

QCoder Benchmark: Bridging Language Generation and Quantum Hardware through Simulator-Based Feedback

Taku Mikuriya^{1,2}, Tatsuya Ishigaki¹, Masayuki Kawarada^{1,3}, Shunya Minami¹,
Tadashi Kadowaki¹, Yohichi Suzuki¹, Soshun Naito⁴, Shunya Takata⁵,
Takumi Kato⁶, Tamotsu Basseda⁷, Reo Yamada⁸, Hiroya Takamura¹

¹National Institute of Advanced Industrial Science and Technology (AIST)

²Yokohama National University ³CyberAgent, Inc. ⁴The University of Tokyo

⁵Keio University ⁶NTT DATA GROUP Corporation ⁷Miletos inc. ⁸University of Tsukuba

Abstract

Large language models (LLMs) have increasingly been applied to automatic programming code generation. This task can be viewed as a language generation task that bridges natural language, human knowledge, and programming logic. However, it remains underexplored in domains that require interaction with hardware devices, such as quantum programming, where human coders write Python code that is executed on a quantum computer. To address this gap, we introduce QCoder Benchmark, an evaluation framework that assesses LLMs on quantum programming with feedback from simulated hardware devices. Our benchmark offers two key features. First, it supports evaluation using a quantum simulator environment beyond conventional Python execution, allowing feedback of domain-specific metrics such as circuit depth, execution time, and error classification, which can be used to guide better generation. Second, it incorporates human-written code submissions collected from real programming contests, enabling both quantitative comparisons and qualitative analyses of LLM outputs against human-written codes. Our experiments reveal that even advanced models like GPT-4o achieve only around 18.97% accuracy, highlighting the difficulty of the benchmark. In contrast, reasoning-based models such as o3 reach up to 78% accuracy, outperforming averaged success rates of human-written codes (39.98%). We release the QCoder Benchmark dataset along with a public evaluation API to support further research.¹

1 Introduction

Programming code generation has emerged as an important and practical problem in language generation studies (Chen et al., 2021). This task requires models to generate correct and executable code by bridging natural language, human exper-

① LLM-based Coding

Input (a natural language instruction:)
Design a quantum circuit on one qubit that prepares the quantum state $\psi = i|1\rangle$ starting from the $|0\rangle$ state. Output (a function code generation for quantum programming)
Output (domain-specific python code)
`qc = QuantumCircuit(1)`
`qc.x(0) # $|0\rangle \rightarrow |1\rangle$`
`qc.s(0) # apply phase of $i: |1\rangle \rightarrow i|1\rangle$`

② Evaluation using simulated hardware

This circuit cannot be run on a quantum computer
Reason:
- circuit depth exceed
- unsupported gates are used in your circuit

③ Refine the code

Figure 1: Quantum code generation involves generating a python code that constructs a quantum circuit executable on a quantum computer. Due to strict constraints of actual hardware, feedback from the hardware is necessary to generate executable codes on a quantum computer.

tise, and formal programming logic. Recent advances in large language models (LLMs) have led to impressive performance on classical programming benchmarks (Wang et al., 2025; OpenAI et al., 2024). However, these benchmarks are primarily evaluated in software-only environments, where failures are typically limited to runtime errors or syntax violations detected by a software development environment such as Python interpreters. In contrast, little is known about how LLMs perform in domains such as quantum programming (Vishwakarma et al., 2024), where generated code must not only be syntactically correct, but also conform to strict, domain-specific constraints imposed by real or simulated quantum hardware.

Quantum programming serves as a representative example of hardware constraint-driven code generation tasks. As shown in Figure 1, given the instruction as a natural language, this code generation task involves generating a python code to produce a quantum circuit that can be run on a separate hardware, i.e., either a real quantum computer or

¹<https://qcoder-bench.github.io/>

simulator. Unlike classical programs, correctness and executability depend not only on the absence of runtime errors in Python, but also on whether the resulting circuits comply with constraints imposed by quantum hardware. These hardware-level constraints include, for example, limitations on circuit depth (i.e., the number of sequential gate operations) and the availability of only certain types of quantum gates in a quantum computer. As a result, quantum code must be syntactically valid in Python and must generate quantum circuits that conform to the logical requirements of quantum computation.

