QuanBench: Benchmarking Quantum Code Generation with Large Language Models

Xiaoyu Guo†
Kyushu University
guo.xiaoyu.961@s.kyushu-u.ac.jp

Minggu Wang†
Kyushu University
wang.minggu.065@s.kyushu-u.ac.jp

Jianjun Zhao*
Kyushu University
zhao@ait.kyushu-u.ac.jp

***Abstract*—Large language models (LLMs) have demonstrated good performance in general code generation; however, their capabilities in quantum code generation remain insufficiently studied. This paper presents QuanBench, a benchmark for evaluating LLMs on quantum code generation. QuanBench includes 44 programming tasks that cover quantum algorithms, state preparation, gate decomposition, and quantum machine learning. Each task has an executable canonical solution and is evaluated by functional correctness (Pass@K) and quantum semantic equivalence (Process Fidelity). We evaluate several recent LLMs, including general-purpose and code-specialized models. The results show that current LLMs have limited capability in generating the correct quantum code, with overall accuracy below 40% and frequent semantic errors. We also analyze common failure cases, such as outdated API usage, circuit construction errors, and incorrect algorithm logic. QuanBench provides a basis for future work on improving quantum code generation with LLMs.**

***Index Terms*—Quantum Code Generation, Large Language Models, Benchmarking, Quantum Programming, Code Evaluation, Quantum Algorithm**

I. Introduction

Recent advances in LLMs have significantly advanced software engineering tasks such as code generation [1], [2], synthesis [3], and completion [4], [5], [6]. Transformer-based models such as ChatGPT [7], DeepSeek [8], Gemini [9], and LLaMA [10] demonstrate strong performance on general-purpose programming benchmarks, including HumanEval [11], MBPP [3], and DS-1000 [12], enabling the generation of standalone functions directly from natural language prompts. These benchmarks have supported the rapid adoption of LLMs in classical software development workflows.

While these models have achieved strong results on classical programming tasks, their capabilities in specialized domains such as quantum programming remain underexplored. Quantum programming differs from classical programming in both syntax and semantics [13], [14]. It operates on quantum bits (qubits) rather than classical bits, involves unitary and reversible transformations, and depends on linear algebra and measurement-based computation [15]. These characteristics present unique challenges for LLM-based code generation, including the generation of semantically valid quantum circuits, the preparation of entangled states, gate decomposition, and the correct implementation of quantum algorithms.

† These authors contributed equally to this work.
* Corresponding author.

Although Qiskit HumanEval [16] has been proposed to evaluate quantum code generation by adapting Qiskit-specific APIs, its focus remains largely on API compliance rather than on testing algorithmic reasoning or quantum semantic correctness. Currently, no benchmark systematically evaluates whether LLMs can generate functionally correct and semantically meaningful quantum programs from natural language descriptions, which is essential for practical quantum software development.

To address this gap, we present QuanBench, a benchmark designed to evaluate LLMs on quantum code generation tasks. Built on the Qiskit framework (version 0.46.0), QuanBench comprises 44 carefully curated tasks, each accompanied by a natural language prompt, a canonical implementation, and unit tests. The benchmark covers four categories of quantum programming tasks: quantum algorithm implementation (e.g., Grover's algorithm [17], Quantum Fourier Transform [18]), quantum state preparation (e.g., Bell and GHZ states), gate decomposition, and quantum machine learning (e.g., variational circuits [19]).

We evaluate model outputs using both functional correctness and quantum semantic equivalence. Functional correctness is measured by Pass@K, which evaluates whether at least one generated solution passes the correctness tests. Quantum semantic equivalence is assessed using Process Fidelity, which measures the similarity between the quantum operations implemented by the generated code and the canonical solution. We evaluate nine recent LLMs, including general-purpose instruction-tuned models, such as GPT-4.1 [7], Claude 3.7 [20], and Gemini 2.5 [9], and code-specialized models, such as CodeLlama [10] and DeepSeek [21].

Our results show that, while some models can occasionally generate correct quantum code, their overall capabilities remain limited. The Pass@1 accuracy remains below 40%, and the highest Pass@5 score reaches only 50%. Most models solve only a subset of tasks, and several problems remain unsolved by any model. DeepSeek R1 achieves the highest task coverage, particularly in quantum state preparation; however, even the best model reaches only an average Process Fidelity of 51.8%. Furthermore, model performance does not correlate strictly with model size, indicating that domain-specific adaptation has a stronger impact than model scale alone.

This paper makes the following contributions:

- We introduce *QuanBench*, a benchmark for systematically

evaluating LLM-based quantum code generation for various quantum programming tasks.
- We conduct comprehensive experiments on nine LLMs, evaluating both functional correctness and quantum semantic equivalence.
- We analyze common failure patterns in LLM-generated quantum code, identify key limitations in current models, and provide insights for future improvement.

The remainder of this paper is organized as follows. Section II provides background on quantum computing and LLM-based code generation. Section III describes the construction of QuanBench. Section IV presents the experimental setup, research questions, and evaluation metrics. Section V presents the experimental results. Section VI outlines the threats to validity. The related work is discussed in Section VII, and the paper is concluded in Section VIII.

II. Background

This section presents the background necessary to support the design and evaluation of QuanBench. We briefly review LLMs, existing benchmarks for code generation, and the fundamentals of quantum computing relevant to quantum code generation tasks.

A. Large Language Models

Large language models (LLMs) [22], built on the Transformer architecture, have demonstrated strong generalization across various natural language processing tasks, including text summarization, logical reasoning, sentiment analysis, and mathematical problem-solving. Models such as ChatGPT [23], DeepSeek [8], and Gemini [24] are pre-trained on large-scale text corpora and fine-tuned with instruction-following objectives, enabling broad applicability across both general and specialized domains.

Despite these advances, general-purpose LLMs still face limitations in code generation tasks such as program synthesis, bug fixing, and code summarization. For example, LLaMA 2-7B [10] achieves only 44.4% Pass@100 on the HumanEval benchmark [11], indicating the challenges in reliably generating functionally correct code.

To address this, code-specific LLMs have been developed [21], [25]. Models such as StarCoder [26], CodeLlama [27], and Codex [11] use domain-specific pre-training and fine-tuning to enhance programming capabilities. For example, CodeLlama-7B achieves 85.9% Pass@100 on HumanEval [11], showing substantial improvements over general-purpose models. These results highlight the importance of domain specialization in LLM training for code generation tasks.

B. Benchmarks for Code Generation

Multiple benchmarks have been introduced to evaluate LLMs in code generation. HumanEval [11] is widely used, offering short programming problems with test cases to assess functional correctness. CodeContests [28] extends evaluation to competitive programming tasks, while EvalPlus [29] expands HumanEval with additional test coverage. RMCBench [30] focuses on evaluating LLM resistance against malicious code.

For the quantum domain, Qiskit HumanEval [16] adapts the HumanEval format to quantum programming using Qiskit syntax and constructs. However, it primarily evaluates API usage and syntax compliance rather than algorithmic reasoning or quantum semantic correctness.

To address this gap, we propose QuanBench, a benchmark designed to evaluate the ability of LLMs to generate functionally correct and semantically meaningful quantum code. In contrast to Qiskit HumanEval, QuanBench focuses on algorithmic reasoning and quantum-semantic fidelity, covering a broader set of quantum tasks grounded in real algorithmic challenges.

C. Quantum Computing

Quantum computing uses principles of quantum mechanics, including superposition, entanglement, and unitary evolution, to process information in a manner distinct from classical computing [15]. A quantum bit (qubit) can exist in a superposition of basis states $|0\rangle$ and $|1\rangle$, represented as:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle, \text{ where } |\alpha|^2 + |\beta|^2 = 1. \tag{1}$$

Quantum computation applies unitary gates to manipulate the qubit states. Gates include single-qubit operations (e.g., Hadamard, Pauli gates) and multi-qubit gates (e.g., CNOT, Toffoli gates), forming quantum circuits that implement algorithms. For an N-qubit system, a quantum gate is represented by a $2^N \times 2^N$ unitary matrix.

