Scaling Hybrid Quantum–HPC Applications with the Quantum Framework

Srikar Chundury
schundu3@ncsu.edu
North Carolina State University
Raleigh, NC, USA

Amir Shehata
shehataa@ornl.gov
Oak Ridge National Laboratory
Oak Ridge, TN, USA

Seongmin Kim
kims@ornl.gov
Oak Ridge National Laboratory
Oak Ridge, TN, USA

Muralikrishnan Gopalakrishnan Meena
gopalakrishm@ornl.gov
Oak Ridge National Laboratory
Oak Ridge, TN, USA

Chao Lu
luc1@ornl.gov
Oak Ridge National Laboratory
Oak Ridge, TN, USA

Kalyana Gottiparthi
gottiparthik@ornl.gov
Oak Ridge National Laboratory
Oak Ridge, TN, USA

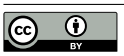
Eduardo Antonio Coello Perez
coellopereea@ornl.gov
Oak Ridge National Laboratory
Oak Ridge, TN, USA

Frank Mueller
fmuelle@ncsu.edu
North Carolina State University
Raleigh, NC, USA

In-Saeng Suh
suhi@ornl.gov
Oak Ridge National Laboratory
Oak Ridge, TN, USA

Abstract

Hybrid quantum-high performance computing (Q-HPC) workflows are emerging as a key strategy for running quantum applications at scale in current noisy intermediate-scale quantum (NISQ) devices. These workflows must operate seamlessly across diverse simulators and hardware backends since no single simulator offers the best performance for every circuit type. Simulation efficiency depends strongly on circuit structure, entanglement, and depth, making a flexible and backend-agnostic execution model essential for fair benchmarking, informed platform selection, and ultimately the identification of quantum advantage opportunities. In this work, we extend the Quantum Framework (QFw), a modular and HPC-aware orchestration layer, to integrate multiple local backends (Qiskit Aer, NWQ-Sim, QTensor, and TN-QVM) and a cloud-based quantum backend (IonQ) under a unified interface. Using this integration, we execute a number of non-variational as well as variational workloads. The results highlight workload-specific backend advantages: while Qiskit Aer's matrix product state excels for large Ising models, NWQ-Sim not only leads on large-scale entanglement and Hamiltonian but also shows the benefits of concurrent subproblem execution in a distributed manner for optimization problems. These findings demonstrate that simulator-agnostic, HPC-aware orchestration is a practical path toward scalable, reproducible, and portable Q-HPC ecosystems, thereby accelerating progress toward demonstrating quantum advantage.



SC Workshops '25, St Louis, MO, USA

ACM ISBN 979-8-4007-1871-7/2025/11
https://doi.org/10.1145/3731599.3767553

CCS Concepts

• **Hardware → Quantum computation**; • **Computer systems organization → Distributed architectures**; • **Computing methodologies → Simulation evaluation**; • **General and reference → Performance**.

Keywords

Quantum computing, High-performance computing, Quantum simulation, Hybrid quantum-classical workflows, Quantum frameworks

ACM Reference Format:
Srikar Chundury, Amir Shehata, Seongmin Kim, Muralikrishnan Gopalakrishnan Meena, Chao Lu, Kalyana Gottiparthi, Eduardo Antonio Coello Perez, Frank Mueller, and In-Saeng Suh. 2025. Scaling Hybrid Quantum–HPC Applications with the Quantum Framework. In *Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3731599.3767553

1 Introduction

As scientific and engineering problems increase in complexity, traditional computational methods face fundamental limitations due to exponential scaling [1, 2]. Domains such as combinatorial optimization, materials discovery, and large-scale linear algebra contain problem instances that become classically intractable beyond modest sizes [3–5]. Quantum computing offers a promising paradigm to address such challenges by exploiting quantum parallelism, entanglement, and superposition [6].

Progress in quantum hardware [1, 7–9] has delivered small- to mid-scale quantum devices, so-called quantum processing units (QPUs), that can act as accelerators alongside CPUs and GPUs in high-performance computing (HPC) environments. Yet practical quantum advantage remains out of reach due to limited qubit counts, device noise, and the need for seamless integration with large-scale classical workflows [10]. These constraints have intensified interest

in hybrid quantum-classical algorithms, where quantum subroutines are embedded within classical outer loops. In contrast to their non-variational counterpart, variational algorithms are less prone to adverse affects of today's noisy quantum devices. Such algorithms, including the Quantum Approximate Optimization Algorithm (QAOA) [11, 12], the Variational Quantum Linear Solver (VQLS) [13], and the Variational Quantum Eigen solver [14], leverage the quantum-classical hybrid structure, coupling near-term QPUs with HPC-scale classical optimization.

To evaluate the capabilities of emerging Q-HPC systems, both non-variational and variational workloads should be considered. The non-variational set that we choose to analyze includes the generation of Greenberger–Horne-Zeilinger (GHZ) [15] fully entangled states, Hamiltonian simulation (HAM) [16], the transverse-field Ising model (TFIM) [17], and the Harrow–Hassidim-Lloyd (HHL) [18] linear solver. The variational set includes QAOA [19], and Distributed QAOA (DQAOA) [20]. While these workloads represent key algorithmic families in physics, engineering, and scientific computing [21–24], they are computationally demanding, and hence are seldom evaluated at large scales.

To address the scalability issue, we extend the Quantum Framework (QFw) [25–27], a portable and HPC-aware orchestration layer for quantum applications, to integrate multiple local and one cloud-based backend under a unified interface. The local backends include NWQ-Sim [28], Qiskit Aer [29], TN-QVM [30], and QTensor [31], each launched in distributed mode via the PMIx Reference RunTime Environment (PRTE) and the Message Passing Interface (MPI). The cloud backend interfaces with IonQ [32]. Deployed on the Frontier supercomputer [33] via SLURM heterogeneous job groups, QFw enables coordinated execution across heterogeneous resources with fine-grained control over scheduling, communication, and device allocation.

Furthermore, we show that QFw can run identical application code across all backends without modification, supporting reproducible benchmarking and rapid backend substitution. Our evaluation reveals workload-specific performance advantages and demonstrates the role of distributed execution and asynchronous orchestration in scaling large variational workloads.

The contributions of this work are the following:

- We integrate Qiskit-Aer, QTensor, and IonQ backends into QFw.
- We scale execution and evaluation of variational and non-variational workloads on QFw across multiple local simulators and a cloud backend.
- We provide an implementation together with large-scale experiments of DQAOA for metamaterial optimization on the Frontier supercomputer.

The remainder of this paper is organized as follows: Section 2 reviews technical background and related work; Section 4 details backend integration and orchestration design; Section 5 describes the compute platform, backends, and benchmarks; Section 6 presents performance and scalability results; Section 7 analyzes observations and implications; and Section 8 summarizes our findings and outlines future directions.

