

# Pushing the Limits of LLMs in Quantum Operations

Dayton C. Closser <sup>1,2</sup> and Zbigniew J. Kabala  <sup>3,4</sup>

<sup>1</sup>dayton.closser@duke.edu

<sup>2</sup>Department of Electrical & Computer Engineering, Pratt School of Engineering, Duke University, Durham, North Carolina, 27708, United States of America

<sup>3</sup>zbigniew.kabala@duke.edu

<sup>4</sup>Department of Civil & Environmental Engineering, Pratt School of Engineering, Duke University, Durham, North Carolina, 27708, United States of America

**What is the fastest Artificial Intelligence Large Language Model (AI LLM) for generating quantum operations? To answer this, we present the first benchmarking study comparing popular and publicly available AI models tasked with creating quantum gate designs. The Wolfram Mathematica framework was used to interface with the 4 AI LLMs, including WolframLLM, OpenAI ChatGPT, Google Gemini, and DeepSeek. This comparison evaluates both the time taken by each AI LLM platform to generate quantum operations (including networking times), as well as the execution time of these operations in Python, within Jupyter Notebook. Our results show that overall, Gemini is the fastest AI LLM in producing quantum gate designs. At the same time, the AI LLMs tested achieved working quantum operations 80% of the time. These findings highlight a promising horizon where publicly available Large Language Models can become fast collaborators with quantum computers, enabling rapid quantum gate synthesis and paving the way for greater interoperability between two remarkable and cutting-edge technologies.**

## 1 Introduction

Today's quantum computers are reaching new milestones in performance and efficacy. These systems, known as Noisy Intermediate Scale Quantum (NISQ) computers, vary in size "with a number of qubits ranging from 50 to a few hundred [1]." Key challenges confronting NISQ systems include reliability, scalability, and, soon we

posit, interoperability.

Reliability-wise NISQ systems face complex quantum noise environments. The term "Noisy emphasizes...imperfect control over those qubits; the noise will place serious limitations on what quantum devices can achieve in the near term [1]." When noise undesirably meshes with quantum information, that information becomes "entangled" with the environment in a process known as decoherence. Approaches to counter undesirable entanglement with the environment, or decoherence induced errors in quantum computers is addressed by the field of quantum error correction. This is a process whereby quantum information is encoded into a larger subspace of Hilbert space by monitoring symmetries of code space [2]. The end-goal is to achieve "fault-tolerance...the property that a circuit overall can be more reliable than the faulty gates that make it up [3]." This is especially relevant in "large-scale quantum computing [which] requires quantum error correction and fault-tolerance protocols to control the growth of errors [4]."

*"The only qubit that has no errors is the qubit which never does anything."*

— Kenneth R. Brown, *Quantum Error Correction and Architectures* [5]

Scalability-wise, the hardware of NISQ systems face challenges with control and confinement or again, coupling to the environment. This warrants approaches as atomic, molecular, and optical physics and human-crafted hardware [2]. It should be mentioned here that some entanglements are desirable by design; "the power of quantum parallelism [multiple operations] relies on the phenomenon [6]." Entanglement plays an "intriguing role...in quantum computing. As with

most good things, it is best consumed in moderation [7]."

While much research in the field focuses on the former two factors, we propose a third critical factor for quantum computers: interoperability, or the ability of quantum systems to effectively collaborate and communicate with other technologies and users. This becomes vital when considering who provides the information quantum computers are given, that is, the quantum operations and gate designs issued. Historically, the design of quantum circuits has relied heavily on human expertise. However, the advent of accessible Artificial Intelligence Large Language Models (AI LLM, or LLM for short) and machine learning methodologies presents a compelling opportunity to systematically explore and generate quantum circuit architectures that implement desired quantum operations. We argue this third factor of interoperability should be seriously considered, especially "given the successes of both machine learning and quantum computing, combining these two strands of research is an obvious direction [8]."

Although research at the intersection of LLMs and quantum computing is still emerging, a crucial gap remains: a rigorous evaluation of publicly available AI models for quantum design applications. Our paper addresses this gap by delivering one of the first systematic benchmarks of leading AI models such as WolframLLM, OpenAI, Gemini, and DeepSeek, focused specifically on quantum circuit design and gate synthesis. In summary, such systematic comparisons of AI models specifically benchmarking fast quantum design and gate synthesis have been limited and nascent, until now.

## 2 Literature Review

The Web of Science is the gold-standard database of peer-reviewed journal articles. At title search within it for "quantum comput\*" (where "\*" stands for a wild card, comput covers computation, computational, computer, computers, etc.) yields 8,139 entries with the earliest published in 1975. Even though still relatively young, the field of quantum computing or, more generally, quantum information science, is maturing rapidly with a number of research and review articles published in top journals.

In Nature, Ladd et al. (2010) explored whether storing, transmitting, and processing information encoded in uniquely quantum systems is feasible, noting that while promising, it remains unclear which technology will prevail [9]. In Science, O'Brien (2007) highlighted that all-optical quantum computing became feasible with single-photon sources and detectors as early as 2001, though practical scaling presents challenges [10]. Belenchia, Wald, and Giacomini argued "that a quantum massive particle should be thought of as being entangled with its own Newtonian-like gravitational field, and thus that a Newtonian-like gravitational field can transmit quantum information [11]."

Wasielewski et al. (2020) in Nature Reviews Chemistry emphasize molecules' quantum properties as new avenues for advancing quantum information science and its applications [12]. Bahrami, et al. "computed the decoherence rate for two quite common types of environment: thermal radiation and background gas [13]. Additionally, Tóth and Apellaniz discussed applications for quantum Fisher information, such as "how it can be used to obtain a criterion for a quantum state to be a macroscopic superposition [14]." Nayak et al. (2008), in Reviews of Modern Physics, focus on topological quantum computation using non-Abelian anyons, which offers fault-tolerance through nonlocal encoding [15].

Trapped-ion quantum computing, reviewed in Applied Physics Reviews by Bruzewicz et al. (2019), shows promise with few-ion systems already demonstrating quantum algorithms [16]. Reiher et al. (2017) argued in PNAS that quantum devices are nearing computational power beyond classical supercomputers and are well suited for studying complex chemical reactions [17].

Raussendorf and Briegel (2001) proposed one-way quantum computers based on cluster states, where computation occurs via one-qubit measurements [18]. And famously, John Preskill (2018) coined the name of Noisy Intermediate-Scale Quantum (NISQ) devices, and in his paper, described the upcoming era of 50-100 qubit machines to outpace classical computers, though still fully expecting fault-tolerant computing to require improved gates first to accomplish this feat [1].