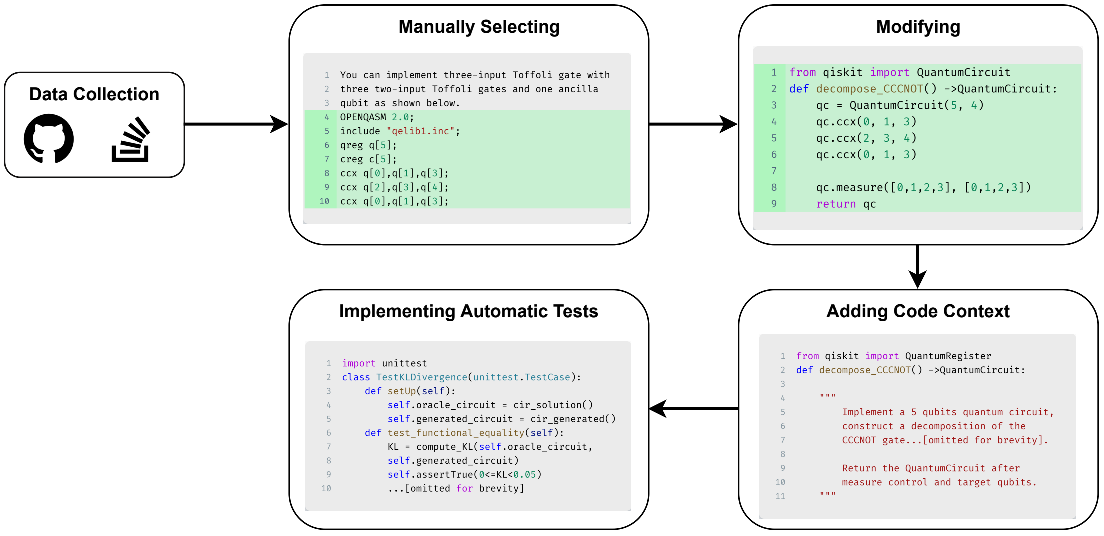
Several frameworks support the development of quantum programs, including Qiskit [31], Cirq [32], and Q# [33]. These toolkits provide abstractions for circuit design, simulation, and execution on both simulators and real quantum hardware.

Classical quantum algorithms such as Grover's search [34], Quantum Fourier Transform (QFT) [18], and Deutsch-Jozsa [35] demonstrate computational advantages over classical methods. More recently, parameterized quantum circuits (also known as variational circuits) [19] have attracted attention for hybrid quantum-classical computation in applications such as quantum chemistry and optimization.

These properties present unique challenges for code generation models, including handling circuit structure, maintaining quantum state coherence, and correctly composing quantum operators.

Despite these developments, quantum programming remains a challenging task [13]. Unlike classical programming, the semantics of quantum gate operations depend on the algorithmic context, making program design, reasoning, and automation complex.

In parallel, LLMs have shown strong capabilities in classical code generation, raising the question of whether LLMs can also handle quantum program semantics and algorithmic intent. To investigate this, we introduce QuanBench, a benchmark designed to systematically assess LLMs' ability to generate quantum programs across diverse algorithmic tasks,



```
You can implement three-input Toffoli gate with
three two-input Toffoli gates and one ancilla
qubit as shown below.
OPENQASM 2.0;
include "qelib1.inc";
qreg q[5];
creg c[5];
ccx q[0],q[1],q[3];
ccx q[2],q[3],q[4];
ccx q[0],q[1],q[3];
```

```
from qiskit import QuantumCircuit
def decompose_CCCNOT() ->QuantumCircuit:
    qc = QuantumCircuit(5, 4)
    qc.ccx(0, 1, 3)
    qc.ccx(2, 3, 4)
    qc.ccx(0, 1, 3)

    qc.measure([0,1,2,3], [0,1,2,3])
    return qc
```

```
from qiskit import QuantumRegister
def decompose_CCCNOT() ->QuantumCircuit:

    """
        Implement a 5 qubits quantum circuit,
        construct a decomposition of the
        CCCNOT gate...[omitted for brevity].

        Return the QuantumCircuit after
        measure control and target qubits.
    """
```

```
import unittest
class TestKLDivergence(unittest.TestCase):
    def setUp(self):
        self.oracle_circuit = cir_solution()
        self.generated_circuit = cir_generated()
    def test_functional_equality(self):
        KL = compute_KL(self.oracle_circuit,
        self.generated_circuit)
        self.assertTrue(0<=KL<0.05)
        ...[omitted for brevity]
```

Fig. 1: QuanBench Design

using semantic evaluation metrics and canonical solutions, to assess both algorithmic reasoning and quantum semantic correctness.

III. Benchmark Construction

The construction of QuanBench consists of three stages, as shown in Figure 1.

- First, we curated a diverse set of high-quality quantum algorithm implementations by collecting programs from open-source GitHub repositories and extracting relevant issues from StackOverflow.
- Second, each collected item was reformulated as a clearly defined programming problem with a corresponding canonical solution. All canonical solutions are executable and faithfully represent the intended quantum algorithms.
- Third, we implemented a multi-criteria evaluation pipeline for each benchmark problem, which includes functional correctness checks, comparative testing, and constraint-based validation. This evaluation framework allows us to assess both the syntactic correctness and the quantum semantic equivalence of the model-generated code.

A. *Problem Selection*

• **Sourcing Problems.** To collect high-quality benchmark problems, we retrieve repositories containing quantum program content by searching for keywords ”Quantum algorithm” and ”Qiskit” from GitHub (questions from Stack Overflow), using the following filtering process:

- First, temporal filtering was applied by including only repositories created after 2023, which helps reduce potential overlap with the training data of current LLMs, whose datasets primarily cover pre-2023 content.
- Second, popularity-based filtering is applied: GitHub repositories with more than 5 stars and Stack Overflow posts with more than 50 views are retained. Using this threshold as a preliminary proxy for community validation regarding utility and correctness.

This process collected a total of 127 GitHub repositories and 39 Stack Overflow questions, which were then manually screened for quality and diversity.

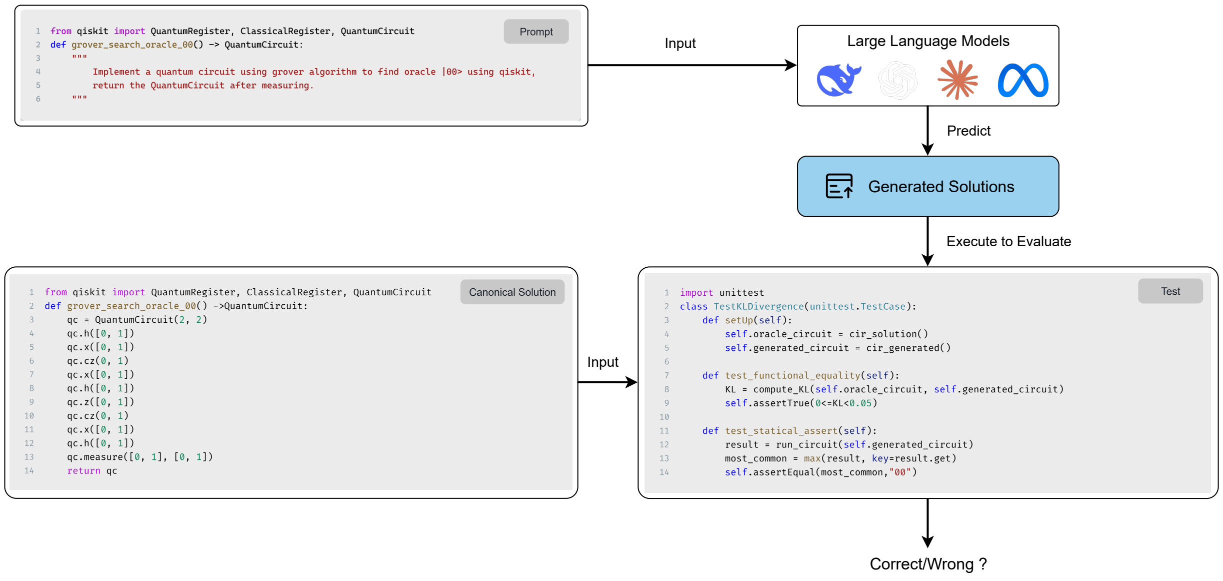
• **Filtering Suitable Problems.** From the initial repository and program pool, we applied additional manual screening based on the following criteria:

- To eliminate ambiguity, each program must be accompanied by a clear and well-defined problem description, and testable behaviors.
- Each collected program is required to successfully execute without errors and to yield correct outputs in accordance with its behavior.
- To eliminate redundancy, programs that employed the same algorithmic approach to address the same problem were excluded.