2 Background

This section provides the context for our study. We first summarize the QFw execution model and services that enable distributed, backend-agnostic orchestration (Section 2.1). We then define the workload taxonomy, covering fixed-circuit non-variational kernels (GHZ, HAM/TFIM, and HHL, see Section 2.2) and parameterized, optimizer-in-the-loop variational algorithms (QAOA and DQAOA, see Section 2.3). Together, these elements motivate the orchestration choices used in our experiments and ground the performance analysis presented later.

2.1 Quantum Framework (QFw)

The QFw [25–27] is a scalable orchestration platform designed to integrate quantum and classical computing resources for large-scale hybrid execution. QFw provides a modular, backend-agnostic interface for managing quantum circuit execution across CPUs, GPUs, and QPUs in HPC environments. By abstracting backend-specific details, QFw enables quantum applications to remain portable and reproducible, allowing researchers to switch between simulation and hardware targets with minimal code changes.

QFw builds on the Process Management Interface for Exascale (PMIx) Reference RunTime Environment (PRTE) in the Distributed Virtual Machine (DVM) mode to enable rapid process spawning and low-latency coordination across distributed nodes. Its architecture comprises three primary services. (1) The *Quantum Platform Manager* (QPM) acts as a central dispatcher, selecting execution backends and managing task configurations. (2) The *Quantum Resource Controller* (QRC) schedules and launches quantum tasks across MPI ranks, ensuring efficient utilization of allocated resources. (3) Communication between these components is handled by the *Distributed Execution Framework* (DEFw), a lightweight remote procedure call (RPC) layer optimized for HPC-scale deployments.

Through this design, QFw supports a variety of quantum backends, including state-vector and tensor-network simulators. Users typically access QFw through a Python API module (`QFwBackend`) that provides a drop-in backend compatible with frameworks such as Qiskit [34] and PennyLane [35]. This interface translates circuit execution requests into QFw API calls, which are then dispatched to the selected backend. Simulator workloads are executed in distributed mode via MPI, while hardware requests are routed to the appropriate service interface. In both cases, results are returned in standardized formats, shielding applications from backend-specific parsing.

By combining HPC process management, distributed task scheduling, and backend-agnostic execution, QFw serves as a bridge between quantum and classical ecosystems [36]. It provides a unified foundation on which scalable, reproducible, and portable quantum workflows can be developed and deployed on leadership-class computing platforms.

2.2 Non-Variational Workloads

Non-variational workloads execute fixed circuits without a classical parameter-update loop and therefore expose the raw execution, memory, and communication characteristics of each backend. We consider (i) GHZ state preparation, which stresses long-range entanglement growth with shallow but highly correlated circuits [15];

(ii) Hamiltonian simulation of Ising-type models, including TFIM, which exercises trotterized time evolution and controlled two-qubit interactions with tunable depth [16, 17, 21]; and (iii) the HHL linear solver, which illustrates deeper coherent subroutines (e.g., phase estimation and controlled rotations) and ancilla management [18]. These kernels span distinct structures and depths, making them informative for contrasting state-vector and tensor-network simulators, including their distributed implementations [28, 30, 31]. Within QFw, identical circuits are issued through standard frontends and dispatched uniformly to multiple local simulators and the cloud backend, enabling portable and reproducible comparisons [34, 35].

2.3 Variational Workloads

Variational workloads couple parameterized quantum circuits with a classical optimizer, repeatedly preparing, measuring, and updating parameters to minimize a task-specific objective [23]. Our focus is on QAOA, whose layered cost–mixer ansatz provides a controllable depth-quality trade-off and well-studied scaling behavior on combinatorial problems [12]. In particular, our work focuses on finding novel meta-materials by formulating the optimization problem as a quadratic unconstrained binary optimization (QUBO).

Distributed Quantum Approximate Optimization Algorithm (DQAOA). The DQAOA is an extension of QAOA designed to address the hardware and scale limitations of near-term quantum computing systems. In this approach, a large combinatorial optimization problem is decomposed into smaller sub-problems, each requiring much fewer qubits and shallower circuits. These sub-problems can be executed simultaneously on multiple quantum simulators or hardware, where their results are combined to form an approximate global solution. Such decomposition mitigates two key constraints in current quantum systems: (1) limited qubit counts and (2) circuit depth restrictions due to noise.

DQAOA naturally lends itself to HPC integration, as many sub-problems can be solved in parallel. A typical workflow allocates classical and quantum resources, dispatches sub-problem circuits concurrently, and aggregates their outputs for classical post-processing. This parallel execution is well-suited for batch schedulers and distributed execution frameworks, enabling substantial reductions in wall time for large-scale problems.

Kim *et al.* [19] proposed DQAOA that coupled HPC resources with quantum backends, distributing sub-problem execution across heterogeneous compute nodes. Their study showed that this approach can handle problems with up to thousands of binary variables with high solution quality. Xu *et al.* [26] extended this model to a GPU-accelerated implementation using Qiskit Aer and MPI-based orchestration, achieving significant performance gains on leadership-class supercomputers.

These studies illustrate how DQAOA can combine quantum parallelism with classical HPC scalability. By exploiting distributed resources at both the algorithmic and simulation levels, DQAOA provides a pathway toward solving larger optimization instances than would be possible on quantum computing alone.

In this context, iterative workloads amplify orchestration demands (many circuit evaluations per optimization step) and therefore benefit directly from QFw's backend-agnostic scheduling and distributed execution model. Using a unified frontend (e.g., Qiskit or PennyLane) with QFw allows the same QAOA/DQAOA application code to target multiple simulators and a cloud backend without modification, supporting fair benchmarking and rapid platform substitution [34, 35].

3 Related Work

Early efforts to integrate quantum acceleration into classical HPC workflows have focused on bridging existing cluster tools with emerging quantum resources. For instance, Esposito and Haus [37] leverage SLURM's heterogeneous jobs feature and MPI to co-schedule hybrid classical-quantum workflows on supercomputers, aiming to eliminate idle time on scarce quantum processors. Their method requires refactoring a monolithic hybrid program into a sequence of smaller jobs, each encapsulating one quantum computation alongside its surrounding classical processing. This allows the quantum device to be released after each quantum step and reused by other jobs, significantly improving utilization. While effective, this approach demands moderate user intervention (splitting code and managing data hand-offs) and relies on the HPC scheduler to interleave tasks, without deeper automation or resource awareness beyond job queuing.

