Beyond Rules: LLM-Powered Linting for Quantum Programs

Pietro Cassieri
Department of Computer Science
University of Salerno
Salerno, Italy
pcassieri@unisa.it

Giuseppe Scanniello
Department of Computer Science
University of Salerno
Salerno, Italy
gscanniello@unisa.it

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*—As quantum computing transitions from theoretical experimentation to its practical application, the reliability of quantum software has become a critical bottleneck. Traditional static analysis techniques for quantum programs—primarily rule-based linters—are increasingly inadequate; they struggle to keep pace with rapidly evolving APIs and fail to capture complex, context-dependent quantum programming problems. This results in high maintenance overhead and limited detection capabilities. In this paper, we introduce LintQ-LLM+CoT and LintQ-LLM+RAG, novel approaches that redefine the detection of quantum programming problems by employing Large Language Models (LLMs) specialized, respectively, via Chain-of-Thought (CoT) prompting and a Retrieval-Augmented Generation (RAG) system that grounds the model's reasoning in a curated knowledge base of verified quantum programming problems and best practices. We conducted a rigorous and manual comparative evaluation against the state-of-the-art rule-based tool, LintQ, using a corpus of 55 Qiskit programs. Our results show that LLM-based approaches, with and without RAG, outperform LintQ in terms of quantum programming problems detection correctness (precision) and completeness (recall). Overall, LLM-based approaches were more effective than LintQ (F1-score equal to 0.70 and 0.68 vs. 0.41). Furthermore, the RAG-enhanced variant demonstrated a slightly superior precision, effectively reducing false positives. Our findings suggest that LLMs provide a scalable and adaptive foundation for the next generation of linters in quantum software engineering.**

***Index Terms*—Quantum Software Engineering, Static Analysis, LLMs, RAG, Qiskit.**

I. Introduction

Ensuring the reliability of quantum software has become a paramount challenge as quantum computing transitions from a theoretical pursuit to a practical engineering discipline. In this evolving landscape, developers increasingly rely on tools like Qiskit [1] to manipulate both quantum and classical bits. However, the unique principles of quantum mechanics—such as superposition, entanglement, and the collapse of quantum states upon measurement—introduce specialized programming "anti-patterns" or bugs that traditional classical software analysis tools are often ill-equipped to detect.

Recent advances in the quantum research field have identified ten common, quantum-specific, programming problems, categorized into families such as gate-related issues, resource allocation errors, and implicit API constraint violations. To address these, state-of-the-art static analysis frameworks like LintQ [2] have been developed. While LintQ achieves notable precision by utilizing declarative queries over formal abstractions of quantum concepts, it relies heavily on manually crafted deterministic rules [2]. This reliance creates a significant maintenance burden and limits the tool's ability to adapt to the rapidly evolving nature of quantum APIs [3].

In previous work [3], some of the authors of this paper introduced the foundational concepts for LintQ-LLM, an approach that leverages Large Language Models (LLMs) to automate the detection of quantum programming problems in Qiskit programs. This approach was limited to single-prompt interactions and did not fully utilize the reasoning capabilities of the models.

In this paper, we improve LintQ-LLM by introducing a multi-prompt Chain-of-Thought (CoT) interaction designed to guide the model through strategic planning, code understanding, and localizing the problems. By moving away from rigid rules and toward the flexible reasoning capabilities of LLMs, this enhanced approach, named LintQ-LLM+CoT, provides a more adaptable solution to detect programming problems. In addition, we also propose LintQ-LLM+RAG, a variant of LintQ-LLM+CoT that uses a Retrieval-Augmented Generation (RAG) extension to further bridge the gap between flexible reasoning and formal correctness. The overarching goal of this variant is to ground the model's reasoning in a specialized knowledge base of manually verified true positive instances (i.e., actual quantum programming problems). By providing the model with "one-shot" learning examples, with semantic similarity to the analyzed code, we aim to improve the correctness of the analysis and reduce the hallucination of false positives (FPs).

We also conducted a comprehensive empirical evaluation of both LintQ-LLM+CoT and LintQ-LLM+RAG in terms of detection correctness (precision), completeness (recall), and effectiveness (F1-score) of quantum programming problems. Specifically, we performed a manual, rigorous comparative evaluation of LintQ-LLM+CoT and LintQ-LLM+RAG against the state-of-the-art rule-based tool, LintQ, using a corpus of 55 Qiskit programs, including both real-world code and synthetic fault-injected files.

Our results show that LintQ-LLM+CoT, with and without RAG, outperforms LintQ in terms of correctness and completeness. Also, the effectiveness of the LLM-based approaches is higher than that of LintQ (0.70 and 0.68 vs. 0.41). Furthermore, the RAG-enhanced variant yielded a slightly superior precision, effectively reducing FPs. In summary, our results suggest that LLMs provide a scalable and adaptive foundation for the next generation of quantum software quality assurance tools (e.g., linters).

To summarize, this paper makes the following contributions:

- A multi-prompt CoT pipeline for LintQ-LLM, namely LintQ-LLM+CoT, which enhances the foundational LLM-based linting concept introduced in prior work [3], as well as LintQ-LLM+RAG, a variant of LintQ-LLM+CoT that uses RAG to ground reasoning in verified quantum programming problem examples.
- A rigorous empirical assessment of LintQ, LintQ-LLM+CoT, and LintQ-LLM+RAG in terms of effectiveness as well as correctness and completeness of the detected quantum programming problems.

II. Background and Related Work

This section establishes the foundational concepts and related literature framing our work.

A. Quantum Software Testing and Analysis

Several automated approaches have been proposed to ensure the correctness of quantum compilers and simulators. One prominent example is QDiff [4], which adapts differential testing to quantum software. It generates semantically equivalent program variants through source-to-source transformations and compares their outputs across different backends and optimization levels using statistical tests such as the Kolmogorov–Smirnov test. Similarly, MorphQ [5] applies metamorphic testing to the Qiskit toolchain by transforming circuits—for instance, by changing basis gates or converting to and from OpenQASM—and checking whether expected relationships between outputs hold. More recently, Fuzz4All [6] explores universal fuzzing for quantum systems by leveraging LLMs to generate and mutate diverse, syntactically valid programs that can stress-test quantum platforms.