Furthermore, we prefer programs that demonstrate scalability and extensibility, meaning that they can be generalized to handle larger numbers of qubits and be adapted to a broader range of quantum tasks. After applying these criteria, we curated a final set of 44 unique quantum algorithm programs to include in the benchmark suite. (42 from GitHub and 2 from Stack Overflow)

B. *Rewriting Problems and Canonical Solutions*

• **Version Control.** Since the training data of LLMs such as ChatGPT typically extends only up to 2023, these models often lack familiarity with subsequent versions of Qiskit. To ensure environmental consistency and minimize potential



```python
from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit
def grover_search_oracle_00() -> QuantumCircuit:
    """
        Implement a quantum circuit using grover algorithm to find oracle |00> using qiskit,
        return the QuantumCircuit after measuring.
    """
```

```python
from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit
def grover_search_oracle_00() ->QuantumCircuit:
    qc = QuantumCircuit(2, 2)
    qc.h([0, 1])
    qc.x([0, 1])
    qc.cz(0, 1)
    qc.x([0, 1])
    qc.h([0, 1])
    qc.z([0, 1])
    qc.cz(0, 1)
    qc.x([0, 1])
    qc.h([0, 1])
    qc.measure([0, 1], [0, 1])
    return qc
```

```python
import unittest
class TestKLDivergence(unittest.TestCase):
    def setUp(self):
        self.oracle_circuit = cir_solution()
        self.generated_circuit = cir_generated()

    def test_functional_equality(self):
        KL = compute_KL(self.oracle_circuit, self.generated_circuit)
        self.assertTrue(0<=KL<0.05)

    def test_statical_assert(self):
        result = run_circuit(self.generated_circuit)
        most_common = max(result, key=result.get)
        self.assertEqual(most_common,"00")
```

Fig. 2: One Example in QuanBench

confounding effects on LLM performance, we standardized the environment to Qiskit 0.46.0 and Python 3.10. This choice does not materially affect either the prompts or the solutions, as our tasks rely solely on the core quantum circuit logic, which remains compatible across different versions. The only differences arise in the construction and execution of the evaluation metrics (e.g., simulators and transpilation tools), which rely on consistent backend behavior.

• **Translation.** Since not all programs were originally implemented in Qiskit, some were written in frameworks such as Cirq or Classiq, and we manually translated them into Qiskit. To ensure the correctness of the translation, we manually verify that the structure of the translated quantum circuit matches the original and that both produce identical outputs.

• **Core Circuit Logic Extraction.** QuanBench is designed around Qiskit-based quantum programming tasks, which may include simulation or hybrid optimization logic beyond the quantum circuit itself (e.g., QAOA integrates classical optimization). Due to substantial syntax differences across Qiskit versions, we manually extract only the circuit part from the translated programs. To ensure clarity and consistency, simulation and hybrid optimization logic are implemented within the test code, isolating them from the main quantum circuit. The docstring specifies that the output should be a QuantumCircuit.

• **Canonical Solutions** The extracted circuits serve as our canonical solutions. Each is validated using test cases to ensure functional equivalence with the original program.

C. Docstring Construction

We wrote docstrings based on the extracted core circuit logic, covering the problem description, qubit allocation, algorithm used, and measurement strategy. All docstrings were cross-validated by three experts in quantum programming. Subsequently, we used ChatGPT to generate a program from the docstring to confirm clarity and reduce the chance of misinterpretation.

D. Implementing Multi-Criteria Evaluations

To evaluate the correctness of quantum programs generated by LLMs, we adopt a multi-criteria evaluation framework that combines probabilistic output comparison and static constraint validation. This framework captures both the functional and structural correctness of the generated code.

• **KL Divergence.** For tasks involving measurement outcomes (e.g., quantum teleportation [36] and Bell state preparation), we compare the output probability distributions of the generated code and the canonical solution using the Kullback-Leibler (KL) divergence. This metric quantifies the divergence between the model's output distribution P and the reference distribution Q:

$$D_{\mathrm{KL}}(P \parallel Q) = \sum_x p(x) \log \frac{p(x)}{q(x)}. \tag{2}$$

A lower KL divergence indicates that the model-generated program produces output distributions statistically closer to the correct quantum implementation.

As shown in Figure 2, we use Python's `unittest` framework to verify functional correctness. Specifically, we apply the `assertTrue` statement with a threshold of 0.05: the assertion passes only if the KL divergence between the generated and reference distributions is below 0.05.

• **Static Constraint Validation.** In addition to output-based evaluation, quantum programs must satisfy certain structural and semantic constraints to be considered valid. In QuanBench, we define multiple static constraints, including:

- **Gate set constraints**: The set of gates used in the generated circuit must be a subset of those used in the reference implementation.
- **Measurement result constraints**: The output distribution must include specific target states and reflect expected probability relationships.
- **Phase constraints**: The generated circuit must preserve relative phases of quantum states where applicable.

Figure 2 illustrates an example of a measurement result constraint. In this case, for a Grover algorithm marking the state $|00\rangle$, the correct behavior is reflected in the output distribution where state 00 should have the highest probability. The generated circuit is accepted only if it satisfies this expected pattern.

E. Dataset Statistics

QuanBench was developed to assess LLM's ability to handle often occurring quantum programming tasks based on natural language task descriptions. Table I summarizes the task distribution in four categories, defined according to their underlying quantum computational objectives. In constructing QuanBench, we aimed to include as many commonly used and practically relevant quantum algorithms as possible (e.g., HHL, QFT, Grover's, and Shor's algorithms) and balance both high-level algorithmic reasoning and low-level circuit construction skills required in quantum programming.

Scalability was another key consideration: tasks were selected for their potential to be extended to more qubits or restructured to accommodate alternative problem formulations. Following these design criteria, we assembled 44 quantum programming tasks for the QuanBench benchmark.

• **Quantum Algorithm.** This category encompasses full implementations of established quantum algorithms that demonstrate quantum computational advantages, including Grover's search [34], Shor's factoring [37], Deutsch-Jozsa [35], Bernstein-Vazirani [38], and Simon's algorithms [39]. Tasks in this category involve multiple components, including state initialization, oracle construction, and algorithm-specific logic. These problems evaluate whether LLMs can understand the overall structure and procedural flow of complete quantum algorithms.

• **Quantum State Preparation.** This category focuses on constructing specific quantum states and analyzing their structural or entanglement properties. Tasks include preparing states such as Bell, GHZ, and W states, which require accurate encoding of entanglement and superposition. These problems assess whether LLMs can translate target states into correct gate-level implementations.

• **Gate Decomposition.** These tasks require decomposing high-level quantum operations into sequences of elementary gates [40], which are supported by the hardware. Gate decomposition is crucial for the practical deployment of current quantum devices. The tasks assess the understanding of the quantum gate algebra, the decomposition rules (for example, the Toffoli gate decomposition), and the ability to maintain the correctness of the circuit during transformation.

