unharmful because they are used to seeing this style of programming / quantum circuits. Regarding the CG smell, one participant that did not agree mentioned that

> *"Using customized gates may be the only possible route for some applications, but if you have a better construction, that should be used, of course."*

Regarding the IM smell, one participant mentioned that

> *"It is not a bad practice. If your algorithm requires it, use it. Else, do not use it. Try to limit their number, but if you need a C-NOT, you use a C-NOT, your choice."*

Interestingly, 40% of all participants did not know whether the example with the ROC smell was or was not smelly. We hypothesize that identifying a sequential set of repeated operations on a quantum circuit drawing was not easy for the participants, leading them to select a more conservative answer (*"do not know"*).

We also had interesting comments that support the other quantum-specific smells we derived, for example, *"Things like ROC and NC are the most 'serious' to resolve."* and *"Aligning the physical and logical qubit topologies is a good idea."*

Additionally, we also asked the participants about their perceptions of the severity of the smells. LC, NC, IdQ, and LPQ smells were identified as the top-3 most severe. Details of the severity reported by participants are shown in Table III.

V. QSMELL

To identify the list of curated smells in code written in Qiskit, we developed a tool called QSMELL. In this section, we first describe QSMELL's operating modes and then describe the evaluation conducted to assess QSMELL's effectiveness at detecting quantum-specific code smells. Instructions on how to install QSMELL and usage examples are provided at https://github.com/jose/qsmell and in the replication package in `tools/qsmell/README.md`.

A. Code Analysis

Code smell tools that other have proposed (e.g., SLIC [42] and PyNose [81]) mostly perform static analysis to detect whether a code smells occurs in a piece of code. QSMELL, on the other hand, depending on the smell metric either performs a dynamic (preferably) or static analysis.

The metrics of CG, ROC, LC, IM, IdQ, and IQ that rely on an accurate set of qubits and/or set of operations performed in the circuit, are computed using dynamic analysis. For these smells, a static analysis would not be able to handle common code as loops or to track objects passed as arguments to other functions. For example, in Listing 3 (lines 8-9), a static analysis approach will not be able to detect to which qubit is applied the `cx` gate. The metrics for the other smells, i.e., NC and LPQ, which do not rely on the quantum circuit but the actions performed on the circuit, e.g., calls to methods that are not part of a quantum circuit object (e.g., `transpile`, or Qiskit backends' methods) cannot be computed with the same dynamic analysis and are therefore computed using static analysis.

1) Dynamic Analysis: To perform a dynamic analysis on a QP, QSMELL takes as input an *execution* matrix, whereas each row represents a quantum or classical bit, each column represents a timestamp in the circuit, and each cell represents a quantum operation performed in the circuit. Given, for example, the QP in Listing 3, one would have to inject the following piece code

```
from quantum_circuit_to_matrix import qc2matrix
qc2matrix(qc, output_file_path='example-matrix.csv')
```

and run it to generate the *execution* matrix. Note that the `quantum_circuit_to_matrix` module was built by us on top of Qiskit's API and is part of the QSMELL distribution. In a nutshell, the module first collects the set of qubits from the `qc` object's data and then iterates over all operations performed in each qubit. The *execution* matrix for the example in Listing 3 is shown below

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
q0	u2(int,float)	cx()				barrier()	rz(float)	barrier()				cx()	u2(int,float)	measure()
q1		cx()	cx()			barrier()	rz(float)	barrier()			cx()	cx()		
q2			cx()	cx()		barrier()	rz(float)	barrier()		cx()	cx()			
q3				cx()	cx()	barrier()	rz(float)	barrier()	cx()	cx()				
q4					cx()	barrier()	rz(float)	barrier()	cx()					
c0														measure()

Once the *execution* matrix has been generated, to compute, e.g., the LC metric, QSMELL first computes the maximum number of operations in any qubit (any row in the matrix), i.e., six in the *execution* matrix above. It then computes the maximum number of operations performed simultaneous (in the same timestamp), i.e., five at the seventh timestamp. Finally, the LC metric value is $(1 - 0.03512)^{6*5} = 0.34$. As this value is lower than the threshold value 0.50 (see Section IV-D1), we can say the LC smell is present in Listing 3. Recall that lower values of LC mean the circuit is more prone to be affected by a gate error.

2) Static Analysis: As the information of the quantum backend (see lines 29-31 in Listing 3, for example) is not kept in the quantum circuit object itself, QSMELL performs a static analysis for smell metrics NC and LPQ. It takes a source code `.py` file and analysis it using Python AST [82]. For instance, to compute the LPQ metric for the example in Listing 3, QSMELL first finds all calls to the `transpile` method in the program's under analysis AST and counts how many do not define the `initial_layout` parameter. As there is one call to the `transpile` method without the `initial_layout` parameter being set, the LPQ metric is 1 and therefore, smelly.

Table IV: Rater A/B vs. QSmell. Values in **bold face** represent disagreement between rater A/B vs. QSmell. Values highlighted in gray point out the values that are above the threshold and hence exhibit the smell.