More automated scheduling frameworks have been proposed to manage quantum tasks in HPC environments. SCIM MILQ by Seitz *et al.* [38] is an HPC quantum scheduler that combines conventional job scheduling techniques with quantum-specific methods like circuit cutting. By partitioning large quantum circuits and intelligently scheduling the pieces, SCIM MILQ seeks to minimize overall makespan and noise impact when running multiple quantum tasks on limited hardware. This is a step toward specialized runtime systems that can optimize hybrid workloads, although such research prototypes have yet to be integrated into general-purpose HPC job managers. Similarly, XACC [39] provides a low-level software infrastructure for heterogeneous quantum-classical computing, exposing a service-oriented framework for quantum program compilation and execution across diverse hardware backends. XACC's hardware-agnostic interfaces lay the groundwork for tightly coupling quantum co-processors with classical code, but require adoption within HPC toolchains.

Another line of work emphasizes unified software stacks to hide quantum-classical complexity from end users. The Munich Quantum Software Stack (MQSS) [40] being developed by TUM and LRZ is a comprehensive platform that deeply integrates quantum devices into HPC centers. MQSS provides a single access point to multiple quantum backends at the LRZ supercomputing facility, accessible via a web portal, command line, or through hybrid jobs tied into the cluster scheduler [41]. The stack automates the workflow from user-submitted quantum programs to execution on available hardware, including dynamic compilation that adapts to device calibration data [42]. Our work shares a similar goal of enabling flexible backend selection and seamless quantum integration, but MQSS is tailored to a specific institutional environment.

Beyond HPC-centric solutions, the rise of cloud quantum services also provides relevant context. Platforms such as Amazon Braket [43], Microsoft Azure Quantum [44], and qBraid [45] aggregate a variety of quantum processors (superconducting, ion-trap,

etc.) from multiple vendors and offer unified interfaces for job submission. These services let users develop algorithms with familiar open-source frameworks (e.g., Qiskit, Cirq, PennyLane) and run them on different backends through a common API. For example, qBraid's environment enables one-click access to over two dozen quantum devices across providers, with cross-framework compatibility and centralized job tracking. Such cloud platforms demonstrate the value of backend flexibility and ease of use, though they operate in remote data centers rather than tightly coupled to local HPC resources. Conqure [46] combines an Amazon-compatible interface with a Qiskit front-end and Slurm-based scheduling coordinated with HPC in an open-source environment aimed at lab-hosted quantum devices and local or remote simulation with a private cloud presence.

In our work, we similarly support multiple quantum backends—combining both local HPC ones and a remote one under a single workflow. However, we differ by focusing on hybrid workflow performance and co-scheduling in an HPC context, rather than pure cloud access. All prior art has either required users to manually split and schedule quantum tasks or has been limited to specific ecosystems, with only small-scale simulation. Our approach strives to generalize hybrid quantum-classical scheduling with minimal user effort. It provides a unified framework that dynamically selects quantum backends and orchestrates their interaction with classical computations, thereby advancing the state of the art in flexible quantum workflow management.

4 Methodology

QFw addresses two complementary goals: (i) providing a clean, modular path to integrate new simulators and hardware backends, and (ii) enabling an application execution model that scales across HPC resources and hybrid cloud endpoints with minimal changes to user code.

QFw exposes a *frontend API* to applications and a *backend QPM API* to simulators or hardware. QPM coordinates a pool of worker processes, each bound to an HPC core, and dispatches batches of quantum circuit tasks either via PRTE/MPI for on-premise execution or through REST for cloud endpoints. Results are returned asynchronously to the frontend, ensuring non-blocking execution for iterative and variational workflows.

The end-to-end workflow is illustrated in Fig. 1. In **step-1**, a SLURM job is submitted with two heterogeneous groups: one for the application (hetgroup-0) and another for QFw-related management and simulator nodes (hetgroup-1). In **step-2**, the QFw setup procedure launches the core infrastructure, including the DEFw with a PRTE/DVM URI shared across all processes, multiple QPM processes for managing quantum circuit execution jobs and queues, and multiple QRC worker threads per QPM process for triggering MPI runs on the nodes allocated to QFw (hetgroup-1). Only the first node in this group houses QFw's management services, while all nodes, including the first, act as workers. In **step-3**, the user application, implemented in frameworks such as Qiskit or PennyLane, is executed.

If the application is distributed (**step-4A**), the programmer may employ MPI (e.g., via `mpi4py`) or use multithreading within the application's node group and must connect to the same backend API specified in **step-5**. If it is not distributed (**step-4B**), the QFwBackend is invoked directly, connecting to the QPM API in hetgroup-1 and preparing to submit circuits to any QPM accessible to the backend. In **step-5**, the QFwBackend, implemented as a Qiskit BackendV2 interface, routes application requests to the designated QPM service via RPC calls handled by DEFw. These calls, which include circuit creation, deletion, execution, and status queries, are processed in **step-6** by the appropriate QPM implementation depending on the application's configuration. Circuit execution (**step-7**) may occur on local HPC resources in hetgroup-1 or on remote QPUs via REST calls.

In **step-9**, execution results are marshaled into the common QPM API format before being forwarded for application-level post-processing (**step-10**). Depending on the application logic (**step-11**), e.g., whether additional iterations for a classical optimizer are required, control is either returned to the QFwBackend for continued execution or the application concludes (**step-12**). In **step-13**, QFw performs a controlled teardown in hetgroup-1, shutting down RPC services, releasing worker allocations, and freeing any remote QPU queue reservations (currently using QCUP's shared queue). Finally, the SLURM job terminates in **step-14**.

The next two subsections detail (1) how new simulators or hardware backends plug into QFw, and (2) how applications interface with the frontend API to submit batched circuits and retrieve asynchronous results.

4.1 Integrating Backends

Integrating a new backend into QFw requires implementing a *Backend QPM* that conforms to a predefined QPM-API. Consequently, each backend that is to be integrated with QFw as a new QPM should (1) accept a standardized circuit/problem description, (2) configure backend-specific runtime parameters, (3) launch execution via MPI/PRTE (on-prem) or REST (cloud), and (4) marshal results into QFw's unified return format. Hence, the frontend and backend are cleanly decoupled and the application code remains unchanged when swapping backends or retargeting sub-backends. QFw also unifies logging, error propagation, and timing instrumentation so QPM can maintain comparable per-backend performance profiles.