Bourassa et al. (2021) advocated photonics as a modular, room-temperature platform for

scalable, fault-tolerant quantum computing [19], while Steiger et al. (2018) introduced ProjectQ, an open-source framework to develop and simulate quantum algorithms [20].

Lohani et al. (2022) proposed data-centric machine learning methods that enhance quantum state reconstruction accuracy without changing model architectures [21]. Tóth and Apellaniz (2014) also reviewed quantum metrology advances and how noise limits precision [14]. Loss and DiVincenzo (1998) discussed universal quantum gates using spin states in coupled quantum dots [22]. Kitaev (2003) described fault-tolerant 2D quantum computation using anyonic excitations and their braiding operations [23].

In Current Opinion in Structural Biology, Lapala (2024) highlighted how integrating AI, machine learning, and quantum computing is revolutionizing molecular dynamics simulations, calling for multidisciplinary approaches to meet emerging challenges [24].

And in their works, senior research scientists Srinivasan Arunachalam and Ronald de Wolf hinted at interoperability in several ways: This included using "quantum machine learning for "quantum supremacy", i.e., for solving some task using 50–100 qubits in a way that is convincingly faster than possible on large classical computers [8]." In essence, could certain "practical machine learning problems [dovetail] with a large provable quantum speed-up? [8]."

At the same time, understanding the potential impacts of interoperability is also imperative, especially considering "the integration of artificial intelligence and quantum computing...could introduce new issues related to data quality [24]."

Other researchers have also suggested interoperability between LLMs and quantum computers: work by Nicolas Dupuis et al. has discussed "an area of interest...to develop specialized LLMs for quantum code generation [25]."

Furthermore, the authors recognized, "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 [25]."

As well, Sanjay Vishwakarma et al. employed Qiskit HumanEval for "quantum computing tasks, each accompanied by a prompt, a canonical solution, a comprehensive test case, and

a difficulty scale to evaluate the correctness of the generated solutions [26]." Lastly, Basit et al. introduced "a novel, high-quality dataset comprising 3,347 PennyLane-specific quantum code samples and contextual descriptions, specifically curated to support LLM training and fine-tuning for quantum code assistance [27]."

### 3 Materials and Methods

The experiment's first step involved creating a master quantum prompt table (available individually in the Appendix as Table 8, Table 9, Table 10, Table 11, Table 12), which lists every experiment that would be input. The concepts chosen were an array of topics pulling from quantum gates (such as the Pauli gate set), to experiments in the quantum computing field (such as quantum fourier transforms). Twenty-five prompts were elected and chosen to get a sense of how LLMs would handle the subject materials. This number was primarily chosen because of the limitations to how much execution time is permitted on each LLM service respective to cost and the project's budget.

The Wolfram Mathematica framework was used to interface with the 4 LLM AIs. Their programmatic results were saved to text files, then carefully preserved, copied, then filtered into text files that could be executed in the Python3 language. This entailed painstakingly and carefully removing content such as header information that was not directly called for as part of the programmatic prompt. The removal was careful to only carve out content specifically between the comments START and END, as requested for in the prompts. The programmatic results were then executed in a Jupyter Notebook environment. The test platform computer was a Red Hat Enterprise Linux laptop running an Intel i7-7700HQ (3.80 GHz) with 16 gigabytes of ram.

Figure 1: Testbed Platform.

As well, we also measured the quantities of time for how long each AI platform took to respond to initial requests or connections with Mathematica. At the time of this writing, this was not possible for Wolfram LLM, as there was no native API support for measuring the connection time for this internal Wolfram service. Crucially, this captured the total time from the sending of the prompt request to the LLM, the "thinking time of the LLM" and the response to Mathematica. We also solely measured the connection time to the LLMs themselves, to get a gist of the actual LLM "thinking" time, versus networking latencies.

Then, we compared each of the LLM's programmatic output by the execution times of prompts on the testbed machine with a test harness we devised to benchmark the performance of the output. We evaluated the overall execution time in seconds. Furthermore, we looked at success rates of each prompt measured by the output created and whether or not it would compile with our test harness and compiler. If not, which ones did not compile? Lastly, we investigated the output to see how it evaluated to the expected criteria. The following results are from Wolfram Mathematica, Jupyter Notebook, Python3, and GenAI to produce open source graphics from the corresponding data.

## 4 Results

We first graphed the results of all the LLM models, with an optimal window between 0-15 seconds. 3 of the 4 LLM models generated consistently and relatively similarly low execution times for synthesizing quantum gates (Gemini, WolframLLM, and OpenAI). In contrast, DeepSeek was the relative slowest with execution times that exceeded the 15 second window. We ranked the order of the LLMs here by on average (number of

peaks) fastest to slowest. Important to note, this also included the time to file the request with the LLM service, as well as the "return trip" of the prompt.

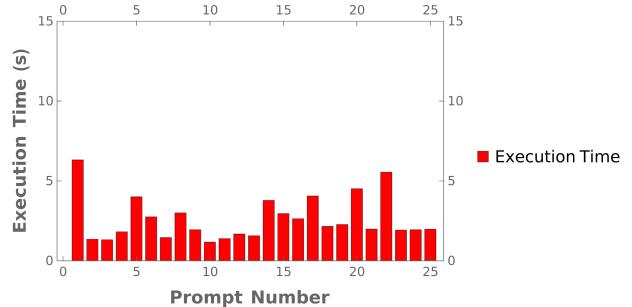


Figure 2: Gemini AI Execution Times.

Gemini was the fastest LLM on average to execute our prompts.

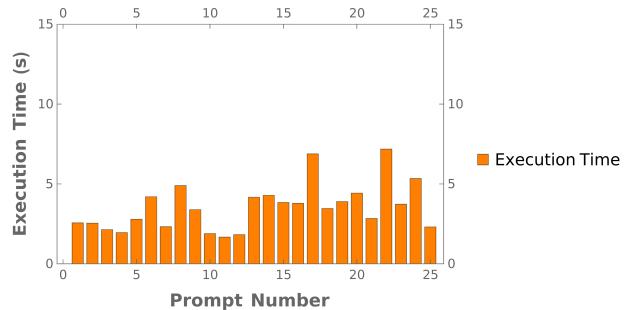


Figure 3: WolframLLM AI Execution Times.