Name	CG	ROC	NC	LC	IM	IdQ	IQ	LPQ
Metric values assigned by human raters								
(qiskit-machine-learning) qsvc	0	1	0	0.49	0	1	0	0
(qiskit-terra) fae	0	4	1	0.10	0	0	0	0
Metric values assigned by QSmell								
(qiskit-machine-learning) qsvc	0	1	0	0.49	0	1	0	0
(qiskit-terra) fae	0	4	**2**	0.10	0	0	0	0

B. Tool Evaluation

We conducted an evaluation on the effectiveness of QSmell at correctly computing quantum-specific smell metrics and then at detecting them. As no existing dataset contains information for quantum smells, we constructed our validation *oracle* dataset by applying closed coding [83], where a rater identifies a pre-determined pattern. Other researchers have used this approach while preparing a validation dataset for other code smell detection tools [42, 81].

We start by randomly selecting 10% of the programs collected in Section VI-A, i.e., a total of two programs: qsvc and fae. Then two raters (i.e., authors of the paper but not developers of QSmell) equipped with the (1) definitions of each smell along with its threshold, and (2) the program's source code and its quantum circuit (draw and *execution* matrix), manually computed each smell metric and labeled the programs with the smells they exhibit, if any. To ensure an unbiased annotation process, the raters individually labeled programs and discussed their results afterward to reach a consensus. Finally, we also compared the raters' data against QSmell output on the same set of programs.

1) Rater A vs. Rater B: Once both raters finished their task, metric values and labels of raters were calculated using the inter-rater reliability Cohen's Kappa (k) [84]. Regarding metric values, raters agreed on 87.5% of the values with an inter-rater reliability k of 0.78, which indicates an *almost perfect* agreement. There was one disagreement for the ROC smell in both programs. Rater A computed the value 2 for qsvs and the value 5 for fae. Rater B computed the value 1 for qsvs and the value 1 for fae. Raters met and reviewed them together, and for each program, the raters discussed their respective reasoning and the source of disagreement. In this experiment, the root cause of the disagreement was due to whether the first occurrence of the repeated set of operations should be counted as repeated. Both raters agreed that it should not count, i.e., if a set of operations appears three times in the quantum circuit, only two are considered a repetition. This further clarified the ROC's metric definition and achieved an agreement of 100% with an inter-rater reliability k of 1, indicating a *perfect* agreement. Table IV (top) reports the metric values and the labels computed and assigned by both raters for each program per smell. Regarding labels (i.e., whether a program exhibits a code smell), raters agreed on 100% of all smells. LC, NC, and ROC are present in fae; LC and ROC are present in qsvs.

2) Rater A/B vs. QSmell: Next, we ran QSmell on the same set of programs and compared our results against the oracle. Table IV reports the metric values and the labels computed and assigned by QSmell for each program per smell. Regarding metric values, rater A/B and QSmell agreed on all values but the NC value for fae, 1 vs. 2. Although there are indeed two calls to the `execute` method in the fae program, one is in an `if` and the other one is the correspondent `else`. Thus, at runtime, only one would be executed. As the threshold value for the LC metric is 1, such disagreement did not affect QSmell's labeling of the LC smell. Regarding labels (i.e., whether a program exhibits a code smell), first, no occurrences of the CG, IM, IQ, and LPQ smells exist in the oracle dataset or QSmell's output. Second, QSmell achieved 100% precision, recall, and F1 on all smells. Such high precision/recall is due to QSmell's dynamic analysis, which reliable computes the required data for six out of the eight smells.

VI. Empirical Evaluation

In this section, we aim to answer the following research question using QSmell:

RQ2: What is the prevalence of quantum-specific code smells in quantum programs?

A. Experimental Subjects

To select representative quantum projects, we started by doing a keyword search using the GitHub search API. We searched for projects with a description containing the word "quantum computing". We scoped our search to projects written in Python and that use the Qiskit library by searching for projects containing the words "qiskit" and "import qiskit". This choice was due to two main reasons: (1) Qiskit is one of the most popular quantum frameworks [61] and one of the most frequently used frameworks in research [13, 15, 57, 58, 59, 60], and (2) QSmell only supports Qiskit's API. The search procedure returned 628 quantum projects.

Next, we further filtered the projects following the guidelines proposed by Kalliamvakou et al. [85]. Specifically, we selected projects based on two criteria: number of commits in 2022 and number of contributors. (1) We filtered out projects with less than 100 commits in total in 2022, to ensure we only elect projects with sufficient development activity that have not been abandoned or considered inactive [35]. (2) We discarded projects with just one contributor to avoid selecting toy projects or students assignments. The two criteria, when applied, resulted in 21 projects (which can be found in the replication package in `subjects/README.md`). (3) We manually inspected the 21 projects and discarded those related to documentation, lecture notes, and hardware platforms. We ended up with a total of three projects: qiskit-machine-learning, qiskit-terra, and qiskit-nature, which are umbrella projects containing multiple QPs. (4) We then collected all QPs (15 in total) available in these three projects. The list of the 15 programs and some metrics per program can be found in the replication package in `subjects/README.md`. On average, the programs have 229 lines of code, 4 qubits, 1 classical bit, and 1130 operations.

