Quantum Program Linting with LLMs: Emerging Results from a Comparative Study

Seung Yeob Shin
University of Luxembourg
Luxembourg, Luxembourg
seungyeob.shin@uni.lu

Fabrizio Pastore
University of Luxembourg
Luxembourg, Luxembourg
fabrizio.pastore@uni.lu

Domenico Bianculli
University of Luxembourg
Luxembourg, Luxembourg
domenico.bianculli@uni.lu

***Abstract*—Ensuring the quality of quantum programs is increasingly important; however, traditional static analysis techniques are insufficient due to the unique characteristics of quantum computing. Quantum-specific linting tools, such as LintQ, have been developed to detect quantum-specific programming problems; however, they typically rely on manually crafted analysis queries. The manual effort required to update these tools limits their adaptability to evolving quantum programming practices.**

To address this challenge, this study investigates the feasibility of employing Large Language Models (LLMs) to develop a novel linting technique for quantum software development and explores potential avenues to advance linting approaches. We introduce LintQ-LLM, an LLM-based linting tool designed to detect quantum-specific problems comparable to those identified by LintQ. Through an empirical comparative study using real-world Qiskit programs, our results show that LintQ-LLM is a viable solution that complements LintQ, with particular strengths in problem localization, explanation clarity, and adaptability potential for emerging quantum programming frameworks, thus providing a basis for further research. Furthermore, this study discusses several research opportunities for developing more advanced, adaptable, and feedback-aware quantum software quality assurance methods by leveraging LLMs.

***Index Terms*—quantum software, linting, static analysis, large language models (LLMs), quality assurance**

I. Introduction

Quantum computing has made notable advancements in recent years, offering the potential to efficiently solve a certain set of problems in various domains, such as chemistry [1], cryptography [2], and optimization [3]. As the field grows, the quality assurance of quantum programs becomes increasingly important. Quantum programs manipulate both quantum and classical bits, where quantum bits (i.e., qubits) follow the principles of quantum mechanics, such as superposition and entanglement. Due to the unique characteristics of quantum programs, traditional software analysis techniques are insufficient for ensuring the quality of quantum programs. Hence, specialized analysis techniques that account for the specificities of quantum programs are needed.

Static program analysis techniques [4], particularly linting techniques, have been successfully applied to the development of classical software programs. Linting techniques automatically detect potential (or definitive) problems, including violations of coding rules, deviations from best practices, or defects. Linting tools scan the source code under analysis, matching it against a predefined set of rules to detect potential problems. These tools can warn developers about problems before they execute the code, aiming at improving software quality. In the quantum domain, linting tools play an even more important role, as executing quantum programs is costly and often restricted by limited access to quantum computers.

To account for the unique characteristics of quantum programs during linting, some tools have been introduced recently, including QSmell [5], QChecker [6], and LintQ [7]. Among them, LintQ is the state-of-the-art tool. It targets Qiskit programs [8] and implements a set of analyses to detect quantum-specific programming problems, such as those related to measurements, gate usage, resource allocation, and implicit constraint violations in API usage.

While quantum-specific linting tools have demonstrated their feasibility and effectiveness, they rely on manually crafted patterns, rules, or queries, which limit their adaptability to evolving quantum programming practices. As the field of quantum computing advances, new libraries, frameworks, and programming paradigms are expected to emerge, making it increasingly challenging to manually maintain and update these linting tools.

To address the above-mentioned challenge, this study investigates the feasibility of using LLMs to detect quality problems in quantum programs. Given the remarkable success of LLMs in assisting programming activities in classical software quality assurance [9], we aim to assess whether LLMs could offer a promising alternative or complement existing linting tools. Our hypothesis is that LLM can leverage their extensive knowledge of programming patterns, best practices, and implicit coding conventions, obtained from vast amounts of code repositories, including those related to quantum programs. In addition, LLMs have the potential to provide contextualized information and suggestions tailored to the quantum source code under analysis. Furthermore, LLM-enabled interactive, conversational analysis—which leverages chatbot-style feedback loops during the implementation of quantum programs—may enhance the linting process by offering explanations and recommendations in a more intuitive and user-friendly manner, thereby lowering the entry barrier for developers in developing high-quality quantum software programs.