Testing quantum software poses challenges distinct from those in classical computing, as highlighted by Miranskyy et al. [7]. Quantum programs are inherently probabilistic, quantum states cannot be copied due to the no-cloning theorem, and any measurement irreversibly collapses the state. Together, these properties make it difficult to observe, reproduce, and validate program behavior, complicating the entire testing process. Testing has largely focused on dynamic techniques that can handle probabilistic outputs. Quito, proposed by Ali et al. [8], introduces input and output coverage criteria combined with statistical oracles, such as the Wilcoxon signed-rank test, to assess whether a test passes or fails over multiple executions. In a similar vein, QuanFuzz [9] uses search-based techniques, including genetic algorithms, to generate inputs that maximize coverage of quantum-relevant behaviors, particularly around measurement operations.

Researchers have also proposed runtime assertion mechanisms to address the intrinsic difficulty of inspecting intermediate quantum states. For example, Huang and Martonosi [10] introduced statistical assertions based on Chi-square tests to check whether observed measurement distributions match expected outcomes. Differently, Li et al. [11] proposed Proq, a projection-based approach grounded in Birkhoff–von Neumann quantum logic. By representing predicates as projections onto subspaces, Proq enables partial verification of intermediate states using local measurements and ancilla qubits, reducing the risk of collapsing the entire quantum state.

Despite their shown effectiveness, dynamic approaches generally require repeated program execution. This can be costly in terms of time and resources, especially when running on noisy intermediate-scale quantum (NISQ) hardware, where results are affected by noise and limited qubit fidelity. These limitations have motivated increasing interest in static analysis techniques. In fact, static analysis avoids execution altogether, sidestepping both measurement-related issues and hardware constraints. For example, QSmell [12] identifies quantum-specific code smells—such as overly long circuits or misaligned qubit mappings—that may degrade performance or reliability. QChecker [13] focuses on problem detection by constructing specialized abstract syntax trees (ASTs) and matching extracted properties against known problem patterns from the Bugs4Q benchmark. Meanwhile, QCPG [14] extends classical code property graphs to the quantum domain, integrating ASTs, control-flow graphs, and program-dependence graphs to model quantum constructs like qubits, gates, and measurements, enabling problem detection through graph queries. Unlike these dynamic and rule-based static approaches, our work leverages LLM to perform quantum linting. By employing multi-prompt CoT reasoning and RAG, we provide a more flexible approach that generalizes across evolving APIs without requiring manually crafted rules.

B. Quantum Problem Detection: LintQ

Quantum programming problems frequently arise when developers misuse quantum constructs or misunderstand the underlying quantum properties. Recently, studies have cataloged these issues to aid in the development of automated analysis tools. In particular, ten common quantum-specific programming problems have been identified and categorized into three main families: measurement- or gate-related problems, resource allocation problems, and implicit API constraint violations [2]. In Figure 1, we show an example of the

TABLE I
QUANTUM-SPECIFIC PROGRAMMING PROBLEMS AND THEIR DESCRIPTIONS.

Problem	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.

```
dreg = QuantumRegister(1, 'd')
nreg = QuantumRegister(2, 'n')
creg = ClassicalRegister(2, 'c')
circ = QuantumCircuit(dreg,nreg,creg)
while found_path == False:
    circ.h(0)
    circ.cx(0,1)
    circ.x(0)
    circ.cx(0,2)
    circ.x(0)
    circ.measure(nreg[:2],creg)
```

Fig. 1. An example of code affected by *OpAfterMeas*. The portion of the quantum programming affected by this problem is highlighted in the red ellipse: starting from the second iteration of the while loop, quantum gates are applied to qubits that have already been measured.

OpAfterMeas problem, while Table I summarizes these quantum programming problems (presented in the LintQ work [2]) by providing a short description for each of them.

LintQ [2] is the state-of-the-art static analysis tool specifically designed to detect quantum-specific problems in source code written using Qiskit [1]. It introduces a formal set of abstractions representing common quantum computing concepts, such as quantum registers, classical registers, quantum circuits, gates, qubit usage, and measurements. These abstractions provide a high-level representation of the program, enabling LintQ to perform robust static analyses rapidly without needing to thoroughly process the lower-level Python implementation details.

Built upon these abstractions, LintQ leverages CodeQL [15], a general-purpose, robust static analysis engine, to execute

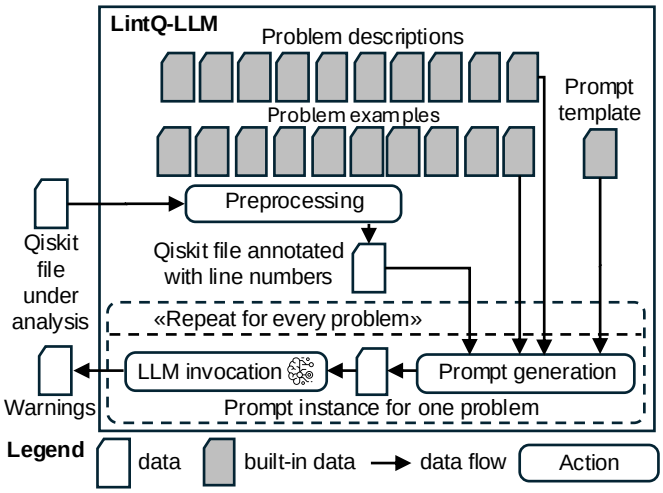

Fig. 2. Architecture and data flow overview of LintQ-LLM.

declarative queries over the behavioral representation of the Qiskit code. Each analysis corresponds to a query verifying against the ten problems described in Table I. A previous empirical evaluation of LintQ on 7568 real-world Qiskit programs achieved an overall precision of 62.5% [2]. While this demonstrated the feasibility and effectiveness of query-based problem detection, LintQ inherently relies on manually crafted deterministic patterns and rules. This architectural choice demands significant manual effort to maintain and limits the tool's adaptability to evolving quantum programming practices, thus highlighting the need for novel, more adaptable solutions to quality assurance in quantum software.

C. LLM-based Quantum Problem Detection: LintQ-LLM

To overcome the rigid nature of rule-based linters, prior work introduced LintQ-LLM [3], an LLM-based linting approach designed to identify quantum programming problems in Qiskit source code. The core logic to analyze a Qiskit code fragment relies on querying an LLM independently for each type of problem, ensuring high specificity while strictly respecting the operational context window of the model. Figure 2 illustrates the overall architecture of LintQ-LLM. LintQ-LLM analyzes one file at a time and prepares a separate prompt for each type of problem: in each prompt, it inserts only the instructions related to that check (i.e., a description of the problem and an example) along with the code to be analyzed. This keeps the context shorter, the code occupies more space in the token budget, and the model does not have to handle so many instructions at once, reducing the risk of confusion between the various constraints.