• **Quantum Machine Learning.** This emerging category includes tasks involving parameterized quantum circuits (PQCs) [41], which are widely used in hybrid quantum-classical learning algorithms. PQCs are applied in tasks such as classification, regression, and quantum state discrimination. These problems test whether LLMs can synthesize circuit templates that correctly implement variational models, even when the problem specifications are incomplete or abstract.

This category design enables QuanBench to address a broad spectrum of quantum programming challenges, ranging from classical algorithmic reasoning to low-level circuit manipulation and the synthesis of quantum components on hybrid quantum-classical models. The combination provides a comprehensive evaluation setting for analyzing the quantum programming capabilities of LLMs.

F. Dataset Expansion

QuanBench currently includes the majority of common quantum algorithms and supports scalable task design, but full balance across task types has not yet been achieved. The benchmark is actively expanding to include variations of existing algorithms across different qubit sizes, adaptations to new problem types, and tasks related to Quantum Machine Learning. Table I shows that the current stage of QuanBench comprises 117 tasks, with improved task diversity and relative category balance.

The current version of QuanBench is scheduled for open source release in the near future, and the benchmark remains under active development and expansion

TABLE I: Task Categorization in QuanBench

Category	**Number of Tasks**	
	QuanBench	Current Stage
Quantum Algorithm	32	64
Quantum State Preparation	6	24
Gate Decomposition	5	14
Quantum Machine Learning	1	15
Total	44	117

IV. Evaluation of LLM Models

A. Research Questions (RQs)

We evaluate the capabilities of LLMs on quantum algorithm generation tasks through the following research questions:

- **RQ1:** What is the current capability of LLMs in generating quantum algorithm programs? This question focuses on the overall performance of LLMs in producing correct, functional, and executable quantum code from natural language descriptions.
- **RQ2:** What is the gap between LLM-generated quantum algorithms and canonical solutions? This question examines the semantic and structural differences between model-generated code and reference implementations.

- **RQ3:** What are the common causes of errors in LLM-generated quantum programs? This question analyzes both qualitative and quantitative error patterns, identifying failure modes such as deprecated APIs, incorrect qubit indexing, and semantic deviations.
- **RQ4:** What is the effect of temperature and batch size on the performance of LLMs in QuanBench? The question aims to identify suitable parameter settings for fair and efficient benchmarking and to provide insights that can guide both future research and the practical optimization of LLMs for quantum code generation.

B. Model Selection and Settings

• **Model Selection.** To evaluate the quantum code generation capabilities of LLMs, we selected a diverse set of models that have demonstrated strong performance in code generation tasks. The selected models are listed in Table II.

TABLE II: List of Evaluated LLMs in QuanBench

Category	**LLM**	**Organization**	**Year**	**Open**
Gen. LLM	GPT-4.1 [7]	OpenAI	2025	no
	DeepSeek V3 [8]	DeepSeek-AI	2024	yes
	DeepSeek R1 [42]	DeepSeek-AI	2025	yes
	Claude 3.7 Sonnet [20]	Anthropic	2025	no
	Gemini 2.5 [9]	Google	2025	no
	Llama-4-Maverick-17B [10]	Meta	2025	yes
	Qwen2.5-7B-Instruct [43]	Qwen	2025	yes
Code LLM	CodeLlama-7B [27]	Meta	2023	yes
	CodeLlama-34B [27]	Meta	2023	yes

The model selection followed the criteria below:

- We prioritized models with demonstrated strong performance in code generation, particularly those fine-tuned on programming or instruction datasets.
- We excluded open-source models that lack version identifiers or checkpoint information to ensure full reproducibility.
- We include models with more than 7B parameters only if their inference APIs are available, due to local hardware inference limitations.
- Following prior studies [16], which showed that small-scale LLMs perform poorly in quantum programming tasks, we focused on medium-to-large models, including 30B+ instruction-tuned variants.

This selection strategy ensures a well-balanced model set that allows meaningful comparison across model size, architecture, and deployment modes.

• **Model Settings.** To ensure fair and reproducible evaluation, we applied unified inference configurations across all models and maintained controlled execution environments. Proprietary models (e.g., GPT-4.1, Claude 3.7 Sonnet) were accessed via official APIs, while open-source models were executed locally or via Hugging Face Inference Endpoints.

For open-source models such as CodeLLaMA-7B and Qwen2.5-7B-Instruct, we downloaded official model weights from Hugging Face repositories and ran inference locally. The proprietary models were accessed through their official providers or authorized third-party APIs, including OpenAI, DeepSeek, Anthropic, Google, Novita, and Replicate.

All models were evaluated using consistent inference parameters, with the temperature set to 0.8. We used a HumanEval-style prompt format to standardize task presentation across models.

C. Evaluation Metric

To comprehensively assess both the correctness and the quantum functionality of the code generated by LLMs, we adopt two complementary evaluation metrics. Pass@K and Process Fidelity. These metrics jointly capture syntactic/functional correctness as well as quantum semantic equivalence.

Similar to HumanEval [11], we use Pass@K to measure the probability that an LLM generates at least one correct solution among its top-K outputs for a given problem. Specifically, for each of the 44 benchmark problems in *QuanBench*, Pass@K reports the proportion of problems where at least one of the K generated samples pass all correctness tests, including static verification and simulation-based evaluation.

In our experiments, each model generates $n = 5$ samples per problem. We report both Pass@1 and Pass@5 to assess model performance under single-sample and multi-sample settings. To reduce sampling variance, we apply the unbiased estimator of Pass@K used in HumanEval [11], defined as:

$$\text{Pass}@K = 1 - \frac{\binom{n-C}{K}}{\binom{n}{K}}, \tag{3}$$

where C is the number of correct generations. This formulation enables statistically robust comparisons across models with varying generation quality.

In addition to Pass@K, we introduce Process Fidelity to evaluate the quantum semantic correctness of generated circuits. Unlike Pass@K, which is based on test outcomes, Process Fidelity directly measures the similarity between the quantum operations implemented by the generated circuit and the canonical solution.

Formally, for each benchmark problem, let U_{ref} be the unitary matrix of the canonical solution, and U_{gen} be the unitary matrix of the generated circuit. The process fidelity $\mathcal{F}$ is calculated as:

$$\mathcal{F}(U_{\text{ref}}, U_{\text{gen}}) = \left| \frac{1}{d} \text{Tr}\left(U_{\text{ref}}^{\dagger} U_{\text{gen}} \right) \right|^2, \tag{4}$$

where $d = 2^n$ is the Hilbert space dimension for a n-qubit system, and $\text{Tr}(\cdot)$ denotes the matrix trace. A fidelity score of 1.0 indicates exact functional equivalence up to a global phase.

In our experiments, for each of the 44 benchmark problems, fidelity is computed between the canonical circuit and each of the 5 generated circuits, yielding 220 fidelity comparisons per model. This metric enables fine-grained evaluation of whether the generated circuit preserves the intended quantum

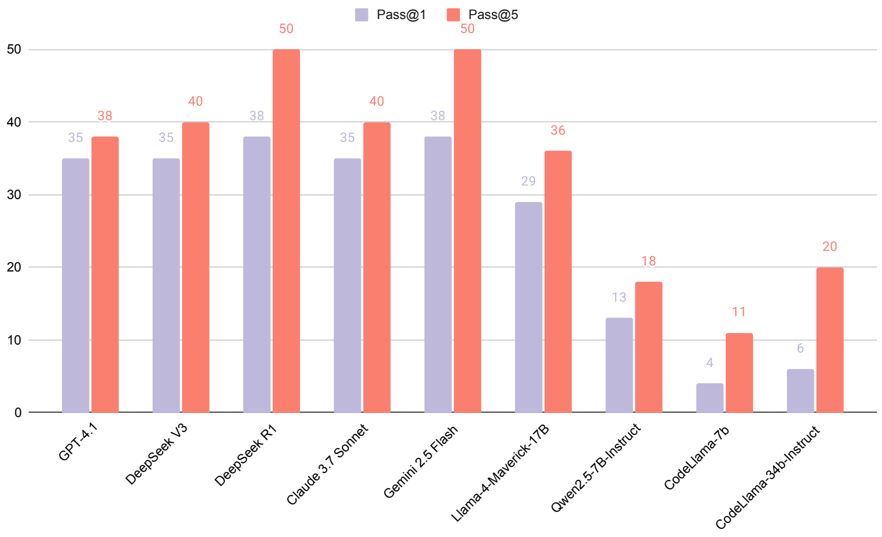


Fig. 3: Effectiveness comparison between LLMs on QuanBench

transformation, regardless of low-level variations such as gate decomposition or gate ordering.