Given the potential advantages of LLMs in enhancing the linting process, we first investigate the feasibility of an LLM-based linting approach in comparison to the state-of-the-art

quantum-specific linting tool, LintQ. In addition, we explore how LLMs can complement and extend existing linting techniques, providing a more adaptable, accurate, and developer-friendly approach to quantum software quality assurance.

To summarize, this paper makes the following contributions: (1) We introduce LintQ-LLM, an LLM-based linting tool that performs analyses comparable to those of LintQ. (2) We conduct a comparative study of LintQ-LLM and LintQ, empirically evaluating both tools on the same dataset of real-world Qiskit programs. (3) We analyze the strengths and weaknesses of these tools and, based on our findings, propose future research directions. To our knowledge, this is the first attempt to leverage LLMs for automatically detecting quantum programming problems. We believe this work serves as a stepping stone toward more advanced LLM-powered quantum software quality assurance tools that provide adaptable and precise analyses along with context-aware explanations and suggestions.

The rest of the paper is organized as follows. Section II provides background on LintQ. Section III describes our approach to developing LintQ-LLM. Section IV presents our experimental results. Section V discusses the findings from our experiments and outlines directions for future work. Section VI surveys related work. Section V concludes the paper.

II. Background: LintQ

LintQ [7] is a static analysis framework that detects quantum-specific problems in Qiskit source code. Specifically, LintQ introduces a set of abstractions for common quantum concepts, such as quantum registers, classical registers, quantum circuits, gates, qubit usage, and measurements. These abstractions enable LintQ to perform static analyses of Qiskit source code. LintQ provides ten analyses, each of which identifies a quantum-specific programming problem. For the analyses, LintQ leverages CodeQL [10], a general-purpose static analysis engine for source code. Each analysis is constructed as a query supported by CodeQL over the behavioral representation of the Qiskit code under analysis, which is expressed using the abstractions.

Table I presents the quantum-specific programming problems that LintQ identifies in Qiskit source code. Each problem corresponds to an invalid or undesirable use of Qiskit programming constructs that may arise during quantum program development. The ten problems listed in Table I are categorized into three groups based on their nature: (1) measurement- or gate-related problems, (2) resource allocation problems, and (3) implicit API constraint violations.

LintQ was applied to 7,568 real-world Qiskit programs and achieved an overall precision of 62.5%, which was computed by manually inspecting 361 warnings generated by the tool and observing 261 correctly reported problems. Further, its application to a benchmark of 42 quantum programs led to a recall of 7.1%, which shows large room for improvement despite LintQ being the state-of-the-art approach.

TABLE I
Problems identified by LintQ and their descriptions.

Problem[a]	Description
Measurement- or Gate-related Problems	
DoubleMeas	Two consecutive measurements are performed on the same qubit state.
OpAfterMeas	A gate is applied to a qubit after it has already been measured.
MeasAllAbuse	Measurement results are stored in a newly and implicitly created register, despite the presence of an existing classical register.
CondWoMeas	A conditional gate is applied without measuring the associated register.
ConstClasBit	A qubit is measured without undergoing any prior transformation.
Resource Allocation Problems	
InsuffClasReg	There are not enough classical bits to store the measurement results of all qubits.
OversizedCircuit	The quantum register includes qubits that remain unused.
GhostCompose	Two circuits are composed, but the resulting composed circuit is not utilized.
Implicit API Constraint Violations	
OpAfterOpt	A gate is applied to the circuit after transpilation.
OldIdenGate	An identity gate is created using an API that has been removed.

[a] Note that the problem names match those used in the LintQ paper [7] for each analysis.