The problem detection strategy in LintQ-LLM relies on a single-prompt interaction. A zero-shot prompt is constructed to instruct the model to act as a source code linter with expert knowledge in quantum software. The prompt specifies the exact problem to detect, including its name, a detailed description, a generic example of the problem, and embeds the target source code to be analyzed. Finally, it mandates

that the response must be a single JSON object. While LintQ-LLM demonstrated the foundational feasibility of employing LLMs for quantum code analysis, the single-prompt mechanism structurally can limit the reasoning depth and context awareness of the model, occasionally resulting in FPs when analyzing complex quantum circuits. This represents the main motivation behind the definition of LintQ-LLM+CoT and LintQ-LLM+RAG, both introduced in the next section.

III. LintQ-LLM+CoT and LintQ-LLM+RAG

To enhance the performance of LintQ-LLM, we propose LintQ-LLM+CoT and LintQ-LLM+RAG, which significantly evolve the LLM interaction mechanism through CoT and incorporate RAG.

1) Multi-prompt Chain-of-Thought Pipeline: The problem detection strategy is implemented through a multi-prompt CoT interaction, which consists of one system prompt and two sequential user prompts (note that despite being called user prompts, their generation is automated by our approach). To ensure optimal structural layout and instruction clarity, these prompts were systematically generated and refined during development, leveraging the OpenAI Prompt Optimizer [16]. The overall structure of our prompting mechanism is shown in Figure 3 and summarized below.

The system prompt instructs the model to act as a source code linter with expert knowledge in quantum software and mandates that its response must be a single JSON object.

The first automated user prompt provides a comprehensive task definition: it specifies the exact problem to detect, including its name, a detailed description, and a concrete code example of the quantum programming problem (dynamically provided by the RAG module in LintQ-LLM+RAG). Specifically, this prompt guides the LLM through a multi-step reasoning process:

1) **Strategic Planning:** Formulate a "Detection Strategy" outlining the conceptual plan to identify the problem, including primary API elements and logical checks.
2) **Code Understanding:** Create a "Code Summary" to briefly describe the essential components and operations within the source code.
3) **Problem Detection Logic:** Apply the strategy to inspect the code and identify all instances of the problem.
4) **Report Results:** For each detected case, generate a JSON object containing the exact code "snippet", an array of "lines" numbers, and a detailed explanation. If no problems are found, an empty JSON object is returned.

The second automated user prompt delivers the line-numbered source code, instructing the model to perform the analysis based on the strategy established in the previous turn.

2) Knowledge base creation: LintQ-LLM+RAG incorporates a RAG system, leveraging a specialized vector database as its knowledge retrieval backend. The creation of this knowledge base followed a rigorous inclusion and exclusion protocol operating on the dataset previously evaluated by the original LintQ authors [2]. Figure 4 depicts the process responsible for building and indexing this knowledge base.

TABLE II
Composition of the RAG Knowledge Base by number of examples per Quantum problem Type.

Quantum problem	Count	Quantum problem	Count
OpAfterMeas	38	DoubleMeas	15
ConstClasBit	25	CondWoMeas	14
OversizedCircuit	20	OldIdenGate	10
InsuffClasReg	18	GhostCompose	6
MeasAllAbuse	16	OpAfterTransp	4

The foundation for the valid context injections is composed of manually verified True Positives (TPs). From a set of 7568 Qiskit files, Paltenghi et al. obtained 4699 warnings of quantum-related problem detection [2]. Of this 4699 warnings, they manually analyzed 345 warnings and annotated the warnings as TPs (True Positives), FPs (False Positives), or NW (Noteworthy, used for borderline cases). Although the meaning of TP and FP is well recognized, please note that they are formally defined in Section IV-C. At the end of the manual labeling, they obtained 165 TP instances [2]. We excluded FPs and NW cases from our knowledge base to ensure that the retrieval system grounds its reasoning strictly on unambiguous examples. We programmatically annotated the 165 files with the TP selected for our knowledge base: the lines explicitly causing the warning were commented with a standardized label `# Problem: <Problem Description>`.

After preparing the source files, we systematically embedded all candidate examples using OpenAI's `text-embedding-3-large` model [17]. This model ensures optimal semantic compatibility with the GPT-5 model family ultimately responsible for the reasoning step. To avoid exceeding the strict context window limit during execution, we filtered the selected files using the `tiktoken` library [18]: examples with more than 8192 tokens were excluded, resulting in the removal of eight files. The remaining subset corresponds to a knowledge base of 157 files. These files were sequentially vectorized and stored in a Facebook AI Similarity Search (FAISS) index [19] capable of determining similarity through Euclidean distance. The ultimate composition of the RAG database varies in available examples per problem type, distributed as reported in Table II.

3) Retrieval mechanism: In the LintQ-LLM+RAG execution phase, dynamically analyzing a target file translates into a comprehensive nearest-neighbor search paired with one-shot learning. As depicted in Figure 5, this approach evolves the LintQ-LLM architecture from an ungrounded inference model to a dynamically parameterized static analyzer. The integration of RAG into LintQ-LLM+RAG augments each LLM query with a retrieved example selected from the indexed knowledge base. Given the input Qiskit file, a preprocessing step annotates the code (e.g., with line numbers) and prepares it for analysis, while in parallel the system retrieves the most relevant problem example based on similarity with the current input.

When processing an incoming Qiskit program, its source code is initially tokenized to verify its bounds against the em-

System Instructions

You are a source code linter with expert knowledge in quantum software. Your response format must be a single JSON object.

