Qiskit Code Assistant: Training LLMs for generating Quantum Computing Code

Nicolas Dupuis*, Luca Buratti†, Sanjay Vishwakarma‡, Aitana Viudes Forrat‡,
David Kremer‡, Ismael Faro‡, Ruchir Puri* and Juan Cruz-Benito‡
*IBM Research, Yorktown Heights, NY, USA
†IBM Research, Zurich, Rüschlikon, Switzerland
‡IBM Quantum, Yorktown Heights, NY, USA

2024 IEEE LLM Aided Design Workshop (LAD) | 979-8-3503-7608-1/24/$31.00 ©2024 IEEE | DOI: 10.1109/LAD62341.2024.10691762

***Abstract*—Code Large Language Models (Code LLMs) have emerged as powerful tools, revolutionizing the software development landscape by automating the coding process and reducing time and effort required to build applications. This paper focuses on training Code LLMs to specialize in the field of quantum computing. We begin by discussing the unique needs of quantum computing programming, which differ significantly from classical programming approaches or languages. A Code LLM specializing in quantum computing requires a foundational understanding of quantum computing and quantum information theory. However, the scarcity of available quantum code examples and the rapidly evolving field, which necessitates continuous dataset updates, present significant challenges. Moreover, we discuss our work on training Code LLMs to produce high-quality quantum code using the Qiskit library. This work includes an examination of the various aspects of the LLMs used for training and the specific training conditions, as well as the results obtained with our current models. To evaluate our models, we have developed a custom benchmark, similar to HumanEval, which includes a set of tests specifically designed for the field of quantum computing programming using Qiskit. Our findings indicate that our model outperforms existing state-of-the-art models in quantum computing tasks. We also provide examples of code suggestions, comparing our model to other relevant code LLMs. Finally, we introduce a discussion on the potential benefits of Code LLMs for quantum computing computational scientists, researchers, and practitioners. We also explore various features and future work that could be relevant in this context.**

***Index Terms*—Code Large Language Models, code LLMs, Qiskit, Quantum Computing**

I. Introduction

We are digitally surrounded by Large Language Models (LLMs) that guide us while writing to serve as assistants for several problems, reduce user writing effort, suggest different options for words/sentences to enhance our style, or fix our grammatical errors. The same applies in the context of source code, where LLMs are being used in different stages of the software development cycle: from the generation of code to bug fixing, from generating documentation to migration [1], [2]. Generic LLMs such as GPT-4 [3], Claude [4], or Gemini [5] are very good at coding, but smaller code-LLMs can reach almost the same coding skills while being easier and cheaper to train, and to infer. For example, to cite just a few, StarCoder [6], Code Llama [7], and DeepSeek Coder [8] all show impressive performances on various code benchmarks.

979-8-3503-7608-1/24/$31.00 ©2024 IEEE

There is a growing interest in the application of Artificial Intelligence (AI) methods to enhance the quantum computing field [9], [10]. In recent years, it has become apparent that the majority of research and development efforts in this area have been focused on devising novel quantum algorithms for artificial intelligence [11]–[17] or integrating classical AI features with quantum systems to optimize specific domains or processes [14], [15], [18]–[20]. However, there is a noticeable gap in the application of machine learning and classical intelligence systems and algorithms to augment quantum ecosystems and platforms and empowering quantum computing practitioners, a concern that has gained increasing attention within the quantum community [21]–[23].

With the proliferation of Large Language Models (LLMs) for code assistance, and considering the previous comments in the field of quantum computing, an area of interest is to develop specialized LLMs for quantum code generation. Quantum code generation poses unique challenges that make it a more complex task than standard code generation:

- It requires a basic knowledge of quantum computing
- There is limited amount of data and code examples
- The field evolves quickly, and new techniques appear frequently and the relevant libraries are updated accordingly.

The same challenges apply to some extent to human coders: quantum computing has a high barrier of entry for developers that are not familiar with the field but want to explore its capabilities. With specialized code assistants for quantum, we aim to make quantum computing more accessible to new adopters, and to make development workflows more efficient for current users.

In this work, we introduce specialized LLMs that can work as code assistants for Qiskit SDK [24], [25] users. Qiskit is the lead open-source quantum computing framework [26] that provides a comprehensive set of tools, libraries, and documentation for building quantum algorithms, simulating quantum systems, and working with quantum hardware.

The paper is organized as follows: Section 2 presents the methods and materials employed to train the LLMs, including details about the dataset, training procedures and foundation models used. Section 3 presents a summary of the results achieved, including a description of our evaluation benchmark, how our model compare to others and some prompt results. Section 4 presents some conclusions and future directions.

