QSpark: Towards Reliable Qiskit Code Generation

Kiana Kheiri[1,*], Aamna Aamir[1], Andriy Miranskyy[1] and Chen Ding[1]

[1]Toronto Metropolitan University, Toronto, Canada

Abstract

Quantum circuits must be error-resilient, yet LLMs like Granite-20B-Code and StarCoder often output flawed Qiskit code. We fine-tuned the Qwen2.5-Coder-32B model with two RL methods, Group Relative Policy Optimization (GRPO) and Odds-Ratio Preference Optimization (ORPO), using a richly annotated synthetic dataset. On the Qiskit HumanEval benchmark, ORPO reaches 56.29% Pass@1 (≈ +10 pp over Granite-8B-QK) and GRPO hits 49%, both beating all general-purpose baselines; on the original HumanEval they score 65.90% and 63.00%. GRPO performs well on basic tasks (44/78) and excels on intermediate ones (41/68), but neither GRPO nor ORPO solves any of the five advanced tasks, highlighting clear gains yet room for progress in AI-assisted quantum programming.

Keywords

Quantum computing, Qiskit, Large Language Models, Code generation, Reinforcement learning, Quantum software engineering

1. Introduction

Imagine writing a quantum teleportation protocol with just a natural language prompt. Although this might sound futuristic, recent advancements in AI bring us closer to such a reality. Quantum computing has the potential to solve certain classes of problems faster than classical computing, but programming quantum computers remains a specialized and challenging task [1]. Developing correct and optimized quantum programs requires expertise in quantum mechanics, quantum algorithms, and software engineering, a combination that is not widespread among developers. Even with high-level frameworks like IBM's Qiskit [2], which provide libraries to design and run quantum circuits, writing quantum code is often an error-prone process that requires careful handling of quantum-specific concepts (for example, superposition, entanglement, non-cloning) and resource constraints. As quantum hardware scales up, the complexity of the software needed to harness it also increases, calling for more sophisticated development methodologies [3].

In recent years, researchers have begun to explore how advances in artificial intelligence, particularly large language models (LLMs), can help make quantum programming more accessible and efficient (see [4] and [5, Sec. 4.7] for review). Early work in this area, such as Cruz-Benito et al. [6], demonstrated that deep learning-based approaches could effectively provide customized assistance during the quantum coding process, paving the way for more advanced AI-driven tools.

Large language model (LLM) based coding assistants have already transformed classical software development by providing code autocompletion, generation, and error detection. However, applying these models to quantum programming presents unique challenges. Quantum programming uses a distinct set of languages, libraries, and idioms (such as constructing quantum circuits gate by gate) that differ significantly from classical programming [5].

The code examples available for training are scarce compared to the vast repositories of classical code since quantum computing is an emerging field with relatively few expert developers and open-source projects. Moreover, quantum code must adhere to the rules of quantum mechanics and often requires domain knowledge (for example, understanding the effects of certain gate sequences or the need for qubit reuse and error mitigation). As a result, even advanced coding AIs may generate incorrect or suboptimal quantum code without domain-specific training. In fact, the need for specialized tooling

AIQxQIA 2025: International Workshop on AI for Quantum and Quantum for AI | co-located with ECAI 2025, Bologna, Italy

kiana.Kheiri@torontomu.ca (K. Kheiri); aamna.aamir@torontomu.ca (A. Aamir); avm@torontomu.ca (A. Miranskyy); cding@torontomu.ca (C. Ding)

in quantum software has been highlighted by the Quantum Software Engineering community [5, 7], which argues that simply porting classical development techniques to the quantum realm is insufficient due to fundamental differences in how quantum programs operate.

To bridge this gap, our work proposes a Qiskit-based quantum computing coding assistant, an AI-driven tool designed to help developers write and refine quantum programs using the Qiskit SDK. QSpark[1] focuses specifically on Qiskit, IBM's widely used quantum SDK, and supports tasks such as circuit construction, optimization, and code debugging. We envision a system that can understand high-level intentions (e.g., "prepare a Bell state" or "optimize this circuit section") and provide context-sensitive suggestions or code snippets, much like modern code assistants do for classical languages. Using a large language model trained on quantum programming data, the assistant can generate Qiskit code, recommend quantum algorithmic patterns, and catch common mistakes, all within the developer's workflow. The goal is to lower the barrier to entry for quantum programming and to accelerate the development process for experts and beginners alike.

In this paper, we detail the design of such a Qiskit-based coding assistant, discuss the training and integration of the underlying LLM model, and evaluate its effectiveness in aiding quantum programmers. We begin by reviewing related work in two key areas: quantum programming environments and AI-assisted coding tools for quantum software. This overview will contextualize our contributions and highlight how our approach builds on recent advances. By merging the power of Qiskit and LLMs, we aim to push the boundaries of developer tools in quantum computing, making quantum programming not only more efficient but also more accessible to a broader audience.

2. Literature Review

2.1. Quantum Programming Environments and Tools

Several efforts have been made to create better software environments for developing quantum applications. IBM's Qiskit is one of the leading frameworks, providing an open source SDK with tools for circuit design, simulation, and execution on quantum hardware [4].