Automated User Prompt 1

You are an **expert Qiskit source-code linter**. Your task is to analyze the provided Qiskit source code and detect all instances of the **[PROBLEM NAME AND SHORT DESCRIPTION]**.
Problem Description
Identify **every** occurrence of the **[PROBLEM NAME]** issue. Leverage your in-depth knowledge of Qiskit APIs and programming practices to accurately detect such cases.
Details / Examples:
{PROBLEM LONG DESCRIPTION AND EXAMPLE}
Step 1 Strategic Planning
Begin with a concise checklist (3-1 bullets) outlining the conceptual sub-tasks you will perform to ensure a complete and reproducible linting process. Start by formulating a **detection strategy** tailored to the [PROBLEM NAME AND SHORT DESCRIPTION] issue. Include:
- A concise conceptual overview of this issue in Qiskit.
- The **primary API elements or code patterns** that may reveal it.
- The **logical checks or contextual factors** that confirm its presence.
- A clear, stepwise search and verification plan.
Create this as the "Detection Strategy" before analyzing the code.
Step 2 Code Understanding
Carefully review the code between `<code>` and `</code>`. Briefly summarize its essential components, including:
- Definitions of QuantumCircuit, QuantumRegister, and ClassicalRegisterS
- Core Qiskit operations (e.g., `.measure`, `.h`, `.cx`, `.compose`, `.transpile`, etc.)
- Significant sequences, control flows, or resets
Create a short **"Code Summary"** (3-6 bullet points).
Step 3 Problem Detection Logic
Apply your **Detection Strategy** to thoroughly inspect the code for every instance of **[PROBLEM NAME AND SHORT DESCRIPTION]**:
1. Pinpoint relevant function invocations or Qiskit API usages.
2. Examine the context around each pattern for logical soundness.
3. Confirm that all criteria for **[PROBLEM NAME]** are satisfied.
4. Track usage of quantum and classical bits as needed.
5. Note and document each confirmed occurrence.
After analysis, validate your detection by quickly verifying that findings are directly supported by the Detection Strategy and the observed code; if not, self-correct before reporting.
Step 4 Report Results
For each detected case, provide an object matching this JSON schema:
```
  "snippet": <string, exact affected line(s) of code>,
  "lines": <integer array, 1-based line numbers>,
  "explanation": <string, justification based on your reasoning>
```
- List each occurrence as a separate object in a JSON array.
- Always use line numbers that match the input code.
- If there are no instances, output only: the JSON with a single warning but the warning with all empty fields.

Automated User Prompt 2

Source Code to Analyze
Please analyze the following source code for the **[PROBLEM NAME AND SHORT DESCRIPTION]** problem. Each line includes its line number in a comment.

<code>
[LINE NUMBERED CODE]
</code>

Fig. 3. The multi-prompt structure used by LintQ-LLM+CoT and LintQ-LLM+RAG. The interaction consists of a system prompt and two sequential automated user prompts designed to first establish a detection strategy with examples and then analyze the target source code.

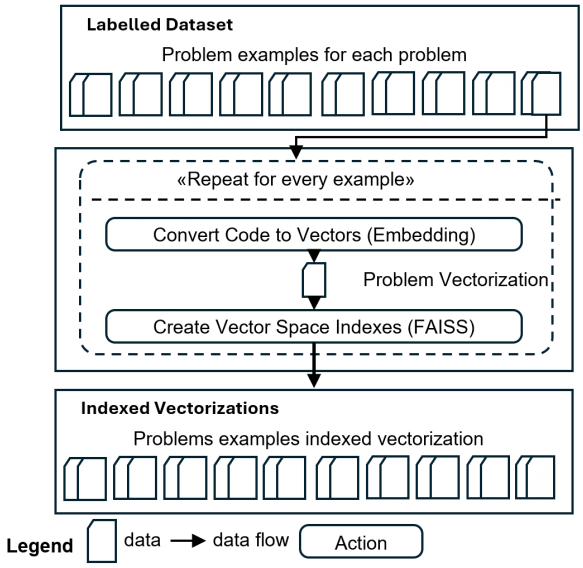

Fig. 4. Knowledge Base creation process for the RAG integration.

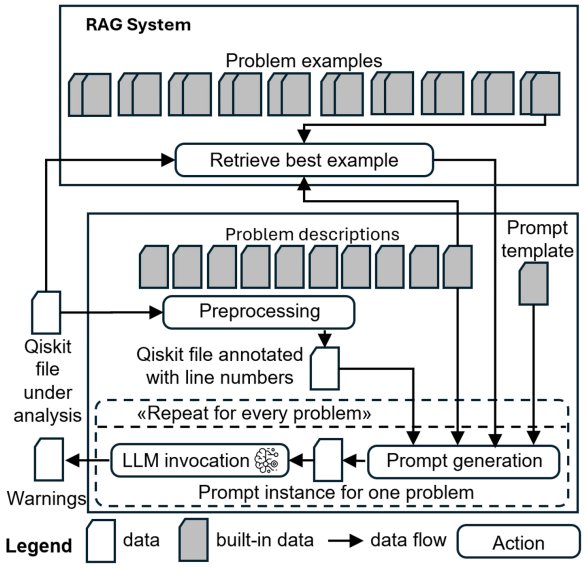

Fig. 5. Retrieval mechanism integration mapped in LintQ-LLM+CoT.

bedding constraint window. Assuming the valid 8192 bounds stand, the `text-embedding-3-large` model builds a vector representation of the current file. Due to the domain specificity of distinct problem types, the system operates across separate FAISS indices designed independently for each of the ten investigated problems.

For a specific quantum programming problem analysis, the framework queries the aligned FAISS index, recovering the single most semantically similar verified file ($k = 1$). By

retrieving a known, mathematically comparable instance containing the explicit problem, the LintQ-LLM+RAG architecture dynamically structures a reference snippet. This snippet, coupled with its embedded description of the problem's exact location, is then injected into the prompt. Consequently, this guides the final LLM validation by grounding its judgment in a high-quality, context-aware reference example retrieved from the FAISS knowledge base.

IV. Empirical Study Design

Our overarching goal is to assess whether shifting from rule-based static analysis to LLM-driven reasoning provides benefits in the detection of quantum problems in Qiskit programs. Given this goal, we formulate the following two research questions (RQs):

- **RQ1:** *How do LintQ-LLM+CoT and LintQ-LLM+RAG compare to LintQ in terms of effectiveness, correctness, and completeness in detecting quantum programming problems?*
  It focuses on a comparative evaluation against the current state of the art, aiming to determine whether the proposed LLM-based techniques represent a meaningful advancement over traditional rule-based static analysis.
- **RQ2:** *To what extent does LintQ-LLM+RAG enhance LintQ-LLM+CoT's ability to detect quantum programming problems with respect to correctness, completeness, and effectiveness?*
  This RQ narrows the focus to the individual LLM-based approaches, specifically investigating the contribution of RAG (if any) in improving detection performance (correctness, completeness, and effectiveness).

The rest of this section describes the experimental setup, the construction of the evaluation corpus, and the protocol followed to assess the efficacy of LintQ-LLM+CoT and LintQ-LLM+RAG. Figure 6 provides an overview of the used design.

A. Evaluation Corpus Construction

To address our RQs, we constructed a balanced stratified evaluation corpus. The process began with the original LintQ corpus consisting of 7568 Qiskit files. We first excluded the 157 files manually verified by Paltenghi et al. [2] and already incorporated into the RAG knowledge base. The remaining subset, consisting of 7411 files, was further filtered to include only those with a token count within the 8192 limit of the chosen embedding model, ensuring that the RAG retrieval mechanism could operate on the entire file content.