II. Methods and materials

We start training on top of a Granite code model [27], part of a family of decoder-based models for generative AI code tasks. The Granite series of models shows state-of-the-art performance across open Code-LLMs in a variety of coding tasks. Furthermore, Granite models are among the most open models available as of today [28], providing clear details about data, training and architecture. Our base model is granite-20b-code which uses gpt_bigcode [29] architecture, has 20 Billion parameters, learned positional encodings, multi-query attention, and a context length of 8192 tokens. The tokenizer is identical to StarCoder [30] and has a vocabulary size of 49152. The Granite base model was pre-trained on 1.6 T tokens of code data including 116 programming languages.[1]

To improve the performances of the model at generating high-quality Qiskit code, we extend its pretraining with additional Qiskit data containing python scripts, and Jupyter notebooks. We crawled GitHub using its API, searching for all publicly available repositories with a permissive open-source license that contain the keyword "qiskit" in the name or in the description, keeping only the main branch at the latest commit available, and omitting forks. All the data was collected on April 19, 2024. As common with technical SDKs, Qiskit evolves fast and deprecates features often, so training a model with the latest data available ensure compatibility with the latest releases. After collecting data, we filtered out samples with deprecated code, keeping only samples updated after 2022, and applied exact-match deduplication. For the Jupyter notebooks, we followed a similar approach to StarCoder [6], and used sentinel tokens to separate out code and markdown fields. We also filter out cells containing decoded base64 image data. We did not use the output cells from the notebooks.

After filtering, the total number of tokens is 88 M, of which 80 M were never seen by the base model. We set up training data mixing ratios to ensure diversity and quality. Table I shows the data and token distribution used. The data includes python scripts, and Jupyter notebooks that contain mix of Qiskit tutorials and code. In the table, we highlight the difference between Qiskit Official (qko) and other non-official Qiskit data (qk). The qko data comes from one of the following GitHub organizations, Qiskit [31], Qiskit-Community [32], or Qiskit-Extensions [33], and is considered of the highest quality, hence the large oversampling factors (10.3 and 11.2). We tested different weights and mixing ratios and found the values in Table I to work the best on our evaluation benchmark and when sampling from the model.

For extend-pretraining, the data is packed and we use a special token to separate each samples. The total number of tokens after oversampling is 193 M and we train for 1400 steps (≈ 3 epochs). We use a global batch size of 64, and a learning rate warmed-up from 0 to 1×10^{-5} on 140 steps then decayed with a cosine schedule.

TABLE I: Data distribution and token count used for extend pretraining. Each subset has a weight and is oversampled. Total token count in one training epoch is 193 M. "qko" refers to any sample originating from an official Qiskit GitHub organization.

Dataset	Weight	Epochs	Raw tokens (M)	Eff tokens (M)
qko-code	0.35	10.3	6.5	67.7
qk-code	0.3	1	58	58
qko-notebook	0.24	11.2	4.1	46.4
qk-notebook	0.11	1	20	20

In order to improve natural language understanding, we further instruct-tune the model. We use the octopack approach as described in [34], mixing chat data from openassistant (8k samples) and commit data from commitpackft (5k samples). We also add synthetic data specific to Qiskit: 2.7k question/answer pairs synthetically generated from tutorial using mixtral instruct model [35], and 1k synthetic prompt/code pairs, whose execution accuracy were validated using synthetically-generated unittests. We left-pad all the sequences and use a 2048 sequence length. We train the model for 3.2 epochs using a global batch size of 32, a learning rate of 8×10^{-6} decayed using a cosine schedule and a warm-up of 160 steps.

III. Results

A. Qiskit HumanEval

For testing our model, we have created a new execution-based benchmark called Qiskit HumanEval (QHE). The benchmark is similar to HumanEval [36], including multiple challenging code problems to solve, all based on the Qiskit library. Among the different problems evaluated through the benchmark we can find management of basic quantum code instructions, writing of particular quantum algorithms, handling of executions in either simulators or quantum devices, basic understanding of quantum information theory-related problems or terminology. QHE currently comprises 101 examples. Similar to HumanEval, the context provided to the model consists in a function definition, followed by a docstring which details the task the model is required to solve. Each example also includes a reference canonical solution, as well as unittests in order to evaluate the correctness of the generated solutions. This benchmark will be released later in 2024.