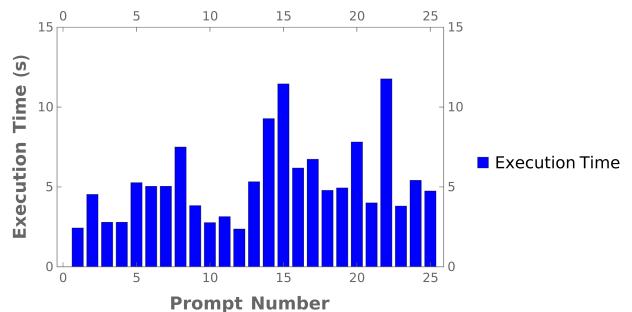


Figure 4: OpenAI Execution Times.

Gemini was followed by WolframLLM, OpenAI, and DeepSeek, respectively. Because DeepSeek was extraordinarily slower than the other LLMs on the 0-15 second scale, we elected to use a larger 0-35 second window to accommodate this extra latency, as shown in Figure 5. To showcase this difference, we also compared using the original 15-second scale with OpenAI; note the difference in Figure 6.

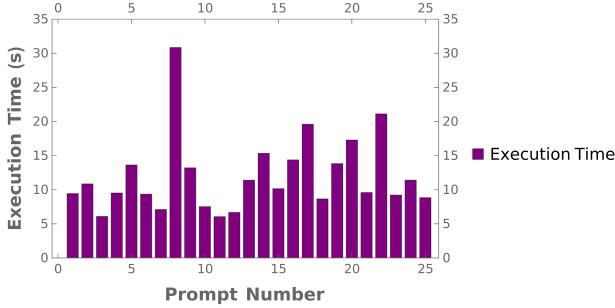


Figure 5: DeepSeek AI Execution Times.

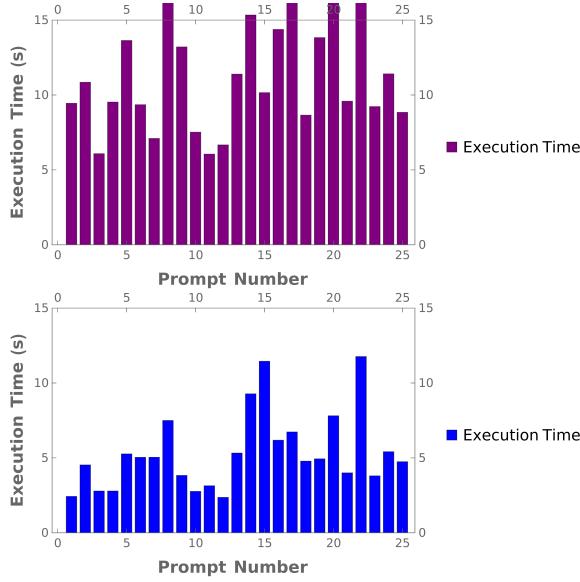


Figure 6: Comparison DeepSeek (Top) and OpenAI (Bottom)

## Average LLM Execution Time Table

LLM Model	Average Execution Time (s)
Gemini	2.67
WolframLLM	3.48
OpenAI	5.22
DeepSeek	12.23

Table 1: Average LLM Execution Time Per Model (Rounded to 3 Decimal Places, in seconds)

### 4.1 ServiceConnect Times

Next, ServiceConnect times were compared. These are composites of the total execution time measured above, but specifically the time that Wolfram Mathematica took to connect to each LLM service. We measured this to give us a picture of the networking latencies that could exist

during testing. Because of the volatility in these graphs, we plotted a 5-point moving average of the data, seeing that DeepSeek was the relative average fastest (only just) as measured by number of points (twenty-five points) and their respective times, averaged, as shown in Table 2.

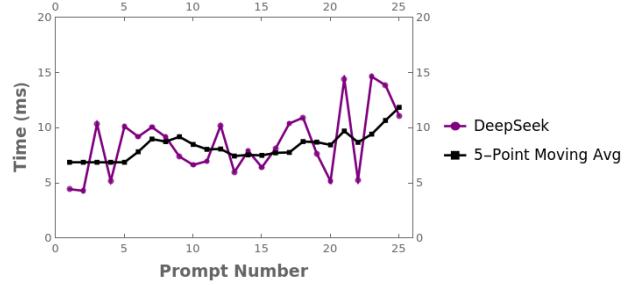


Figure 7: DeepSeek ServiceConnect Moving Averages.

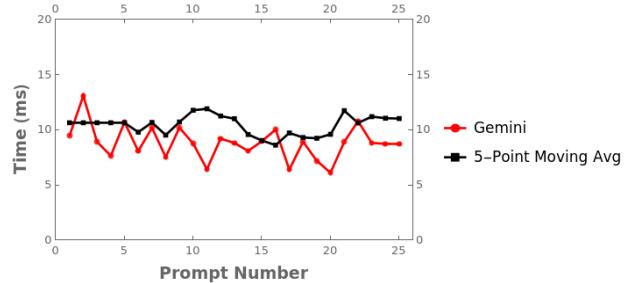


Figure 8: Gemini ServiceConnect Moving Averages.

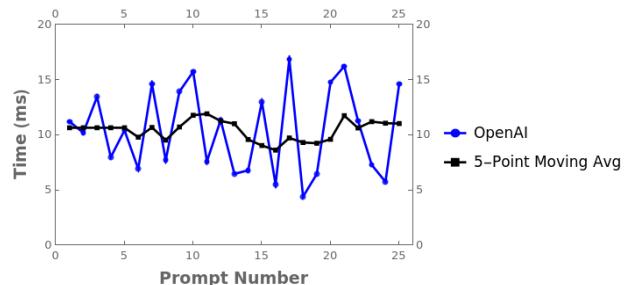


Figure 9: OpenAI ServiceConnect Moving Averages.

#### 4.1.1 Three Way Comparison

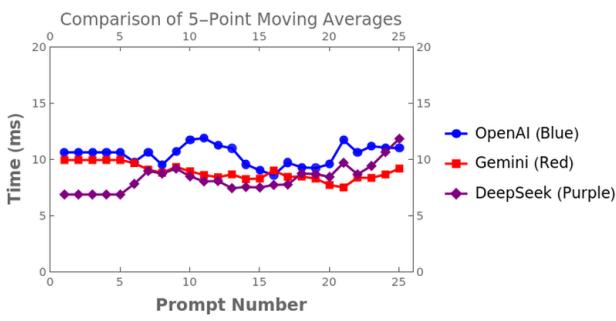


Figure 10: Three Way Three-Point Moving Average Comparison.