From this filtered subset, we performed a stratified selection focused on potential quantum problems identified by unverified LintQ reports. Our objective was to curate a balanced dataset of 55 files: up to five files for each of the ten supported quantum problem categories and five files flagged as clean (zero warnings) to serve as a control group. At this stage, the actual presence of the problems was unknown, as the selection relied on the original tool's unverified output.

For categories where the filtered subset contained fewer than five eligible files (i.e., extremely rare problem types), we utilized problem injection [20]. We manually introduced specific quantum problems into verified clean files following the patterns documented by Paltenghi et al. [2]. Concretely, we first analyzed the formal definitions and example instances of each problem provided in the LintQ work [2], and then reproduced those patterns by modifying existing circuits that did not originally exhibit the problem. This process resulted in a final Evaluation Corpus composed of 43 real-world files and 12 synthetic (injected) files, as detailed in Table III. Some of the files selected had multiple instances of different LintQ warnings; in fact, at the end of the sampling, and after the analysis of the synthetic files with LintQ, we had a total of 77 warnings across the 55 files.

TABLE III
Composition of the Evaluation Corpus across all quantum problem categories.

Quantum Problem	Real Files	Injected Files	Total
Clean Files	5	0	5
CondWoMeas	3	2	5
ConstClasBit	5	0	5
DoubleMeas	5	0	5
GhostCompose	1	4	5
InsuffClasReg	5	0	5
MeasAllAbuse	4	1	5
OldIdenGate	5	0	5
OpAfterMeas	5	0	5
OversizedCircuit	5	0	5
OpAfterTransp	0	5	5
Total	**43**	**12**	**55**

B. Experimental Procedure

The effectiveness of the proposed approaches was evaluated through four parallel identification actions performed on the Evaluation Corpus:

1) **Manual Validation**: A human annotator (the first author) conducted a line-by-line analysis of all 55 files to identify every occurrence of each quantum problem, establishing the Ground Truth for the study.
2) **Baseline Tool Execution**: We retrieved the original LintQ warnings for the 43 real-world files from the primary reports. For the 12 synthetic files, we executed the original rule-based LintQ to generate the baseline warnings.
3) **LintQ-LLM+CoT Inference**: The 55 files were analyzed by LintQ-LLM+CoT to generate warnings.
4) **LintQ-LLM+RAG Inference**: The Evaluation Corpus was processed by the retrieval-enhanced approach to generate warnings.

Finally, the outputs from LintQ, LintQ-LLM+CoT, and LintQ-LLM+RAG were compared against the established Ground Truth to calculate classification metrics.

C. Evaluation Metrics

To compute the metrics needed to study our RQs, we had to determine:

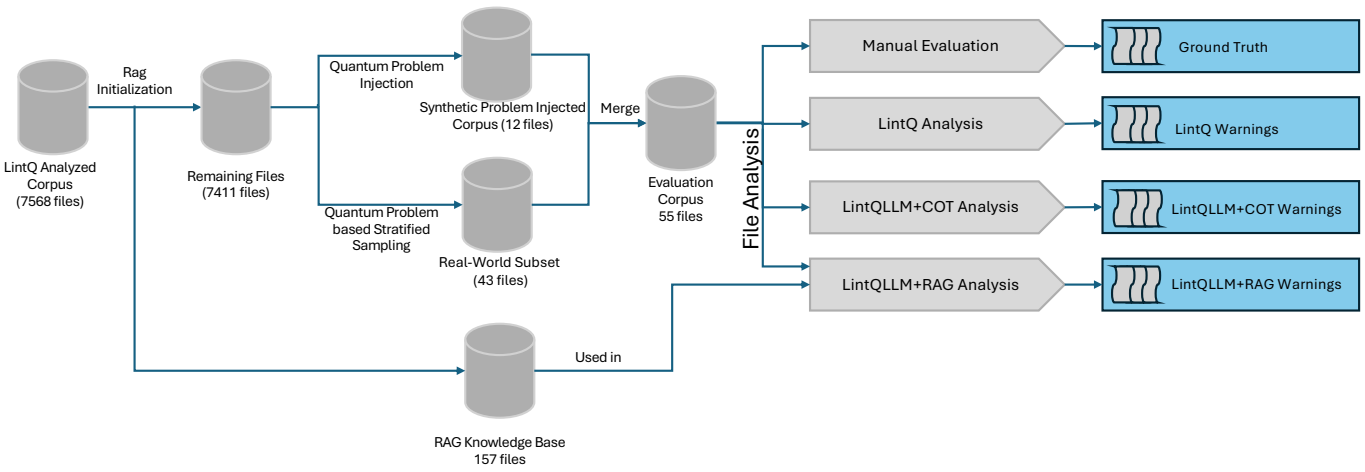

Fig. 6. Experimental design for the construction and validation of the Evaluation Corpus.

- **True Positive (TP)**: The tool correctly generates a warning for a line of code containing the specific quantum programming problem.
- **False Positive (FP)**: The tool generates a warning for a line of code that does not contain that specific quantum programming problem.
- **False Negative (FN)**: The tool fails to generate a warning for a line that contains a quantum programming problem, as confirmed by the ground truth.

We used TP, FP, and FN to calculate the following metrics:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{1}$$

It estimates the correctness/reliability of the warnings issued by a given approach. A high precision value implies that developers will waste less time investigating false alarms.

$$\text{Recall} = \frac{TP}{TP + FN} \tag{2}$$

It assesses the approach's completeness (i.e., sensitivity). A high recall value indicates the tool misses very few quantum programming problems.

$$\text{F1-score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{3}$$

It provides a single, balanced assessment of the approach's overall effectiveness by computing the harmonic mean of precision and recall. It accounts for the inherent trade-off between overwhelming the developer with false warnings (low precision) and missing critical quantum programming problems entirely (low recall). The higher its value, the better.

V. Results

This section presents the results of our empirical assessment and some complementary results to better understand the contribution of our research. Specifically, Table IV summarizes the findings across the 55 files forming our Evaluation Corpus.

TABLE IV
Overall Performance Comparison across all Problem Types.

Approach	TP	FP	FN	Precision	Recall	F1-Score
LintQ (Baseline)	30	47	40	0.39	0.43	0.41
LintQ-LLM+CoT	67	54	3	0.55	**0.96**	**0.70**
LintQ-LLM+RAG	60	47	10	**0.56**	0.86	0.68

The results suggest that both LintQ-LLM+CoT and LintQ-LLM+RAG substantially outperform the LintQ baseline in identifying quantum programming problems. In detail, LintQ-LLM+CoT achieved the highest recall value (0.96) and overall F1-score (0.70), demonstrating its capacity to identify almost every injected and real quantum programming problem in our corpus, keeping FNs to a minimum (3). Meanwhile, the integration of the semantic knowledge base in LintQ-LLM+RAG led to the highest precision value (0.56), minimizing the number of FPs reported compared to the base LLM approach, albeit at a cost in recall.