B. Evaluation Results

We tested the Qiskit model and other baselines on QHE and HumanEval (HE). To compute the execution accuracy, we used the bigcode harness framework [37] and ran the code generated by the models in a docker environment setup with the latest version of Qiskit SDK (at the time of writing this paper, v1.0.2). Table II presents the results. The pass scores were computed on greedy-decoded model outputs. We compare Granite with 3 base models and one instruct model. Of the three base, DEEPSEEK-CODER-33B-BASE has the highest QHE pass score at 39.6%. The instruct version of DeepSeek Coder is best at HE but the QHE pass rate is slightly lower. STARCODER2-15B is also a strong baseline on QHE with a

[1] We started this work using an early version of granite code which saw less tokens than the recently published moded [27].

```
# Import qiskit ibm runtime service and print which backends are available for
# my account. Get backend ”ibm brisbane” and define a function in which I
# can show the properties thermal relaxation time, dephasing time and gate error
# of sx gate for the qubit 1 in that backend.

from qiskit_ibm_runtime import QiskitRuntimeService
service = QiskitRuntimeService()
print(service.backends())
backend = service.backend(”ibm_brisbane”)

def show_properties(backend):
    properties = backend.properties()
    qubit = 1
    print(f’Qubit {qubit} thermal relaxation time: {properties.qubit_property(qubit, ’
        T1’)}”)
    print(f’Qubit {qubit} dephasing time: {properties.qubit_property(qubit, ’T2’)}”)
    print(f’Qubit {qubit} sx gate error: {properties.gate_error(’sx’, qubit)}”)

show_properties(backend)
```

(a) GRANITE-20B-CODE-QK

```
# Import qiskit ibm runtime service and print which backends are available for
# my account. Get backend ”ibm brisbane” and define a function in which I
# can show the properties thermal relaxation time, dephasing time and gate error
# of sx gate for the qubit 1 in that backend.

from qiskit_ibm_runtime import QiskitRuntimeService
from qiskit.providers.ibmq.runtime import UserMessenger

service = QiskitRuntimeService()
print(service.backends())
backend = service.get_backend(”ibmq_qasm_simulator”)

def show_properties(backend):
    print(backend.properties().to_dict())
    print(backend.properties().qubits[1].to_dict())
    print(backend.properties().gates[0].to_dict())
show_properties(backend)
```

(b) DEEPSEEK-CODER-33B-BASE

```
from qiskit.circuit.library import LinearFunction
from qiskit.synthesis.linear.linear_matrix_utils
import random_invertible_binary_matrix

def get_random_linear_function(n_qubits, seed):
    ”””
    Generate a random linear function circuit using the input parameters
    n_qubits, seed and through using the random_invertible_binary_matrix
    method.
    ”””
    matrix = random_invertible_binary_matrix(n_qubits, seed)
    linear_function = LinearFunction(matrix)

    return linear_function
```

(c) GRANITE-20B-CODE-QK

```
from qiskit.circuit.library import LinearFunction
from qiskit.synthesis.linear.linear_matrix_utils
import random_invertible_binary_matrix

def get_random_linear_function(n_qubits, seed):
    ”””
    Generate a random linear function circuit using the input parameters
    n_qubits, seed and through using the random_invertible_binary_matrix
    method.
    ”””
    random_invertible_binary_matrix.seed(seed)
    matrix = random_invertible_binary_matrix.get_random_invertible_binary_matrix(
        n_qubits)
    return LinearFunction(matrix)
```

(d) DEEPSEEK-CODER-33B-BASE

Fig. 1: Output samples generated with GRANITE-20B-CODE-QK and DEEPSEEK-CODER-33B-BASE. In (a) and (b) the models are prompted with an instruction set as a python comment, while in (c) and (d), the models are prompted with the import statements, a function header and a python docstring.

TABLE II: HumanEval (HE) and Qiskit-HumanEval (QHE) pass@1 computed using greedy decoding.

Model	HE	QHE
CODELLAMA-34B-PYTHON-HF	52.43%	26.73%
DEEPSEEK-CODER-33B-BASE	49.39%	39.6%
DEEPSEEK-CODER-33B-INSTRUCT	**68.9%**	35.64%
STARCODER2-15B	45.12%	37.62%
GRANITE-20B-CODE	38.41%	20.79%
GRANITE-20B-CODE-QK	36.58%	**46.53%**

pass score of 37.62%. Our granite base model (GRANITE-20B-CODE) has a QHE pass score of 20.79%, however, after extend training, the pass score reaches 46.53%, beating all models.