Together, Pass@K and Process Fidelity provide a robust framework to assess both the functional correctness and the quantum mechanical soundness of LLM-generated quantum programs.

V. Experimental Results

In this section, we present the experimental results of the evaluation of selected LLMs on *QuanBench*. The analysis addresses each research question (RQ1–RQ4) and provides quantitative and qualitative insights into the quantum code generation capabilities of the models.

A. RQ1: What is the current capability of LLMs in generating quantum algorithm programs?

1) Overall Performance: Figure 3 presents the performance of the LLMs evaluated in QuanBench. The x-axis lists the models, and the y-axis shows their Pass@K scores. Both Pass@1 and Pass@5 results are reported for each model.

Across all models, the highest Pass@1 accuracy does not exceed 0.4, and the best Pass@5 accuracy reaches only 0.5, indicating that current LLMs have limited capability to generate correct quantum algorithm implementations. Among the models, DeepSeek R1 and Gemini 2.5 achieve the highest performance, with Pass@1 = 0.38 and Pass@5 = 0.5, respectively.

In contrast, CodeLlama-7B shows the weakest performance, with a Pass@5 of only 0.11. Models such as GPT-4.1, DeepSeek V3, and Claude 3.7 produce comparable results. In particular, CodeLlama-34B performs between Qwen2.5-7B and Llama-4-Maverick-17B, suggesting that the performance of quantum code generation does not strictly correlate with the size of the model, and CodeLlama models generally underperform on these tasks.

2) Task Coverage: We further analyzed task coverage by counting the number of problems for which each model generated at least one correct solution within its five generated samples. This provides a complementary perspective to Pass@K by focusing on success at the task level rather than per-sample accuracy.

As shown in Figure 4, the scope of tasks varies significantly between models. DeepSeek R1 and Gemini 2.5 achieve the highest coverage, solving 22 out of 44 problems each. Notably, DeepSeek R1 uniquely solves three tasks that no other model completed, and it is the only model that successfully solves all Quantum State Preparation tasks, indicating particular strength in that category.

In contrast, other models perform well only on narrower subsets of tasks, with generally lower success rates for state preparation and gate decomposition problems. These results indicate that performance differences among LLMs extend beyond aggregate Pass@K scores and reflect variations in their ability to handle diverse quantum programming tasks.

Answer to RQ1: Overall, current LLMs exhibit limited capability to generate complete and correct implementations of quantum algorithms. Although some models show strength in specific categories, their performance does not consistently generalize across the diverse set of quantum programming tasks in QuanBench. Different models display different strength patterns, suggesting varying internal representations and reasoning strategies for quantum circuits.

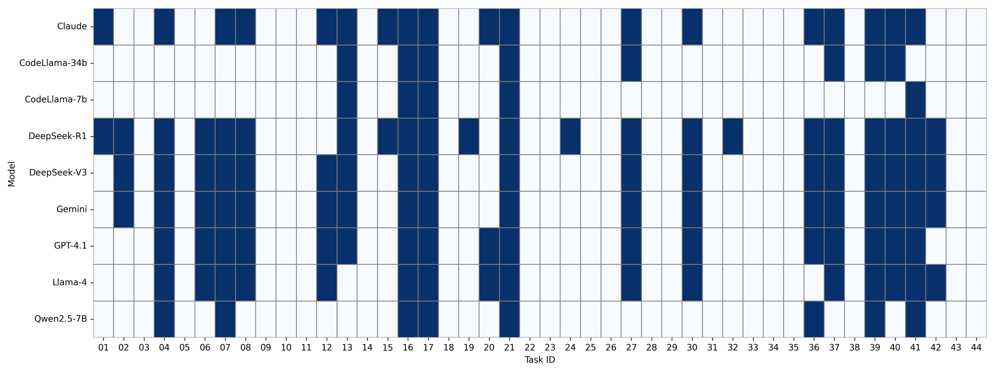


Fig. 4: Number of problems solved on QuanBench

B. RQ2: What is the gap between LLM-generated quantum algorithms and canonical solutions?

To evaluate the fidelity and structural correctness of quantum circuits generated by LLMs, we introduced the Process Fidelity metric, which quantifies the similarity between a generated circuit and its canonical counterpart in terms of their underlying quantum transformations. During evaluation, generated circuits that failed to compile were excluded.

As shown in Table III, the average process fidelity scores across models are significantly lower than 100%, indicating that most generated circuits diverge from the intended quantum operations. While larger models tend to achieve higher fidelity (typically around 50%), GPT-4.1 achieves only 47%, consistent with its relatively low Pass@K performance.

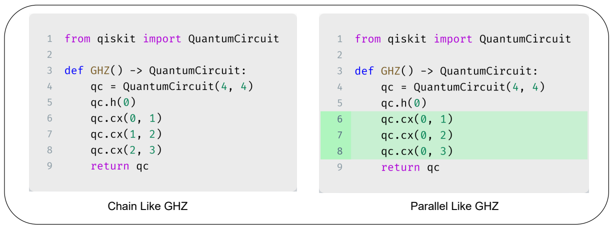
TABLE III: Process Fidelity Scores of Different LLMs

LLM	Passed	Failed	Average
GPT-4.1	70	28	47
DeepSeek V3	75	33	51
DeepSeek R1	49	51	50
Claude 3.7 Sonnet	75	32	52
Gemini 2.5 Flash	57	41	50
Llama-4-Maverick-17B	60	30	41
Qwen2.5-7B-Instruct	64	8	20
CodeLlama-7b	65	5	14
CodeLlama-34b-Instruct	72	10	24

To gain deeper insight into the relationship between functional correctness and quantum-semantic similarity, we separately computed Process Fidelity for generated circuits that pass and fail the test cases.

Our results show that for models such as GPT-4.1, DeepSeek V3, and Claude 3.7, the passed process fidelity reaches approximately 70%, indicating that the generated circuits are structurally and semantically close to the canonical solutions. In contrast, their failing samples exhibit much lower fidelity (around 30%), suggesting a significant divergence from the intended quantum behavior.

Interestingly, for larger models, such as DeepSeek R1 and Gemini 2.5, the process fidelity remains consistently around 50% for both successful and failed samples. Further inspection



```python
from qiskit import QuantumCircuit

def GHZ() -> QuantumCircuit:
    qc = QuantumCircuit(4, 4)
    qc.h(0)
    qc.cx(0, 1)
    qc.cx(1, 2)
    qc.cx(2, 3)
    return qc
```

```python
from qiskit import QuantumCircuit

def GHZ() -> QuantumCircuit:
    qc = QuantumCircuit(4, 4)
    qc.h(0)
    qc.cx(0, 1)
    qc.cx(0, 2)
    qc.cx(0, 3)
    return qc
```

Fig. 5: Different Constructions of GHZ State

of some low-fidelity but functionally correct samples reveals that discrepancies often arise from structural variations that do not affect the final measurement results. These include differences in qubit ordering and global or relative phase shifts that lead to identical observable behavior.