In addition to Qiskit, practical resources such as the book by Silva [8] provide essential hands-on strategies for programming quantum rigs using Python, the quantum assembly language, and cloud-based platforms, including IBM QExperience. This resource highlights the inherent challenges in quantum programming and emphasizes the need for developer-friendly methodologies and tools.

Researchers have also acknowledged that improving quantum software development requires higher-level abstractions and more systematic design approaches. For example, Ammermann et al. [9] introduce a view-based development approach that unifies diverse stakeholder perspectives within a quantum IDE. This model suggests that future quantum IDEs may provide synchronized views, such as algorithm logic, circuit layout, and hardware mapping, to better manage the complexity of quantum programs.

Another notable platform is QuantumPath, developed by Hevia et al. [10], which takes an engineering-oriented approach to quantum software creation. QuantumPath provides an application lifecycle management platform for quantum software, supporting developers from algorithm conception through testing, deployment, and maintenance. Providing an ecosystem of modules and enforcing best practices, it simplifies the development of hybrid quantum-classical solutions for real-world use.

These efforts highlight a common theme: quantum software development requires more than just programming libraries. It also needs robust tools and processes, similar to classical software engineering, but tailored for the quantum domain. Our proposed Qiskit-based assistant complements these initiatives by focusing on the coding phase of quantum development. It can be viewed as a plugin designed to enhance quantum programming environments such as Qiskit, providing intelligent support during the development process.

[1]To facilitate reproducibility, we release our full implementation at https://github.com/TMUDeV/QSPARK.

2.2. AI-Assisted Quantum Code Generation

With the growing use of AI in coding-related tasks, it is natural that researchers have started applying LLMs to quantum programming. Dupuis et al. [4] introduced the Qiskit Code Assistant, an AI model specifically trained to generate Qiskit code and help quantum programmers. Their research highlighted the challenges in training a code model for quantum computing, such as understanding quantum gate operations and circuit semantics, as well as the limited availability of training data. Despite these challenges, their specialized model outperformed general-purpose code generation models on Qiskit programming tasks.

Similarly, Vishwakarma et al. [3] developed Qiskit HumanEval, a benchmark suite designed to evaluate how well different LLMs can generate correct quantum code. Their results showed that advanced LLMs, like GPT-style models, can produce executable quantum programs from prompts, successfully passing many tests in the suite. This finding is encouraging, as it confirms the potential of AI-assisted quantum coding and provides a benchmark for future improvements.

Beyond Qiskit-focused tools, researchers have also explored broader applications of AI in quantum algorithm design. For example, Liang et al. [11] examined how LLMs, such as GPT-4, can be used to suggest viable quantum circuit structures. Their work suggests that AI can play a key role in supporting the design of quantum architectures when guided appropriately. Similarly, Aragonés-Soria and Oriol (2024) introduced C4Q, a specialized chatbot that uses pre-trained language models for user request classification and then utilizes its own engine to generate accurate responses [12]. This approach highlights the potential of AI to streamline the development of quantum algorithms. It also demonstrates how such tools can make quantum computing more accessible to beginners by simplifying the learning curve and coding process.

These studies contribute to a growing consensus that generative AI can be a valuable tool in the quantum software development process. Our work builds directly on these prior developments. In particular, we utilize the insights from the Qiskit Code Assistant and the Qiskit HumanEval benchmark to train and evaluate our coding assistant. While prior models have demonstrated the viability of quantum code generation, our contribution lies in tightly integrating this model with the Qiskit user experience by embedding it directly into IDE workflows and extending its capabilities with features tailored to better support quantum developers. These features include recognizing when a qubit needs to be measured or reset, suggesting circuit optimizations, and aligning with Qiskit's latest API.

By positioning QSpark within existing quantum programming environments and AI coding tools, we aim to advance the support available to quantum computing developers. Ultimately, this work contributes to the broader goal of quantum software engineering: to bring the productivity and reliability benefits of modern software development to the field of quantum computing [5], thereby accelerating innovation and adoption.

3. Methodology

Our Qiskit-based code assistant is built upon the Qwen2.5-Coder-32B model, a 32-billion-parameter large language model (LLM) specialized for code generation. It was selected for its strong performance in both general-purpose programming and domain-specific reasoning. We fine-tune the model using a curated dataset of Qiskit programs, detailed in the following subsections.

3.1. Generation of Training Data

To enable robust supervised and reinforcement learning, we construct a high-quality dataset, for training purposes, comprising 522 Qiskit programming tasks. The data set was created through a multistage pipeline that includes code retrieval, function extraction, annotation, validation, deduplication, and formatting. The entire process is automated and designed to ensure consistency, reproducibility, and broad coverage of real-world quantum programming challenges.

We start by collecting approximately 10,819 Qiskit-related source code samples from public repositories. Source files are parsed to extract quantum-relevant functions along with accompanying docstrings and structural metadata. The extracted functions are filtered for completeness and relevance, and each is assigned a unique task identifier.