C. Prompt results

In Fig. 1, we present examples of prompt queries sent to granite-qiskit (a), (c) and DeepSeek Coder (b), (d). First comparing (a) and (b), DeepSeek Coder, does not correctly follow the instructions provided. It starts by introducing an unnecessary import with “from qiskit.providers.ibmq.runtime import UserMessenger”, which is irrelevant and outdated. Furthermore, while it does manage to list available backends, it incorrectly chooses “ibmq_qasm_simulator” instead of the specified “ibm_brisbane” backend. Also, displaying the backend properties, though somewhat informative, misses the mark by not focusing on the specified properties of thermal relaxation time, dephasing time, and sx gate error for qubit 1. Whilst the output is technically correct in a broader context, it fails to address the prompt specifics. The Qiskit model correctly responds to the prompt. It accurately selects the “ibm_brisbane” backend and correctly defines the “show_properties” function to focus on the requested qubit and gate properties. When the model proposes code to get qubit properties, it shows a good knowledge about quantum computing terminology as it associates the prompt statement “thermal relaxation time” to the term “T1” or “dephasing time” as “T2”. Now looking at prompts (c) and (d), granite-qiskit correctly addresses the prompt specifications, creating a “get_random_linear_function” method that accurately employs the input parameters of “n_qubits” and seed. This is achieved through the correct application of generating a random invertible binary matrix and leveraging this matrix to construct a LinearFunction, thereby aligning with the prompt. DeepSeek Coder’s response, while attempting to address the same task, falls short on several fronts. It uses an unnecessary method call “random_invertible_binary_matrix.seed(seed)” that disrupts the standard workflow and makes the code unable to run. Furthermore, the misapplication of the LinearFunction object construction, characterized by an erroneous approach to matrix generation, further underscores a fundamental misinterpretation of the task requirements.

IV. Discussion

In terms of the utility of such solutions, we view these tools as a potential catalyst that could accelerate the adoption and utilization of quantum computing, particularly among newcomers and students, much like how LLMs have facilitated the learning of programming in classical languages/-paradigms [38]. For more experienced researchers, computational scientists, and similar user personas, we anticipate that LLMs could improve the coding experience, exploration, and overall happiness [36], [39], [40], although further evaluation is required to confirm this observation [41]. In relation to the public release of these LLMs, the models trained under the auspices of the Qiskit Code Assistant project will be made available to IBM Quantum users in the upcoming months through a suite of services and extensions that can integrate with existing IDEs or various programming environments. Similarly, the evaluation benchmark, the Qiskit HumanEval, will be released publicly to enable other LLMs to compare and enhance their code suggestions in the field of quantum computing. Enabling LLMs in a rapidly evolving context like quantum computing presents unique challenges. We anticipate continuous updates to new approaches, algorithms, and libraries. A crucial aspect of this project is the ability to regularly update the models to reflect the most recent code, trends, and best practices, ensuring that they remain valuable and relevant for all user personas. As part of this evolving landscape, we expect the need for features such as code explanation, translation between different libraries or versions, automatic test generation and code repair to arise soon.

Acknowledgment

The authors thank the data and model factory who collected the code data and trained the base model, in particular Mayank Mishra, Rameswar Panda, Gaoyuan Zhang, Matthew Stallone, and Hima Patel. We also thank Atin Sood for helping setting up the service, and Xuan Liu for support and management.