For example, as shown in Figure 5, the construction of a GHZ state can be implemented using different but functionally equivalent circuit structures. One approach applies CNOT gates from the initial qubit in superposition to all other qubits in parallel, while another applies CNOT gates sequentially from one qubit to the next in a chain-like manner. Although both implementations generate the same entangled state, their circuit representations differ, which may result in lower process fidelity despite being functionally correct.

We hypothesize that this behavior reflects the partial ability of these models to capture deeper quantum algorithmic structures. As a result, even some incorrect samples may exhibit unitary behavior closer to the reference circuit, while successful samples may achieve correctness through alternative circuit constructions, resulting in lower process fidelity despite functional equivalence. This suggests that stronger models may generalize better by producing semantically diverse but functionally valid circuits, an important trait for practical quantum code generation.

Answer to RQ2: This highlights a critical limitation: current LLMs often succeed in capturing surface-level functional objectives, but struggle to preserve deeper quantum semantics, such as correct operator composition, entanglement structures, or unitary decompositions. Bridging this semantic gap is essential to achieve reliable and generalizable quantum code generation.

C. RQ3: What are the common causes of errors in LLM-generated quantum programs?

To gain insight into the failure modes of LLM-generated quantum circuits, we conducted a detailed error analysis of the failed samples. Several recurring patterns were identified that significantly contribute to the observed failure rates.

- **Use of Deprecated Gate APIs.** A notable portion of compilation failures stemmed from the use of outdated gate APIs. For example, the `cu1` gate has been deprecated in Qiskit versions after 0.45.0, replaced by `cp`, or alternatively accessible via `qiskit.circuit.library.CU1Gate`. All evaluated models, except Gemini 2.5, DeepSeek V3, and DeepSeek R1, incorrectly invoked the deprecated `cu1` gate, resulting in compilation errors. This indicates that several LLMs rely on outdated training data and do not know about version-specific API updates.
- **Structural Errors in Circuit Construction.** Many errors were caused by incorrect qubit indexing, such as assigning the same qubit as both control and target in a controlled operation (e.g., a `cx` gate with identical indices). This pattern was observed across all models, indicating a limited understanding of quantum gate constraints and circuit construction rules.
- **Overuse or Misuse of Gates.** In some cases, circuits exhibited high process fidelity, but failed test cases due to extraneous gate operations inserted at the beginning or end of the circuit. Although the core algorithmic structure was correct, redundant gates altered the expected measurement outcomes. This behavior reflects an incomplete grasp of quantum circuit minimality and end-to-end semantics.
- **Semantic Deviations in Logic.** Even syntactically valid circuits frequently implement incorrect algorithmic logic. For example, a common failure involved improperly constructed oracles in Grover's algorithm, leading to significant drops in process fidelity despite syntactic correctness. These errors suggest that LLMs frequently struggle to comprehend the algorithmic intent and structural requirements of quantum programs.

We summarize the distribution of the four identified error types in Figure 6. Among them, **Semantic Deviations in Logic** account for the largest proportion of errors, particularly in cases where models fail to generate any correct solution. These errors often involve incorrect oracle construction or missing gate operations, especially in complex tasks such as state preparation and gate decomposition.

The second most frequent error is **Structural Errors in Circuit Construction**, which often leads to compilation failures due to issues such as invalid qubit indexing or incorrect gate syntax.

Overuse or Misuse of Gates ranks third. In these cases, generated circuits may pass partial tests, but fail under stricter

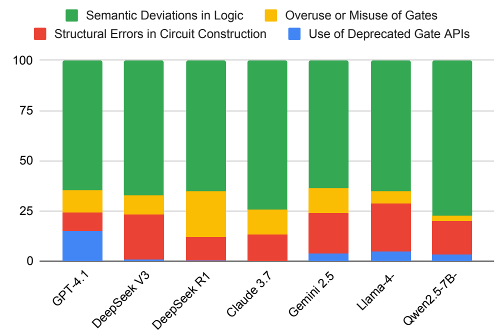


Fig. 6: Distribution of identified error types across all models

correctness checks. For example, in quantum-state preparation, the absence of necessary phase gates may lead to failures when relative phase relationships are tested. Similarly, in Grover's algorithm, incorrectly inserted Hadamard gates may destroy the amplitude amplification effect, causing functional errors even when the overall structure appears plausible.

The least frequent error is **Use of Deprecated Gate APIs**, which is mitigated mainly by standardizing the Qiskit version across evaluations. However, we observe a notable exception with GPT-4.1, which frequently invokes outdated gates such as `cu1`. We hypothesize that this is due to stale training data or limited exposure to updated quantum programming resources.

Answer to RQ3: Overall, these patterns highlight that while LLMs can generate plausible quantum code snippets, they often lack robust semantic grounding and version-aware reasoning, both of which are critical for producing correct and executable quantum programs. Semantic deviations remain the most critical and challenging issue for LLMs in generating quantum programs.

D. RQ4: What is the effect of temperature and batch size on the performance of LLMs in QuanBench

Given that LLMs demonstrate different levels of output diversity depending on the temperature setting [44], and that batch size can also affect diversity, we performed experiments to assess their performance on quantum programming tasks under multiple temperature settings and to investigate the influence of batch size when temperature is held constant.

- **Temperature Settings.** We first examined the performance of LLMs under different temperature settings, specifically 0.1, 0.5, 0.8, and 1.0. It should be noted that for certain models, the maximum temperature setting is restricted to 1, which restricts the range of diversity experiments that can be conducted. Tables IV and V report the results for Pass@1 and Pass@5, respectively, with the best results highlighted in green.

As shown in Table IV, the Pass@1 results indicate that a temperature of 0.8 yields the highest accuracy for most models. For example, DeepSeek V3 achieves its best performance at a temperature of 0.1 with an accuracy of 36%, which is only marginally higher than its score of 35% at 0.8. In contrast, CodeLlama models achieve their best results at a temperature of 0.5, which we hypothesize may be related to the model configuration. Overall, for Pass@1, most LLMs perform best when the temperature is set to 0.8.

In Table V, we again observe that most LLMs achieve their best performance at a temperature of 0.8. At the same time, some models exhibit identical performance across different temperatures (e.g., DeepSeek V3, Gemini, and CodeLlama), while others perform better at a temperature of 1.0. These results suggest that with the current batch size of 5, a temperature of 0.8 generally leads to superior performance for most models.

TABLE IV: LLMs Pass@1 performance across different temperatures

Model	**Temperature**			
	0.1	**0.5**	**0.8**	**1.0**
GPT-4.1	34	33	**35**	34
DeepSeek V3	**36**	35	35	33
DeepSeek R1	35	37	**38**	35
Claude 3.7 Sonnet	33	33	**35**	32
Gemini 2.5 Flash	35	36	**38**	36
Qwen2.5-7B-Instruct	10	10	**13**	8
CodeLlama-7B	4	**7**	4	3
CodeLlama-34B-Instruct	10	**11**	6	6

TABLE V: LLMs Pass@5 performance across different temperatures

Model	Temperature			
	0.1	**0.5**	**0.8**	**1.0**
GPT-4.1	**40**	36	38	38
DeepSeek V3	**40**	**40**	**40**	38
DeepSeek R1	35	47	**50**	45
Claude 3.7 Sonnet	36	40	40	**45**
Gemini 2.5 Flash	**50**	47	**50**	**50**
Qwen2.5-7B-Instruct	13	15	**18**	13
CodeLlama-7B	4	**11**	**11**	9
CodeLlama-34B-Instruct	15	**20**	**20**	13

• **Batch Size Settings.** On the other hand, a batch size of 5 may constrain the diversity of outputs for certain LLMs with higher temperature. To further examine the impact of batch size, we selected two representative models with different optimal temperature settings: GPT-4.1 (which achieves its best Pass@5 at temperature 0.1) and Claude 3.7 (which achieves its best Pass@5 at temperature 1.0). We then increased the batch size from 5 to 100. Table VI presents the comparison. (Due to computational constraints, the 100 batch size experiments were limited to GPT-4.1 and Claude 3.7, and not conducted for the other models.) For GPT-4.1, expanding the batch size had virtually no effect on Pass@1, while Pass@5 improved by 1%. For Claude 3.7, batch size expansion led to a 1% increase in Pass@1, but a 2% decrease in Pass@5. These findings indicate that even under high temperature and large batch size, the overall performance remains virtually the same. Considering computational resources, a batch size of 5 is sufficient to evaluate the performance of LLMs. Therefore, it is recommended that users adopt a batch size of 5 for experimentation.