For each function, a natural language prompt is derived from its docstring or signature. This prompt is paired with the corresponding canonical implementation, a designated function entry point, and a difficulty score. The difficulty rating is calculated using a set of code-level features such as circuit depth, gate complexity, use of measurement or entanglement, and algorithmic structure. This scoring system enables the construction of a curriculum-aligned dataset that spans tasks ranging from basic quantum operations to advanced algorithmic workflows (see Table 2).

Table 1
Difficulty scale with representative Qiskit programming tasks.

Level	Criteria and Example Task
Basic	Simple circuits with few gates and minimal measurement. Example: prepare a single qubit in superposition using an H-gate.
Intermediate	Circuits with measurement, moderate depth, or algorithmic structure. Example: implement a 4-qubit Quantum Fourier Transform (QFT).
Difficult	Complex circuits involving entanglement, variational methods, or hybrid workflows. Example: build a Variational Quantum Eigensolver (VQE) ansatz circuit and connect it to a classical optimizer.

To ensure correctness, each solution is automatically validated through simulation-based unit tests. These tests verify the functional behavior of the quantum circuit, including correct output shape, gate behavior, and fidelity of simulation results. Tasks that fail validation are excluded from the final dataset or flagged for manual inspection.

To improve training diversity and reduce redundancy, we apply structural and semantic deduplication techniques. Near-duplicate solutions or trivial variants are filtered out using syntactic similarity and abstract syntax tree (AST) comparisons, ensuring a more diverse set of training signals. To illustrate the filtering process, approximately 10,819 raw Qiskit-related functions were initially collected, of which fewer than 5% passed all validation steps. The majority of rejections were due to incomplete docstrings, missing test coverage, or trivial circuits (e.g., functions that only returned an empty register). The final curated set of 522 tasks was therefore deliberately biased toward code that was both executable and semantically meaningful. Difficulty scores were computed automatically using heuristics: basic tasks had depth ≤ 3 and no entanglement; intermediate tasks required either measurements or circuit depth > 3; and difficult tasks contained multi-qubit entanglement or hybrid classical-quantum structures. Unlike the QHE benchmark, which evaluates generalization, this dataset was designed for training, and thus prioritizes diversity and coverage across circuit patterns.

Each finalized task in the dataset consists of:

- A unique identifier,
- A natural language task description,
- A validated Qiskit implementation,

Table 2
Difficulty Scale and Distribution of Qiskit Programming Tasks

Level	Criteria	Number of Tasks
Basic	Simple circuits with a few gates, no measurement, no entanglement.	259
Intermediate	Circuits with measurements, moderate depth, or basic algorithmic structure.	223
Advanced	Complex circuits involving entanglement, variational algorithms, or multi-step workflows.	40

- A unit test suite,
- A function entry point, and
- A difficulty level categorized as *basic*, *intermediate*, or *advanced*.

Based on this curated dataset, we derived two specialized training subsets to support preference-based reinforcement learning. For the Odds-Ratio Preference Optimization (ORPO) setting, we constructed a collection of pairwise comparisons consisting of a preferred ("chosen") and a suboptimal ("rejected") output for the same prompt. The chosen samples were selected on the basis of code correctness, readability, and alignment with quantum programming best practices, while the rejected examples were synthetically perturbed or drawn from lower-quality outputs. For the Group Relative Policy Optimization (GRPO) setting, we generated multiple candidate completions per prompt and assigned relative scores based on their simulated execution fidelity and resource efficiency. These two subsets enable distinct learning objectives: ORPO promotes human-aligned code generation through direct preference modeling, while GRPO reinforces code quality by optimizing for group-level performance differentials.

3.2. Reinforcement Learning with Preferences

To further refine the behavior of the model, we employ two independent reinforcement learning strategies: Group Relative Policy Optimization (GRPO) and Odds-Ratio Preference Optimization (ORPO), each targeting a different aspect of quantum code quality.

Odds-Ratio Preference Optimization (ORPO) ORPO aligns the model with human-like coding preferences, focusing on readability and maintainability. It uses pairwise preference data where a "chosen" response is preferred over a "rejected" one, based on manual review and synthetic annotations [13].

The ORPO objective increases the likelihood of preferred output while regularizing the divergence from the original (pre-trained) policy. The loss is defined as

$$\mathscr{L}_{\text{ORPO}} = \text{KL}(\pi_\theta \| \pi_0) - \beta \log_2 \frac{\pi_\theta(\hat{y} \mid x)}{\pi_\theta(y \mid x)}. \tag{1}$$

Here, π_θ is the current policy, π_0 is the pre-trained policy, x is the input prompt, $\hat{y}$ is the chosen output, and y is the rejected one. The hyperparameter β controls the strength of the preference signal relative to the regularization term. The term $\text{KL}(\pi_\theta \| \pi_0)$ represents the Kullback-Leibler (KL) divergence between the current policy π_θ and the pre-trained policy π_0. For two discrete probability distributions P and Q, the KL divergence is generally defined as:

$$\text{KL}(P \| Q) = \sum_i P(i) \log \left(\frac{P(i)}{Q(i)} \right) \tag{2}$$

In this context, it measures how much π_θ deviates from π_0, acting as a regularization to prevent the current policy from straying too far from the original model's capabilities.