Table V: Quantum-specific smells. Values highlighted in gray point out the values that are above the threshold for CG, ROC, NC, IM, IdQ, IQ, and LPQ or below the threshold for the LC smell, and therefore exhibits the smell.

Name	CG	ROC	NC	LC	IM	IdQ	IQ	LPQ
(qiskit-machine-learning) qgan	0	0	0	0.75	0	0	0	0
(qiskit-machine-learning) vqc	0	2	0	0.39	0	1	0	0
(qiskit-nature) adapt_vqe	0	7	0	0.00	0	19	5	0
(qiskit-nature) qeom	0	0	0	0.01	0	19	0	0
(qiskit-terra) ae	0	49	2	0.00	0	821	0	0
(qiskit-terra) grover	0	1	4	0.22	0	2	1	0
(qiskit-terra) hhl	0	0	0	0.75	0	1	1	0
(qiskit-terra) iae	0	4	3	0.22	0	0	0	0
(qiskit-terra) ipe	0	0	4	0.96	0	0	0	0
(qiskit-terra) mlae	0	15	2	0.01	0	0	0	0
(qiskit-terra) phase_estimation	0	0	1	0.96	0	0	0	0
(qiskit-terra) qaoa	0	0	0	0.81	0	0	0	0
(qiskit-terra) shor	0	0	2	0.00	0	15650	15650	0
(qiskit-terra) vqd	0	0	0	0.70	0	0	0	0
(qiskit-terra) vqe	0	0	0	0.70	0	0	0	0
Median	0.00	0.00	0.00	0.39	0.00	0.00	0.00	0.00
Average	0.00	5.20	1.20	0.43	0.00	1100.87	1043.80	0.00
Threshold	1.00	1.00	1.00	0.50	1.00	0.00	0.00	1.00

B. *Experimental Procedure*

We first executed QSmell on 15 of the 17 programs and computed all quantum-specific code smells metrics. Note that two programs were used to evaluate QSmell's effectiveness in Section V-B and therefore discarded from the empirical evaluation. Then, as there is no prior work based on which we could define the thresholds for QPs, we have (1) used the already defined thresholds in Section IV for smells CG, ROC, NC, LC, IM, and LPQ; and (2) calculated the median of IdQ and IQ smell metrics on all programs and used that as the threshold of each metric. Finally, for each smell metric, we identified which programs have a smell metric value higher than the threshold as being smelly. Exception for the LC smell metric, where a program has the LC smell when the metric value is lower than the threshold.

C. *Results*

Table V reports the quantum-specific smells found on all QPs. Overall, at least one smell occurs in 11 out of 15 programs (73.33%), and the ratio of smells in a program is 26.67% (i.e., a program has, on average, 2.13 smells). shor has the highest number of smells (ROC, NC, LC, IdQ, and IQ), ipe and phase_estimation has the lowest number of smells (NC), and no smell was detected on qaoa, qgan, vqd, and vqe. Regarding individual quantum-specific smells, LC occurs in most programs (8), followed by NC and IdQ (7), ROC (6), and IQ (4). The CG, IM, and LPQ do not occur in any program.

1) Occurring code smells: LC code smell is related to the error of physical devices. If the circuit has a high depth or width, then the execution of such circuit could lead to inconclusive results as the total error may be too high. As the survey results suggest (Table II), developers are aware of this smell's severity. However, it is possible that they have to use a larger number of gates to perform simple operations as no built-in gate exists for the intended purpose. One interesting future research direction would be to analyze frequent co-occurring operations, optimally implement the required gates and offer them as a single built-in gate (similar to the Toffoli gate discussed in Section IV-B1).

IdQ and IQ smells were defined due to quantum hardware limitations at keeping qubits in the correct state for long periods of time. Even though developers are aware of their severity, these smells' prevalence might be explainable by the fact that looking only at the source code, it is hard to realize how long any qubit is idle. Thus, having the circuit depicted (e.g., directly in the IDE) might help alleviate this situation.

We hypothesize that one reason for the prevalence of the ROC smell could be due to the lack of developer knowledge regarding the built-in *repeat* operator and the negative consequence ROC has in the circuit's performance. Most, Qiskit online tutorials [86], experimental examples [87, 88], and books [89] do not mention the *repeat* operator.

2) Non-occurring code smells: We hypothesize that built-in gates have been sufficient for the developers' needs and, therefore, the usage of customized gates has not been required by developers to implement their programs. The survey results also suggest that CG is the least severe smell.

Due to the lack of access to physical quantum computers, developers still execute their QPs mostly on simulators which do not require an alignment between logical and physical qubits. This can be one of the likely reasons for not finding any occurrence of the LPQ smell. As the technology becomes more mature and quantum computing resources become more available, this smell will become frequent and critical.

VII. Implications

The study performed in this paper is a call for action for the community to proactively (1) investigate code smells tailored for QPs, (2) develop novel tools to assist developers in detecting quantum-specific code smells, and (3) train developers. We list some implications of our study below.

1) Further studies: Although our results show that QPs have a wide variety and prevalence of code smells, only one study investigated code smells in QC [32]. Nevertheless, our findings show the uniqueness, prevalence, and perceived severity of code smells from the developer's perspective. Thus, we encourage others to investigate further code smells specially tailored for QPs that explore other quantum properties not explored by our set of smells.

