

Qonductor: A Cloud Orchestrator for Quantum Computing

Emmanouil Giortamis
Technical University of Munich
Munich, Germany
emmanouil.giortamis@tum.de

Francisco Romão
Technical University of Munich
Munich, Germany
francisco.romao@tum.de

Nathaniel Tornow
Technical University of Munich
Munich, Germany
nathaniel.tornow@tum.de

Dmitry Lugovoy
Technical University of Munich
Munich, Germany
dmit.lugovoy@tum.de

Pramod Bhatotia
Technical University of Munich
Munich, Germany
pramod.bhatotia@tum.de

Abstract

We describe Qonductor, a cloud orchestrator for hybrid quantum-classical applications that run on heterogeneous hybrid resources. Qonductor abstracts away the complexity of hybrid programming and resource management by exposing the *Qonductor API*, a high-level and hardware-agnostic API. The *resource estimator* strategically balances quantum and classical resources to mitigate resource contention and the effects of hardware noise. The *hybrid scheduler* automates job scheduling on hybrid resources and balances the tradeoff between users' objectives of QoS and the cloud operator's objective of resource efficiency.

We implement an open-source prototype and evaluate Qonductor using more than 7000 real quantum runs on the IBM quantum cloud to simulate real cloud workloads. Qonductor achieves up to 54% lower job completion times (JCTs) while sacrificing 3% execution quality, balances the load across QPU, which increases quantum resource utilization by up to 66%, and scales with growing system sizes and loads.

CCS Concepts

• **Hardware** → **Quantum computation**; • **Software and its engineering** → **Cloud computing**.

Keywords

Quantum Computing, Quantum Software, Hybrid Quantum-Classical, Quantum Cloud Orchestration

ACM Reference Format:

Emmanouil Giortamis, Francisco Romão, Nathaniel Tornow, Dmitry Lugovoy, and Pramod Bhatotia. 2025. Qonductor: A Cloud Orchestrator for Quantum Computing. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25)*, November 16–21, 2025, St Louis, MO, USA. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3712285.3759785>

1 Introduction

Quantum computing offers the potential to solve computational problems beyond the capabilities of classical computers by leveraging the principles of quantum mechanics [20, 30, 41, 81]. Quantum

computing is realized in the form of Quantum Processing Units (QPUs) [26], which are characterized by inherent noise and small qubit counts [74] and are now offered by all major cloud providers in a quantum-as-a-service fashion [2, 3, 5, 7].

However, since QPUs are not general-purpose processors, the quantum programming and execution models are *hybrid*, consisting of classical and quantum code. For instance, quantum applications can use classical pre- and post-processing steps to mitigate or correct hardware noise errors [32, 86, 89]. These steps often leverage classical accelerators such as GPUs [85] or FPGAs [58] for improved performance.

On top of that, the quantum cloud landscape is characterized by resource contention. Specifically, due to manufacturing and operational requirements [51], QPUs are an expensive and scarce resource, with less than 100 QPUs available *globally* by all cloud vendors combined [2, 3, 5, 7], with the largest provider, IBM, typically offering fewer than ten online at any given time [7]. In contrast, demand is constantly increasing, with IBM recently celebrating *three trillion* program executions on their platform [12].

Despite this scarcity, QPUs are vastly heterogeneous since there is an abundance of quantum technologies, architectures, and models, each presenting different tradeoffs between performance metrics, manufacturing complexity, and operational requirements [43]. More importantly, heterogeneity extends to QPU performance, with same-model QPUs experiencing significant performance differences (we detail this in §3), and this performance changes over time unpredictably [69, 77].

Naturally, QPU heterogeneity and scarcity drive users to select the highest-fidelity QPUs available, inevitably leading to QPU load imbalance, where best-performing QPUs become hotspots while the rest are underutilized [77, 78, 95]. Consequently, there is an inherent conflict between achieving high fidelity and maintaining low job completion times (JCTs), as users ideally desire both but must often compromise on one, typically sacrificing JCTs.

To summarize, quantum application development and orchestration are characterized by hybrid workflows and resources, scarce and vastly heterogeneous QPUs, and fundamentally conflicting objectives, posing three critical challenges.