ORPO Reward Construction. Odds-Ratio Preference Optimization (ORPO) aligns the model with human-like coding preferences, focusing on readability and maintainability. For each prompt, we construct a pairwise comparison between a *chosen* output $\hat{y}$ and a *rejected* output y. The chosen output is correct, executable, and stylistically aligned with Qiskit best practices. while the rejected output is either synthetically perturbed or sampled from lower-quality generations.

Group Relative Policy Optimization (GRPO) GRPO improves execution fidelity by ranking outputs within a group of candidates generated for each prompt[14]. Each output y_i is evaluated using Qiskit v.2.0.0 and Qiskit Aer simulations v. 0.17.1 and assigned a reward $r(y_i)$. G represents the number of

candidate outputs in a group generated for each prompt. The group mean μ and the standard deviation σ are used to compute the normalized advantage:

$$\mu = \frac{1}{G}\sum_{i=1}^{G} r(y_i), \quad \sigma = \sqrt{\frac{1}{G}\sum_{i=1}^{G}(r(y_i)-\mu)^2}, \quad A(y_i) = \frac{r(y_i)-\mu}{\sigma}. \tag{3}$$

The policy is updated using a clipped objective to ensure training stability:

$$\mathscr{L}_{\text{GRPO}} = \mathbb{E}_{x,y\sim\pi_\theta}\left[\min\left(\frac{\pi_\theta(y\mid x)}{\pi_{\theta_{\text{old}}}(y\mid x)}A(y),\ \text{clip}\left(\frac{\pi_\theta(y\mid x)}{\pi_{\theta_{\text{old}}}(y\mid x)}, 1-\epsilon, 1+\epsilon\right)A(y)\right)\right]. \tag{4}$$

The clip function $\text{clip}(v, L, U) = \max(L, \min(v, U))$ bounds r_θ, and ϵ (e.g., 0.1–0.2) sets the range $[1-\epsilon, 1+\epsilon]$. This process guides the model toward producing more executable and resource-efficient quantum circuits by emphasizing outputs that outperform others in the same generation group.

GRPO Reward Construction. For each prompt, we generate a group of candidate completions. Each candidate is executed with `Qiskit v2.0.0` and `Qiskit Aer v0.17.1` simulators. The execution is scored using three criteria:

1. **Unit test pass rate** (r_1): fraction of test cases passed (primary correctness signal).
2. **Circuit depth penalty** (r_2): normalized inverse of circuit depth to reward more efficient solutions.
3. **Qubit count penalty** (r_3): normalized inverse of the number of qubits used, discouraging wasteful allocations.

These are combined into a single scalar reward:

$$r(y) = \alpha \cdot r_1 + \beta \cdot r_2 + \gamma \cdot r_3, \tag{5}$$

with weights $\alpha = 0.7$, $\beta = 0.2$, and $\gamma = 0.1$ chosen empirically to emphasize correctness while still encouraging efficiency.

Within each group, rewards are normalized using the group mean and variance:

$$A(y_i) = \frac{r(y_i)-\mu}{\sigma}, \quad \mu = \frac{1}{G}\sum_{i=1}^{G} r(y_i), \quad \sigma = \sqrt{\frac{1}{G}\sum_{i=1}^{G}(r(y_i)-\mu)^2}, \tag{6}$$

where G is the number of candidates. This normalization ensures that the rewards are *relative*: a candidate only receives a high advantage if it is better than its peers, even if all solutions are weak. The policy update (Eq. 4) then uses this normalized advantage to push the model towards consistently producing correct and efficient circuits.

In practice, this setup allows GRPO to prefer structurally sound and resource-efficient quantum circuits for simple tasks, while avoiding overfitting to a single absolute scoring heuristic.

4. Results and Discussion

4.1. Evaluation Setup

We evaluated our models using the Qiskit HumanEval (QHE) benchmark introduced by Vishwakarma et al. [3], which extends the original HumanEval benchmark to assess LLMs on quantum programming tasks. Following the evaluation framework used in that work, we compare our GRPO and ORPO fine-tuned models to both general-purpose open-source LLMs and a specialized QHE-tuned baseline, using the following key metrics:

1. **Pass@1 Accuracy:** The percentage of completions that pass a unit test on the first attempt.

Table 3
Hyperparameter Settings for GRPO and ORPO Fine-Tuning

Hyperparameter	GRPO	ORPO
Learning Rate	5×10^{-6}	4×10^{-5}
Weight Decay	0.1	0.1
Warmup Ratio	0.1	0.1
LR Scheduler	Cosine	Linear
Batch Size	64	32
Sequence Length	2048 tokens	2048 tokens
Optimizer	`adamw_8bit`	`adamw_8bit`
Training Epochs	3	3

2. **Performance by Difficulty Level:** Evaluation of the accuracy of the model in the tasks labeled Basic, Intermediate, and Advanced.
3. **General-Purpose vs. Domain-Specific Models:** A comparison to understand the impact of domain adaptation on performance.