To enhance studies on this constrained generation task, we introduce QCoder Benchmark, a dataset and evaluation framework specifically designed for quantum code generation. Our benchmark contains 1) pairs of a programming contest problem and human-written solutions and 2) an evaluation tool to provide hardware-specific feedback. Unlike prior benchmarks that rely on generic Python execution, our evaluation tool uses a quantum simulator that returns quantum-specific feedback about domain-specific constraints, e.g., circuit depth and inappropriate uses of unsupported quantum gates. This evaluation tool allows feedback-driven iterative language generation: a generation paradigm in which models incorporate feedback from a hardware to refine their generated codes (Madaan et al., 2023).

This paper uses our benchmark to investigate whether LLMs can improve their quantum code generation performance by incorporating domain-specific feedback. Our experiments show that even advanced LLMs like GPT-4o achieve only around 18.97% accuracy, while the best performing LLM o3 reaches 65.52% and it outperforms averaged success rate of human-written codes submitted to programming contests (39.98%). We also find that incorporating feedback into prompt to refine codes can significantly improve generation performance, emphasizing the importance of feedback from a simulated hardware.

We release QCoder Benchmark as public resources to support further research on code generation under complex constraints. This paper makes the following contributions: 1) we implement iterative code generators that use feedback from a simulated hardware-based evaluation tool, 2) we empirically demonstrate that such feedback effectively enhances LLMs’ performances, and 3) our benchmark data and evaluation API will be made public.

2 Related Work

Various coding benchmarks have been proposed for general-purpose code generation tasks, such as HumanEval (Chen et al., 2021), Mostly Basic Python Problems (MBPP) (Austin et al., 2021), and the APPS dataset (Hendrycks et al., 2021). These benchmarks primarily focus on solving basic algorithmic problems written in Python and are evaluated using predefined input-output test cases.

Our benchmark differs from these benchmarks in two key aspects. First, it targets a domain-specific coding task—quantum programming—which involves generating circuits that must conform to real-world hardware constraints. Second, rather than relying solely on static test cases, our benchmark evaluates generated code using a simulated quantum computer, offering a new paradigm for evaluating executable and hardware-aware code.

Domain-specific coding benchmarks have been introduced for various domains, including data science (DS-1000 (Lai et al., 2022)), secure coding (LLMSEval (Tony et al., 2023)), database query generation (Spider (Yu et al., 2018)), and bioinformatics (BioCoder (Tang et al., 2024)).

In the quantum programming domain, Qiskit HumanEval (Vishwakarma et al., 2024) shares similar motivations with ours. However, our benchmark differs in two important ways: (1) it provides a hardware simulator-based evaluation framework for assessing generated quantum circuits, and (2) each programming task is accompanied by multiple human-written implementations, enabling comparative analysis between human and LLM-generated code.

Quantum programming is a form of code generation aimed at controlling hardware, but only a few attempts exist in this direction e.g., a study that generates code to control robots (Luo et al., 2024). Refinement of generated code has also been shown to be effective in various setups (Ding et al., 2024; Madaan et al., 2023; Bi et al., 2024; Liu et al., 2023). Our study extends this line of work by incorporating hardware-aware feedback.

3 QCoder Benchmark

Our benchmark consists of pairs of programming contest problems and human-written solutions, together with an evaluation tool that provides quantum hardware-aware feedback². Our benchmark

²This evaluation tool will be released as a Web API for easier usage for future researches.

differs from existing general-purpose or quantum benchmarks (Vishwakarma et al., 2024) in two aspects: (1) it enables fine-grained evaluation from domain-specific perspectives (e.g., circuit depth), not just functional correctness, and (2) it includes human-written solutions collected from programming contest submissions. These submissions often contain errors and variations useful for studying the differences between LLM-generated and human-written code.

3.1 Dataset

Formulation of Quantum Code Generation

We define the task of quantum code generation as generating a quantum circuit implementation in response to a natural language prompt. The input prompt describes a quantum programming problem in natural language, including constraints on hardware such as supported quantum gates or maximum circuit depth. The expected output is a quantum program written using the Qiskit library (Javadi-Abhari et al., 2024) (or a compatible framework) that can be transpiled into a valid quantum circuit. An example of input prompt is shown in Figure 2.

Problem

Design a quantum circuit on one qubit that prepares the quantum state $|\psi\rangle = i|1\rangle$ starting from the $|0\rangle$ state.

Constraints

States with different global phases will be considered incorrect.