First, hybrid programming and execution models are required. The standard practice for developing hybrid applications is through tedious and *manual* composition of classical and quantum tasks into workflows with virtually no standardization. Users navigate a largely heterogeneous landscape unguided to *manually* select the resources required to execute their workflows, amplifying



This work is licensed under a Creative Commons Attribution 4.0 International License. SC '25, St Louis, MO, USA

© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1466-5/2025/11
<https://doi.org/10.1145/3712285.3759785>

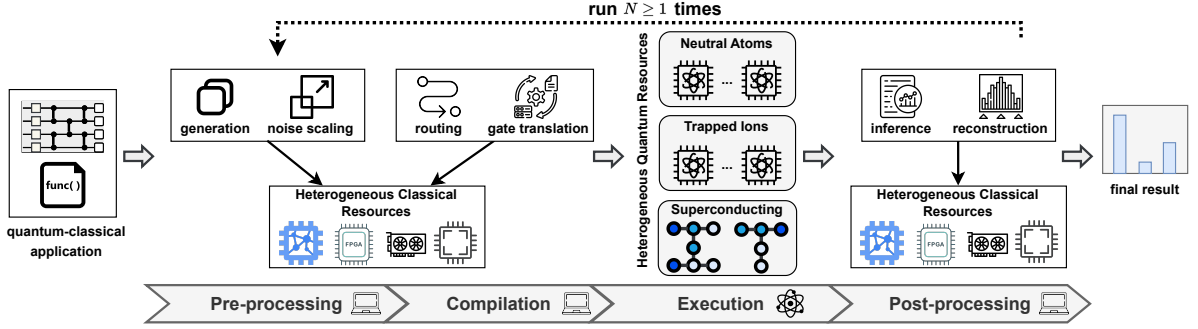


Figure 1: Quantum cloud and hybrid computational model (§ 2.2). *Quantum applications are hybrid, i.e., require quantum and classical resources. The pre-processing, compilation, and post-processing steps run on classical heterogeneous accelerators. QPUs are vastly heterogeneous across space and time, i.e., QPU technologies, architectures, and calibration data.*

QPU load imbalance [40, 77, 95]. **Second, quantum performance estimation depends on classical resources.** While research and industry efforts are constantly developing classical error mitigation techniques [32, 86, 89], their impact on quantum performance metrics (i.e., fidelity and execution time) is typically not included in the cloud scheduling decisions [78, 84, 95]. **Third, the conflicting objectives of the quantum cloud require multi-objective optimization.** Designing a system for the growing cloud that doesn't operate at the extremes of the fidelity-JCT tradeoff, i.e., does not simply choose the least busy or the highest-fidelity machine [15, 78], requires scalable multi-objective scheduling algorithms.

Thus, we pose the following research question: *How to design a scalable hybrid quantum-classical orchestrator that balances the conflicting objectives of the hybrid cloud?*

To answer this question, we introduce Qonductor, a cloud orchestrator for deploying hybrid applications on hybrid and heterogeneous clusters. **First**, the *Qonductor APIs* abstract away the complexity of hybrid application development and execution using hybrid resources. **Second**, our *resource estimator* systematically explores hybrid resource configurations that reduce resource contention while increasing execution quality (or fidelity). **Third**, our *hybrid scheduler* balances the tradeoff between fidelity vs. job completion times (JCTs).

We implement an open-source prototype of Qonductor in Python [91] and Go [19] by building on top of Kubernetes' scheduler, key-value store, and custom resource definitions to support QPUs [9], and the resource estimator based on the Qiskit framework [1], the scikit-learn ML library [70], and the pymoo optimization library [23].

To evaluate Qonductor's effectiveness, we first analyze real quantum cloud load conditions, construct a simulation environment resembling this workload, and evaluate Qonductor in this environment using more than 70,000 benchmark circuits. Our results show that Qonductor: (1) achieves up to 54% lower JCTs for a ~ 3% fidelity penalty on average, (2) evenly balances the load across QPUs and achieves 66% higher QPU utilization, (3) accurately estimates fidelities and runtimes in at least ~ 75% of the times, (4) scales linearly with an increasing cluster size and up to 3× the current quantum cloud load.

Contributions. We make the following contributions:

- (1) **Exposing hybrid cloud tradeoffs:** We demonstrate that quantum performance can be increased using much cheaper classical

resources and that users can experience vastly lower waiting times for minimal fidelity penalties.