2) Tools: Some code smells are easier to identify in a quantum circuit drawing than in the source code (e.g., ROC). Thus, we foresee a system where developers can directly write source code and/or manipulate a quantum circuit, which would help developers at writing smell-free programs.

3) Training: The variety and prevalence of code smells in QPs illustrate the importance of educating developers about code smells. Educators can illustrate design principles by showing well-designed programs and those that exhibit code smells (as we have done in Sections IV-B to IV-D). Developers must also educate themselves about the types of code smells that may occur in QPs and how to mitigate them. Or even better, being conscious about code smells when programming in the first place and avoiding them altogether.

VIII. Threats to Validity

Our study, like any other empirical research, has its risks. Based on the guidelines reported by Wohlin et al. [90], we have taken all reasonable steps to mitigate the effect of potential threats, which are described in detail in this section.

1) Threats to construct validity: There is a threat that participants might have misunderstood the survey's questions. To mitigate it, we conducted pilot studies with developers with different experience levels to assess if there were unclear or leading questions and updated the questions based on their feedback. As the responses are distributed across the provided choices instead of one choice dominating all responses, we believe our effort to minimize any confirmation bias was successful.

Another threat is the detection of specific smells procedure as identifying whether a smell occurs in a piece of code relies on threshold values. For smells CG, ROC, NC, LC, IM, and LPQ, we have identified a threshold value based on our experience. Although this might sound biased, the (median) results reported in Table V corroborate our choices. The threshold value for smells IdQ and IQ were identified using empirical analysis (described in Section VI-B). It is, however, possible that these thresholds would be different for a different set of programs.

Regarding the likelihood of false positives in our results, similar to code smells in classical programming, some quantum-specific smells might be unavoidable in certain situations. However, as experienced practitioners have identified these smells, we believe they capture and highlight problematic patterns. We hope future research will investigate, e.g., the relationship between quantum bugs and maintenance issues with code smells in QPs, and address the false positive concern.

2) Threats to internal validity: The manual analysis used to map QC best practices to code smells could have introduced unintentional bias. Three authors individually did the best practices to code smells mapping to minimize this threat and discussed it with the other two authors until a consensus was reached. We then surveyed 35 quantum developers to validate our mapping and the definition of code smells.

Although the quantum-specific smells were derived from Cirq teams' best practices, the smells are related to characteristics of the quantum domain (i.e., IdQ, LM), and some smells are related to the programming style (i.e., NC, LC). Thus, these smells apply to the Qiskit framework and any other quantum framework as almost all relevant quantum computation platforms use circuit-based computation in their computational platform model. Therefore, the smell's specificity is transversal to almost all (if not all) quantum programming platforms.

The QSMELL's effectiveness was assessed by multiple researchers independently using the standard approach in literature (e.g., [81]). Two authors of the paper built an *oracle* dataset and evaluated QSMELL's output against the oracle. Note that we could not use external annotators (classical software developers or professional services (e.g., Amazon Mechanical Turk) due to the nature of the topic. Nevertheless, we took steps to minimize annotation bias. We followed the usual literature procedure where annotators individually completed the annotations, discussed the results, and then reached an agreement. The annotators are authors of the paper but not developers of QSMELL, so they were not aware of the internals of the implementation nor the procedures employed by the detector. Table IV reports the results, highlighting the agreement between tool-identified and human-identified smells. Furthermore, we randomly selected some of the smells reported in Table V and verified whether they were correct. We did not manually and exhaustively analyze all smells identified by QSMELL, as we assume that since QSMELL performed well on the validation dataset with high precision and recall, it should at least do reasonably well on the larger dataset. That is the usual assumption in the literature.

To address any other threat to the internal validity, QSMELL's source code and R source code (developed to perform the statistical analysis) were reviewed by two authors of the paper.

3) Threats to external validity: We conducted our empirical evaluation on 15 open-source QPs from GitHub. Although we believe we selected the largest and most diverse set of open-source QPs, our results might be different on other programs, e.g., industrial quantum programs. Moreover, during empirical evaluation, we could not compare QSMELL with any other tool since there exists no other code smell detection tool for QPs.

IX. CONCLUSIONS AND FUTURE WORK

In this paper, we studied the prevalence of code smells in QPs. We first derived eight quantum-specific smells, primarily inherited from the advocated QC programming practices, that may negatively affect the maintainability and quality of QPs. This is not an exhaustive list of quantum-specific smells for QPs, but we hope that future research will identify additional smells and enrich the catalog. We then developed QSMELL, the first tool to detect smells in QPs.

Experiments on a set of 15 real-world QPs showed that smells are prevalent in QC code: 11 programs (73.33%) contain at least one code smell, and, on average, programs have three code smells. Results also showed that LC is the most prevalent code smell in 53.33% of the programs.

The results reported in this paper lay the foundation for our future work. We plan to explore the evolution of the quantum-specific code smells and their effect on QPs' overall quality. Furthermore, we plan to run a user study on the usage of QSMELL and extend QSMELL to support other quantum frameworks (e.g., Cirq).