Use the following code format:

```
from qiskit import QuantumCircuit

def solve() -> QuantumCircuit:
    qc = QuantumCircuit(1)
    # Write your code here:

    return qc
```

The LLM is expected to generate only the body of the solve() function.

Figure 2: Example prompt for quantum code generation.

Data Collection

QCoder Benchmark is constructed by collecting quantum programming problems from QCoder, a publicly available platform for quantum program-

ming education website³. We have obtained permission from the QCoder’s developers to redistribute the dataset.

Each problem includes a reference solution and the corresponding target quantum state vector. Each problem is also paired with human-written solutions submitted by participants of real-world quantum programming contests hosted on QCoder. Unlike many generation tasks where model outputs are compared to references using metrics such as BLEU (Papineni et al., 2002), code generation typically does not use references for evaluation, instead, we execute the generated code and verify functional correctness.

For each of the 58 problems, we collected approximately 30 human-submitted codes on average, resulting in a total of 1,740 problem–solution pairs. All solutions are written using the Qiskit library. Although these 30 codes represent the final submitted versions, each was typically created through multiple rounds of editing and refinement by a human coder. The dataset also includes revision histories for each submission, with an average of 20 intermediate versions per code, capturing the iterative development process of human programmers. This rich set of revisions reflects a diverse range of implementation strategies and coding styles, providing a challenging and realistic benchmark for LLM-based code generation.

3.2 Simulator-based Evaluator

Our benchmark has a simulator-based evaluation tool that assesses quantum codes. This evaluator ensures that the submitted quantum circuits are not only functionally correct but also comply with hardware constraints. Given a quantum program, the evaluator performs three evaluation steps:

1. **Runtime check:** The program is executed using the Python interpreter to detect syntax or runtime errors. If an error occurs, the remaining evaluation steps are skipped.
2. **Unsupported gates check:** If no runtime error is detected, the program is transpiled into a quantum circuit. The evaluator then checks whether any gates not supported by the hardware side are used. Such violations are critical, as they make the circuit non-executable on real quantum hardware⁴.

³<https://www.qcoder.jp/en>

⁴In such cases, the code refinement can make the circuit

3. **Circuit depth check:** The evaluation tool measures the circuit depth and compares it against the problem’s specified threshold, if any.
4. **Output state fidelity check:** The circuit is executed on either a real quantum computer or a simulator, and the resulting quantum state is compared against the reference state to assess correctness. Note that our experiments use⁵ simulator, but it can be also replaced by a real quantum computer for more precise feedback.

This evaluation tool checks for constraint violations in order of severity from top to bottom. From the software development environment, python’s runtime errors are considered critical. From the hardware side, the use of unsupported gates is treated as the most critical violation, as it renders the circuit incompatible with real quantum hardware. If a depth limit is specified in the prompt, depth violations are also flagged, as they may impact execution feasibility on NISQ (Noisy Intermediate-Scale Quantum) devices.

All evaluation steps are performed systematically on our web-based API, which will be made public. The API takes a generated quantum code as input and returns a textual report including runtime success, constraint violations, and correctness against the reference state represented in a predefined format, e.g, if an generated program is correct, the following report is produced: { "runtime_error": false, "gate_violation": false, "depth_violation": false, "state_match": true }. This feedback is converted into a natural language prompt used to refine generated code as explained in Section 4.

3.3 Statistics and Comparisons with Other Datasets

Table 1 compares our QCoder Benchmark with existing code generation benchmarks. While prior datasets such as HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) focus on general programming tasks, QCoder is tailored for quantum programming as with an existing benchmark Qiskit HumanEval (Vishwakarma et al., 2024). In contrast to Qiskit HumanEval (Vishwakarma et al.,

2024), QCoder provides human-written solutions and hardware-aware evaluation tool.

4 Methods

This paper compares prompt-based code generators rather than finetuning-based models because developing prompt-based techniques is particularly important for domains like quantum programming, where users are often not experts in natural language processing. The following subsections describe our prompt and refinement process using hardware-aware feedback expressed in natural language.

4.1 Baseline Prompting Strategy

For all LLMs used in our experiments, we use a consistent prompting strategy to ensure fair comparison. Each prompt includes:

- A natural language problem description
- Explicit constraints (e.g., unsupported gates and depth constraints)
- A Python code template with a placeholder function `solve()` that uses the Qiskit library

Models are explicitly instructed to generate only the function body, with no additional imports or code outside the template. A sample prompt is shown in Figure 3.