III. Approach: LLM-based Linter

This section describes LintQ-LLM, our LLM-based linting approach for quantum programs. Our implementation of LintQ-LLM is available online [11].

A. Overview

Figure 1 provides an overview of LintQ-LLM. LintQ-LLM takes a Qiskit source code file as input and produces warnings for any quantum-specific problems identified, including their specific locations and explanations. Its key characteristics are to process one source code file a time, which is common for other linters, and more importantly, querying the LLM independently for each type of problem to be identified in the source code. An alternative would have been to query the LLM all at once, for all the potential problems to be identified, but this would have led to longer prompts with a lot of instructions for the LLM. Such long prompts reduce the number of tokens available for the program under analysis; indeed, an LLM can process only a fixed number of tokens (for simplicity, characters), thus reducing the maximum length of the file under analysis, which is provided within the prompt, as described below. Further, querying the LLM for multiple problems a time is more likely to introduce mistakes in the generated answer due to the increased length of the prompt and potential ambiguities among the multiple instructions.

B. Prompt Engineering

We created LLM prompts aimed at detecting quantum-specific programming problems. Specifically, the prompts targeted the ten problems listed in Table I, ensuring a fair comparison between LintQ and LintQ-LLM. We note that the first arXiv version of the LintQ paper [12] was posted on 1 October 2023, and the initial commit on the LintQ repository [13] was made on 26 October 2022. To avoid potential biases or confounding factors resulting from LLMs having access to publicly available data about LintQ, we selected the GPT-3.5 Turbo model from OpenAI, whose knowledge cutoff date is 1 September 2021.

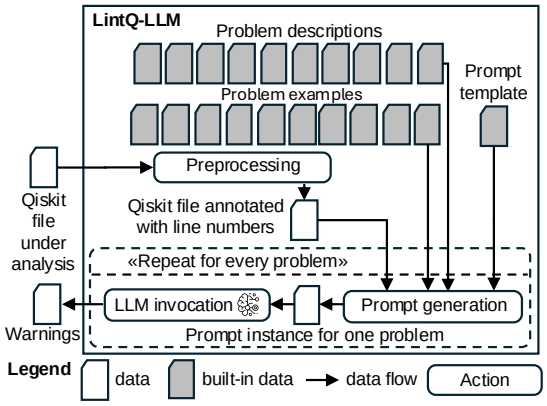

Fig. 1. An overview of the data flow in LintQ-LLM.

```
1  ## Situation
2  You are analyzing the source code to detect ?problem?
       occurrences.
3  <code>
4  ?code?
5  </code>
6
7  ## Your Role
8  Act as a source code linter tool to detect all occurrences of
       the problem:
9  ?problem_description?
10
11 ## Output Format
12 If the code contains ?problem?, return this JSON object:
13 {
14   "problem": "?problem?",
15   "snippets": ["string"], // extract code verbatim where
         ?problem? occurs.
16   "lines": ["integer"], // list line numbers where ?problem?
         occurs.
17   "explanations": ["string"] // explain why each line has
         ?problem?.
18 }
```

Fig. 2. Prompt template. `?param?` indicates a parameter in the template.

To detect a particular quantum-specific problem in the source code file, LintQ-LLM generates a dedicated prompt for that problem, which is then fed to the LLM. To this end, LintQ-LLM uses a prompt template, which is common for all the types of problems addressed by LintQ-LLM and is instantiated with appropriate data at each invocation.