DATA AVAILABILITY

All the research artifacts (i.e., tools, scripts, and data) created for this study or generated by it are available in the replication package at https://doi.org/10.5281/zenodo.7556360.

ACKNOWLEDGMENTS

We thank the reviewers for their comments and suggestions during the rebuttal phase. This work was supported by the LASIGE Research Unit, ref. UIDB/00408/2020 and ref. UIDP/00408/2020, and by the Instituto de Telecomunicações Research Unit, ref. UIDB/50008/2020 and ref. UIDP/50008/2020.

References

[1] M. Benedetti, J. Realpe-Gómez, R. Biswas, and A. Perdomo-Ortiz. "Estimation of effective temperatures in quantum annealers for sampling applications: A case study with possible applications in deep learning". In: *Physical Review A* 94.2 (2016), p. 022308.

[2] C. Bennett and G. Brassard. "Quantum cryptography: Public key distribution and coin tossing". In: *arXiv preprint arXiv:2003.06557* (2020).

[3] R. Barends, J. Kelly, A. Megrant, A. Veitia, D. Sank, E. Jeffrey, T. White, J. Mutus, A. Fowler, B. Campbell, et al. "Superconducting quantum circuits at the surface code threshold for fault tolerance". In: *Nature* 508.7497 (2014), pp. 500–503.

[4] J. Zhao. "Quantum software engineering: Landscapes and horizons". In: *arXiv preprint arXiv:2007.07047* (2020).

[5] R. Shaydulin, C. Thomas, and P. Rodeghero. "Making quantum computing open: Lessons from open source projects". In: *Proc. of the IEEE/ACM 42nd Int. Conference on Software Engineering Workshops*. 2020, pp. 451–455.

[6] H. Ball, M. Biercuk, A. Carvalho, J. Chen, M. Hush, L. De Castro, L. Li, P. Liebermann, H. Slatyer, C. Edmunds, et al. "Software tools for quantum control: Improving quantum computer performance through noise and error suppression". In: *Quantum Science and Technology* 6.4 (2021), p. 044011.

[7] F. Leymann, J. Barzen, M. Falkenthal, D. Vietz, B. Weder, and K. Wild. "Quantum in the cloud: application potentials and research opportunities". In: *arXiv preprint arXiv:2003.06256* (2020).

[8] G. Li, L. Zhou, N. Yu, Y. Ding, M. Ying, and Y. Xie. "Projection-based runtime assertions for testing and debugging quantum programs". In: *Proc. of the ACM on Programming Languages* 4.OOPSLA (2020), pp. 1–29.

[9] J. Liu, G. T Byrd, and H. Zhou. "Quantum circuits for dynamic runtime assertions in quantum computation". In: *Proc. of the Twenty-Fifth Int. Conference on Architectural Support for Programming Languages and Operating Systems*. 2020, pp. 1017–1030.

[10] Y. Huang and M. Martonosi. "Statistical Assertions for Validating Patterns and Finding Bugs in Quantum Programs". In: *Proc. of the 46th Int. Symposium on Computer Architecture*. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, 541–553. DOI: 10.1145/3307650.3322213.

[11] S. Honarvar, M. Mousavi, and R. Nagarajan. "Property-based testing of quantum programs in Q#". In: *Proc. of the IEEE/ACM 42nd Int. Conference on Software Engineering Workshops*. 2020, pp. 430–435.

[12] S. Ali, P. Arcaini, X. Wang, and T. Yue. "Assessing the effectiveness of input and output coverage criteria for testing quantum programs". In: *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2021, pp. 13–23.

[13] P. Zhao, J. Zhao, and L. Ma. "Identifying Bug Patterns in Quantum Programs". In: *2021 IEEE/ACM 2nd Int. Workshop on Quantum Software Engineering (Q-SE)*. CA, USA: IEEE Computer Society, 2021, pp. 16–21. DOI: 10.1109/Q-SE52541.2021.00011.

[14] Y. Huang and M. Martonosi. "Statistical assertions for validating patterns and finding bugs in quantum programs". In: *Proc. of the 46th Int. Symposium on Computer Architecture*. 2019, pp. 541–553.

[15] D. Fortunato, J. Campos, and R. Abreu. "Mutation Testing of Quantum Programs: A Case Study With QISKit". In: *IEEE Transactions on Quantum Engineering* (2022), pp. 1–16. DOI: 10.1109/TQE.2022.3195061.

[16] J. Campos and A. Souto. "QBugs: A Collection of Reproducible Bugs in Quantum Algorithms and a Supporting Infrastructure to Enable Controlled Quantum Software Testing and Debugging Experiments". In: *2021 IEEE/ACM 2nd Int. Workshop on Quantum Software Engineering (Q-SE)*. 2021, pp. 28–32. DOI: 10.1109/Q-SE52541.2021.00013.

[17] I. Ahmed, C. Brindescu, U. Mannan, C. Jensen, and A. Sarma. "An Empirical Examination of the Relationship between Code Smells and Merge Conflicts". In: *Empirical Software Engineering and Measurement (ESEM), 2017 ACM/IEEE Int. Symposium on*. IEEE. 2017, pp. 58–67.