Problem: Design a quantum circuit on one qubit that prepares the quantum state $|\psi\rangle = i|1\rangle$ starting from $|0\rangle$.

Constraints: States with different global phases will be considered incorrect.

Code template:

```
from qiskit import QuantumCircuit
def solve() -> QuantumCircuit:
    qc = QuantumCircuit(1)
    # Write your code here
    return qc
```

Figure 3: Example of the baseline prompt.

We use the default tokenizer and decoding strategy of each model. No maximum token length is specified; decoding is stopped upon encountering a stop token or natural termination. The generated function body is inserted into the provided template and directly passed to the evaluation tool for assessment.

be executable by decomposing unsupported gates into the supported gate set.

⁵<https://qiskit.github.io/qiskit-aer/>

Dataset	Domain	Submissions	Test Case	Hardware	Dataset Size
HumanEval	General	N/A	Yes	N/A	3,200 problem-answer pairs
MBPP	General	N/A	Yes	N/A	1,000 problem-answer pairs
Qiskit HumanEval	Quantum Programming	N/A	Yes	Partial	151 problem-answer pairs
QCoder (Ours)	Quantum Programming	Yes	Yes	Yes	58 problems × 30 human coders × 20 submissions (on avg)

Table 1: Comparison of QCoder Benchmark with existing code generation benchmarks. "Submissions" indicates whether the dataset includes multiple human-written solutions. "Test Case" refers to the use of predefined functional tests, and "Hardware" denotes whether the evaluation considers hardware-level constraints.

4.2 Feedback-aware Code Refinement

Your answer was

```
'''python
{LLMs' submitted code}
'''
```

Branching according to labels:

WA: This is wrong. Try again.

DLE: The circuit depth exceeded the given constraint. Please revise your implementation to improve efficiency. Try again.

UME: Unauthorized modules has been used. Try again.

UGE: An unauthorized quantum gate has been used. Try again.

RE: The occurring error is: {error text}. Try again.

Figure 4: The prompt used for iterative refinement.

In addition to the baseline prompting, we also evaluate an iterative refinement approach that utilizes feedback from the simulator-based evaluator described in Section 3. This method aims to improve code correctness by performing multiple rounds of generation and correction.

As shown in the example prompt in Figure 4, at each iteration the model receives the baseline prompt along with structured feedback from the evaluator, such as Python error messages, circuit depth violations, or gate usage issues, in the predefined format explained in Section 3. The model is instructed to revise its previous code while adhering to the original constraints.

This iterative process is repeated up to a fixed number of rounds (e.g., 3), or until the generated program passes all evaluation checks. This approach simulates a human-like refinement loop and allows us to assess whether LLMs can incorporate domain-specific feedback to improve their solutions over iterations. This pipeline relies on the

simulator-based evaluator introduced in Section 3, which serves both as a verifier and as a source of structured feedback during refinement.

5 Experiments

5.1 Compared LLMs

We evaluate three proprietary models: gpt-3.5-turbo, gpt-4o-mini, and o3. We also compare two open-source models: Qwen-1.5-14B-Chat (Bai et al., 2023) and DeepSeek-R1-Distill-Llama-70B (DeepSeek-AI, 2025).

These models were selected to cover both proprietary and open-source systems, as well as a range of model sizes and architectural designs, enabling broad comparisons of capabilities. The proprietary models are accessed via the OpenAI API, while the open-source models are deployed locally with their default tokenizers and decoding configurations. All models are prompted using the same format and are evaluated under identical conditions using our simulator-based evaluator.

5.2 Evaluation Metrics

We use the following evaluation metrics to assess the generated programs:

Success rate. A generation is counted as successful if the produced code:

1. runs without any Python execution errors,
2. passes the simulator-based checks for unsupported gate usage and circuit depth,
3. produces the correct quantum state vector as specified in the task.

Fine-grained Failure Rates. To better understand the reasons behind failures, we compute the proportion of failed generations at each stage of the evaluation process. Specifically, we calculate the following stage-wise failure rates:

Model	Success Rate (%)
GPT-4o-mini	18.97
GPT-3.5-turbo	10.34
o3	65.52
DeepSeek-R1-Distill-Llama-70B	29.31
Qwen-1.5-14B-Chat	10.34
Averaged Human	39.98

Table 2: Success rate (%) of baseline prompting without code refinements.