Figure 2 shows a shortened version of our template; the complete prompt template is available online [11]. As shown in Figure 2, we structured the prompt template based on prompt engineering guidelines [14] and leveraged few-shot learning. The template includes three parameters that shape the concrete prompts, as follows: `(?problem?)` the name of the quantum-specific problem to detect, i.e., one of the problems listed in Table I, `(?code?)` the source code to be analyzed for detecting occurrences of the problem, `(?problem_description?)` a detailed description of the problem, including example cases for few-shot learning. Before querying the LLM, these parameters are programmatically replaced with the corresponding data. For `(?code?)`, we provide the source code under analysis verbatim, but we annotate each line with the corresponding line number, which we found to be necessary to avoid erroneous outputs from the LLM.

```
1  DoubleMeas (Double measurement) --- Any two subsequent
       measurements on the same qubit produce the same classical
       result, making the second measurement not only redundant
       but also a possible sign of unintended behavior or a
       misunderstanding of the properties of quantum information.
2  The code example below shows the DoubleMeas problem.
3  <example>
4    circuit = QuantumCircuit(3, 3)
5    circuit.ccx(0, 1, 2)
6    circuit.measure(0, 0) # Measure qubit 0
7    circuit.measure(2, 2)
8    circuit.measure(0, 1) # Problem: Qubit 0 already measured
9  </example>
```

Fig. 3. Prompt description of the double measurement problem (DoubleMeas in Table I). We reuse the problem description from the LintQ paper [7].

An example problem description is provided in the following paragraphs, after an overview of the template structure.

In the `Situation` section of the template (Figure 2), the context in which the LLM operates is explicitly defined, ensuring the LLM understands its objective of identifying a specific quantum-related programming problem (provided by `?problem?`) in the provided source code (given by `?code?`). The `Your Role` section instructs the LLM to act as a source code linter, specifically focusing on detecting the problem described by `?problem_description?`. The `Output Format` section enforces a strict JSON-based output format to facilitate automated parsing and analysis. The output includes: (`problem`), i.e., the name of the detected quantum-specific problem; (`snippets`), i.e., extracted code segments where the problem occurs; (`lines`), i.e., the line numbers corresponding to the instructions with the identified problem; (`explanations`), i.e., descriptions of why the reported code segments exhibit the problem.

As an example, Figure 3 shows the prompt description for the double measurement problem (DoubleMeas in Table I). The description explains how consecutive measurements on the same qubit always yield the same classical result, making the second measurement redundant and potentially indicating unintended behavior or a misunderstanding of quantum information principles. The `<example>` block in the description presents the DoubleMeas problem in a Qiskit circuit. In the circuit, qubit 0 is measured twice on lines 6 and 8, with comments explaining the problem. We reuse the problem descriptions from the LintQ paper [7] to ensure consistency with the LintQ work, allowing for comparability in our evaluation. However, LintQ-LLM does not use any descriptions of how LintQ identifies quantum-specific problems, ensuring that LintQ-LLM relies solely on the LLM's analysis capabilities for a fair comparison with LintQ.

IV. Experiments: A Comparative Study

In this section, we report on our experiments comparing LintQ and LintQ-LLM. Specifically, we aim to address the following research question (RQ): *How does LintQ-LLM compare to LintQ in terms of effectiveness in detecting quantum-specific problems in Qiskit code?*

A. Datasets and Metrics

We used the annotated dataset from the LintQ study to enable a direct comparison with LintQ and to avoid introducing threats to internal validity. The dataset includes 345 warnings, randomly sampled by LintQ authors from the LintQ-identified warnings for the 7,568 Qiskit files processed in their experiments. The LintQ authors annotated these warnings as true positives (TPs), false positives (FPs), and noteworthy (NWs). TPs are warnings that correctly identify real quantum-specific problems in the analyzed source code. FPs refer to warnings that do not correspond to actual problems in the Qiskit code. NWs are potential problems for which the authors were unable to definitively determine whether the behaviors caused by such problems were intended by developers or not.

The 345 warnings in the dataset belong to 268 Qiskit files. Since LintQ-LLM uses ten example cases, one for each problem, for few-shot learning (six from the Qiskit files and four from the LintQ paper), we excluded the annotated warnings of the six files from our evaluation dataset. As a result, our evaluation dataset contains 338 annotated warnings belonging to 262 Qiskit files.

