When Large Language Models Meet Quantum Programming: A Review

Yitong Chen[1], Zhuohua Li[2], Cheng Wen[1], Shiyuan Yu[1], Jie Su[1], Jianhao He[2], Shengchao Qin[1]

[1] Guangzhou Institute of Technology, Xidian University, China

[2] The Chinese University of Hong Kong, Shatin, N.T., Hong Kong

Introduction

Quantum computing promises to solve problems beyond the reach of classical computers, but programming quantum devices remains exceptionally difficult, which requires expertise in quantum mechanics, linear algebra, and rapidly evolving platform-specific frameworks. Large language models (LLMs), which have demonstrated strong capabilities in classical code generation and software engineering, offer a promising path to lower this barrier. This review surveys the emerging intersection of LLMs and quantum programming, organized around the **quantum software development lifecycle**.

We distinguish **two** categories of tasks: those analogous to classical software engineering, such as code generation, bug detection and repair, linting, and cross-platform transpilation, but complicated by quantum-specific constraints; and those unique to quantum computing, such as circuit optimization, noise-aware compilation, and equivalence verification of quantum circuits. We comparatively analyze LLM-based approaches spanning prompt engineering, supervised fine-tuning, reinforcement learning, retrieval-augmented generation, and multi-agent systems, and review existing benchmarks, datasets, and evaluation metrics, highlighting the gap between syntactic correctness and physical semantic correctness.

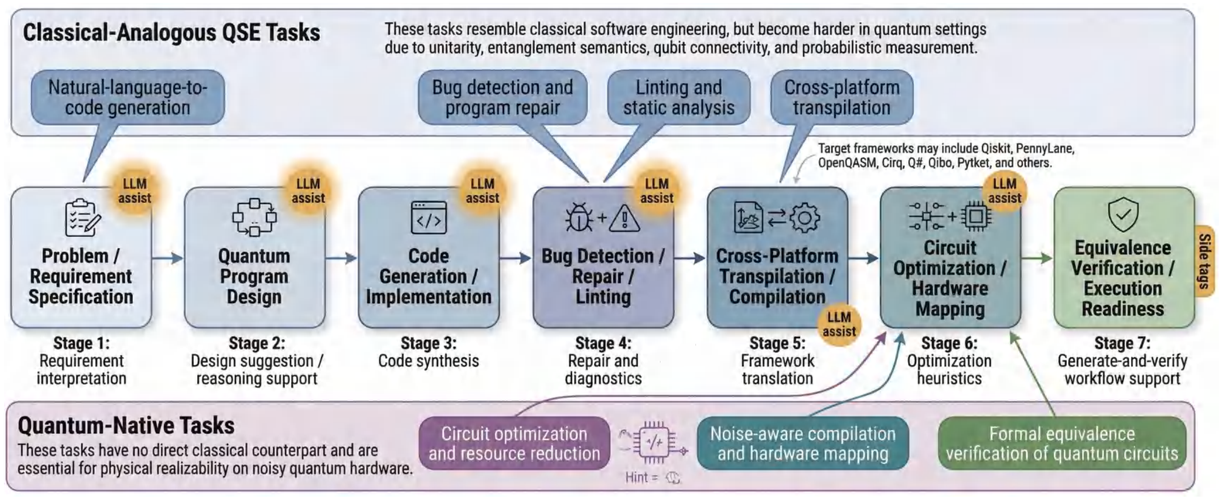


FIGURE 1 Overview of the quantum software development lifecycle (QSDLC).

LLMs for Classical-Analogous QSE Tasks

Code generation from natural language is the most studied task in LLM-assisted quantum programming. Figure 2 visualizes the composition of the reviewed literature in terms of both model families and training strategies:

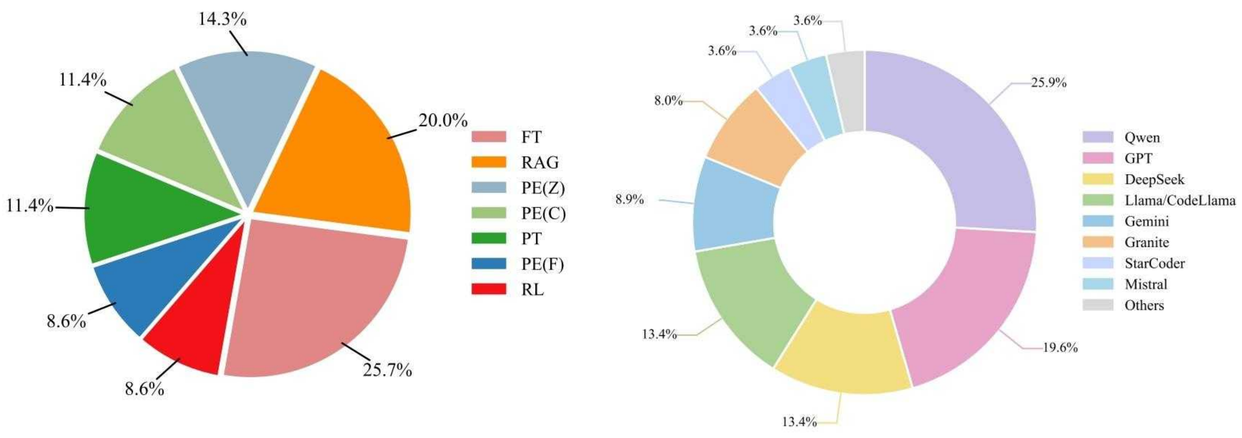


FIGURE 2 Distribution of LLM models and techniques used across reviewed studies..