#### Average Service Connect Time Table

LLM Model	Average Service Connect Time (milliseconds)
OpenAI	10.93
Gemini	8.99
DeepSeek	8.73

Table 2: Average Service Connect Times for Each LLM (ms)

#### 4.2 Execution by Python

To account for any system noise of the testbed, each program was executed twice, with the second execution logged. DeepSeek was first place to execute the most code the fastest, at all prompts under 40 milliseconds.

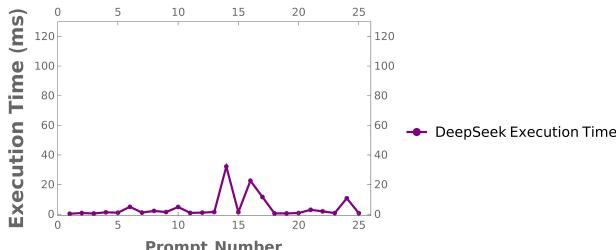


Figure 11: DeepSeek Python Execution Times.

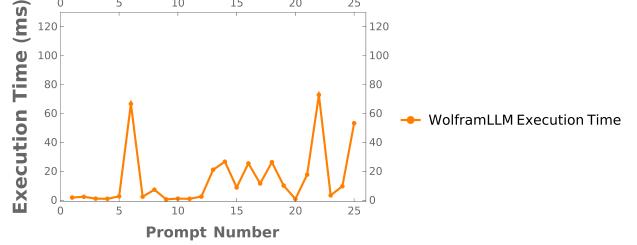


Figure 12: WolframLLM Python Execution Times.

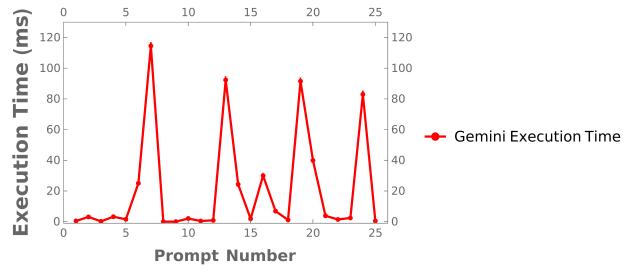


Figure 13: Gemini Python Execution Times.

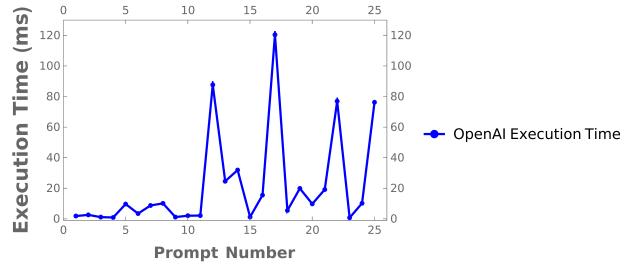


Figure 14: OpenAI Python Execution Times.

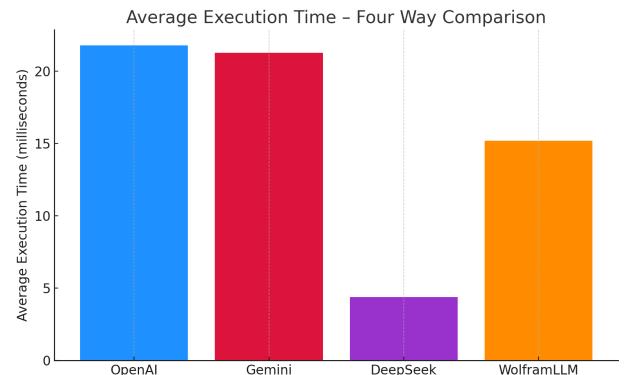


Figure 15: Four Way Average Comparison of Python Execution Times.

It is noteworthy that of all the LLMs, both OpenAI and Gemini were within a few tenths of a millisecond average of each other across executing all prompts. This average was computed by taking each prompt's execution time, adding it, and divided by the number of prompts (twenty-five).

While the averages are very tight, the prompts were of varying execution times (not the same for each prompt).

## Python Average Execution Time Table

LLM Model	Avg. Execution Time (ms)
DeepSeek	4.38
WolframLLM	15.17
Gemini	21.25
OpenAI	21.77

Table 3: Average Execution Time Per Model (Rounded to 3 Decimal Places, in milliseconds)

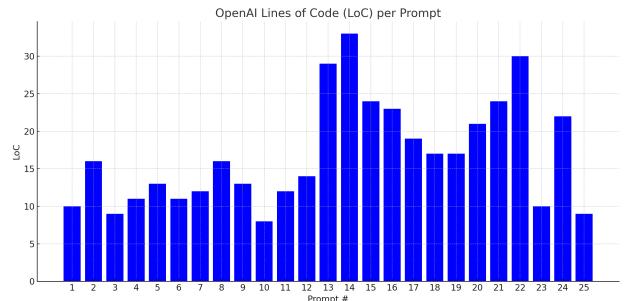


Figure 18: OpenAI LoC.

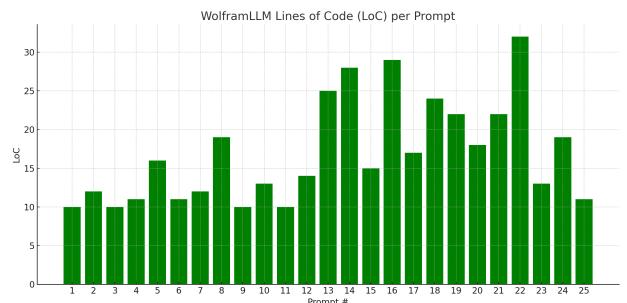


Figure 19: WolframLLM LoC.

### 4.3 Lines of Code (LoC) by Prompt

Puzzling, both OpenAI and WolframLLM had exactly the same lines of code (averaged) for all programmatic output as shown in Table 4.

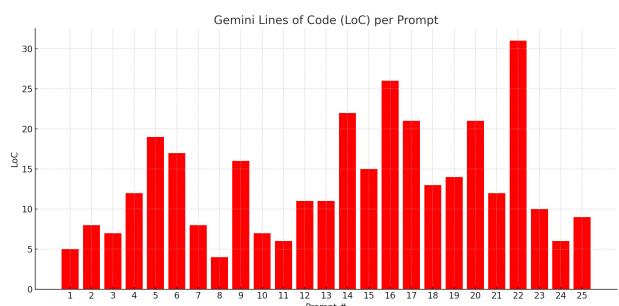


Figure 16: Gemini LoC.