For our experiment, we applied LintQ-LLM to each file in the evaluation dataset and measured effectiveness in terms of precision and recall. Precision is the number tp of TPs identified by LintQ-LLM divided by the number of warnings that overlap between the evaluation dataset and LintQ-LLM's detected warnings. These overlapping warnings contain tp, fp, and nw instances of TPs, FPs, and NWs; hence, precision is calculated as $\frac{tp}{(tp+fp+nw)}$. Recall is the number tp of TPs identified by LintQ-LLM divided by the total number of TPs in the evaluation dataset. This total includes both the tp TPs identified by LintQ-LLM and the fn TPs missed by LintQ-LLM; hence, recall is calculated as $\frac{tp}{(tp+fn)}$.

B. Methodology

To answer the RQ, LintQ-LLM and LintQ were applied to the same set of real-world Qiskit source code files. We then compared the warnings generated by both tools, using the aforementioned annotated dataset, to assess their ability to identify actual quantum-specific problems in the code.

C. Results

Table II presents the number of TPs, FPs, and NWs reported by each tool for each quantum-specific problem listed in Table I. We note that the number of TPs, FPs, and NWs obtained from LintQ-LLM was determined through our manual inspection of the results it produced. From our inspection, we found several warnings where LintQ and LintQ-LLM identified the same problem occurrences but reported the corresponding code lines (i.e., locations) differently, as described below.

For example, LintQ and LintQ-LLM provide different but correct locations for the same occurrences of the ConstClasBit problem. LintQ identifies occurrences of the ConstClasBit problem when a quantum circuit is created. However, LintQ-LLM identifies the same problem occurrence when a qubit is measured. Recall from Section II that the ConstClasBit problem occurs when a qubit is measured without undergoing any prior transformation. Hence, by definition, the warning locations (i.e., measurements in this case) identified by LintQ-LLM are more aligned with the intended semantics of the problem. Among the 137 TPs produced by LintQ-LLM, 110 are found at the same locations identified by LintQ. Of the remaining 27 TPs, 24 are found at locations involving measurements (21 TPs), operations (2 TPs), and register creation (1 TP), rather than at the quantum circuit creation locations where LintQ detects them. The remaining three TPs occur when LintQ-LLM and LintQ identify the same problem on different lines while referring to the same code segment (1 TP) and the double measurements on the same qubit (2 TPs). This suggests that LintQ-LLM provides a more precise localization of the problem compared to LintQ.

TABLE II
TPs, FPs, AND NWs IDENTIFIED BY LINTQ AND LINTQ-LLM.

Problem	LintQ			LintQ-LLM		
	#TPs	#FPs	#NWs	#TPs	#FPs	#NWs
DoubleMeas	18	4	3	17	3	3
OpAfterMeas	44	0	0	30	0	0
MeasAllAbuse	16	0	1	16	0	1
CondWoMeas	27	0	0	22	0	0
ConstClasBit	28	21	10	17	17	6
InsuffClasReg	24	22	21	9	3	7
OversizedCircuit	27	16	13	8	5	3
GhostCompose	7	0	3	4	0	0
OpAfterOpt	6	0	0	3	0	0
OldIdenGate	13	11	3	11	8	2
Sum	210	74	54	137	36	22

LintQ-LLM achieved a precision of 70%, comparable to LintQ's 62.6%. Although our results concern only the lines reported by both LintQ and LintQ-LLM, and thus might be imprecise (e.g., TPs might be distributed differently in lines not annotated by LintQ), they are promising.

Table III presents the recall values for each problem detected by LintQ-LLM. Recall from Section IV-A that the LintQ dataset was created by randomly sampling its own warnings; hence, computing LintQ's recall using this dataset is infeasible. LintQ-LLM detected most of the cases of MeasAllAbuse, DoubleMeas, and OldIdenGate problems, with recall values above 85%. In contrast, LintQ-LLM missed many InsuffClasReg and OversizedCircuit cases, with recall values of 38% and 30%, respectively. By comparing these two groups of high-recall and low-recall problems, we found that LintQ-LLM correctly detects TPs when the problems are related to API usage, such as measuring all qubits (MeasAllAbuse), individual measurements (DoubleMeas), and deprecated API usage (OldIdenGate). However, LintQ-LLM misses many TPs when problem identification requires complex analysis, such as data flow analysis on classical and quantum registers, corresponding to InsuffClasReg and OversizedCircuit, respectively. The results suggest that future improvements should focus on enhancing LintQ-LLM's capabilities for analyzing the usage flows of quantum and classical bits (i.e., data flows). Overall, LintQ-LLM achieved a recall score of 65%; such a result is promising since it indicates that LintQ-LLM can detect a high