QFw currently supports five backends spanning state-vector and tensor-network simulation, plus a cloud QPU provider. Each integration reflects challenges in terms of robustness, scaling model, and ease of deployment on HPC systems:

- **TN-QVM (ORNL).** TN-QVM is a tensor-network simulator that uses the ExaTN library underneath. TN-QVM can target multiple topologies (MPS, TTN, PEPS). We wrote a thin wrapper to select the topology as a sub-backend. In QFw, we currently support and test *ExaTN-MPS*.
- **NWQ-Sim (PNNL).** The state-vector engine (SV-Sim) is fully integrated. Historically, SV-Sim offers several sub-backends (OpenMP, MPI, CPU, AMD GPU) that are supported and selectable at runtime. NWQ-Sim's native MPI distribution makes it a good fit for multi-node CPU/GPU HPC runs.
- **QTensor (ANL).** QTensor is a tree tensor-network approach built around the `qtree` library. Although QTensor is primarily designed for QAOA-style workloads (emphasizing

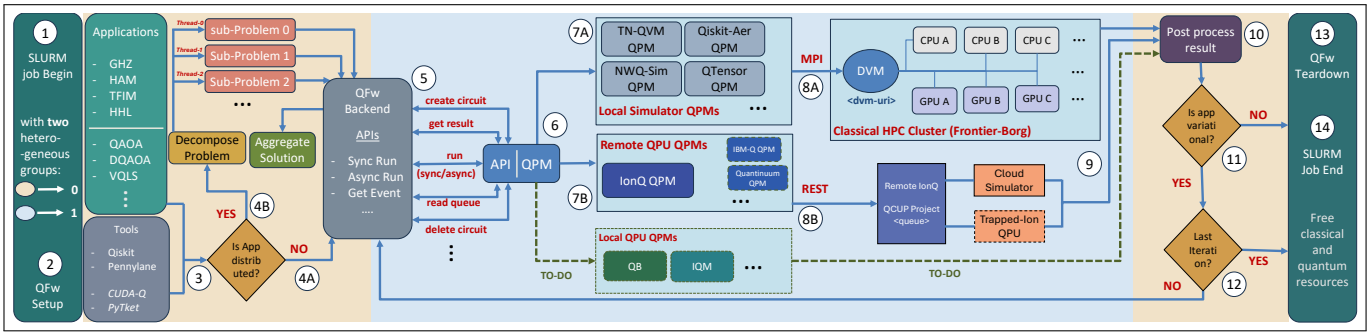


Figure 1: QFw architecture and execution flow. QPM manages worker pools, dispatches batches to local MPI ranks or cloud endpoints, and streams results back to applications. ***Note: Background shading corresponds to the heterogeneous groups as pointed out in*** `step-1`***, with the warmer tone indicating hetgroup-0 components and the cooler tone indicating hetgroup-1 components.***

expectation estimation on sparse QUBOs via light-cone style optimizations), QFw uses it for full-state contraction through `qtree`. QFw has been tested thoroughly with `numpy`. MPI distribution is enabled via `mpi4py`.

- **Qiskit Aer.** Aer remains a state-of-the-art single-node circuit simulator and provides several sub-backends: `statevector`, `matrix_product_state`, `stabilizer`, and `automatic` (which selects among others). MPI support exists via chunking across ranks. QFw has been exercised with *mps*, *statevector*, and *automatic*. GPU acceleration is available (CUDA by default); HIP/ROCm requires a custom Aer build for AMD GPUs.
- **IonQ (cloud).** For the cloud path, simple REST suffices. In practice, we leverage IonQ's Qiskit `BackendV2` plugin, which handles the REST plumbing but adds helpful boilerplate. This work extensively tested the *simulator* sub-backend.

Table 1 summarizes capabilities and caveats. In all cases, backends and sub-backends are selected via lightweight runtime properties (e.g., {"backend": "qtensor", "subbackend": "numpy"}), so users can shift between engines without refactoring their algorithms. The key contribution here is the ease of integrating new backends and combing components to this end. What's more, the same QFw backend object can be plugged into popular frameworks (e.g., Qiskit, PennyLane); swapping `backend`/`subbackend` toggles engines and device targets without changing the user's quantum program as illustrated in Figure 2.

4.2 Integrating Applications

QFw is deployed on HPC systems via SLURM's heterogeneous job allocation model. A typical launch reserves two resource groups: `hetgroup-0` for the application's classical control logic, and `hetgroup-1` for QFw services and backend execution. The QPM service is started on the lead node of `hetgroup-1`, instantiating a PRTE-DVM that spans all backend nodes. This separation ensures that backend execution is isolated from application orchestration overheads and allows for independent scaling of the classical and quantum-simulation components.

On the frontend, applications interact with QFw through a lightweight Python client, `QFWBackend`, which exposes a uniform

```
# Qiskit + QFw example
from qfw_qiskit import QFwBackend
backend = QFwBackend(
    properties = {
        "backend":"nwqsim",
        "subbackend":"MPI"
    }
)
# Pennylane + QFw example
import pennylane as qml
dev = qml.device('qiskit.remote', wires=2,
     backend=backend)
```

Figure 2: Selecting the QFwBackend.

`execute()` method. Applications generate circuits or problem instances using their preferred SDK (e.g., Qiskit) and pass them directly to QFw. Execution calls are translated into RPCs to QPM, which schedules them across worker pools.

For **non-variational workloads** (GHZ, HAM, TFIM, HHL), QFw batches independent circuit instances across available cores, maximizing throughput. For **variational workloads** (QAOA, DQAOA), the application issues multiple asynchronous calls within each optimization iteration, enabling parameter sweeps without idle time.

To enable scalable quantum optimization workflows, we extended QFw to **DQAOA applications**. The workflow (Fig. 1) begins with a dual-group heterogeneous SLURM allocation: one group dedicated to the DQAOA application and another to QFw services. In the QFw group, a unique DVM-URI is created and QPM services are launched. On the DQAOA side, a Qiskit-based Python application initializes with a QUBO matrix as input. During initialization, the QFwBackend establishes a secure connection to the QPM services.

The large QUBO is decomposed into multiple subQUBOs using either random partitioning or decomposition methods directed by an impact factor. The application uses Python's `threading` module to issue concurrent `solve` calls for each subQUBO, as the workload is primarily I/O-bound. For local HPC simulators, these calls trigger asynchronous RPCs to QFw, which dispatch the jobs via PRTE and launch MPI-parallel circuit executions across the allocated nodes.

Table 1: Backends used with QFw.

Backend	Sub-backend(s)	CPU	GPU	Native MPI	Notes
TN-QVM (ORNL)	exatn-mps TTN (pending) PEPS (planned)	Yes	Yes*	Yes*	Tensor-network simulator; QFw wrapper selects topology. QFw is tested with *ExaTN-MPS*. TTN currently blocked by `.xasm` vs. `.qasm`; PEPS is architecturally supported.
NWQ-Sim (SV-Sim)	OpenMP MPI CPU AMDGPU	Yes	Yes	Yes	Fully integrated. At the time of development, HIP+MPI lacked complete upstream support; other sub-backends are supported via QFw.
Qiskit Aer	statevector matrix_product_state automatic	Yes	Yes†	Yes	Strong single-node performance; MPI uses chunking. QFw tested with *mps*, *statevector*, and *automatic*. HIP/ROCm requires a custom Aer build.
QTensor (ANL)	numpy cupy (planned) pytorch (planned)	Yes	Yes	Yes	Tree TN (`qtree`). Designed for QAOA expectation estimation on sparse QUBOs, but used in QFw for full-state contraction; currently tested thoroughly with `numpy`. MPI via `mpi4py`.
IonQ (cloud)	simulator hardware (planned)	N/A			Integrated via IonQ's Qiskit `BackendV2` plugin (REST under the hood).