Since the original evaluation script was not publicly released and the HumanEval [15] framework was incompatible with QHE tasks, we implemented a custom benchmarking script customized for the QHE setting. Although the QHE paper reports 101 tasks, the publicly released dataset contains 151 entries. This resulted in 78 Basic, 68 Intermediate, and 5 Advanced tasks. This script automatically executes model completions against the associated unit tests and logs pass/fail outcomes, enabling consistent and scalable evaluation across all models. Our results are therefore based on a fully automated, reproducible evaluation pipeline that faithfully adheres to the QHE benchmark structure.

We compared our GRPO and ORPO models with the following strong baseline models.

- General-Purpose Open-Source LLMs: These models, such as CodeLLaMA-34B [16], DeepSeek-33B [17], StarCoder2-15B [18], and CodeGemma-7B [19], are large language models trained on vast datasets of general programming code. They serve as a benchmark for how well unspecialized models perform on quantum programming tasks.
- Granite-8B-Base[20]: This is a general-purpose base model. Its performance helps to understand the impact of any quantum-specific fine-tuning.
- Granite-8B-QK (QHE-tuned baseline)[3]: This model is a specialized version of Granite-8B, fine-tuned specifically for the Qiskit HumanEval benchmark. It represents the state-of-the-art in domain-adapted models for Qiskit code generation and provides a direct comparison to our preference-based fine-tuning approaches.

The results for the baseline models are taken from [3].

Fine-Tuning Hyperparameters: Table 3 summarizes the hyperparameter settings used during fine-tuning for both approaches that we ran on the A100 GPU with 80 GB for VRAM.

4.2. Results

We evaluated the performance of our GRPO and ORPO models on the Qiskit HumanEval (QHE) benchmark, comparing them to several strong general-purpose code LLMs and the QHE-tuned baseline. Table 4 reports the Pass@1 accuracy on both HumanEval (HE) and QHE under greedy decoding.

Our models achieve significant improvements over all baseline models in QHE. ORPO achieves the highest Pass@1 accuracy at 56.29%, outperforming the domain-adapted Granite-8B-QK model by nearly 10 percentage points. GRPO also performs competitively, achieving 49.00%, and surpasses all general-purpose models. These results demonstrate the effectiveness of preference-based fine-tuning in the quantum domain, even compared to models trained specifically for QHE.

Table 4
Pass@1 on HumanEval (HE) and Qiskit HumanEval (QHE) with Greedy Decoding. The results for models other than ours are taken from [3].

Model	HE	QHE
CodeLLaMA-34B	52.43%	26.73%
DeepSeek-33B	49.39%	39.60%
StarCoder2-15B	45.12%	37.62%
CodeGemma-7B	42.68%	24.75%
Granite-8B-Base	39.02%	28.71%
Granite-8B-QK	38.41%	46.53%
GRPO (Ours)	63.00%	49.00%
ORPO (Ours)	**65.90%**	**56.29%**

Table 5
QHE Pass Counts by Difficulty Level (78 Basic, 68 Intermediate, 5 Advanced). The results for models other than ours are taken from [3].

Model	Basic	Intermediate	Advanced
CodeLLaMA-34B-Python	19/54	8/45	0/2
DeepSeek-Coder-33B	30/54	10/45	0/2
StarCoder2-15B	26/54	12/45	0/2
CodeGemma-7B	20/54	5/45	0/2
Granite-8B-Code-Base	21/54	8/45	0/2
Granite-8B-Code-QK	**32/54**	15/45	0/2
GRPO (Ours)	42/78	32/68	0/5
ORPO (Ours)	44/78	**41/68**	0/5

Interestingly, both GRPO and ORPO also show strong generalization on the original HumanEval benchmark, with Pass@1 scores of 63.00% and 65.90%, respectively, outperforming larger models like CodeLLaMA-34B and DeepSeek-33B. This suggests that preference optimization not only improves performance on domain-specific tasks but also may enhance general code generation capabilities.

To better understand the behavior of the model across the complexity of tasks, Table 5 presents pass counts grouped by difficulty level. ORPO ranks third in basic-level tasks with 44/78 (slightly behind Granite-8B-Code-QK at 32/54 and DeepSeek-Coder-33B at 30/54 in terms of completion percentage), achieves the highest pass count on intermediate tasks (41/68), and outperforms others in total completions. GRPO performs worse than ORPO but still surpasses many baseline models. Neither model succeeds on the five advanced tasks, consistent with all other baselines.

These results offer complementary insights: GRPO appears to be more effective for simpler structurally consistent circuits, benefiting from group-level ranking rewards, while ORPO demonstrates stronger reasoning and robustness on moderately complex tasks due to its fine-grained preference alignment objective.

Both models perform on par with or exceed these reference rates, further validating their practical utility. Overall, the results highlight the strength of preference-driven optimization in quantum programming and emphasize the importance of evaluating across difficulty levels to capture nuanced model capabilities.

4.2.1. Training Dynamics

Figure 1 illustrates the training dynamics of our preference-optimized models. The plot on the left shows the reward trajectory for GRPO, while the plot on the right presents the loss curve for ORPO.