- **Runtime Error:** The proportion of generated programs that fail due to Python runtime errors.
- **Gate Constraint Violation:** The proportion of programs that use quantum gates unsupported by the specified hardware.
- **Depth Constraint Violation:** The proportion of programs whose circuit depth exceeds the specified limit.
- **Wrong Output:** The proportion of programs that pass all checks but still produce an incorrect final quantum state vector.

These fine-grained metrics help isolate specific failure points and provide a more detailed characterization of model weaknesses.

Success Rates of Human Submitted Codes

For each test problem, approximately 30 human-written code samples are available on average. We compute both the overall success rates and fine-grained failure rates on these human submitted solutions to investigate humans’ coding skills.

Changes of Success Rates over iterations As the iterative feedback method generates new code at each round, we track the changes of success rate over iterations.

6 Main Results

In this section, we present the results of our experiments. We begin by comparing overall success rates, followed by a fine-grained failure analysis to understand the types of errors commonly produced. We then examine the effects of iterative refinement using simulator-based feedback and compare model performance against human-written code.

Which LLMs did work better?

As shown in Table 2, o3, a reasoning-based model, achieved the highest success rate (**65.52%**) among

all compared LLMs. It significantly outperforms other proprietary models: GPT-4o-mini (18.97%), or GPT-3.5-turbo (10.34%). This result suggests the substantial advantage of reasoning-oriented models in quantum code generation. As expected, open-sourced models, i.e., DeepSeek-R1-Distill-Llama-70B and Qwen-1.5-14B, obtain lower success rates than all proprietary models possibly due to smaller parameter sizes.

Fine-grained Failure Rates

Next, we discuss failure rates in each evaluation step.

Comparing Among LLMs: As shown in Table 3, GPT-4o-mini and GPT-3.5-Turbo exhibit higher failure rates of runtime errors (17.24% and 36.21%, respectively), indicating frequent failures during the basic Python code execution stage. Even when programs avoid runtime errors and meet the circuit depth constraints, a significant portion still fail to produce the correct quantum state vector upon simulation—accounting for 53.45% in GPT-4o-mini and 31.03% in GPT-3.5-Turbo.

In contrast, o3 demonstrates a remarkable reduction in runtime errors (0%) and shows improved robustness across all error types. However, it still fails to produce the correct quantum output in 18.97% of the cases, suggesting that despite being a reasoning-based model, there remains room for improvement in fully automating quantum programming via LLMs.

Comparing Human coders and LLMs: Runtime Errors are common for both GPT-3.5-Turbo (36.21%) and human coders (27.40%), suggesting that generating syntactically valid codes remains a challenge for both. For the depth violations, human coders (10.03%) and GPT-3.5-Turbo (12.06%) show similar violation rates, suggesting that understanding and complying with quantum-specific constraints such as circuit depth is equally challenging for both. In contrast, o3 maintains a remarkably low violation rate (1.72%), indicating its superior adaptation to quantum programming constraints.

Finally for the wrong outputs, the most prominent failure mode for GPT-4o-mini is generating incorrect output state vectors despite producing syntactically valid code. While humans also exhibit a noticeable rate of wrong outputs (24.69%), o3 performs better (18.97%), suggesting its stronger task comprehension and ability to generate semantically accurate code.

Model	Success (%)	Runtime Err. (%)	Depth (%)	Wrong Output (%)
GPT-4o-mini	18.97	17.24	5.17	53.45
GPT-3.5-Turbo	10.34	36.21	12.06	31.03
o3	65.52	0.00	1.72	18.97
Averaged Human	39.98	27.40	10.03	24.69

Table 3: Distribution of fine-grained failure rates identified by the hardware-aware evaluation tool when the number of iterations is set to one without refinement.

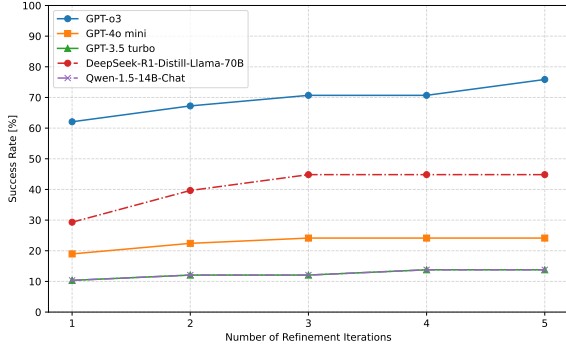


Figure 5: The changes of success rate when we change the number of refinement iterations.