TABLE III
RECALL FOR EACH PROBLEM DETECTED BY LINTQ-LLM.

MeasAllAbuse	DoubleMeas	OldIdenGate	CondWoMeas	OpAfterMeas	ConstClasBit	GhostCompose	OpAfterOpt	InsuffClasReg	OversizedCircuit
100%	94%	85%	81%	68%	61%	57%	50%	38%	30%

proportion of faults detected by means of static analysis. Our recall score is much higher than the 7.2% recall score reported for LintQ [7]; however, it is worth noting that we could not compare LintQ-LLM on the same dataset used to compute LintQ recall (i.e., Bugs4Q [15]). Indeed, the Bugs4Q dataset was made public before the training of any available LLM capable of code comprehension; specifically, the oldest LLMs capable of code comprehension that are available are GPT-3.o and llama-2, both trained after the release of Bugs4Q.

Regarding the execution time of LintQ and LintQ-LLM for linting a program, LintQ took ≈1s, whereas LintQ-LLM took ≈35s. This result indicates that network communication and the GPT model introduce inefficiencies, requiring further optimization.

D. Threats to Validity

The main threats to the validity of our results is the possible bias introduced by the annotation dataset curated by the authors of the LintQ paper. Since the dataset includes *samples* of warnings obtained from LintQ, it naturally cannot represent the entire population of warnings produced by LintQ-LLM. Nevertheless, we opted to use the annotated dataset, since it provides a common ground for comparison. For a fair comparison, further studies are needed with a new set of annotations that are free from bias toward either LintQ or LintQ-LLM, such as those created independently from both tools.

To prevent any threats to validity caused by confounding factors, as discussed in Section III-B, we selected the GPT-3.5 Turbo model, as its knowledge cutoff date is September 1, 2021, which is prior to the introduction of LintQ. However, given the rapid advancements in the field of LLMs, applying newer models with improved reasoning capabilities and better understanding of source code may yield different results. Hence, future studies should consider evaluating LintQ-LLM using more recent LLMs, such as GPT-o1 or beyond, to assess their capabilities in linting quantum programs.

V. DISCUSSION AND FUTURE WORK

This section discusses LintQ and LintQ-LLM based on our experience of reproducing LintQ and developing LintQ-LLM. From these experiences, we have identified their strengths and limitations, as well as potential research opportunities.

Adapting to quantum programming frameworks. Although LintQ introduces abstractions for quantum programming constructs, it currently applies only to Qiskit programs. Extending it to support other quantum programming frameworks, such as Cirq [16] and PennyLane [17], requires manually mapping their APIs to the abstractions, which may need modification, and redefining queries. In contrast, LintQ-LLM is more adaptable than LintQ. The only component requiring significant modification is the few-shot learning component, as it depends on framework-specific examples.

Adapting linting techniques. LintQ relies on the query-based problem detection scheme provided by CodeQL, making it capable of deterministically detecting problems. However, detecting new quantum-specific problems—an expected challenge due to the advancements in quantum computing—requires manually crafting precise queries, which demands in-depth knowledge of both CodeQL and the target quantum programming framework. In contrast, LintQ-LLM leverages LLM-based analysis, allowing it to more easily adapt to detecting new problems without requiring manually defined queries. This is because engineers facing new problems inherently have concrete examples that can be used for few-shot learning.