* Engine-dependent via ExaTN build options. † CUDA by default; HIP/ROCm requires a custom Aer build.

For remote quantum services such as the IonQ cloud simulator, QPM translates the request into REST API calls, managing job submission and result retrieval transparently.

As subQUBO results return, they are aggregated to update the classical optimizer, and the cycle repeats until convergence. Upon completion, QPM services are terminated, the DVM-URI is released, and SLURM resources are deallocated. This integration design unifies local MPI-based simulation and remote cloud backends under a single asynchronous workflow, enabling DQAOA to scale efficiently across heterogeneous resources while preserving full backend flexibility.

This modular integration model enables new backends and applications to be combined arbitrarily, supporting reproducible, large-scale quantum workloads without rewriting core application logic.

5 Experimental Setup

Compute Platform: The experiments are run on a Frontier test cluster at OLCF with 32 nodes. Each Frontier compute node contains one 64-core "Optimized 3rd Gen" AMD EPYC CPU with 512 GiB DDR4 memory, and four AMD Instinct MI250X GPUs. Each MI250X comprises two GCDs (8 logical GPUs per node), each with 64 GiB HBM2e. Nodes are connected via HPE Slingshot 200 with an aggregate node-injection bandwidth of 800 Gbit s^{-1}. Each node exposes eight last-level cache (LLC) domains, each serving eight CPU cores; to minimize OS noise, we reserve one core per LLC for kernel/system processes, leaving **56 application cores per node** available to QFw.

Job Orchestration, Builds, and Protocol: We use SLURM heterogeneous job groups: *hetgroup-0* hosts the application layer and *hetgroup-1* hosts Quantum Framework (QFw) simulation workers. The QPM (on the first node of *hetgroup-1*) spawns eight worker threads and distributes circuit-execution tasks round-robin, using PRTE across *hetgroup-1* for local simulators or REST API calls for IonQ cloud backends. All simulators/backends are built from their latest public releases with `-O3`; GPU builds are enabled when available (e.g., cuQuantum/GPU options for Aer). Each experiment is repeated three times (limited by allocation) for which we report the mean and standard deviation. Unless stated otherwise, one node is reserved for the application layer (*hetgroup-0*) while QFw scales on *hetgroup-1*.

Table 2: Benchmarks and problem sizes grouped by category. QAOA reports *QUBO size*. DQAOA reports *QUBO size* and subQUBO settings as *(subqsize*, nsubq†)*.

Non-variational	
Application	**#qubits**
SupermarQ GHZ	4, 8, 12, 16, 20, 24, 28, 30, 32
SupermarQ HAM	4, 8, 12, 16, 20, 24, 28, 30, 32
TFIM	4, 8, 12, 16, 20, 24, 28, 30, 32
HHL	5, 7, 9, 11, 13, 15, 17
Variational	
Application	**QUBO size**
QAOA	4, 8, 10, 20, 30
DQAOA	30 with *(subqsize*, nsubq†)*: (16,2), (8,4), (12,3)
	40 with *(subqsize*, nsubq†)*: (16,4), (12,4)

* subqsize refers to the sub-QUBO size. † nsubq refers to the number of sub-QUBOs.

Backends: We use the set of simulators and the IonQ cloud emulator (with hardware executions planned for future work) described in Section 4.1 and Table 1.

Benchmarks: We evaluate both non-variational and variational applications as described in Section 4.2; problem sizes are enumerated in Table 2.

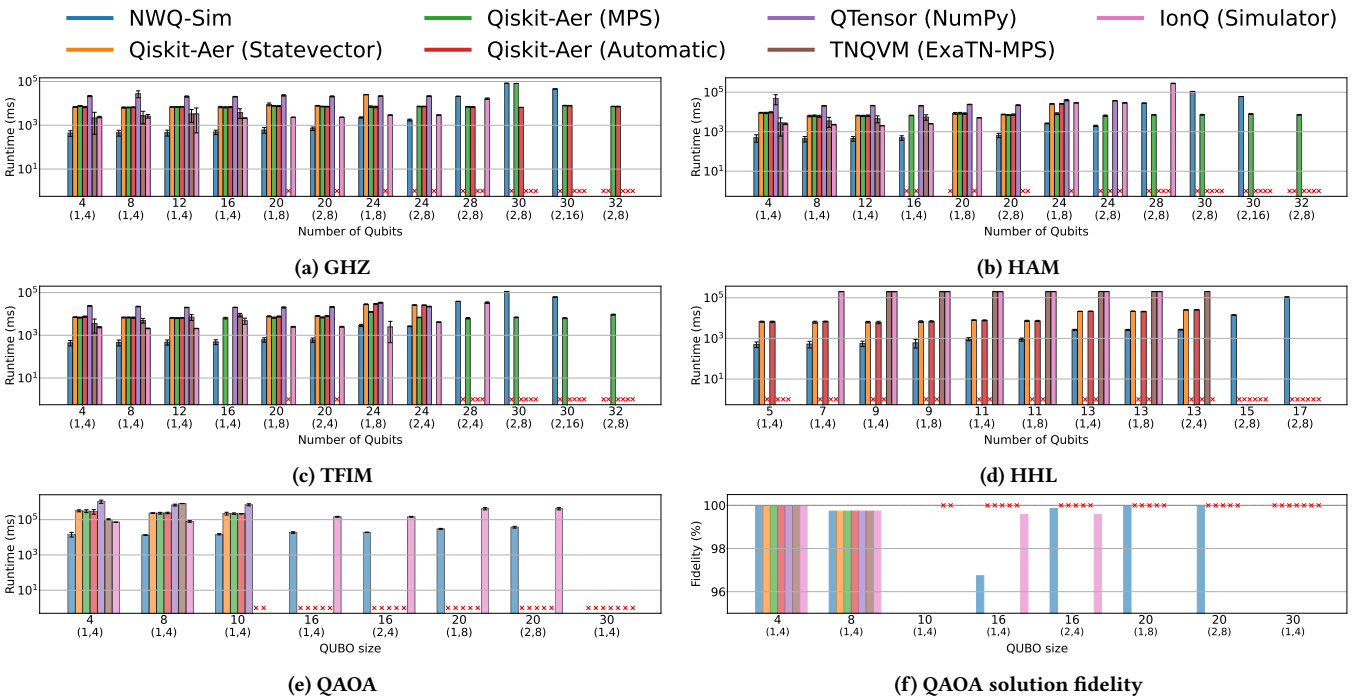


Figure 3: (a)-(e) Results per benchmark across backends. (f) QAOA solution fidelity (referenced to D-Wave's hybrid quantum annealing solver [47]). This figure represents an approximate weak scaling study where we increase both the number of qubits and the number of processes but not exactly proportionally. Note: The secondary x-axis in each panel denotes $(\#N, \#P)$, i.e., (num nodes, processes per node).