How did iterative refinement improve generation?

As shown in Figure 5, the success rate improves significantly with iterative refinement across all models—GPT-3.5, GPT-4o, and o3. Notably, the most substantial gain is observed after the first refinement, particularly when increasing the iteration count from 1 to 2. Beyond the second iteration, the improvements become more incremental, indicating diminishing returns. These results suggest that leveraging feedback from the hardware-aware evaluation tool to refine the generated code is highly effective, especially in the early iterations.

How LLMs’ performances against Human-written Submissions?

The best-performing model, o3, achieves a success rate of 65.52%, which significantly surpasses the averaged human performance of 39.98%. In contrast, GPT-4o-mini (18.97%) and GPT-3.5-Turbo (10.34%) perform notably worse than human coders, indicating that mid-tier LLMs still fall short in quantum programming tasks.

Figure 6 illustrates the comparison between the success rates of the best-performing LLM (o3) and the averaged human performance, across different numbers of refinement iterations ranging from 1 to 15. Note that both blue lines in Figure 5 and

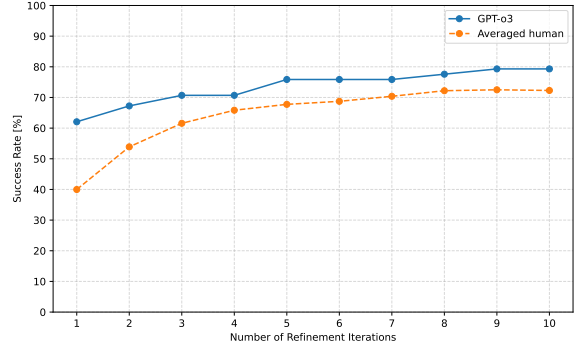


Figure 6: Changes of success rate of the best performing LLM (o3) and human submissions when we increase the number of code refinement. o3 achieves higher success rates than values obtained by averaging human success rates.

Figure 6 represent the performances of the same model. The orange line represents the success rate of averaged human submissions. While the human performance shows a steady upward trend as the number of refinements increases, o3 occasionally exhibits sudden gains in performance—for instance, between iteration 4 and 5. However, it is worth noting that the success rate of o3 does not always improve monotonically; in some cases (e.g., from iteration 3 to 4), performance may stagnate or even slightly drop. This fluctuation highlights the non-deterministic nature of LLM-based generation and suggests that, although o3 generally outperforms human coders in our datasets, its iterative refinement process is not always consistently effective.

7 Case Study: Actual Outputs

This section presents a specific example from the QCoder Benchmark to illustrate how feedback can effectively improve code generation. The example problem is shown in Figure 7.

We show the reference program for this problem in Figure 8. In this problem, it is not necessary to consider the amplitudes of the ground states. Instead, it suffices to view the ground state as a

Problem: Implement the operation of preparing the state $|\psi\rangle$ from the zero state on a quantum circuit qc with 2 qubits. The state $|\psi\rangle$ is defined as

$$|\psi\rangle = a_0 |00\rangle + a_1 |10\rangle + a_2 |01\rangle,$$

where a_0 , a_1 , and a_2 denote arbitrary non-zero probability amplitudes (any values are permitted).

Constraints: Global phase is ignored in judge.

Code template: (omitted)

Figure 7: An example of the problems in QCoder (QPC001-A4)

```
from qiskit import QuantumCircuit

def solve() -> QuantumCircuit:
    qc = QuantumCircuit(2)
    qc.h(0)
    qc.ch(0, 1)
    qc.cx(1, 0)
    return qc
```

Figure 8: An example of the answer for QPC001-A4

superposition formed by dividing the initial state $|00\rangle$ into three components. This transformation can be implemented through the following steps.