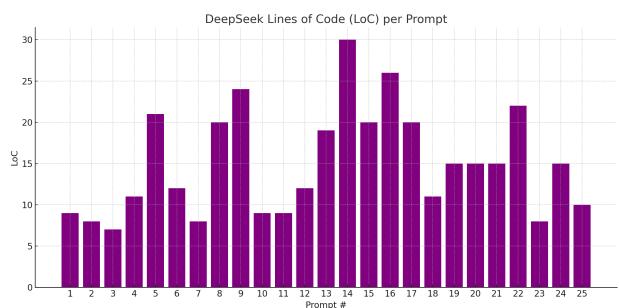


Figure 17: DeepSeek LoC.

### 4.3.1 Average Four Way Comparison

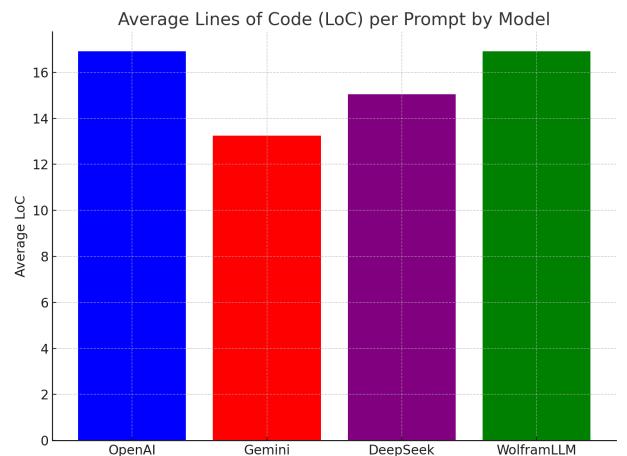


Figure 20: Average Four Way Comparison LoC.

Important to note, the prompts that failed to compile were different (see Table 7), and the size and tallies of each program varied as well. In other words, the outputs were individually different but population-wise, identical!

## Average LoC Table

Model	Average LoC
Gemini	13.24
DeepSeek	15.04
OpenAI	16.92
WolframLLM	16.92

Table 4: Average Lines of Code (LoC) per Prompt by Model

#### 4.4 Execution by Memory Usage

We also compared memory usage by platform.

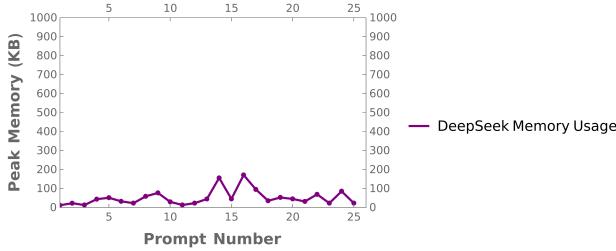


Figure 21: DeepSeek Python Memory Usage.

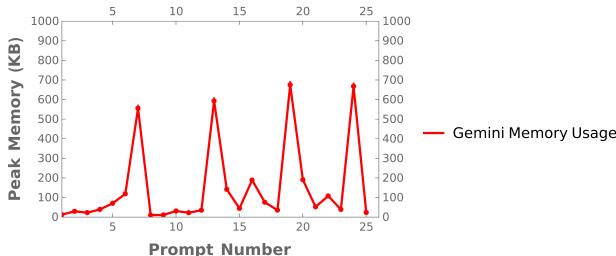


Figure 22: Gemini Python Memory Usage.

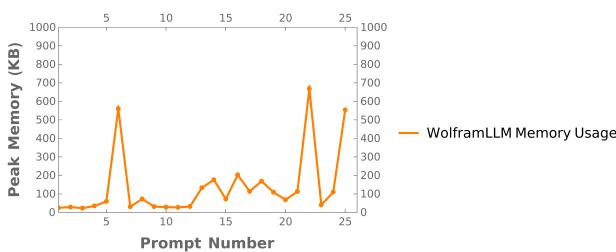


Figure 23: WolframLLM Python Memory Usage.

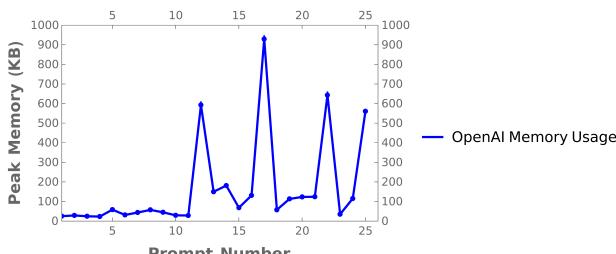


Figure 24: OpenAI Python Memory Usage.

Following this analysis, we found that the lowest memory usage and resources evaluated according to our testbed was from DeepSeek (as shown in Table 5).

#### Average Memory Usage by LLM Table

Rank	LLM	Average Memory Usage (KB)
1	DeepSeek	49.90
2	Gemini	154.66
3	WolframLLM	156.54
4	OpenAI	169.59

Table 5: Ranked Average Memory Usage of LLMs (KB)

#### 4.5 Execution Compilation success rate

Astonishingly, we found for every LLM, OpenAI, Gemini, WolframLLM and DeepSeek, none performed better overall in their success rate at producing executable code. The best and worst success rate of every LLM was 80% (as shown in Figure 25). This rate stemmed from the fact each LLM had at least two prompts that would not compile.

#### Program Execution Success Rate

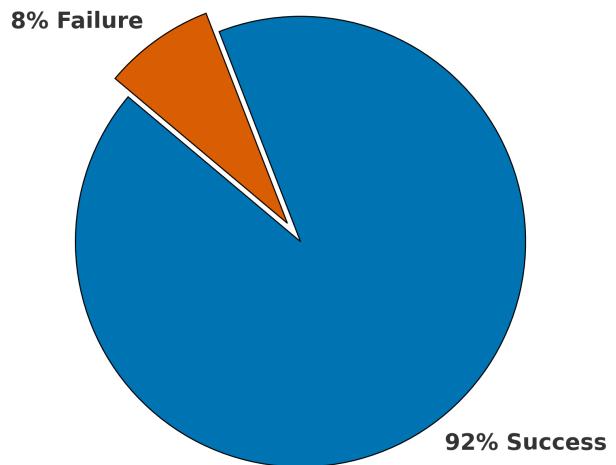


Figure 25: Success rate of all Python prompts for each LLM.

To curious readers, a list of the prompts that did not compile, with these prompts listed and

compared to other LLMs is provided in the appendix. As evident in Table 7, no prompt other than Prompt 13 was failed equally across LLMs. Even in this singular case of Prompt 13, the errors between LLMs (DeepSeek V. WolframLLM) are completely different (note a QasmQobj error versus a classical bit index out-of-range error as shown in Table 7).