In the GRPO setup (left), the model is optimized using group-based reinforcement signals derived from task-specific XML output properties. The observed rewards display high variance throughout

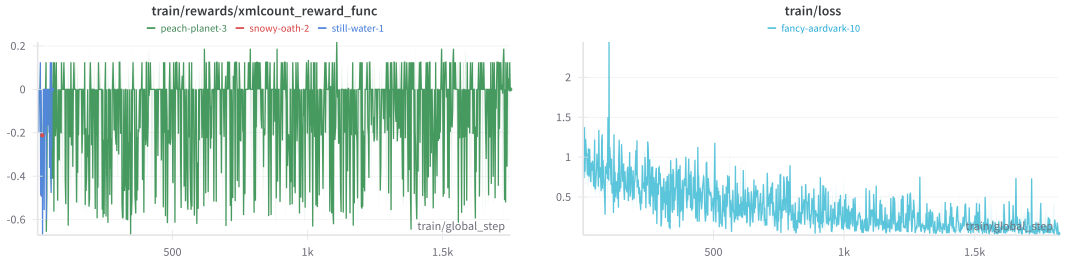

Figure 1: Training dynamics of QSpark fine-tuning. Left: GRPO reward trajectory with normalized advantage values (y-axis) across training steps (x-axis). Right: ORPO pairwise ranking loss (y-axis) across training steps (x-axis). Legends and axis labels are explicitly provided to highlight reward and loss behavior.

training, a result of simulation-based reward assignment and the stochastic nature of quantum program outputs. Despite fluctuations, the trend demonstrates that the model is able to consistently explore and exploit high-reward completions. Importantly, this noisy, yet bounded, reward signal is characteristic of preference-driven reinforcement learning in sparse-reward domains.

The ORPO training curve (right) exhibits more stable and gradual convergence. The pairwise ranking loss steadily decreases as the model learns to align its output with preference-labeled completions. The initial sharp drop is followed by continued fine-tuning and refinement, reflecting effective optimization using contrastive supervision. Huang et al. [21] provide theoretical insights into pairwise learning for ranking, supporting our observation of a smooth convergence process under such loss formulations. This steady progression contrasts with the volatility of GRPO and highlights the complementary strengths of the two methods. GRPO encourages exploration and robustness through output diversity, while ORPO guides the model toward aligning with desired behavior patterns.

Together, these training signals validate the design of our preference learning pipeline. GRPO encourages structural diversity and correctness in simpler tasks, while ORPO promotes nuanced alignment and interpretability in more complex scenarios.

4.3. Discussion

Our updated evaluation highlights the effectiveness of reinforcement learning with preferences, through GRPO and ORPO, for quantum code generation. Both models substantially outperform general-purpose LLMs on the Qiskit HumanEval (QHE) benchmark under greedy decoding. ORPO achieves the highest QHE pass@1 score at 56.29%, surpassing even the domain-specific Granite-8B-Code-QK baseline (46.53%), while GRPO also delivers a strong 49.00%. These improvements were achieved without explicit supervised instruction tuning or access to the original QHE fine-tuning scripts, emphasizing the strength of our preference optimization pipeline. At the task difficulty level, ORPO consistently outperforms GRPO.

Interestingly, even the benchmark reference implementation, executed via the latest version of Qiskit, only achieves 69/78 on Basic, 63/68 on Intermediate, and 2/5 on Advanced tasks. This highlights potential fragility and version sensitivity in quantum execution environments. It also raises a critical point: our models were tested under realistic run-time conditions and still matched or exceeded these reference pass counts, reinforcing their practical applicability. Compared to previous work by Vishwakarma et al. [3], our results are competitive despite several evaluation challenges. Although their paper reports 101 tasks, the public release contained 151 files. Furthermore, their evaluation script was not released. To address this, we wrote our benchmark evaluation script and validated completions using the unit tests provided, ensuring consistent and reproducible results.

Finally, the persistent failure across all models, including ours and the benchmark baseline, on the five advanced tasks (0/5) underscores the difficulty of complex quantum reasoning. These results suggest that success in advanced quantum programming likely requires novel strategies, potentially involving curriculum learning, richer supervision signals, or domain-specific memory mechanisms.

Our findings validate reinforcement learning with preferences as a promising direction for quantum LLMs. They also highlight the urgent need for standardized, version-controlled benchmarks and tooling in quantum code generation research.

4.4. Challenges and Ethical Considerations

Although the integration of AI into quantum computing offers significant promise, it also introduces important challenges, both technical and ethical, that must be carefully considered [22].

Lim et al. [23] and Hernandez and Patel [24] highlight the dual role of generative AI in education, portraying it as both a transformative enabler and a potential threat to traditional learning paradigms. This perspective aligns with our work, where reinforcement learning is used not to replace human quantum programmers but to augment their workflows. Our use of preference optimization explicitly reflects this balance: the goal is to guide AI-generated code toward human-aligned styles, best practices, and interpretable solutions, rather than generate opaque or overly optimized outputs that lack usability.

Ahmadi [25] explores the convergence of quantum computing and artificial intelligence, emphasizing the revolutionary potential of this union in fields such as cryptography and optimization. However, he also underscores several concerns, ranging from algorithmic reliability to ethical deployment, which are directly relevant to our work. In particular, issues like execution fidelity, qubit resource constraints, and reproducibility are amplified in the quantum domain, where small errors in AI-generated code can lead to significant deviations in output. Our manual validation of test cases, necessitated by the lack of standardized evaluation tools, further reflects the importance of transparency and accountability in quantum AI development.