First, applying a Hadamard gate to the first qubit yields the transformation

$$|00\rangle \xrightarrow{H(0)} \frac{1}{\sqrt{2}}(|00\rangle + |10\rangle). \quad (1)$$

Next, a controlled-Hadamard gate is applied, targeting the second qubit and controlled by the first. This results in

$$\frac{1}{\sqrt{2}}(|00\rangle + |10\rangle) \xrightarrow{CH(0,1)} \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{2}(|10\rangle + |11\rangle). \quad (2)$$

Finally, a controlled- X gate is applied, targeting the first qubit and controlled by the second, producing

$$\frac{1}{\sqrt{2}}|00\rangle + \frac{1}{2}(|10\rangle + |11\rangle) \xrightarrow{CX(1,0)} \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{2}(|10\rangle + |01\rangle). \quad (3)$$

Through this sequence of operations, a superposition of the three computational basis states $|00\rangle$,

```
def solve() -> QuantumCircuit:
    qc = QuantumCircuit(2)
    a0 = 1
    a1 = 1
    a2 = 1
    norm = np.sqrt(np.abs(a0)**2
        + np.abs(a1)**2 + np.abs(a2)**2)
    state = np.array([a0, a2, a1, 0],
        dtype=complex) / norm
    qc.initialize(state, [0, 1])
    return qc
```

Figure 9: DeepSeek-R1’s answer at the first iteration without refinement.

```
def solve() -> QuantumCircuit:
    qc = QuantumCircuit(2)
    a0 = 1
    a1 = 1
    a2 = 1
    norm = math.sqrt(a0**2+a1**2+a2**2)
    a0_norm = a0 / norm
    a1_norm = a1 / norm
    a2_norm = a2 / norm
    denominator = math.sqrt(a0_norm**2
        + a2_norm**2)
    theta = 2 * math.atan(a1_norm
        / denominator)
    qc.ry(theta, 0)
    phi = 2 * math.atan(a2_norm/a0_norm)
    c_ry = RYGate(phi).control(1)
    qc.append(c_ry, [0, 1])
    return qc
```

Figure 10: DeepSeek-R1’s answer at second iteration.

$|10\rangle$, and $|01\rangle$ is successfully prepared. The available code is shown in Fig. 8.

Fig. 9 shows the code generated at the first iteration. In this code, the `.initialize()` method is used, however, since this method is not included in the allowed set of quantum gates. Therefore, the gate constraint violation error is raised. This violation report is then included in the prompt for refinement as shown in Section 4. The second code generated by the model is shown in Fig. 10. This code passes the runtime error check, however, the evaluator produces the wrong output error although the runtime or other constraints’ checks are passed. The generated code at third iteration shown in Fig. 11 can successfully generate the desired state. Note that the model employs a different approach from

the example solution shown in Fig. 8. Through iterative refinement, the model gradually modifies the code to produce one that satisfies the input constraints.

Finally, we also observed that many failure cases involved Python’s runtime errors that are relatively easy for humans to fix, such as missing import statements (e.g., `import math`). They still struggle with basic coding hygiene that would typically be trivial for humans.

```
import math
from qiskit import QuantumCircuit
from qiskit.circuit.library.
    standard_gates import RYGate

def solve() -> QuantumCircuit:
    qc = QuantumCircuit(2)
    a0 = 1
    a1 = 1
    a2 = 1
    norm = math.sqrt(a0**2+a1**2+a2**2)
    a0_norm = a0 / norm
    a1_norm = a1 / norm
    a2_norm = a2 / norm

    theta = 2 * math.asin(a1_norm)
    phi = 2*math.atan(a0_norm/a2_norm)

    qc.ry(theta, 0)
    qc.x(0)
    qc.cry(phi, 0, 1)
    qc.x(0)
    return qc
```

Figure 11: DeepSeek-R1’s answer at third iteration. (Success)

8 Conclusions

We introduced QCoder Benchmark, which consists of a dataset and evaluation tool for quantum code generation. This benchmark is designed to investigate the capabilities of LLMs under domain-specific hardware-aware constraints. By integrating a quantum simulator that returns hardware-aware feedback, we implemented a feedback-driven iterative code generator. Our experiments revealed that even advanced models like GPT-4o struggle with quantum programming tasks, while reasoning-oriented models like o3 show superior performance and can even outperform human-written code submissions for programming contests. These findings

suggest the importance of refinement of codes by domain-specific feedback. While our benchmark and experiments focus on quantum programming, the proposed feedback-driven generation framework—where domain-specific constraint violations are systematically detected and incorporated into iterative code refinement—could generalize to other domains that impose strict execution constraints. Potential applications include robotics and embedded system programming, where functional correctness alone is insufficient and compliance with real-world constraints (e.g., timing, resource usage, device compatibility) must be enforced. We leave the exploration of such domains to future work.