LLM-based approaches for quantum programming span diverse operational strategies, frameworks, and evaluation standards. As highlighted in Table X, current methods range from foundation code completion tools and interactive simulator feedback pipelines, to advanced agentic frameworks for complex circuits and cross-framework support beyond Qiskit.

Note: Due to space constraints, Table 1 highlights selected key representatives. For the full comparative table covering all 19+ evaluated systems, see our full paper.

TABLE 1 Representative LLM-Based Approaches for Quantum Programming.

Method/System	Task	Framework	Benchmark/Dataset	Primary Metric
Qiskit Code Assistant [12]	Code Gen.	Qiskit	QHE, HumanEval	Pass@k
QCoder [15]	Code Gen.	Qiskit	QCoder Benchmark	Success Rate
Agent-Q [28]	Code Gen.	OpenQASM 3.0	AGENT-Q Dataset	Syntactical Correctness, Expectation Value Analysis
PennyLang [26]	Code Gen.	PennyLane	PennyLang Dataset	Pass@k

LLMs for Quantum-Native Tasks

This section evaluates tasks fundamentally bounded by quantum-physical constraints, where classical algorithms cannot directly apply. Across three key sub-domains, LLM adoption varies significantly: LLMs act as active heuristic proposal generators for circuit optimization, remain largely absent in classical-dominated noise-aware mapping, and integrate with formal verifiers under an emerging neuro-symbolic paradigm.

TABLE 2 Representative LLM-Based Approaches for Quantum Programming.

Sub-domain	Core Task	LLM Role	Classical Baseline / Method	Status / Key Finding
Circuit Optimization	Resource Reduction	Heuristic Generator	ZX-calculus rewrites, DFS search	Active
Noise-Aware Mapping	Qubit Routing	Minimal	Pytket, Qiskit Transpiler	Classical dominant (Research Gap)
Formal Verification	Equivalence Check	Proposal Verifier	ZX diagram reduction, Model Counting	Neuro-symbolic (Generate + Verify)

Benchmarks & Dataset

Recent efforts in quantum LLM evaluation and instruction tuning span multiple target frameworks, including multi-framework setups. As shown in Table 3, benchmarks have evolved from basic unit-test execution to large-scale automated verification, interactive simulator feedback, and quantum-semantic evaluation using process fidelity. In parallel, Table 4 summarizes the key instruction datasets driving model alignment.

TABLE 3 Comparison of Quantum Code Generation Benchmarks.

Benchmark	Year	Target Framework	# Tasks	Evaluation Method	Key Innovation
Qiskit HumanEval [25]	2024	Qiskit	101	Unit test; Pass@1	First execution-based benchmark for Qiskit.
QHackBench [16]	2025	PennyLane	49	exec. success, Pass@k	First PennyLane-focused evaluation.
QuanBench [46]	2025	Qiskit	44	process fidelity; Pass@k	Quantum-semantic fidelity evaluation
QCircuitBench [14]	2025	QASM	120,290	Syntax; semantic; efficiency	Large-scale automatic verification.
QuantumBench [55]	2025	Qiskit	784	Exact match; execution	Conceptual + math + programming.
QCoder [15]	2025	Qiskit	58	Simulator feedback; acc.	Hardware-aware iterative evaluation.
PennyLang [26]	2025	PennyLane	3,347	RAG pipeline; Pass@5	First large-scale PennyLane dataset.

TABLE 4 Quantum Code Instruction-Tuning Datasets.

Dataset	Year	Target FW	Size	Construction	Availability
QuantumLLMInstruct [56]	2024	Multiple	500k+	Synthetic 4-stage pipeline	HuggingFace
PennyLang [26]	2025	PennyLane	3,347	GitHub scraping + GPT-4o	Open-source
Agent-Q [28]	2025	OpenQASM	14,000	Eigensolvers + QAOA/VQE	HuggingFace/ GitHub
Qiskit Instr. [48]	2025	Qiskit	3,000	GitHub scraping + Llama synthesis	—
Synthetic Qiskit [36]	2025	Qiskit	10k SFT; 4.9k DPO	Synthetic + teacher models	—
QSpark Dataset [24]	2025	Qiskit	522 curated	Crawled + simulation tests	GitHub

Conclusions

This paper surveyed the intersection of large language models and quantum programming, organized around the quantum software development lifecycle. We distinguished classical-analogous tasks, where quantum constraints make familiar SE problems harder, from quantum-native tasks with no classical counterpart. Current LLMs show strong potential in code generation and program repair, but remain limited by the quantum semantic gap, benchmark fragmentation, and the absence of real hardware evaluation. Bridging these gaps through neuro-symbolic integration and unified evaluation frameworks will be essential for advancing LLM-assisted quantum programming from research prototypes toward practical deployment.