Looking ahead, we advocate for the development of community-driven benchmarks, shared evaluation pipelines, and stronger documentation practices. These are critical steps not only for reproducibility but also for building trustworthy AI systems that can be safely and ethically deployed in quantum research and education.

5. Conclusion and Future Work

In this work, we present a Qiskit-based quantum code assistant built on the Qwen2.5-Coder-32B model, fine-tuned using reinforcement learning with preferences. By introducing Group Relative Policy Optimization (GRPO) and Odds-Ratio Preference Optimization (ORPO), we explore how domain-aligned feedback can improve quantum code generation beyond conventional supervised fine-tuning. Our models demonstrate competitive performance on the Qiskit HumanEval benchmark, particularly excelling at Basic and Intermediate tasks, where they outperform several general-purpose LLMs. These results underscore the promise of preference-based optimization for aligning large language models with quantum programming best practices.

However, this work also presents important challenges. We encountered inconsistencies in benchmark releases, missing evaluation scripts, and had to manually run and validate test cases, which affected reproducibility. Furthermore, none of the evaluated models, including ours, succeeded in the Advanced-level tasks, pointing to the need for better instruction tuning, longer-horizon reasoning, and deeper integration with quantum hardware constraints. Although our results demonstrate the potential for preference-based optimization for quantum code generation, there are several limitations to be acknowledged. First, the training data remains relatively small compared to classical code datasets, which may restrict generalization to novel quantum tasks. Second, due to the absence of an official evaluation script and inconsistencies in the published benchmark, we relied on manual validation for scoring, introducing potential subjectivity, and making direct comparisons to other models less precise.

In future work, our aim is to:

- Integrate GRPO and ORPO into a unified reward framework [26].
- Develop sampling-based decoding strategies that align with human-in-the-loop workflows.

- Broaden the data set to encompass a wider range of quantum use cases, including error correction, hybrid quantum-classical algorithms, and hardware-specific optimizations.

In addition, we intend to work with a more comprehensive and clearly defined benchmark and develop a robust, automated evaluation pipeline to support consistent testing and comparison across models. We also advocate for the open release of standard evaluation tools to support fair benchmarking and collaborative development in quantum LLM research.

By addressing these open challenges, we hope to push the boundaries of AI-assisted quantum programming, making it more accessible, reliable, and aligned with human intent.

6. Acknowledgements

This work is partially sponsored by Natural Science and Engineering Research Council of Canada (grants # 2020-04760 and RGPIN-2022-03886).

Declaration on Generative AI

During the preparation of this work, the authors used ChatGPT in order to: Grammar and spelling check, paraphrase, and reword. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the content of the publication.

References

[1] A. Ajagekar, T. Humble, F. You, Quantum computing based hybrid solution strategies for large-scale discrete–continuous optimization problems, Computers & Chemical Engineering 132 (2019) 106630. doi:10.1016/j.compchemeng.2019.106630.

[2] 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, J. M. Gambetta, Quantum computing with qiskit, arXiv preprint (2024). URL: https://arxiv.org/abs/2405.08810. doi:10.48550/arXiv.2405.08810. arXiv:2405.08810.

[3] N. Dupuis, L. Buratti, S. Vishwakarma, A. V. Forrat, D. Kremer, I. Faro, R. Puri, J. Cruz-Benito, Qiskit code assistant: Training llms for generating quantum computing code, in: Proceedings of the IEEE LLM-Aided Design Workshop (LAD), 2024. URL: https://arxiv.org/abs/2405.19495. doi:10.1109/LAD62341.2024.10691762.

[4] S. Vishwakarma, F. Harkins, S. Golecha, V. S. Bajpe, N. Dupuis, L. Buratti, D. Kremer, I. Faro, R. Puri, J. Cruz-Benito, Qiskit humaneval: An evaluation benchmark for quantum code generative models, in: Proceedings of the IEEE International Conference on Quantum Computing and Engineering (QCE), 2024, pp. 1169–1176. doi:10.1109/QCE57912.2024.00001.

[5] J. M. Murillo, J. Garcia-Alonso, E. Moguel, J. Barzen, F. Leymann, S. Ali, T. Yue, P. Arcaini, R. Pérez-Castillo, I. García-Rodríguez de Guzmán, M. Piattini, A. Ruiz-Cortés, A. Brogi, J. Zhao, A. Miranskyy, M. Wimmer, Quantum software engineering: Roadmap and challenges ahead, ACM Transactions on Software Engineering and Methodology 34 (2025). doi:10.1145/3712002.

[6] J. Cruz-Benito, I. Faro, F. Martín-Fernández, R. Therón, F. J. García-Peñalvo, A deep-learning-based proposal to aid users in quantum computing programming, in: International Conference on Learning and Collaboration Technologies, Springer, Springer International Publishing, 2018, pp. 421–430. doi:10.1007/978-3-319-91152-6_32.