A. RQ1: How do LintQ-LLM+CoT and LintQ-LLM+RAG compare to LintQ in terms of effectiveness, correctness, and completeness in detecting quantum programming problems?

As far as the effectiveness is concerned, both LLM-based approaches outperform the baseline LintQ (see Table IV). LintQ-LLM+CoT achieves the highest F1-score of 0.70, and LintQ-LLM+RAG closely follows with 0.68, whereas the rule-based LintQ baseline only reaches 0.41. This suggests that transitioning from static, manually crafted queries to the flexible reasoning capabilities of LLMs provides a massive uplift in overall detection capability.

In terms of correctness, the integration of the semantic knowledge base in LintQ-LLM+RAG proves beneficial, achieving the highest overall precision value of 0.56. This means the RAG-enhanced approach minimizes the number of FPs reported. LintQ-LLM+CoT achieves a similar precision value of 0.55, but both represent a substantial improvement over the 0.39 precision value of LintQ. We attribute this

increased correctness to the CoT prompting and, for the RAG variant, to the one-shot learning examples that help ground the LLM's reasoning and reduce hallucinations.

Finally, in terms of completeness, LintQ-LLM+CoT demonstrates an outstanding capability of retrieving almost every quantum programming problem in the corpus, reaching a peak recall value of 0.96 with only 3 FNs. In comparison, LintQ-LLM+RAG achieves a recall value of 0.86, and the baseline falls short at 0.43. While RAG slightly penalizes recall by introducing stricter semantic constraints from the retrieved snippets, both LintQ-LLM+CoT and LintQ-LLM+RAG are evidently far superior to the baseline in comprehensively identifying quantum programming problems.

Summary of RQ1

Both LintQ-LLM+CoT and LintQ-LLM+RAG substantially outperform the rule-based LintQ baseline across all evaluated metrics. LintQ-LLM+CoT achieves the highest effectiveness (F1-score of 0.70) and completeness (Recall of 0.96), while LintQ-LLM+RAG provides the highest correctness (Precision of 0.56) by minimizing FPs.

B. RQ2: To what extent does LintQ-LLM+RAG enhance LintQ-LLM+CoT's ability to detect quantum programming problems with respect to correctness, completeness, and effectiveness?

To answer RQ2, we compare the performance of LintQ-LLM+CoT with its RAG augmented counterpart, LintQ-LLM+RAG, both shown in Table IV. As for the correctness of the detected quantum program problems, the introduction of the RAG architecture yielded an improvement, even if it does not seem significant. Indeed, the precision value increased from 0.55 to 0.56, actively reducing the aggregate number of FPs from 54 down to 47. By grounding the reasoning on manually verified data, it seems that the LLM becomes more calibrated. However, this gain in correctness incurs a penalty in completeness. The recall metric dropped from a near-perfect 0.96 in the LintQ-LLM+CoT configuration down to 0.86 for LintQ-LLM+RAG, with FNs increasing from 3 to 10. A plausible explanation is that the explicit nature of the retrieved one-shot snippets acts as an overly rigid template. In other words, the strict semantic constraints introduced by RAG blind the model to valid, yet structurally diverse, implementations of the same quantum programming problem.

In terms of effectiveness, the RAG integration marginally underperformed LintQ-LLM+CoT. The decrease in Recall outweighed the slight improvement in Precision, resulting in a minor decline in the overall F1-score from 0.70 to 0.68.

Summary of RQ2

The use of RAG introduces a trade-off between correctness and completeness in the detection of quantum programming problems. Its use slightly improves correctness (Precision increases from 0.55 to 0.56) at the cost of a reduced completeness of the detected program problems (Recall drops from 0.96 to 0.86). As a result, the overall effectiveness (0.68 vs. 0.70) slightly decreases, indicating that the gain in precision does not compensate for the loss in recall. Therefore, RAG enhances reliability but limits generalization, making it beneficial in scenarios where precision is prioritized, but less suitable when comprehensive detection is strongly required.

C. Additional Analyses

1) On the Impact of Multi-prompt and CoT : For completeness, we performed a comparative analysis between LintQ-LLM+CoT and LintQ-LLM introduced in prior work [3]. LintQ-LLM achieved a high completeness with a recall value of 0.99 (TP = 69, FN = 1). However, this result comes at the expense of correctness, as the precision value is only 0.32 due to an exceptionally high number of FPs (145). This outcome suggests that while single-prompt strategies are effective at identifying potential problems, they lack the necessary capability to distinguish between correct quantum programming code and quantum programming problems, resulting in an F1-score of 0.49.

The transition to a multi-prompt CoT pipeline (LintQ-LLM+CoT) marks a performance milestone. By guiding the LLM, we achieved an increase in Precision to 0.55, effectively reducing the number of FPs by 62.76% (from 145 to 54). Remarkably, this refinement in correctness does not significantly compromise completeness, as the Recall remains high at 0.96. Consequently, the overall effectiveness (F1-score) rises to 0.70, representing a 30% improvement over LintQ-LLM. Based on the achieved results, similar results and considerations can also be done when comparing LintQ-LLM and LintQ-LLM+RAG.

2) Obfuscation and Data Leakage: A significant validity threat in LLM-based software engineering research is data leakage—the possibility that the underlying model may have encountered our test corpus during its pre-training phase. To empirically address this, we conducted a preliminary experiment using a custom obfuscator, structurally mapping user-defined identifiers to random strings while preserving syntactic integrity and Qiskit API interactions. Testing a sample of 14 files under both obfuscated and clear conditions (as detailed in Table V) revealed that LintQ-LLM+CoT maintained comparable problem detection capability regardless of identifier obfuscation. This suggests the model anchors its reasoning on the topological properties of the quantum circuit construction rather than memorized sequences due to data leakages. Conversely, the semantic representations within the RAG solution proved highly sensitive to obfuscation, disjointing the retrieval

TABLE V
PERFORMANCE COMPARISON BETWEEN OBFUSCATED (OBF) AND NON-OBFUSCATED (CLEAR) INPUTS ON A SAMPLE OF 14 FILES.

Configuration	TP	FP	FN
LintQ-LLM+RAG (OBF)	4	0	8
LintQ-LLM+CoT (OBF)	8	3	4
LintQ-LLM+RAG (Clear)	5	0	7
LintQ-LLM+CoT (Clear)	7	1	5

correlations (only 97 out of 140 retrieval queries successfully aligned with their un-obfuscated counterparts). Consequently, since obfuscation hampered retrieval metrics without yielding significant data leakage issues in LintQ-LLM+CoT execution, the primary experiment was conducted using un-obfuscated Qiskit programs to maximize the structural efficacy.