Ethical Considerations

This work evaluates LLMs on quantum code generation using a custom benchmark dataset and simulator-based feedback. The dataset includes human-written quantum programming codes, which were collected with permission from the QCoder platform. All collected data are free of personally identifiable information and originate from programming contest submissions.

While the dataset is not yet publicly released, we plan to make it available for academic research use under a license that prohibits commercial use. The dataset and evaluation API will be distributed to promote transparency, reproducibility, and responsible research in quantum programming. Final license terms will be announced at the time of release.

Acknowledgements

This work was supported in part by the Council for Science, Technology and Innovation (CSTI) through the Cross-ministerial Strategic Innovation Promotion Program (SIP), “Promoting the application of advanced quantum technology platforms to social issues” (Funding agency: QST), and by the AIST policy-based budget project “R&D on Generative AI Foundation Models for the Physical Domain.”

References

- Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.
- Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei

- Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, and 29 others. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609*.
- Zhangqian Bi, Yao Wan, Zheng Wang, Hongyu Zhang, Batu Guan, Fangxin Lu, Zili Zhang, Yulei Sui, Hai Jin, and Xuanhua Shi. 2024. [Iterative refinement of project-level code context for precise code generation with compiler feedback](#). In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 2336–2353, Bangkok, Thailand. Association for Computational Linguistics.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. [Evaluating large language models trained on code](#).
- DeepSeek-AI. 2025. [Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning](#). *Preprint*, arXiv:2501.12948.
- Yanguibo Ding, Marcus J. Min, Gail Kaiser, and Baishakhi Ray. 2024. [Cycle: Learning to self-refine the code generation](#). *Proc. ACM Program. Lang.*, 8(OOPSLA1).
- Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with apps. *NeurIPS*.
- Ali Javadi-Abhari, Matthew Treinish, Kevin Krsulich, Christopher J. Wood, Jake Lishman, Julien Gacon, Simon Martiel, Paul D. Nation, Lev S. Bishop, Andrew W. Cross, Blake R. Johnson, and Jay M. Gambetta. 2024. [Quantum computing with Qiskit](#). *Preprint*, arXiv:2405.08810.
- Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-Tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2022. Ds-1000: A natural and reliable benchmark for data science code generation. *ArXiv*, abs/2211.11501.
- Jiate Liu, Yiqin Zhu, Kaiwen Xiao, QIANG FU, Xiao Han, Yang Wei, and Deheng Ye. 2023. [RLTF: Reinforcement learning from unit test feedback](#). *Transactions on Machine Learning Research*.
- Hanbin Luo, Jianxin Wu, Jiajing Liu, and Maxwell Fordjour Antwi-Afari. 2024. [Large language model-based code generation for the control of construction assembly robots: A hierarchical generation approach](#). *Developments in the Built Environment*, 19:100488.
- Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegrefe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Sean Welleck, Bodhisattwa Prasad Majumder, Shashank Gupta, Amir Yazdanbakhsh, and Peter Clark. 2023. [Self-refine: Iterative refinement with self-feedback](#). *Preprint*, arXiv:2303.17651.
- OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, and 262 others. 2024. [Gpt-4 technical report](#). *Preprint*, arXiv:2303.08774.
- Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. [Bleu: a method for automatic evaluation of machine translation](#). In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.
- Xiangru Tang, Bill Qian, Rick Gao, Jiakang Chen, Xinyun Chen, and Mark B Gerstein. 2024. [Biocoder: a benchmark for bioinformatics code generation with large language models](#). *Bioinformatics*, 40(Supplement-1):i266–i276.
- Catherine Tony, Markus Mutas, Nicolas Díaz Ferreyra, and Riccardo Scandariato. 2023. [Llmseceval: A dataset of natural language prompts for security evaluations](#). In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*.
- Sanjay Vishwakarma, Francis Harkins, Siddharth Golecha, Vishal Sharathchandra Bajpe, Nicolas Dupuis, Luca Buratti, David Kremer, Ismael Faro, Ruchir Puri, and Juan Cruz-Benito. 2024. [Qiskit humaneval: An evaluation benchmark for quantum code generative models](#). *Preprint*, arXiv:2406.14712.
- Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, and 5 others. 2025. [Openhands: An open platform for AI software developers as generalist agents](#). In *The Thirteenth International Conference on Learning Representations*.
- Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. [Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task](#). In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921, Brussels, Belgium. Association for Computational Linguistics.