#### 4.6 Evaluation Of Programmatic output

The next evaluation was how did the results of the LLMs stack up to expectation? Of the programs compiled, 23 individual programs were compared for each LLM (this is minus two non-compiled programs). Because of how the test harness was devised, evaluation results were a combination of "successes" listed as well as gate designs. The test bed program counted successes at twenty-three of twenty-five for each LLM. Important to note, many of the LLM prompts and outputs relied upon the Qiskit Framework. This discretion was not prompted as evident from the quantum prompt table (available individually in the Appendix as Table 8, Table 9, Table 10, Table 11, Table 12). More about this will be covered in the discussion.

### OpenAI Prompt-2 LLM Prompt Output

```
# START
from qiskit import QuantumCircuit
# Create a quantum circuit with 1 qubit
qc = QuantumCircuit(1)
# Apply H gate
qc.h(0)
# Apply Z gate
qc.z(0)
# Apply H gate again
qc.h(0)
# Draw the circuit
print(qc.draw())
# END
```

### OpenAI Prompt-2 LLM In Python Output

```
q: H Z H
```

Curious readers can see the full programmatic results from each LLM (filtered as well as raw data), including related project files available on a public DukeBox repository: <https://duke.box.com/v/pushing-the-limits-of-llms>. These are the comprehensive prompts that were generated by OpenAI, Gemini, WolframLLM, and DeepSeek during testing.

## 5 Discussion

Following testing, it became clear from these results that on average, the 4 LLMs, OpenAI, Gemini, WolframLLM, and DeepSeek perform roughly equal to each other in producing working, quantum designs. The meaningful difference is in computation and execution time.

Another important note was that many of the LLM programmatic results relied upon the same Qiskit framework, as opposed to OPENQASM (which Gemini did for Prompt #9) or manually defining the gates in Python's numpy (which DeepSeek did for prompt # 4). This suggests the LLM training data may come from the same sources and therefore be bias for quantum operations. A possible cause for this is simply that IBM Qiskit is that much more ubiquitous today in open source literature versus other sources, and henceforth, the training data for LLMs.

In all, if we average the LLM Execution Time data from Table 1 (measured in seconds) and Table 3 (measured in milliseconds), we can see the overall averages in the following Table 6:

### Average Overall LLM Table

Rank	LLM	Average Execution Time (s)
1	Gemini	1.35
2	WolframLLM	1.75
3	OpenAI	2.62
4	DeepSeek	6.12

Table 6: Ranked Average Execution Times of LLMs (Seconds)

This indicates that during testing considering both the LLM's "thinking" execution time, and the actual Python execution time, that Gemini is the fastest overall LLM to produce quantum

operations, followed by WolframLLM, and lastly OpenAI. Even if we account for ServiceConnect averages that were available for three of the four LLM models (as shown in Table 2), this still does not change these rankings. In total, the LLMs performed with an accuracy of 80%.

These are significant findings, as it demonstrates that current LLMs are both relatively fast, and achieve notable success rates in producing functional quantum operations. But how does this impact quantum gate synthesis in terms of the third key factor introduced earlier—interoperability, alongside reliability and scalability? We anticipate that as LLMs continue to improve, their applications in designing the quantum circuits for quantum computers of the future will expand accordingly.

Future research should focus on tracking the progression of gate synthesis performance in current and upcoming LLMs, as well as analyzing historical improvements from earlier models. Meanwhile, the data presented here suggest that publicly available LLMs employed for gate synthesis remains an exciting avenue to watch!

## 6 Conclusion

We presented the first benchmarking study comparing popular and publicly available AI models tasked with creating quantum gate designs. The Wolfram Mathematica framework was used to interface with the 4 LLM AIs, including WolframLLM, OpenAI ChatGPT, Google Gemini, and DeepSeek. This comparison evaluates both the time taken by each AI platform to generate quantum operations (including networking times), as well as the execution time of these operations in Python, within Jupyter Notebook. Our results show that overall, Gemini is the fastest LLM in producing quantum gate designs. At the same time, the LLMs tested achieved working quantum operations 80% of the time. These findings highlight a promising horizon where publicly available Large Language Models can become fast collaborators with quantum computers, enabling rapid quantum gate synthesis and paving the way for greater interoperability between two remarkable and cutting-edge technologies.

## A Appendix

Herein are tables pertaining to the experiments performed.

### Prompt Failure And Errors Table

LLM	Prompt File	Error Description
DeepSeek	Prompt13_Output.txt	'Classical bit index 0 is out-of-range.'
DeepSeek	Prompt18_Output.txt	name 'np' is not defined
Gemini	Prompt9_Output.txt	invalid syntax (<string>, line 1)
Gemini	Prompt22_Output.txt	'Index 1 out of range for size 1.'
OpenAI	Prompt3_Output.txt	'InstructionSet' object has no attribute 'num_...'
OpenAI	Prompt15_Output.txt	'duplicate qubit arguments'
WolframLLM	Prompt13_Output.txt	'QasmQobj' object has no attribute 'name'
WolframLLM	Prompt20_Output.txt	'duplicate qubit arguments'

Table 7: LLM Prompt Errors and Descriptions

### Quantum Prompts Table

#	Concept	Prompt	Goal	Expected Output	Eval Criteria
1	Identity	Write code that builds a quantum circuit performing the identity operation. The code must begin with # START and end with # END. Return only executable code.	No change in state	Empty or canceling gates	Unitary = I
2	Pauli X	Using only Hadamard and Z gates, write code that implements an X gate. The code must begin with # START and end with # END. Output the circuit.	Basis flip	$H \rightarrow Z \rightarrow H$	Gate equivalence
3	Pauli Y	Write code that constructs a Pauli-Y gate using only X and Z gates. The code must begin with # START and end with # END. Output only the quantum code.	Phase + bit flip	iXZ form	Functional correctness
4	Pauli Z	Write code that expresses the Z gate using a sequence of T and $T^\dagger$ gates. The code must begin with # START and end with # END. Return only executable code.	Construct Z from phase gates	T; T; T; T or similar	Phase match
5	Hadamard	Construct a Hadamard gate from rotation gates using code. The code must begin with # START and end with # END. Output only executable code.	X-basis projection	Rx/Rz decomposition	Bloch rotation fidelity

Table 8: Quantum Prompts Table Part 1 (Prompts 1–5)