TABLE VI: GPT and Claude Pass@K in batch size 5 and 100

Model	**Pass@K**	
	1	5
GPT-4.1 (5)	34	38
GPT-4.1 (100)	34	39
Claude 3.7 Sonnet (5)	32	45
Claude 3.7 Sonnet (100)	33	43

Answer to RQ4: Overall, these results suggest that a batch size of 5 combined with a temperature around 0.8 is sufficient for a reliable evaluation in QuanBench. Larger batch sizes offer negligible benefits while incurring significantly higher computational costs.

VI. Threats to Validity

While *QuanBench* includes a diverse quantum programming tasks, several factors may affect the generality of our findings. First, the benchmark is currently limited to the Qiskit framework. Although Qiskit is widely used, models may exhibit different performance when generating code for alternative frameworks such as Cirq or PennyLane. Second, the benchmark focuses primarily on algorithm-level tasks; low-level hardware calibration, pulse-level control, or error mitigation techniques are not included. Third, while we standardize the prompt format and inference parameters, model performance may still vary under alternative prompting strategies, sampling temperatures, or decoding methods. Fourth, although QuanBench includes multiple task categories, the number of quantum machine learning tasks remains limited due to the scarcity of publicly available implementations. These factors may affect the absolute performance reported, although the relative performance trends observed across the models are likely to remain stable. Future work will address these limitations by expanding the task set and supporting additional quantum programming paradigms.

VII. Related Work

A. *Benchmarks for Classical Code Generation*

LLMs have demonstrated strong capabilities in code generation in various programming domains [45]. One of the most influential benchmarks in this area is HumanEval [11], which contains 164 handwritten Python programming problems described via natural language docstrings. It is designed to evaluate the ability of LLMs to synthesize function-level code from natural language specifications.

Following HumanEval, MBPP [3] expanded the scope with approximately 1,000 Python tasks created through crowdsourcing. These problems further assess model performance in synthesizing short programs from descriptive natural language, offering a broader evaluation of generalization and task diversity.

Several domain-specific benchmarks have also been introduced to examine LLM performance in specialized contexts.

DS-1000 [12] focuses on data science code generation, collecting real-world tasks from StackOverflow in seven popular Python libraries. RMCBench [30] evaluates LLM robustness against adversarial and malicious prompts, assessing model behavior in security-critical scenarios and prompt-level perturbations.

B. Benchmarks for Quantum Code Generation

Quantum programming is an emerging field, but developing quantum algorithms remains a significant challenge due to the abstract nature of quantum logic and the constraints imposed by circuit design. To evaluate LLMs in this domain, Qiskit HumanEval [16] was proposed as a curated dataset of Qiskit-based tasks designed to test models' familiarity with quantum programming syntax and APIs. However, this benchmark primarily measures adaptation to the Qiskit syntax, rather than deeper algorithmic reasoning or quantum semantic correctness.

To address this gap, we propose QuanBench, a benchmark specifically designed to evaluate LLMs' ability to generate quantum algorithm implementations that are both syntactically correct and semantically faithful. To better illustrate the distinction between Qiskit HumanEval and QuanBench, we give two explicit examples. Example 1 shows a Qiskit HumanEval docstring, which focuses on API usage: the task specifies adding Hadamard gates, inserting a delay, and returning the circuit, emphasizing step-by-step API compliance. Example 2 shows a QuanBench docstring that defines a task in terms of a quantum algorithm (e.g., Bernstein-Vazirani) with additional hints such as qubit ordering. This highlights QuanBench's focus on algorithmic reasoning and semantic correctness, rather than low-level API calls.

C. Refined LLM-Generated Quantum Programs

Current benchmarks have highlighted the limitations of LLMs in generating quantum code. To mitigate this gap, Qiskit HumanEval [16] introduced a training dataset and fine-tuned LLMs. In parallel, QSpark [46] fine-tuned the Qwen2.5-Coder-32B model using two reinforcement learning methods and evaluated it on Qiskit HumanEval. However, the performance of these fine-tuned models remains limited, which underscores the need for more effective instruction tuning strategies.

```python
from qiskit import QuantumCircuit

def quantum_circuit_with_delay():
    """
    Create a one-qubit quantum circuit,
    apply hadamard gate, then add a delay
    of 100 and then again apply hadamard
    gate and return the circuit.
    """
    qc = QuantumCircuit(1)
    qc.h(0)
    delay_duration = 100
    qc.delay(delay_duration, 0, unit="dt")
    qc.h(0)
    return qc
```

Example 1: An example of Qiskit HumanEval benchmark

```python
from qiskit import QuantumCircuit

def Bernstein_Vazirani_011()->QuantumCircuit:
    """
    Implement the Bernstein Vazirani
    algorithm for a 3-bit hidden string
    a = '011', qubits are ordered from
    right to left (little-endian).
    Return the QuantumCircuit after measure.
    """
    qc = QuantumCircuit(4,3)
    qc.x(3)
    qc.h([0,1,2,3])
    qc.cx(1,3)
    qc.cx(2,3)
    qc.h([0,1,2])
    qc.measure([0,1,2],[0,1,2])
    return qc
```

Example 2: An example of QuanBench benchmark

VIII. Conclusion and Future Work

In this work, we presented QuanBench, a comprehensive benchmark to evaluate the ability of LLMs to generate code for quantum algorithms. Based on the widely adopted Qiskit framework, QuanBench comprises 44 curated programming tasks spanning diverse algorithmic categories, including Grover's search, the quantum Fourier transform, state preparation, and variational circuits. Each task is paired with an executable canonical solution and evaluated using both functional correctness (via Pass@K) and semantic equivalence (via Process Fidelity).

Our empirical evaluation across multiple state-of-the-art LLMs shows that while models occasionally generate correct quantum programs, their overall performance remains limited. Pass@1 accuracy does not exceed 40%, and the Process Fidelity scores reveal substantial semantic deviations from canonical solutions. Further error analysis identifies common failure modes, including outdated API usage, structural inconsistencies, and semantic misinterpretation of algorithmic logic.

In particular, models such as DeepSeek R1 and Gemini 2.5 demonstrate relatively higher task coverage and fidelity, suggesting that certain LLMs exhibit emerging strengths in quantum domains. However, a significant performance gap remains, particularly for tasks requiring deeper quantum reasoning and algorithmic understanding.

QuanBench provides a foundation for a systematic and fine-grained evaluation of quantum code generation. Future work includes extending the benchmark to additional frameworks (e.g., Cirq, PennyLane) and investigating techniques such as prompt engineering, fine-tuning, and retrieval-augmented generation to improve performance on quantum tasks.

Data Availability

QuanBench used in this paper is publicly available in https://github.com/GuoXiaoYu1125/Quanbench.

Acknowledgments

This work was supported by JSPS KAKENHI Grant No. JP24K02920, JP23K28062, JP24K14908, and JST BOOST Grant No. JPMJBS2406.

References

[1] N. Huynh and B. Lin, "Large language models for code generation: A comprehensive survey of challenges, techniques, evaluation, and applications," *arXiv preprint arXiv:2503.01245*, 2025.

[2] B. Athiwaratkun, S. K. Gouda, Z. Wang, X. Li, Y. Tian, M. Tan, W. U. Ahmad, S. Wang, Q. Sun, M. Shang *et al.*, "Multi-lingual evaluation of code generation models," *arXiv preprint arXiv:2210.14868*, 2022.