[18] G. Sculley D.and Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J. Crespo, and D. Dennison. "Hidden technical debt in machine learning systems". In: *Advances in neural information processing systems* 28 (2015), pp. 2503–2511.

[19] Martin Fowler. "Refactoring: Improving the design of existing code". In: *11th European Conference. Jyväskylä, Finland*. 1997.

[20] H. Jebnoun, H. Braiek, M. Rahman, and F. Khomh. "The Scent of Deep Learning Code: An Empirical Study". In: *Proc. of the 17th Int. Conference on Mining Software Repositories*. 2020, pp. 420–430.

[21] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk. "When and why your code starts to smell bad". In: *2015 IEEE/ACM 37th IEEE Int. Conference on Software Engineering*. Vol. 1. IEEE. 2015, pp. 403–414.

[22] R. Arcoverde and E. Garcia A.and Figueiredo. "Understanding the longevity of code smells: preliminary results of an explanatory survey". In: *Proc. of the 4th Workshop on Refactoring Tools*. 2011, pp. 33–36.

[23] A. Chatzigeorgiou and A. Manakos. "Investigating the evolution of bad smells in object-oriented code". In: *2010 Seventh Int. Conference on the Quality of Information and Communications Technology*. IEEE. 2010, pp. 106–115.

[24] A. Lozano, M. Wermelinger, and B. Nuseibeh. "Assessing the impact of bad smells using historical information". In: 2007, pp. 31–34.

[25] D. Rapu, S. Ducasse, T. Gîrba, and R. Marinescu. "Using history information to improve design flaws detection". In: *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proc.*. IEEE. 2004, pp. 223–232.

[26] M. Abbes, F. Khomh, Y. Gueheneuc, and G. Antoniol. "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension". In: *2011 15Th european conference on software maintenance and reengineering*. IEEE. 2011, pp. 181–190.

[27] F. Khomh, M. Di Penta, and Y. Gueheneuc. "An exploratory study of the impact of code smells on software change-proneness". In: *16th Working Conference on Reverse Engineering*. IEEE. 2009, pp. 75–84.

[28] M. D'Ambros, A. Bacchelli, and M. Lanza. "On the impact of design flaws on software defects". In: *10th Int. Conference on Quality Software*. IEEE. 2010, pp. 23–31.

[29] D. Sjøberg, A. Yamashita, B. Anda, A. Mockus, and T. Dybå. "Quantifying the effect of code smells on maintenance effort". In: *IEEE Transactions on Software Engineering* 39.8 (2012), pp. 1144–1156.

[30] A. Yamashita and L. Moonen. "Do code smells reflect important maintainability aspects?" In: *2012 28th IEEE international conference on software maintenance (ICSM)*. IEEE. 2012, pp. 306–315.

[31] A. Yamashita and Moonen L. "Exploring the impact of inter-smell relations on software maintainability: An empirical study". In: *2013 35th Int. Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 682–691.

[32] M. Openja, Mo. Morovati, L. An, F. Khomh, and M. Abidi. "Technical Debts and Faults in Open-source Quantum Software Systems: An Empirical Study". In: *arXiv preprint arXiv:2206.00666* (2022).

[33] P. Yadav, P. Mateus, N. Paunković, and A. Souto. "Quantum Contract Signing with Entangled Pairs". In: *Entropy* 21.9 (2019). ISSN: 1099-4300. DOI: 10.3390/e21090821. URL: https://www.mdpi.com/1099-4300/21/9/821.

[34] C. Vlachou, W. Krawec, P. Mateus, N. Paunković, and A. Souto. "Quantum key distribution with quantum walks". In: *Quantum Information Processing* 17.11 (2018), p. 288. ISSN: 1573-1332. DOI: 10.1007/s11128-018-2055-y. URL: https://doi.org/10.1007/s11128-018-2055-y.

[35] H. Li, F. Khomh, L. Tidjon, et al. "Bug Characteristics in Quantum Software Ecosystem". In: *arXiv preprint arXiv:2204.11965* (2022).

[36] Google inc. *Google best practice*. Accessed: 2022-07-13. 2022. URL: https://quantumai.google/cirq/google/best_practices.

[37] F. Khomh, M. Di Penta, Y. Guéhéneuc, and G. Antoniol. "An exploratory study of the impact of antipatterns on class change-and fault-proneness". In: *Empirical Software Engineering* 17.3 (2012), pp. 243–275.

[38] I. Deligiannis, I. Stamelos, L. Angelis, M. Roumeliotis, and M. Shepperd. "A controlled experiment investigation of an object-oriented design heuristic for maintainability". In: *Journal of Systems and Software* 72.2 (2004), pp. 129–143.

[39] W. Li and R. Shatnawi. "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution". In: *Journal of systems and software* 80.7 (2007), pp. 1120–1128.

[40] A. Nikanjam and F. Khomh. "Design smells in Deep Learning programs: an empirical study". In: *IEEE Int. Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2021, pp. 332–342.

[41] B. Muse, M. Rahman, C. Nagy, A. Cleve, F. Khomh, and G. Antoniol. "On the prevalence, impact, and evolution of SQL code smells in data-intensive systems". In: *Proc. of the 17th international conference on mining software repositories*. 2020, pp. 327–338.