6 Results

We benchmark the integrated backends from Table 1 on the workloads and sizes listed in Table 2. Each point is the mean of three runs with error bars indicating standard deviation.

Non-variational workloads: Figures 3a–3d present runtime scaling for GHZ, HAM, TFIM, and HHL circuits, respectively. In GHZ (Fig. 3a) and HAM from SupermarQ (Fig. 3b), all backends scale to 32 qubits, but runtime separation increases with size: NWQ-Sim and Qiskit Aer (MPS) remain competitive, while QTensor slows notably beyond 24 qubits.

For TFIM (Fig. 3c), Qiskit Aer's *mps* solver sustains low runtimes up to 33 qubits, outperforming NWQ-Sim at large sizes. The TFIM-28 workload, when executed with varying process counts (an approximate strong scaling study, since the increase was not strictly proportional). We observe that state-vector-based simulators such as NWQ-Sim and Qiskit Aer exhibit improved performance with increased resources, whereas MPS-based approaches do not scale as effectively.

For HHL (Fig. 3d), increasing circuit depth reduces scalability. NWQ-Sim outperforms at smaller problem sizes, Qiskit Aer achieves comparable performance for medium instances, and NWQ-Sim again leads for larger instances, subject to the resource constraints indicated on the secondary x-axis.

Variational workloads: Fig. 3e and Fig. 3f show QAOA runtime and fidelity trends as a function of QUBO size. Runtimes increase with problem size, with sharper growth when scaling process counts (#P) beyond a single LLC domain due to MPI communication overheads. Missing points (red X) correspond to runs exceeding the two-hour cutoff. Fidelity remains consistently above 95% across tested sizes, with minor variation due to backend-specific numerical differences. A red X in the plots indicates a configuration omitted due to exceeding walltime or resource constraints on the Frontier test cluster, a scaled-down environment with limited processes per node. These constraints particularly impact backends such as Qiskit Aer, which, when run via `mpi4py`, does not benefit from usual multi-core optimizations due to not being natively designed for strong scaling beyond a single node.

DQAOA results with NWQ-Sim and IonQ backends are shown in Figs. 4 and 5. The former plot compares total execution times across different problem sizes and subproblem configurations, while the latter zoomed view highlights iteration-level timing patterns for a specific configuration (subqsize=12, nsubq=4). The zoomed plot clearly shows NWQ-Sim completing iterations faster (and about four concurrently) and with more uniform timing compared to the IonQ simulator which involves calls over the internet and cloud queues.

7 Discussion

Our evaluation highlights key factors influencing simulator performance, scalability, and QFw's role in enabling portable quantum workflows.

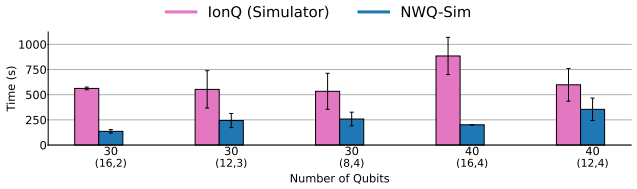


Figure 4: Solving DQAOA using QFw. *Note: The secondary x-axis refers to (subqsize, nsubq) as defined in Table 2.*

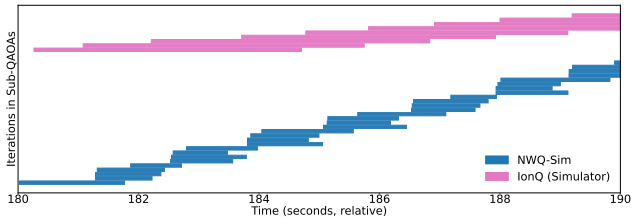


Figure 5: A zoomed portion of the DQAOA-40 run with subqsize=12 and nsubq=4.

No single backend dominates across all problems. Qiskit Aer's *mps* solver is highly efficient for structured, low-entanglement circuits such as TFIM, sustaining low runtimes even beyond 30 qubits. NWQ-Sim's state-vector kernels and MPI parallelization excel for GHZ and HAM workloads, while QTensor's TTN decomposition favors shallow or tree-like circuits but slows sharply on deeper or densely connected topologies.

DQAOA results demonstrate QFw's ability to execute many subQUBOs concurrently, overlapping RPC communication with backend computation. Moderate-sized subQUBOs achieve good scaling, but very small subQUBOs incur fixed overheads from RPC and scheduling, reducing efficiency. The same application code runs without modification across HPC simulators and cloud backends, confirming the effectiveness of QFw's asynchronous, backend-agnostic execution model.

System-level optimizations help reduce OS noise and improve reproducibility at scale. This includes LLC core reservation, SLURM heterogeneous job groups, and PRTE-based MPI dispatching. By fully decoupling application logic from backend orchestration, QFw enables fair, repeatable performance comparisons and rapid backend substitution.

The ability to seamlessly move between local MPI-based simulators and remote quantum hardware positions QFw as a practical platform for hybrid execution, large-scale algorithm tuning, and reproducibility studies in production HPC environments. This portability is critical as quantum computing moves toward integrated HPC-quantum deployments.

8 Conclusion

We integrated a diverse set of quantum circuit backends into QFw, making it a hybrid quantum-HPC execution framework that spans multiple local simulators (NWQ-Sim, Qiskit Aer, TN-QVM, QTensor) via PRTE/MPI and a remote IonQ service. This unified integration supports both non-variational and variational workloads, implemented in Qiskit and PennyLane, and deployable without code changes across heterogeneous HPC and cloud resources.

Our evaluation on GHZ, HAM, TFIM, and HHL circuits, as well as QAOA, and DQAOA, confirms that backend performance is highly workload-dependent: Tensor-network methods such as Qiskit Aer's *mps* excel for structured, low-entanglement problems, while state-vector engines like NWQ-Sim perform strongly for shallow but highly entangled workloads. For deeper circuits and large-scale variational algorithms, distributed MPI execution is essential for scaling, and asynchronous orchestration enables effective overlap of computation and communication.