#	Concept	Prompt	Goal	Expected Output	Eval Criteria
6	CNOT decomposition	Write code that builds a CNOT gate using only CZ and Hadamard gates. The code must begin with # START and end with # END. Return the working circuit.	Functional CNOT	H; CZ; H	Control-target flip
7	SWAP	Write a quantum circuit that swaps two qubits without using a SWAP gate. The code must begin with # START and end with # END. Output only executable code.	Qubit exchange	3 CNOTs	Measurement match
8	Toffoli (CCNOT)	Decompose the Toffoli gate using only H, T, and CNOT gates. The code must begin with # START and end with # END. Write executable code.	Control-control NOT	Standard decomposition	Functional truth table
9	Fredkin (CSWAP)	Construct a Fredkin (CSWAP) gate using a Toffoli gate or other gates. The code must begin with # START and end with # END. Output only the code.	Controlled swap	Toffoli-based or ancilla-based	Swap conditionally
10	Bell state	Write code that prepares a Bell state between two qubits. The code must begin with # START and end with # END. Output the executable circuit only.	Create entanglement	H; CNOT	Result = 00, 11

Table 9: Quantum Prompts Table Part 2 (Prompts 6–10)

#	Concept	Prompt	Goal	Expected Output	Eval Criteria
11	Equal superposition	Create a 3-qubit quantum circuit that produces a uniform superposition. The code must begin with # START and end with # END. Return only code.	8-state	$ \psi\rangle$	3 Hadamards
12	GHZ state	Build a GHZ state circuit for 3 qubits using H and CNOT gates. The code must begin with # START and end with # END. Output only code.	Multi-qubit entanglement	H + CNOT + CNOT	Result = 000, 111
13	W state	Write a circuit that prepares a W state using any method. The code must begin with # START and end with # END. Output only the code.	3-qubit, one-hot entanglement	Controlled superposition	Output = 001, 010, 100
14	Deutsch (no CNOT)	Write code that implements Deutsch's algorithm without using the CNOT gate. The code must begin with # START and end with # END. Output executable code only.	Functional test of $f(x)$	H; CZ; H or Oracle variant	0 = constant, 1 = balanced
15	Bernstein-Vazirani	Construct the Bernstein–Vazirani algorithm in code for the string $s = 101$ . The code must begin with # START and end with # END. Return only the quantum circuit.	Reveal secret string	Hs + Oracle + H	Result = s

Table 10: Quantum Prompts Table Part 3 (Prompts 11–15)

#	Concept	Prompt	Goal	Expected Output	Eval Criteria
16	Grover's 1-step	Write code for Grover's algorithm on 2 qubits with a single marked item. The code must begin with # START and end with # END. Output executable code only.	Amplify marked amplitude	Oracle + diffuser	Output = target index
17	Simon's algorithm	Implement Simon's algorithm for a 2-bit function in executable code. The code must begin with # START and end with # END. Return only the quantum circuit.	Detect hidden XOR pattern	Register logic	Measurement structure
18	Quantum Fourier Transform	Write a 3-qubit quantum Fourier transform circuit in code. The code must begin with # START and end with # END. Return only the code.	Phase encoding	Swap + Hadamard + CP gates	Matches QFT matrix
19	Logical AND (measured)	Construct a quantum circuit that behaves like a logical AND gate via measurement. The code must begin with # START and end with # END. Output only the code.	Output = 1 if both inputs = 1	CCNOT or entangled logic	Measured truth table
20	Logical OR (measured)	Build a circuit that behaves like a logical OR gate using quantum measurement. The code must begin with # START and end with # END. Return executable code.	Output = 1 if any input = 1	Toffoli + ancilla or phase	Output probability logic

Table 11: Quantum Prompts Table Part 4 (Prompts 16–20)

#	Concept	Prompt	Goal	Expected Output	Eval Criteria
21	Entangle then swap	Entangle two qubits, then swap them. Write the code that does both. The code must begin with # START and end with # END. Output only code.	Confirm state transfer	Bell + SWAP	Measure fidelity
22	Quantum teleportation	Write a circuit that implements quantum teleportation for a single-qubit state. The code must begin with # START and end with # END. Output executable code.	Reconstruct qubit on receiver	Bell + CX + classical correction	Final qubit = input
23	Controlled-Z from CNOT	Construct a Controlled-Z gate using only CNOT and H gates. The code must begin with # START and end with # END. Write the code.	Control-Z logic	H; CNOT; H	Matrix equivalence
24	XOR by measurement	Write code that encodes XOR into a quantum circuit and decodes by measurement. The code must begin with # START and end with # END. Output only the circuit.	0 if same, 1 if different	CNOT or parity circuit	Output truth table
25	Random entangled pair	Prepare a random Bell pair from an ancilla and Hadamard gates. The code must begin with # START and end with # END. Return only the executable circuit.	Bell state variation	Randomized entangler	Result = 00, 11

Table 12: Quantum Prompts Table Part 5 (Prompts 21–25)