3) RAG and Types of Quantum Programming Problems: Table VI details the values for Precision, Recall, and F1-score for both LintQ-LLM+CoT and LintQ-LLM+RAG across the ten different quantum programming problem categories. The differences in the achieved results between these approaches are also reported. The effect of introducing RAG varies significantly depending on the specific problem type. For certain categories, RAG provides a notable performance uplift: *DoubleMeas* elevated its F1-score from 0.80 to 0.92 (due to improved precision), and *InsuffClasReg* improved from 0.16 to 0.33 (also due to improved precision). Conversely, other problem categories showed a regression in F1-scores when implementing RAG. Specifically, in *OpAfterMeas* F1-score drop from 0.82 to 0.71 (driven by a decrease in recall), and *ConstClasBit* decreased from 0.59 to 0.55 (driven by a slight decrease in precision). Furthermore, several problem types, such as *OpAfterTransp*, *GhostCompose*, *OldIdenGate*, and *MeasAllAbuse*, maintained a perfect classification score (F1-score = 1.0) across both configurations.

These findings suggest that the effectiveness of RAG is highly dependent on the quantum programming problem. We can also postulate on the dimension of the knowledge base available for each problem: the sheer number of examples does not appear to be directly associated with better performance. In fact, some of the most populated categories (e.g., *OpAfterMeas* with 38 examples) suffered deteriorations, while categories with very few examples (e.g., *OpAfterTransp* with 4 examples) maintained perfect scores. This suggests that the quality and structural representation of the examples are more critical than their quantity.

VI. DISCUSSION AND PRACTICAL IMPLICATIONS

Our study highlights a shift in the potential mechanisms for quality assurance within quantum software engineering. The substantial outperformance of LLM-based approaches over the state-of-the-art rule-based tool (LintQ) suggests that generative models can effectively function as "expert linters". Unlike static analyzers that rely on rigid, declarative queries, LLMs can dynamically adapt to the rapidly evolving scene of quantum computing programming, effectively transferring the maintenance burden from manual rule creation to prompt engineering and knowledge base curation. For practitioners and tool developers/vendors, this implies that integrating LLMs into IDEs or continuous integration pipelines could offer a more robust and adaptable first line of defense against quantum programming problems.

While RAG successfully grounded the LLM and improved the correctness of the detection of quantum program problems, it simultaneously constrained the model, causing it to dismiss valid quantum programming problems that deviated structurally from the retrieved templates. To some extent, this finding reveals a trade-off that could be of interest tp practitioners. That is, in a context where reducing FNs is the priority, the use of CoT and RAG (i.e., LintQ-LLM+RAG) could be preferable. Conversely, if high completeness is required to ensure no quantum programming problem slips through, the LintQ-LLM+CoT approach could be preferred. The results also suggest that when it comes to RAG for quantum programming type, bigger is not necessarily better. Success seems driven by the structural and syntactic clarity of the problems rather than the sheer volume of the knowledge base. This outcome is clearly relevant for researchers interested in understanding when well-structured examples are more valuable than several generic data points (thus favoring data curation over data collection).

Furthermore, our investigation into code obfuscation revealed that the base generative model accurately identifies quantum programming problems based on their topological and structural properties. This resilience is promising in a context where proprietary quantum algorithms might need to be obfuscated before being analyzed by external cloud-based LLMs. Unfortunately, the evidence gathered so far shows that the semantic embeddings utilized by the FAISS index are highly sensitive to obfuscation, severely degrading the retrieval phase. These considerations point toward the necessity of developing structural or AST-aware embedding techniques that can withstand syntactic obfuscation while preserving the semantic context of quantum operations.

Overall, these considerations are not only clearly relevant to researchers but maybe of interest also to tool developers/vendors developing resilient, privacy-preserving automated analysis tools and advancing structural code representation models for quantum software development.

VII. THREATS TO VALIDITY

In the following, we present the possible threats that could affect the validity of the obtained results.

a) Internal Validity: The performance of LintQ-LLM+CoT and LintQ-LLM+RAG heavily depends on the reasoning capabilities of the used LLM and its non-determinism. In our study, we utilized a single state-of-the-art GPT-5 model for all inference tasks. We acknowledge that different LLMs might exhibit varying degrees of proficiency, potentially altering the observed effectiveness. As for the non-determinism, we used the default configuration for the LLM.

TABLE VI
COMPARISON BETWEEN LINTQ-LLM+COT AND LINTQ-LLM+RAG BY TYPE OF QUANTUM PROGRAMMING PROBLEM.

Quantum Problem	Number of examples available in RAG Knowledge base	LintQ-LLM+CoT			LintQ-LLM+RAG			RAG Impact (RAG–COT)		
		Prec.	Rec.	F1	Prec.	Rec.	F1	ΔPrec.	ΔRec.	ΔF1
OpAfterTransp	4	1.00	1.00	1.00	1.00	1.00	1.00	0.00	0.00	0.00
GhostCompose	6	1.00	1.00	1.00	1.00	1.00	1.00	0.00	0.00	0.00
OldIdenGate	10	1.00	1.00	1.00	1.00	1.00	1.00	0.00	0.00	0.00
CondWoMeas	14	0.83	1.00	0.91	0.64	0.80	0.73	-0.17	-0.20	-0.18
DoubleMeas	15	0.64	1.00	0.80	0.86	1.00	0.92	0.19	0.00	0.12
MeasAllAbuse	16	1.00	1.00	1.00	1.00	1.00	1.00	0.00	0.00	0.00
InsuffClasReg	18	0.09	0.64	0.16	0.22	0.64	0.33	0.13	0.00	0.17
OversizedCircuit	20	0.11	0.33	0.17	0.11	0.67	0.19	0.00	0.33	0.02
ConstClasBit	25	0.42	1.00	0.59	0.38	1.00	0.55	-0.04	0.00	-0.04
OpAfterMeas	38	0.69	1.00	0.82	0.69	0.72	0.71	0.0	-0.28	-0.11

Finally, to construct the Evaluation Corpus and ensure adequate representation of extremely rare problems, we injected synthetic quantum programming problems into a subset of files. While these injections structurally mirrored documented quantum programming problems, they might not perfectly replicate the complex topologies of organic, developer-induced problems. Finally, the semantic retrieval mechanism in LintQ-LLM+RAG relies exclusively on the `text-embedding-3-large` model. The choice of the embedding model and distance metric directly governs the quality of the injected one-shot examples. A different embedding strategy could potentially impact the results.

b) External Validity: While the abstract generative nature of LLMs theoretically allows for the analysis of other quantum languages such as Cirq or Q#, we cannot claim that our quantitative results generalize beyond the Qiskit ecosystem [1].