Handling large files. LintQ-LLM leverages the GPT model, which has a token limit to optimize efficiency in a multi-user environment where it is accessed simultaneously. Unfortunately, LintQ-LLM failed to analyze 21 files out of 262 (see Section IV-A) due to the token limit of the chosen underlying GPT model (i.e., 16 385 tokens); such limitation, which is absent in LintQ, could be overcome either by using models with a higher token limit or by leveraging slicing techniques.

Potential research avenues. A promising direction for future research is the development of *hybrid* linting approaches that integrate the strengths of both tools. For example, the outputs of LintQ and LintQ-LLM could be cross-referenced to prioritize the warning identified by both. In addition, the context-specific explanations provided by LintQ-LLM could complement LintQ, helping developers better understand the identified warnings. For example, LintQ-LLM provides an explanation for an identified warning as follows: "*The QuantumCircuit 'qc' is created with both a QuantumRegister 'qr' and a ClassicalRegister 'cr', but only one qubit is initialized in 'qr'. This leads to an OversizedCircuit issue where resources are wasted on unused qubits*". Such explanations produced by LintQ-LLM are significantly more informative compared to those produced by LintQ. Building on this capability, interactive analysis powered by LLMs could further enhance the developer experience. By engaging in a chatbot-based conversational feedback loop, developers can ask follow-up questions, seek clarification, or request code improvement suggestions directly within their development environment. Last, LLMs could be used to automatically derive LintQ parsers for new quantum-programming languages, while LintQ-LLM could support the identification of problems for which a LintQ analysis has not been implemented yet.

Our study restricted the comparison between LintQ and LintQ-LLM to the analysis of Qiskit programs. To derive more generalizable findings, further research is needed to evaluate

these tools by applying them to other quantum programming frameworks. In addition, conducting user-involved case studies is essential to assess the practical usefulness of these linting tools in real-world quantum software development.

VI. Related Work

Quantum-specific static analysis. Static analysis has been widely studied and applied for classical software programs; recently it has gained attention for quantum programs [5]–[7]. Chen et al. [5] identified eight quantum-specific code smells (referred to as "problems" in our context) from the Cirq best practices [18], validated them through a developer survey, and developed QSmell, a tool for detecting these smells. QSmell employs both dynamic and static analysis techniques, leveraging execution details for the former and abstract syntax trees (ASTs) for the latter. Zhao et al. [6] introduced QChecker, a static analysis tool for detecting bugs (also referred to as "problems" in this paper) in quantum programs written in Qiskit. QChecker identifies bugs based on predefined patterns in ASTs, derived from real-world quantum bugs [15]. In contrast to LintQ-LLM, these prior approaches rely on predefined rules, patterns, or graph constructs, which require considerable manual effort to update when quantum programming practices evolve.

LLM-assisted code analysis. The use of LLMs for code analysis has attracted considerable interest in the context of classical (non-quantum) software development [19]–[21]. For example, GitHub Copilot [20] provides code review capabilities that help developers identify potential problems and make improvements more efficiently. Amazon CodeWhisperer [21] suggests best coding practices and uncovers potential security vulnerabilities. Despite these advancements, their application to quantum software remains limited. LintQ-LLM is the first known effort to leverage LLMs for quantum-specific linting. We believe our work opens new opportunities for automated, adaptable, and developer-friendly quality assurance in quantum software development.

VII. Conclusion

As quantum computing continues to evolve, ensuring software quality remains a critical challenge. In this paper, we explored the feasibility and potential of LLMs in quantum program analysis, highlighting the need for further advancements in linting techniques. Specifically, we introduced LintQ-LLM, an LLM-based linting tool for detecting quantum-specific programming problems, and conducted a comparative analysis with LintQ, a state-of-the-art query-based quantum-specific linting tool. Our experiment results indicate that LintQ-LLM is capable of detecting quantum-specific programming problems and, in some cases, provides better problem localization and more intuitive explanations than LintQ. However, LintQ-LLM did not outperform LintQ in detecting complex problems that require sophisticated static analysis techniques to track classical and quantum bit usage flows. Based on these findings, we discussed the potential for a hybrid approach that integrates static analysis with LLM-powered analysis capabilities. Such an approach could improve detection accuracy, offer developers context-aware explanations for detected problems, and provide recommendations.