## References

- [1] J. Preskill, “Quantum computing in the nisq era and beyond,” *Quantum*, vol. 2, p. 79, 2018. [Online]. Available: <https://doi.org/10.22331/q-2018-08-06-79>
- [2] K. Brown, “Quantum error correction now!” Invited lecture at JST Moonshot Symposium, Japan Science and Technology Agency (JST), July 18, 2023, 2023. [Online]. Available: [https://www.jst.go.jp/moonshot/sympo/20230718/material/1\\_4\\_ken\\_brown.pdf](https://www.jst.go.jp/moonshot/sympo/20230718/material/1_4_ken_brown.pdf)
- [3] M. McEwen, D. Bacon, and C. Gidney, “Relaxing hardware requirements for surface code circuits using time-dynamics,” *Quantum*, vol. 7, p. 1172, 2023. [Online]. Available: <https://doi.org/10.22331/q-2023-11-07-1172>
- [4] A. Fahimniya, H. Dehghani, K. Bharti, S. Mathew, A. J. Kollár, A. V. Gorshkov, and M. J. Gullans, “Fault-tolerant hyperbolic floquet quantum error correcting codes,” 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2309.10033>
- [5] K. Brown and M. Newman, “Quantum error correction and architectures: A tutorial,” 2024. [Online]. Available: <https://brownlab.pratt.duke.edu/>
- [6] D. H. McIntyre, *Quantum Mechanics*. Cambridge University Press, 2022. [Online]. Available: <https://doi.org/10.1017/9781009310598>
- [7] D. Gross, S. T. Flammia, and J. Eisert, “Most quantum states are too entangled to be useful as computational resources,” *Phys. Rev. Lett.*, vol. 102, no. 19, p. 190501, May 2009. [Online]. Available: <https://doi.org/10.1103/PhysRevLett.102.190501>
- [8] S. Arunachalam and R. de Wolf, “A survey of quantum learning theory,” 2017. [Online]. Available: <https://doi.org/10.1145/3106700.3106710>
- [9] T. D. Ladd, F. Jelezko, R. Laflamme, Y. Nakamura, C. Monroe, and J. L. O’Brien, “Quantum computers,” *Nature*, vol. 464, no. 7285, pp. 45–53, 2010. [Online]. Available: <https://doi.org/10.1038/nature08812>
- [10] J. L. O’Brien, “Optical quantum computing,” *Science*, vol. 318, no. 5856, pp. 1567–1570, 2007. [Online]. Available: <https://doi.org/10.1126/science.1156956>
- [11] A. Belenchia, R. M. Wald, F. Giacomini, E. Castro-Ruiz, C. Brukner, and M. Aspelmeyer, “Information content of the gravitational field of a quantum superposition,” *International Journal of Modern Physics D*, vol. 28, no. 14, p. 1943001, 2019. [Online]. Available: <https://doi.org/10.48550/arXiv.1905.04496>
- [12] M. R. Wasielewski *et al.*, “Exploiting chemistry and molecular systems for quantum information science,” *Nat. Rev. Chem.*, vol. 4, no. 9, pp. 490–504, 2020. [Online]. Available: <https://doi.org/10.1038/s41570-020-0210-7>
- [13] M. Bahrami, A. Shafiee, and A. Bassi, “Decoherence effects on superpositions of chiral states in a chiral molecule,” *Physical Chemistry Chemical Physics*, vol. 14, no. 25, pp. 9214–9218, 2012. [Online]. Available: <https://doi.org/10.48550/arXiv.1202.0201>
- [14] G. Tóth and I. Apellaniz, “Quantum metrology from a quantum information science perspective,” *Journal of Physics A: Mathematical and Theoretical*, vol. 47, no. 42, p. 424006, 2014. [Online]. Available: <https://doi.org/10.1088/1751-8113/47/42/424006>
- [15] C. Nayak, S. H. Simon, A. Stern, M. Freedman, and S. D. Sarma, “Non-abelian anyons and topological quantum computation,” *Rev. Mod. Phys.*, vol. 80, no. 3, pp. 1083–1159, 2008. [Online]. Available: <https://doi.org/10.1103/RevModPhys.80.1083>
- [16] C. D. Bruzewicz, J. Chiaverini, R. McConnell, and J. M. Sage, “Trapped-ion quantum computing: Progress and challenges,” *Applied Physics Reviews*, vol. 6, no. 2, p. 021314, 2019. [Online]. Available: <https://doi.org/10.1063/1.5088164>

- [17] M. Reiher, N. Wiebe, K. M. Svore, D. Wecker, and M. Troyer, “Elucidating reaction mechanisms on quantum computers,” *Proc. Natl. Acad. Sci. U.S.A.*, vol. 114, no. 29, pp. 7555–7560, 2017. [Online]. Available: <https://doi.org/10.1073/pnas.1619152114>
- [18] R. Raussendorf and H. J. Briegel, “A one-way quantum computer,” *Phys. Rev. Lett.*, vol. 86, no. 22, p. 5188, 2001. [Online]. Available: <https://doi.org/10.1103/PhysRevLett.86.5188>
- [19] J. E. Bourassa, R. N. Alexander, M. Vasmer, A. Patil, I. Tzitrin, T. Matsuura, I. Dhand *et al.*, “Blueprint for a scalable photonic fault-tolerant quantum computer,” *Quantum*, vol. 5, p. 392, 2021. [Online]. Available: <https://doi.org/10.22331/q-2021-02-22-392>
- [20] D. S. Steiger, T. Häner, and M. Troyer, “Projectq: an open source software framework for quantum computing,” *Quantum*, vol. 2, p. 49, 2018. [Online]. Available: <https://doi.org/10.22331/q-2018-01-31-49>
- [21] S. Lohani, J. M. Lukens, R. T. Glasser, T. A. Searles, and B. T. Kirby, “Data-centric machine learning in quantum information science,” *Machine Learning: Science and Technology*, vol. 3, no. 4, p. 04LT01, 2022. [Online]. Available: <https://doi.org/10.1088/2632-2153/ac96b5>
- [22] D. Loss and D. P. DiVincenzo, “Quantum computation with quantum dots,” *Phys. Rev. A*, vol. 57, no. 1, p. 120, 1998. [Online]. Available: <https://doi.org/10.1103/PhysRevA.57.120>
- [23] A. Y. Kitaev, “Fault-tolerant quantum computation by anyons,” *Annals of Physics*, vol. 303, no. 1, pp. 2–30, 2003. [Online]. Available: [https://doi.org/10.1016/S0003-4916\(02\)00018-0](https://doi.org/10.1016/S0003-4916(02)00018-0)
- [24] A. Lappala, “The next revolution in computational simulations: Harnessing ai and quantum computing in molecular dynamics,” *Current Opinion in Structural Biology*, vol. 89, p. 102919, 2024. [Online]. Available: <https://doi.org/10.1016/j.sbi.2024.102919>
- [25] N. Dupuis, L. Buratti, S. Vishwakarma, A. V. Forrat, D. Kremer, I. Faro, R. Puri, and J. Cruz-Benito, “Qiskit code assistant: Training llms for generating quantum computing code,” 2024. [Online]. Available: <https://doi.org/10.48550/arXiv.2405.19495>
- [26] S. Vishwakarma, F. Harkins, S. Golecha, V. S. Bajpe, N. Dupuis, L. Buratti, D. Kremer, I. Faro, R. Puri, and J. Cruz-Benito, “Qiskit humaneval: An evaluation benchmark for quantum code generative models,” in *2024 IEEE International Conference on Quantum Computing and Engineering (QCE)*, 2024, pp. 1169–1176. [Online]. Available: <https://doi.org/10.1109/QCE60285.2024.00137>
- [27] A. Basit, N. Innan, H. Asif, M. Shao, M. Kashif, A. Marchisio, and M. Shafique, “Pennylang: Pioneering llm-based quantum code generation with a novel pennylane-centric dataset,” 2025. [Online]. Available: <https://doi.org/10.48550/arXiv.2503.02497>