Finally, we cannot guarantee that our results perfectly generalize to all quantum programs developed in the wild. Although our stratified Evaluation Corpus consists of 55 files covering all ten recognized quantum programming problem families, it may not encapsulate the entirety of syntactic and structural variations present in larger-scale quantum repositories. To partially mitigate this threat, we provide a replication package available online [21].

c) Construct Validity: A significant threat to construct validity lies in how the "ground truth" was established. Specifically, our manual validation was performed by a single human annotator. This introduces inherent risks associated with the subjective interpretation of quantum programming problems. To mitigate this threat, the annotator (the first author) strictly adhered to the formal problem definitions and code patterns exhaustively documented in prior literature [2]. In case of doubts, which never occurred, we planned for the annotator to consult with one of the other authors to resolve them.

While Precision, Recall, and F-score are standard metrics, they treat all quantum programming problems equally. That is, there could be quantum programming problems less critical (e.g., OldIdenGate) than others because they are less detrimental, and we did not take this aspect into account.

VIII. CONCLUSION

We explored the feasibility and potential of LLMs to act as expert linters for quantum program analysis. Specifically, we introduced and assessed LintQ-LLM+CoT and LintQ-LLM+RAG, two variants of the LintQ-LLM proposed in prior work [3], by integrating a multi-prompt Chain-of-Thought (CoT) pipeline with and without a Retrieval-Augmented Generation (RAG) extension grounded in manually verified quantum programming problem examples. To evaluate LintQ-LLM+CoT and LintQ-LLM+RAG, we applied them to a comprehensive, stratified Evaluation Corpus of 55 quantum programs. Our empirical results clearly indicate that transitioning from rigid, deterministic queries to flexible LLM-based reasoning yields a massive uplift in detection capabilities. While state-of-the-art static analysis tools like LintQ inherently struggle with adaptability, the foundational LintQ-LLM+CoT achieved the highest overall capability (F1-score of 0.70) by effectively generalizing across structural nuances. Concurrently, LintQ-LLM+RAG demonstrated the highest correctness metric (Precision of 0.56) by seamlessly utilizing context-informed, one-shot examples to reject misleading patterns and actively minimize FP alerts.

Looking ahead, our study sets the stage for the following future research directions:

- **AST-Aware Embeddings:** Our preliminary testing showed that standard semantic embeddings (like those in FAISS) are highly sensitive to syntactic obfuscation. Future efforts must focus on developing structural or Abstract Syntax Tree (AST)-aware embedding techniques that withstand code obfuscation while preserving the deep semantic context of quantum operations.
- **Human-Centered Interactions:** As LLM-powered linters mature into IDEs, evaluating the human dimension becomes paramount. Future human-based empirical studies should explore how developers perceive, interact with, and act on LLM-generated explanations compared with traditional linter outputs, and possibly explore hybrid systems that merge strict static tracking with generative recommendations.

REFERENCES

[1] I. Quantum. (2024) Qiskit: An open-source framework for quantum computing. [Online]. Available: https://qiskit.org/

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

[3] S. Y. Shin, F. Pastore, and D. Bianculli, "Quantum program linting with llms: Emerging results from a comparative study," in *2025 IEEE International Conference on Quantum Computing and Engineering (QCE)*, vol. 02, 2025, pp. 181–186.

[4] J. Wang, Q. Zhang, G. H. Xu, and M. Kim, "Qdiff: Differential testing of quantum software stacks," in *36th IEEE/ACM International Conference on Automated Software Engineering*, 2021.

[5] M. Paltenghi and M. Pradel, "Morphq: Metamorphic testing of the qiskit quantum computing platform," in *Proceedings of the 45th International Conference on Software Engineering*, 2023.

[6] C. S. Xia, M. Paltenghi, J. L. Tian, M. Pradel, and L. Zhang, "Fuzz4all: Universal fuzzing with large language models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024.

[7] A. Miranskyy, L. Zhang, and J. Doliskani, "Is your quantum program bug-free?" *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, 2020.

[8] S. Ali, P. Arcaini, X. Wang, and T. Yue, "Assessing the effectiveness of input and output coverage criteria for testing quantum programs," in *14th IEEE Conference on Software Testing, Verification and Validation*, 2021.

[9] J. Wang, F. Ma, and Y. Jiang, "Poster: Fuzz testing of quantum program," in *14th IEEE Conference on Software Testing, Verification and Validation*, 2021.

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

[11] G. Li, L. Zhou, N. Yu, Y. Ding, M. Ying, and Y. Xie, "Projection-based runtime assertions for testing and debugging quantum programs," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, 2020.

[12] Q. Chen *et al.*, "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.

[13] 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.

[14] M. Kaul, A. Küchler, and C. Banse, "A uniform representation of classical and quantum source code for static code analysis," in *Proceedings of the 2023 IEEE International Conference on Quantum Computing and Engineering*, 2023, pp. 1013–1019.

[15] P. Avgustinov, O. de Moor, M. P. Jones, and M. Schäfer, "QL: Object-oriented queries on relational data," in *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, pp. 2:1–2:25.

[16] OpenAI, "Prompt optimization cookbook," 2025. [Online]. Available: https://developers.openai.com/cookbook/examples/gpt-5/prompt-optimization-cookbook

[17] ——, "Text embedding 3 large: Model card," 2024. [Online]. Available: https://developers.openai.com/api/docs/models/text-embedding-3-large

[18] ——, "How to count tokens with tiktoken," 2022. [Online]. Available: https://cookbook.openai.com/examples/how_to_count_tokens_with_tiktoken

[19] M. Douze, A. Guzhva, C. Deng, J. Johnson, G. Szilvasy, P.-E. Mazaré, M. Lomeli, L. Hosseini, and H. Jégou, "The faiss library," 2025. [Online]. Available: https://arxiv.org/abs/2401.08281

[20] R. Natella, D. Cotroneo, and H. S. Madeira, "Assessing dependability with software fault injection: A survey," *ACM Comput. Surv.*, 2016.

[21] P. Cassieri, G. Scanniello, S. Y. Shin, F. Pastore, and D. Bianculli, "Replication package - LintQ-LLM+RAG," 2026. [Online]. Available: https://doi.org/10.6084/m9.figshare.32098114