[7] J. Bausch, F. Leditzky, Quantum codes from neural networks, New Journal of Physics 22 (2018). doi:10.1088/1367-2630/ab6cdd.

[8] V. Silva, Practical quantum computing for developers: programming quantum rigs in the cloud using Python, quantum assembly language and IBM QExperience, Apress, 2018.

[9] J. Ammermann, W. Mauerer, I. Schaefer, Towards view-based development of quantum software, 2024. URL: https://arxiv.org/abs/2406.18363. arXiv:2406.18363.

[10] J. L. Hevia, G. Peterssen, M. Piattini, Quantumpath: A quantum software development platform, Software: Practice and Experience 52 (2022) 1517–1530.

[11] Z. Liang, J. Cheng, R. Yang, H. Ren, Z. Song, D. Wu, X. Qian, T. Li, Y. Shi, Unleashing the potential of llms for quantum computing: A study in quantum architecture design, arXiv preprint arXiv:2307.08191 (2023).

[12] Y. Aragonés-Soria, M. Oriol, C4q: A chatbot for quantum, in: Proceedings of the 5th ACM/IEEE International Workshop on Quantum Software Engineering, Q-SE 2024, Association for Computing Machinery, New York, NY, USA, 2024, p. 29–36. URL: https://doi.org/10.1145/3643667.3648222. doi:10.1145/3643667.3648222.

[13] J. Hong, N. Lee, J. Thorne, Orpo: Monolithic preference optimization without reference model, 2024. URL: https://arxiv.org/abs/2403.07691. arXiv:2403.07691.

[14] Z. Shao, P. Wang, Q. Zhu, R. Xu, J. Song, X. Bi, H. Zhang, M. Zhang, Y. Li, Y. Wu, D. Guo, Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. URL: https://arxiv.org/abs/2402.03300. arXiv:2402.03300.

[15] M. C. et al., Evaluating large language models trained on code, 2021. URL: https://arxiv.org/abs/2107.03374. arXiv:2107.03374.

[16] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, G. Synnaeve, Code llama: Open foundation models for code, 2024. URL: https://arxiv.org/abs/2308.12950. arXiv:2308.12950.

[17] DeepSeek-AI, Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. URL: https://arxiv.org/abs/2501.12948. arXiv:2501.12948.

[18] R. L. et al, Starcoder: may the source be with you!, 2023. URL: https://arxiv.org/abs/2305.06161. arXiv:2305.06161.

[19] C. Team, Codegemma: Open code models based on gemma, 2024. URL: https://arxiv.org/abs/2406.11409. arXiv:2406.11409.

[20] M. Mishra, M. Stallone, G. Zhang, Y. Shen, A. Prasad, A. M. Soria, M. Merler, P. Selvam, S. Surendran, S. Singh, M. Sethi, X.-H. Dang, P. Li, K.-L. Wu, S. Zawad, A. Coleman, M. White, M. Lewis, R. Pavuluri, Y. Koyfman, B. Lublinsky, M. de Bayser, I. Abdelaziz, K. Basu, M. Agarwal, Y. Zhou, C. Johnson, A. Goyal, H. Patel, Y. Shah, P. Zerfos, H. Ludwig, A. Munawar, M. Crouse, P. Kapanipathi, S. Salaria, B. Calio, S. Wen, S. Seelam, B. Belgodere, C. Fonseca, A. Singhee, N. Desai, D. D. Cox, R. Puri, R. Panda, Granite code models: A family of open foundation models for code intelligence, 2024. URL: https://arxiv.org/abs/2405.04324. arXiv:2405.04324.

[21] S. Huang, J. Zhou, H. Feng, D.-X. Zhou, Generalization analysis of pairwise learning for ranking with deep neural networks, Neural Computation 35 (2023) 1135–1158. doi:10.1162/neco_a_01585.

[22] D. B. Rawat, C. Bajracharya, The intersection of quantum computing, ai, and cybersecurity: Challenges and opportunities, in: 2024 IEEE 6th International Conference on Trust, Privacy and Security in Intelligent Systems, and Applications (TPS-ISA), 2024, pp. 176–181. doi:10.1109/TPS-ISA62245.2024.00029.

[23] W. M. Lim, A. Gunasekara, J. L. Pallant, J. I. Pallant, E. Pechenkina, Generative ai and the future of education, The International Journal of Management Education (2023). URL: https://www.sciencedirect.com/science/article/pii/S1472811723000289. doi:10.1016/j.ijme.2023.100790.

[24] K. Hernandez, T. Patel, Enhancing early quantum computing education with quantumaied: Bridging the educational gap, in: Proceedings of the ACM SIGCSE, ACM, 2025, p. 1755. URL: https://dl.acm.org/doi/10.1145/3641555.3705028.

[25] A. Ahmadi, Quantum computing and artificial intelligence: The synergy of two revolutionary technologies, American Journal of Engineering and Science (AJES) (2023). URL: https://ajesjournal.org/index.php/ajes/article/view/4118. doi:10.51983/ajes-2023.12.2.4118.

[26] J. Dai, M. O. Gluzman, Queueing network controls via deep reinforcement learning, ArXiv abs/2008.01644 (2020). doi:10.1287/stsy.2021.0081.