By decoupling application logic from backend orchestration, QFw enables reproducible, cross-platform benchmarking and rapid backend substitution, which are critical capabilities for advancing scalable, portable hybrid quantum-HPC workflows. Future extensions will target real-hardware experimentation, GPU-accelerated tensor-network backends, automated workload-driven backend selection, and larger-scale hybrid HPC-cloud studies spanning multiple hardware targets, further informing backend design and deployment strategies.

AI Assistance

Portions of the text in this manuscript were refined with the assistance of the AI tool ChatGPT (OpenAI) [48] to improve clarity and readability.

Acknowledgments

This work was supported in part by NSF awards MPS-2531350, MPS-2410675, PHY-2325080, CISE-2316201, MPS-2120757, CCF-2217020, and PHY-1818914 as well as DOE DE-SC0025384. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. This material is based upon work supported by the U.S. Department of Energy, Office of Science, National Quantum Information Science Research Centers, Quantum Science Center. Notice: This manuscript has been co-authored by UT-Battelle LLC under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

References

[1] Frank Arute et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, October 2019. doi: 10.1038/s41586-019-1666-5.

[2] Seongmin Kim, Wenjie Shang, Seunghyun Moon, Trevor Pastega, Eungkyu Lee, and Tengfei Luo. High-performance transparent radiative cooler designed by quantum computing. *ACS Energy Letters*, 7(12):4134–4141, 2022.

[3] Ashley Montanaro. Quantum algorithms: An overview. *npj Quantum Information*, 2(1):15023, 2016.

[4] Seongmin Kim, Serang Jung, Alexandria Bobbitt, Eungkyu Lee, and Tengfei Luo. Wide-angle spectral filter for energy-saving windows designed by quantum annealing-enhanced active learning. *Cell Reports Physical Science*, 5(3), 2024.

[5] Chao Lu, Muralikrishnan Gopalakrishanan Meena, and Kalyana Chakravarthi Gottiparthi. Lugo: an enhanced quantum phase estimation implementation. *arXiv preprint arXiv:2503.15439*, 2025.

[6] Quantum error correction below the surface code threshold. *Nature*, 638(8052): 920–926, 2025.

[7] Jens Koch, Terri M Yu, Jay Gambetta, Andrew A Houck, David I Schuster, Johannes Majer, Alexandre Blais, Michel H Devoret, Steven M Girvin, and Robert J Schoelkopf. Charge-insensitive qubit design derived from the cooper pair box. *Physical Review A*, 76(4):042319, 2007.

[8] C. Monroe and J. Kim. Scaling the ion trap quantum processor. *Science*, 339(6124): 1164–1169, 2013. doi: 10.1126/science.1231298.

[9] Michael Freedman, Alexei Kitaev, Michael Larsen, and Zhenghan Wang. Topological quantum computation. *Bulletin of the American Mathematical Society*, 40 (1):31–38, 2003.

[10] John Preskill. Quantum computing in the nisq era and beyond. *Quantum*, 2:79, 2018. doi: 10.22331/q-2018-08-06-79.

[11] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. A quantum approximate optimization algorithm, 2014. URL https://arxiv.org/abs/1411.4028.

[12] Edward Farhi, Jeffrey Goldstone, Sam Gutmann, and Leo Zhou. The quantum approximate optimization algorithm and the sherrington-kirkpatrick model at infinite size. *Quantum*, 6:759, July 2022. ISSN 2521-327X. doi: 10.22331/q-2022-07-07-759. URL http://dx.doi.org/10.22331/q-2022-07-07-759.

[13] Carlos Bravo-Prieto, Ryan LaRose, M. Cerezo, Yigit Subasi, Lukasz Cincio, and Patrick J. Coles. Variational quantum linear solver. *Quantum*, 7:1188, November 2023. ISSN 2521-327X. doi: 10.22331/q-2023-11-22-1188. URL http://dx.doi.org/10.22331/q-2023-11-22-1188.

[14] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J Love, Alán Aspuru-Guzik, and Jeremy L O'brien. A variational eigenvalue solver on a photonic quantum processor. *Nature communications*, 5(1):4213, 2014.

[15] Daniel M. Greenberger, Michael A. Horne, and Anton Zeilinger. Going beyond bell's theorem, 2007. URL https://arxiv.org/abs/0712.0921.

[16] Ernst Ising. Beitrag zur theorie des ferromagnetismus. *Zeitschrift für Physik*, 31 (1):253–258, 1925. doi: 10.1007/BF02980577.

[17] Eduardo Fradkin. *Field Theories of Condensed Matter Physics*. Cambridge University Press, Cambridge, 2 edition, 2013. ISBN 9781139015509. doi: 10.1017/CBO9781139015509.

[18] Aram W. Harrow, Avinatan Hassidim, and Seth Lloyd. Quantum algorithm for linear systems of equations. *Physical Review Letters*, 103(15), October 2009. ISSN 1079-7114. doi: 10.1103/physrevlett.103.150502. URL http://dx.doi.org/10.1103/PhysRevLett.103.150502.

[19] Seongmin Kim, Tengfei Luo, Eunji Lee, and In-Saeng Suh. Distributed quantum approximate optimization algorithm on integrated high-performance computing and quantum computing systems for large-scale optimization, 2024. URL https://arxiv.org/abs/2407.20212.

[20] Seongmin Kim and In-Saeng Suh. Distributed variational quantum algorithm with many-qubit for optimization challenges. *arXiv preprint arXiv:2503.00221*, 2025.

[21] Guang Hao Low and Isaac L. Chuang. Hamiltonian simulation by qubitization. *Quantum*, 3:163, jul 2019. doi: 10.22331/q-2019-07-12-163. URL https://doi.org/10.22331%2Fq-2019-07-12-163.

[22] A. V. Uvarov, A. S. Kardashin, and J. D. Biamonte. Machine learning phase transitions with a quantum processor. *Physical Review A*, 102(1), jul 2020. doi: 10.1103/physreva.102.012415. URL https://doi.org/10.1103%2Fphysreva.102.012415.

[23] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J. Love, Alán Aspuru-Guzik, and Jeremy L. O'Brien. A variational eigenvalue solver on a photonic quantum processor. *Nature Communications*, 5(1), jul 2014. doi: 10.1038/ncomms5213. URL https://doi.org/10.1038%2Fncomms5213.

[24] Eric R. Anschuetz, Jonathan P. Olson, Alán Aspuru-Guzik, and Yudong Cao. Variational quantum factoring, 2018.