[3] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le *et al.*, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.

[4] D. Guo, C. Xu, N. Duan, J. Yin, and J. McAuley, "Longcoder: A long-range pre-trained language model for code completion," in *International Conference on Machine Learning*. PMLR, 2023, pp. 12 098–12 107.

[5] S. Lu, N. Duan, H. Han, D. Guo, S.-w. Hwang, and A. Svyatkovskiy, "Reacc: A retrieval-augmented code completion framework," *arXiv preprint arXiv:2203.07722*, 2022.

[6] Y. Wang and H. Li, "Code completion by modeling flattened abstract syntax trees as graphs," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 35, no. 16, 2021, pp. 14 015–14 023.

[7] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[8] DeepSeek-AI, "Deepseek-v3 technical report," 2024. [Online]. Available: https://arxiv.org/abs/2412.19437

[9] G. DeepMind, "Gemini," https://deepmind.google/models/gemini/, 2025, accessed: 2025-05-26.

[10] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[11] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[12] Y. Lai, C. Li, Y. Wang, T. Zhang, R. Zhong, L. Zettlemoyer, W.-t. Yih, D. Fried, S. Wang, and T. Yu, "Ds-1000: A natural and reliable benchmark for data science code generation," in *International Conference on Machine Learning*. PMLR, 2023, pp. 18 319–18 345.

[13] M. Ying, *Foundations of quantum programming*. Morgan Kaufmann, 2016.

[14] S. J. Gay, "Quantum programming languages: survey and bibliography," *Mathematical Structures in Computer Science*, vol. 16, no. 4, pp. 581–600, 2006.

[15] M. A. Nielsen and I. L. Chuang, *Quantum computation and quantum information*. Cambridge university press, 2010.

[16] S. Vishwakarma, F. Harkins, S. Golecha, V. S. Bajpe, N. Dupuis, L. Buratti, D. Kremer, I. Faro, R. Puri, and J. Cruz-Benito, "Qiskit humaneval: An evaluation benchmark for quantum code generative models," in *2024 IEEE International Conference on Quantum Computing and Engineering (QCE)*, vol. 1. IEEE, 2024, pp. 1169–1176.

[17] G.-L. Long, "Grover algorithm with zero theoretical failure rate," *Physical Review A*, vol. 64, no. 2, p. 022307, 2001.

[18] Y. S. Weinstein, M. Pravia, E. Fortunato, S. Lloyd, and D. G. Cory, "Implementation of the quantum fourier transform," *Physical review letters*, vol. 86, no. 9, p. 1889, 2001.

[19] M. Cerezo, A. Arrasmith, R. Babbush, S. C. Benjamin, S. Endo, K. Fujii, J. R. McClean, K. Mitarai, X. Yuan, L. Cincio *et al.*, "Variational quantum algorithms," *Nature Reviews Physics*, vol. 3, no. 9, pp. 625–644, 2021.

[20] Anthropic, "Claude," https://www.anthropic.com/claude, 2025, accessed: 2025-05-26.

[21] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[22] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong *et al.*, "A survey of large language models," *arXiv preprint arXiv:2303.18223*, vol. 1, no. 2, 2023.

[23] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray *et al.*, "Training language models to follow instructions with human feedback," *Advances in neural information processing systems*, vol. 35, pp. 27 730–27 744, 2022.

[24] G. Team, R. Anil, S. Borgeaud, J.-B. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, A. M. Dai, A. Hauth, K. Millican *et al.*, "Gemini: a family of highly capable multimodal models," *arXiv preprint arXiv:2312.11805*, 2023.

[25] C. Team, H. Zhao, J. Hui, J. Howland, N. Nguyen, S. Zuo, A. Hu, C. A. Choquette-Choo, J. Shen, J. Kelley *et al.*, "Codegemma: Open code models based on gemma," *arXiv preprint arXiv:2406.11409*, 2024.

[26] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: may the source be with you!" *arXiv preprint arXiv:2305.06161*, 2023.

[27] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[28] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[29] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," *Advances in Neural Information Processing Systems*, vol. 36, pp. 21 558–21 572, 2023.

[30] J. Chen, Q. Zhong, Y. Wang, K. Ning, Y. Liu, Z. Xu, Z. Zhao, T. Chen, and Z. Zheng, "Rmcbench: Benchmarking large language models' resistance to malicious code," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 995–1006.

[31] A. Javadi-Abhari, M. Treinish, K. Krsulich, C. J. Wood, J. Lishman, J. Gacon, S. Martiel, P. D. Nation, L. S. Bishop, A. W. Cross, B. R. Johnson, and J. M. Gambetta, "Quantum computing with Qiskit," 2024.

[32] Google, "Cirq," https://quantumai.google/cirq, 2022.

[33] K. Svore, A. Geller, M. Troyer, J. Azariah, C. Granade, B. Heim, V. Kliuchnikov, M. Mykhailova, A. Paz, and M. Roetteler, "Q#: enabling scalable quantum computing and development with a high-level DSL," in *Proceedings of the Real World Domain Specific Languages Workshop 2018*, 2018, pp. 1–10.

[34] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, 1996, pp. 212–219.

[35] S. Gulde, M. Riebe, G. P. Lancaster, C. Becher, J. Eschner, H. Häffner, F. Schmidt-Kaler, I. L. Chuang, and R. Blatt, "Implementation of the deutsch–jozsa algorithm on an ion-trap quantum computer," *Nature*, vol. 421, no. 6918, pp. 48–50, 2003.

[36] D. Bouwmeester, J.-W. Pan, K. Mattle, M. Eibl, H. Weinfurter, and A. Zeilinger, "Experimental quantum teleportation," *Nature*, vol. 390, no. 6660, pp. 575–579, 1997.

[37] T. Monz, D. Nigg, E. A. Martinez, M. F. Brandl, P. Schindler, R. Rines, S. X. Wang, I. L. Chuang, and R. Blatt, "Realization of a scalable shor algorithm," *Science*, vol. 351, no. 6277, pp. 1068–1070, 2016.

[38] Arvind, G. Kaur, and G. Narang, "Optical implementations, oracle equivalence, and the bernstein–vazirani algorithm," *Journal of the Optical Society of America B*, vol. 24, no. 2, pp. 221–225, 2007.

[39] G. Brassard and P. Hoyer, "An exact quantum polynomial-time algorithm for simon's problem," in *Proceedings of the Fifth Israeli Symposium on Theory of Computing and Systems*. IEEE, 1997, pp. 12–23.

[40] J. J. Vartiainen, M. Möttönen, and M. M. Salomaa, "Efficient decomposition of quantum gates," *Physical review letters*, vol. 92, no. 17, p. 177902, 2004.

[41] M. Benedetti, E. Lloyd, S. Sack, and M. Fiorentini, "Parameterized quantum circuits as machine learning models," *Quantum science and technology*, vol. 4, no. 4, p. 043001, 2019.

[42] D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi *et al.*, "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," *arXiv preprint arXiv:2501.12948*, 2025.

[43] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Lu *et al.*, "Qwen2. 5-coder technical report," *arXiv preprint arXiv:2409.12186*, 2024.

[44] M. Peeperkorn, T. Kouwenhoven, D. Brown, and A. Jordanous, "Is temperature the creativity parameter of large language models?" *arXiv preprint arXiv:2405.00492*, 2024.

[45] Z. Zheng, K. Ning, Y. Wang, J. Zhang, D. Zheng, M. Ye, and J. Chen, "A survey of large language models for code: Evolution, benchmarking, and future trends," *arXiv preprint arXiv:2311.10372*, 2023.

[46] K. Kheiri, A. Aamir, A. Miranskyy, and C. Ding, "Qspark: Towards reliable qiskit code generation," *arXiv preprint arXiv:2507.12642*, 2025.