Acknowledgment

We thank Matteo Paltenghi for the help with LintQ.

References

[1] S. McArdle, S. Endo, A. Aspuru-Guzik, S. C. Benjamin, and X. Yuan, "Quantum computational chemistry," *Reviews of Modern Physics*, vol. 92, no. 1, p. 015003, 2020.

[2] C. Portmann and R. Renner, "Security in quantum cryptography," *Reviews of Modern Physics*, vol. 94, no. 2, p. 025008, 2022.

[3] S. Ebadi, A. Keesling, M. Cain, T. T. Wang, H. Levine, D. Bluvstein, G. Semeghini, A. Omran, J.-G. Liu, R. Samajdar *et al.*, "Quantum optimization of maximum independent set using rydberg atom arrays," *Science*, vol. 376, no. 6598, pp. 1209–1215, 2022.

[4] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis*. Springer, 1999.

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

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

[7] M. Paltenghi and M. Pradel, "Analyzing quantum programs with lintq: A static analysis framework for qiskit," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 2144–2166, 2024.

[8] I. Quantum, "Qiskit: An open-source framework for quantum computing," 2025, accessed: March 10, 2025. [Online]. Available: https://qiskit.org/

[9] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software Testing With Large Language Models: Survey, Landscape, and Vision," *IEEE Transactions on Software Engineering*, vol. 50, no. 4, pp. 911–936, 2024.

[10] P. Avgustinov, O. de Moor, M. P. Jones, and M. Schäfer, "QL: object-oriented queries on relational data," in *Proceedings of the 30th European Conference on Object-Oriented Programming*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 56. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, pp. 2:1–2:25.

[11] S. Y. Shin, F. Pastore, and D. Bianculli, "Replication package," accessed: July 23, 2025. [Online]. Available: https://doi.org/10.6084/m9.figshare.28636028

[12] M. Paltenghi and M. Pradel, "Analyzing quantum programs with lintq: A static analysis framework for qiskit," 2023. [Online]. Available: https://arxiv.org/abs/2310.00718v1

[13] ——, "Lintq: A static analysis framework for qiskit quantum programs," 2024, accessed: March 31, 2025. [Online]. Available: https://github.com/sola-st/LintQ

[14] DAIR.AI, "Prompt engineering guide," 2025, accessed: March 20, 2025. [Online]. Available: https://www.promptingguide.ai/

[15] P. Zhao, Z. Miao, S. Lan, and J. Zhao, "Bugs4q: A benchmark of existing bugs to enable controlled testing and debugging studies for quantum programs," *Journal of Systems and Software*, vol. 205, p. 111805, 2023.

[16] G. Q. AI, "Cirq: A python framework for near-term quantum computing," 2025, accessed: March 6, 2025. [Online]. Available: https://quantumai.google/cirq

[17] V. Bergholm, J. A. Izaac, M. Schuld, C. Gogolin, and N. Killoran, "Pennylane: Automatic differentiation of hybrid quantum-classical computations," *CoRR*, vol. abs/1811.04968, 2018.

[18] G. Q. AI, "Cirq best practices," 2025, accessed: March 10, 2025. [Online]. Available: https://quantumai.google/cirq/google/best_practices

[19] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Enhancing static analysis for practical bug detection: An llm-integrated approach," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA1, pp. 474–499, 2024.

[20] GitHub, "GitHub Copilot," 2025, accessed: March 10, 2025. [Online]. Available: https://github.com/features/copilot/

[21] Amazon, "Amazon CodeWhisperer," 2025, accessed: March 10, 2025. [Online]. Available: https://aws.amazon.com/codewhisperer