[42] A. Rahman, C. Parnin, and L. Williams. "The seven sins: Security smells in infrastructure as code scripts". In: *IEEE/ACM 41st Int. Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 164–175.

[43] M. Lanza and R. Marinescu. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.

[44] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Francoise Le Meur. "Decor: A method for the specification and detection of code and design smells". In: *IEEE Transactions on Software Engineering* 36.1 (2009), pp. 20–36.

[45] M. Munro. "Product metrics for automatic identification of" bad smell" design problems in java source-code". In: *11th IEEE Int. Software Metrics Symposium (METRICS'05)*. IEEE. 2005, pp. 15–15.

[46] D. Sahin, M. Kessentini, S. Bechikh, and K. Deb. "Code-smell detection as a bilevel problem". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24.1 (2014), pp. 1–44.

[47] I. Deligiannis, M. Shepperd, M. Roumeliotis, and I. Stamelos. "An empirical investigation of an object-oriented design heuristic for maintainability". In: *Journal of Systems and Software* 65.2 (2003), pp. 127–139.

[48] H. Kagdi, M. Gethers, D. Poshyvanyk, and M. Collard. "Blending conceptual and evolutionary couplings to support change impact analysis in source code". In: *17th Working Conference on Reverse Engineering*. IEEE. 2010, pp. 119–128.

[49] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk. "Detecting bad smells in source code using change history information". In: *2013 28th IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*. IEEE. 2013, pp. 268–278.

[50] S. Omari and G. Martinez. "Enabling Empirical Research: A Corpus of Large-Scale Python Systems". In: *Proc. of the Future Technologies Conference*. Springer. 2019, pp. 661–669.

[51] N. Bafatakis N.and Boecker, W. Boon, M. Salazar, J. Krinke, G. Oznacar, and R. White. "Python coding style compliance on stack overflow". In: *IEEE/ACM 16th Int. Conference on Mining Software Repositories (MSR)*. IEEE. 2019, pp. 210–214.

[52] Z. Chen, L. Chen, W. Ma, and B. Xu. "Detecting code smells in Python programs". In: *2016 Int. Conference on Software Analysis, Testing and Evolution (SATE)*. IEEE. 2016, pp. 18–23.

[53] M. A. Nielsen and I. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. 10th. USA: Cambridge University Press, 2011. ISBN: 1107002176.

[54] P. Mateus, A. Sernadas, and A. Souto. "Universality of quantum Turing machines with deterministic control". In: *Journal of Logic and Computation* 27.1 (2017), pp. 1–19. DOI: 10.1093/logcom/exv008.

[55] Qiskit. *XGate*. Accessed: 2022-08-05. 2022. URL: https://qiskit.org/documentation/stubs/qiskit.circuit.library.XGate.html.

[56] Qiskit. *HGate*. Accessed: 2022-08-05. 2022. URL: https://qiskit.org/documentation/stubs/qiskit.circuit.library.HGate.html.

[57] X. Wang, P. Arcaini, T. Yue, and S. Ali. "Generating Failing Test Suites for Quantum Programs With Search". In: *Search-Based Software Engineering*. Ed. by Una-May O'Reilly and Xavier Devroey. Cham: Springer International Publishing, 2021, pp. 9–25. ISBN: 978-3-030-88106-1.

[58] X. Wang, P. Arcaini, T. Yue, and S. Ali. "Quito: a Coverage-Guided Test Generator for Quantum Programs". In: *2021 36th IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*. 2021, pp. 1237–1241. DOI: 10.1109/ASE51524.2021.9678798.

[59] X. Wang, P. Arcaini, T. Yue, and S. Ali. "QuSBT: Search-Based Testing of Quantum Programs". In: *arXiv preprint arXiv:2204.08561* (2022).

[60] X. Wang, T. Yu, P. Arcaini, T. Yue, and . Ali. "Mutation-Based Test Generation for Quantum Programs with Multi-Objective Search". In: *Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO '22. Boston, Massachusetts: ACM, 2022, 1345–1353. ISBN: 9781450392372. DOI: 10.1145/3512290.3528869.

[61] StackExchange. *Is Qiskit more popular than cirq? Why?* 2021. URL: https://quantumcomputing.stackexchange.com/questions/20673/is-qiskit-more-popular-than-cirq-why (visited on 07/31/2021).

[62] Qiskit. *qiskitCompilerTranspile*. Accessed: 2022-08-05. 2022. URL: https://qiskit.org/documentation/stubs/qiskit.compiler.transpile.html.

[63] Qiskit. *UnitaryGate*. Accessed: 2022-08-28. 2022. URL: https://qiskit.org/documentation/stubs/qiskit.extensions.UnitaryGate.html#qiskit.extensions.UnitaryGate.

[64] Qiskit. *HamiltonianGate*. Accessed: 2022-08-28. 2022. URL: https://qiskit.org/documentation/stubs/qiskit.extensions.HamiltonianGate.html#qiskit.extensions.HamiltonianGate.