[25] Srikar Chundury, Amir Shehata, Thomas Naughton III, Seongmin Kim, Frank Mueller, and In-Saeng Suh. QFw: A quantum framework for large-scale hpc ecosystems. Oak Ridge National Laboratory (ORNL), Oak Ridge, TN (United States), 11 2024. URL https://www.osti.gov/biblio/2498439.

[26] Zhihao Xu, Srikar Chundury, Seongmin Kim, Amir Shehata, Xinyi Li, Ang Li, Tengfei Luo, Frank Mueller, and In-Saeng Suh. Gpu-accelerated distributed qaoa on large-scale hpc ecosystems, 2025. URL https://arxiv.org/abs/2506.10531.

[27] Amir Shehata, Thomas Naughton, and In-Saeng Suh. A framework for integrating quantum simulation and high performance computing, 2024. URL https://arxiv.org/abs/2408.08098.

[28] Ang Li, Bo Fang, Christopher Granade, Guen Prawiroatmodjo, Bettina Hein, Martin Rotteler, and Sriram Krishnamoorthy. Sv-sim: Scalable pgas-based state vector simulation of quantum circuits. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021.

[29] Qiskit contributors. Qiskit: An open-source framework for quantum computing, 2023.

[30] Alexander J. McCaskey. Tensor Network Quantum Virtual Machine (TNQVM), 11 2016. URL https://www.osti.gov//servlets/purl/1340180.

[31] Dan Lykov, Ang Chen, Hui Chen, Kevin Keipert, Zixuan Zhang, Thomas Gibbs, and Yuri Alexeev. Performance evaluation and acceleration of the qtensor quantum circuit simulator on gpus. 2021. URL http://dx.doi.org/10.1109/QCS54837.2021.00007.

[32] IonQ Quantum Cloud. Jobs page. https://cloud.ionq.com/jobs, 2025. Accessed: 2025-08-11.

[33] Scott Atchley, Christopher Zimmer, John Lange, David Bernholdt, Veronica Melesse Vergara, Thomas Beck, Michael Brim, Reuben Budiardja, Sunita Chandrasekaran, Markus Eisenbach, Thomas Evans, Matthew Ezell, Nicholas Frontiere, Antigoni Georgiadou, Joe Glenski, Philipp Grete, Steven Hamilton, John Holmen, Axel Huebl, Daniel Jacobson, Wayne Joubert, Kim Mcmahon, Elia Merzari, Stan Moore, Andrew Myers, Stephen Nichols, Sarp Oral, Thomas Papatheodore, Danny Perez, David M. Rogers, Evan Schneider, Jean-Luc Vay, and P. K. Yeung. Frontier: Exploring exascale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701092. doi: 10.1145/3581784.3607089. URL https://doi.org/10.1145/3581784.3607089.

[34] Ali Javadi-Abhari, Matthew Treinish, Kevin Krsulich, Christopher J Wood, Jake Lishman, Julien Gacon, Simon Martiel, Paul D Nation, Lev S Bishop, Andrew W Cross, et al. Quantum computing with qiskit. *arXiv preprint arXiv:2405.08810*, 2024.

[35] Ville Bergholm, Josh Izaac, Maria Schuld, Christian Gogolin, Shahnawaz Ahmed, Vishnu Ajith, M Sohaib Alam, Guillermo Alonso-Linaje, B AkashNarayanan, Ali Asadi, et al. Pennylane: Automatic differentiation of hybrid quantum-classical computations. *arXiv preprint arXiv:1811.04968*, 2018.

[36] Amir Shehata, Pawel Groszkowski, Thomas Naughton III, Muthian Gopalakrishnan Meena, Eric Y. Wong, Daniel Chaves Claudino, Rafael Ferreira Da Silva, and Theodore L. Beck. Bridging paradigms: Designing for hpc-quantum convergence. *Future Generation Computer Systems*, 174:107980, 2025. URL https://www.sciencedirect.com/science/article/pii/S0167739X25002754.

[37] Aniello Esposito and Utz-Uwe Haus. Slurm heterogeneous jobs for hybrid classical-quantum workflows, 2025. URL https://arxiv.org/abs/2506.03846.

[38] Philipp Seitz, Manuel Geiger, Christian Ufrecht, Axel Plinge, Christopher Mutschler, Daniel D. Scherer, and Christian B. Mendl. Scim milq: An hpc quantum scheduler. In *2024 IEEE International Conference on Quantum Computing and Engineering (QCE)*, page 292–298. IEEE, September 2024. doi: 10.1109/qce60285.2024.10294. URL http://dx.doi.org/10.1109/QCE60285.2024.10294.

[39] Alexander J. McCaskey, Dmitry I. Lyakh, Eugene F. Dumitrescu, Sarah S. Powers, and Travis S. Humble. Xacc: A system-level software infrastructure for heterogeneous quantum-classical computing, 2019. URL https://arxiv.org/abs/1911.02452.

[40] Munich Quantum Valley. Q-dessi: Quantum design environment for software, systems, and integration. https://www.munich-quantum-valley.de/research/consortia/q-dessi, 2024. Accessed: 2024-06-13.

[41] Q-Exa Consortium. Q-exa: Quantencomputer-demonstrationsaufbauten. https://www.quantentechnologien.de/forschung/foerderung/quantencomputer-demonstrationsaufbauten/q-exa.html, 2024. Accessed: 2024-06-13.

[42] Robert Wille, Lucas Berent, Tobias Forster, Jagatheesan Kunasaikaran, Kevin Mato, Tom Peham, Nils Quetschlich, Damian Rovara, Aaron Sander, Ludwig Schmid, Daniel Schoenberger, Yannick Stade, and Lukas Burgholzer. The MQT handbook: A summary of design automation tools and software for quantum computing. In *IEEE International Conference on Quantum Software (QSW)*. doi: 10.1109/QSW62656.2024.00013.

[43] Amazon Web Services. Amazon Braket, 2020. URL https://aws.amazon.com/braket/.

[44] Microsoft. Azure quantum. https://azure.microsoft.com/en-us/products/quantum, 2025. Accessed: 2025-08-11.

[45] qBraid. qbraid: Quantum computing platform. https://www.qbraid.com, 2024. Accessed: 2024-06-13.

[46] Atulya Mahesh, Swastik Mittal, and Frank Mueller. CONQURE: A co-execution environment for quantum and classical resources. In *International Workshop on Integrating High-Performance and Quantum Computing*, October 2025. URL https://arxiv.org/abs/2505.02241.

[47] Seongmin Kim, Sang-Woo Ahn, In-Saeng Suh, Alexander W Dowling, Eungkyu Lee, and Tengfei Luo. Quantum annealing for combinatorial optimization: a benchmarking study. *npj Quantum Information*, 11(1):77, 2025.

[48] OpenAI. Chatgpt, 2025. Large language model. Available at https://chat.openai.com.