References

[1] Z. Zheng, *et al.*, "A survey of large language models for code: Evolution, benchmarking, and future trends," *arXiv:2311.10372*, 2023.
[2] GitHub, https://github.com/features/copilot, 2024.
[3] OpenAI, "GPT-4 Technical Report," 2023.
[4] Anthropic, https://claude.ai/, 2024.
[5] M. Reid, *et al.*, "Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context," *arXiv:2403.05530*, 2024.
[6] A. Lozhkov, *et al.*, "StarCoder 2 and The Stack v2: The Next Generation," *arXiv:2402.19173*, 2024.
[7] B. Rozière, *et al.*, "Code Llama: Open Foundation Models for Code," *arXiv:2308.12950*, 2023.
[8] D. Guo, *et al.*, "DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence," *arXiv:2401.14196*, 2024.
[9] J. Preskill, "Quantum computing in the NISQ era and beyond," *Quantum*, vol. 2, p. 79, 2018.
[10] J. Biamonte, *et al.*, "Quantum machine learning," *Nature*, vol. 549, no. 7671, pp. 195–202, 2017.
[11] K. Bharti, *et al.*, "Machine learning meets quantum foundations: A brief survey," *AVS Quantum Science*, vol. 2, no. 3, 2020.
[12] D. Peral-García, *et al.*, "Systematic literature review: Quantum machine learning and its applications," *Computer Science Review*, vol. 51, p. 100619, 2024.
[13] V. Dunjko, *et al.*, "Machine learning & artificial intelligence in the quantum domain: a review of recent progress," *Reports on Progress in Physics*, vol. 81, no. 7, p. 074001, 2018.
[14] P. S. Emani, *et al.*, "Quantum computing at the frontiers of biological sciences," *Nature Methods*, vol. 18, no. 7, pp. 701–709, 2021.
[15] M. Martonosi, *et al.*, "Next steps in quantum computing: Computer science's role," *arXiv:1903.10541*, 2019.
[16] P. Baireuther, *et al.*, "Machine-learning-assisted correction of correlated qubit errors in a topological code," *Quantum*, vol. 2, p. 48, 2018.
[17] J. Bausch, *et al.*, "Learning to decode the surface code with a recurrent, transformer-based neural network," *arXiv:2310.05900*, 2023.
[18] H. P. Nautrup, *et al.*, "Optimizing quantum error correction codes with reinforcement learning," *Quantum*, vol. 3, p. 215, 2019.
[19] V. Nguyen, *et al.*, "Deep reinforcement learning for efficient measurement of quantum devices," *npj Quantum Information*, vol. 7, no. 1, p. 100, 2021.
[20] D. Kremer, *et al.*, "Practical and efficient quantum circuit synthesis and transpiling with reinforcement learning," *arXiv:2405.13196*, 2024.
[21] V. Esbroeck, *et al.*, "Quantum device fine-tuning using unsupervised embedding learning," *New Journal of Physics*, vol. 22, no. 9, p. 095003, 2020.
[22] H. Moon, *et al.*, "Machine learning enables completely automatic tuning of a quantum device faster than human experts," *Nature communications*, vol. 11, no. 1, p. 4161, 2020.
[23] J. Cruz-Benito, *et al.*, "A deep-learning-based proposal to aid users in quantum computing programming," in *Learning and Collaboration Technologies. Learning and Teaching*, P. Zaphiris and A. Ioannou, Eds. Cham: Springer International Publishing, 2018, pp. 421–430.
[24] Qiskit contributors, "Qiskit: An open-source framework for quantum computing," 2023.
[25] A. Javadi-Abhari, *et al.*, "Quantum computing with qiskit," *arXiv:2405.08810*, 2024.
[26] Unitary Fund, "2023 Quantum Open Source Survey," Dec 2023. [Online]. Available: https://unitaryfund.github.io/survey-website/
[27] M. Mishra, *et al.*, "Granite code models: A family of open foundation models for code intelligence," *arXiv:2405.04324*, 2024.
[28] B. Rishi, *et al.*, "The foundation model transparency index v1.1," *arXiv:2310.12941*, 2024.
[29] L. Ben Allal, *et al.*, "SantaCoder: don't reach for the stars!" *arXiv:2301.03988*, 2023.
[30] R. Li, *et al.*, "StarCoder: may the source be with you!" *arXiv:2305.06161*, 2023.
[31] "Qiskit," https://github.com/Qiskit/qiskit, 2024.
[32] "Qiskit Community," https://github.com/qiskit-community, 2024.
[33] "Qiskit Extensions," https://github.com/Qiskit-Extensions, 2024.
[34] N. Muennighoff, *et al.*, "OctoPack: Instruction Tuning Code Large Language Models," *arXiv:2308.07124*, 2023.
[35] A-Q. Jiang, *et al.*, "Mixtral of experts," *arXiv:2401.04088*, 2024.
[36] M. Chen, *et al.*, "Evaluating large language models trained on code," *arXiv:2107.03374*, 2021.
[37] L. Ben Allal, *et al.*, "A framework for the evaluation of code generation models," https://github.com/bigcode-project/bigcode-evaluation-harness, 2022.
[38] J. Finnie-Ansley, *et al.*, "The robots are coming: Exploring the implications of openai codex on introductory programming," in *Proceedings of the 24th Australasian Computing Education Conference*, 2022, pp. 10–19.
[39] A. M. Dakhel, *et al.*, "Github copilot ai pair programmer: Asset or liability?" *Journal of Systems and Software*, vol. 203, p. 111734, 2023.
[40] S. Barke, *et al.*, "Grounded copilot: How programmers interact with code-generating models," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, pp. 85–111, 2023.
[41] P. Vaithilingam, *et al.*, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *Chi conference on human factors in computing systems extended abstracts*, 2022, pp. 1–7.