[65] Qiskit. *SingleQubitUnitary*. Accessed: 2022-08-28. 2022. URL: https://qiskit.org/documentation/stubs/qiskit.extensions.SingleQubitUnitary.html.

[66] R. Acharya et al. *Suppressing quantum errors by scaling a surface code logical qubit*. 2022. DOI: 10.48550/ARXIV.2207.06431.

[67] Qiskit. *qiskit.circuit.QuantumCircuit.repeat*. Accessed: 2022-08-05. 2022. URL: https://qiskit.org/documentation/stubs/qiskit.circuit.QuantumCircuit.repeat.html.

[68] Qiskit. *Reducing compilation cost*. Accessed: 2022-08-05. 2022. URL: https://qiskit.org/documentation/tutorials/circuits_advanced/01_advanced_circuits.html#Reducing-compilation-cost.

[69] Qiskit. *Parameterized circuits*. Accessed: 2022-08-05. 2022. URL: https://qiskit.org/documentation/tutorials/circuits_advanced/01_advanced_circuits.html#Parameterized-circuits.

[70] Qiskit. *qiskit.providers.ibmq.IBMQBackend.run*. Accessed: 2022-08-05. 2022. URL: https://qiskit.org/documentation/stubs/qiskit.providers.ibmq.IBMQBackend.run.html.

[71] Qiskit. *Source code for qiskit.execute*. Accessed: 2021-05-25. 2021. URL: https://qiskit.org/documentation/stable/0.24/_modules/qiskit/execute.html.

[72] Qiskit. *Bind_parameters*. Accessed: 2022-08-28. 2022. URL: https://qiskit.org/documentation/stubs/qiskit.circuit.QuantumCircuit.bind_parameters.html.

[73] S. Johnstun and J. Van Huele. "Understanding and compensating for noise on IBM quantum computers". In: *American Journal of Physics* 89.10 (Oct. 2021), pp. 935–942. DOI: 10.1119/10.0006204.

[74] Qiskit. *ZGate*. Accessed: 2022-08-05. 2022. URL: https://qiskit.org/documentation/stubs/qiskit.circuit.library.ZGate.html.

[75] R. Boyer and J. Moore. "MJRTY—A Fast Majority Vote Algorithm". In: *Automated Reasoning: Essays in Honor of Woody Bledsoe*. Ed. by Robert S. Boyer. Dordrecht: Springer Netherlands, 1991, pp. 105–117. DOI: 10.1007/978-94-011-3488-0_5.

[76] Qiskit. *U2Gate*. Accessed: 2022-08-28. 2022. URL: https://qiskit.org/documentation/stubs/qiskit.circuit.library.U2Gate.html.

[77] Qiskit. *Transpiler*. Accessed: 2022-08-05. 2022. URL: https://qiskit.org/documentation/apidoc/transpiler.html.

[78] T. Mombach and M. Valente. *GitHub REST API vs GHTorrent vs GitHub Archive: A comparative study*. 2018.

[79] Qualtrics. *Qualtrics XM - Experience Management Software*. [Online; accessed 2022-03-14]. 2015.

[80] M. Wessel, A. Serebrenik, I. Wiese, I. Steinmacher, and M. Gerosa. "What to Expect from Code Review Bots on GitHub? A Survey with OSS Maintainers". In: *Proc. of the 34th Brazilian Symposium on Software Engineering*. SBES '20. New York, NY, USA: Association for Computing Machinery, 2020, 457–462. ISBN: 9781450387538.

[81] T. Wang, Y. Golubev, O. Smirnov, J. Li, T. Bryksin, and I. Ahmed. "PyNose: A Test Smell Detector For Python". In: *36th IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*. IEEE. 2021, pp. 593–605.

[82] Python. *ast — Abstract Syntax Trees*. 2022. URL: https://docs.python.org/3.7/library/ast.html (visited on 03/16/2022).

[83] David Wicks. "The coding manual for qualitative researchers". In: *Qualitative research in organizations and management: an international journal* (2017).

[84] K. Gwet. "Computing inter-rater reliability and its variance in the presence of high agreement". In: *British Journal of Mathematical and Statistical Psychology* 61.1 (2008), pp. 29–48.

[85] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. German, and D. Damian. "An in-depth study of the promises and perils of mining GitHub". In: *Empirical Software Engineering* 21.5 (2016), pp. 2035–2071.

[86] Qiskit. *Qiskit Tutorials*. Accessed: 2022-08-05. 2022. URL: https://github.com/Qiskit/qiskit-tutorials.

[87] Qiskit. *Qiskit Experiments*. Accessed: 2022-08-05. 2022. URL: https://github.com/Qiskit/qiskit-experiments.

[88] Qiskit. *Qiskit IBM Experiments*. Accessed: 2022-08-05. 2022. URL: https://github.com/Qiskit/qiskit-ibm-experiments.

[89] N. Johnston E.and Harrigan and M. Gimeno-Segovia. *Programming Quantum Computers: Essential Algorithms and Code Samples*. USA: O'Reilly Media, Incorporated, 2019. ISBN: 9781492039686.

[90] C. Wohlin, P. Runeson, M. Hst, M. Ohlsson, B. Regnell, and A. Wessln. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012. ISBN: 3642290434.