- (2) **Hardware-agnostic programming model:** We introduce a hardware-agnostic API that simplifies programming hybrid applications and abstracts the underlying heterogeneous resources away.
- (3) **Hybrid resource estimation:** We introduce hybrid quantum-classical resource estimation, the first systematic hardware-aware estimation of fidelity, runtime, and cost (\$) when involving heterogeneous hybrid resources.
- (4) **Hybrid scheduler:** We propose the first hybrid scheduler that balances the tradeoff between the conflicting objectives of fidelity vs. JCTs by employing Pareto-optimal multi-objective optimization techniques.

2 Background

2.1 Quantum Computing Basics

Noisy quantum hardware. Modern quantum hardware, classified as noisy intermediate-scale quantum (NISQ) devices [74], operates with tens to a few hundred qubits [7] and is affected by a variety of error channels. Quantum operations deviate from their ideal unitary evolution due to stochastic Pauli errors, decoherence-induced amplitude damping and phase damping, and control inaccuracies that lower gate fidelities [18]. Moreover, idle qubits experience state degradation through T1 relaxation and T2 dephasing [50], while unwanted qubit-qubit interactions induce correlated errors via crosstalk [28]. The probabilities of these errors are characterized during periodic calibration procedures [88], which yield comprehensive datasets—detailing parameters such as T1, T2, and gate fidelities—that are publicly available [7, 17] but can fluctuate unpredictably between calibration cycles.

Quantum performance metric. To evaluate the quality of circuit execution on NISQ devices, we use the *Hellinger fidelity* metric [14], which quantifies the similarity between the noisy probability distribution obtained from the actual device and the ideal distribution that would be produced by noiseless, perfect hardware. Fidelity ranges from 0 to 1, with higher values indicating better quality results.

Quantum error mitigation. A suite of techniques has been developed to reduce the impact of noise on quantum computations

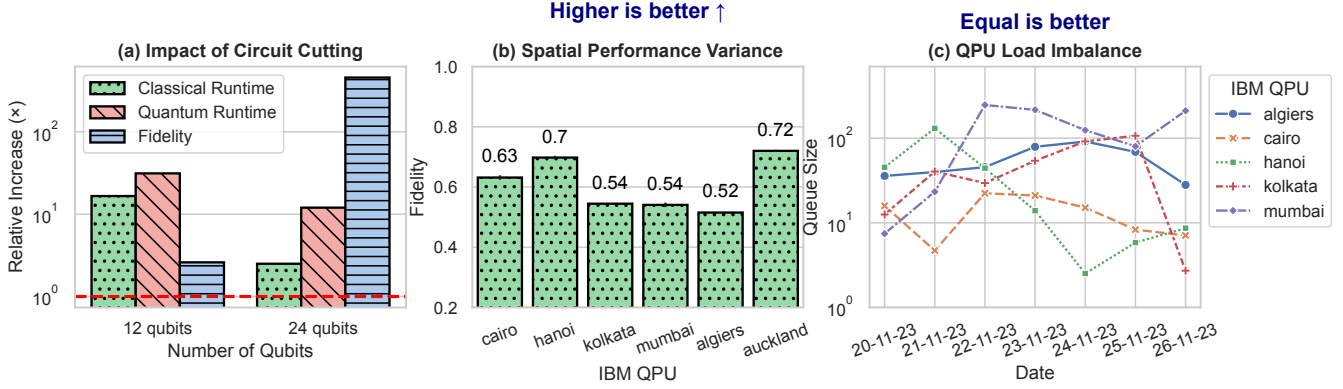


Figure 2: Quantum orchestration challenges (§ 3). (a) Impact of circuit cutting as a relative increase in execution fidelity, quantum, and classical runtime for 12-qubit and 24-qubit circuits. (b) Spatial performance variance: fidelity of a 12-qubit GHZ circuit on different IBM QPUs. There is a 38% fidelity difference from best to worst QPU. (c) QPU load imbalance: number of pending jobs on different IBM QPUs. There is up to $\sim 100\times$ load difference across QPUs.

without the need for full fault tolerance. These methods generally adhere to a three-stage workflow: (1) optimization and/or generation of circuit(s), (2) execution on noisy quantum hardware, and (3) post-processing to reconstruct a less noisy result. Among these methods, Zero-Noise Extrapolation (ZNE) infers the zero-noise limit by running circuits at different noise levels [56, 87]; Probabilistic Error Cancellation (PEC) models noise inverses via probabilistic resampling [36, 87]; Dynamical Decoupling (DD) applies pulse sequences to suppress decoherence [31, 73]; Readout Error Mitigation (REM) corrects measurement errors [24], and Pauli Twirling converts general noise into stochastic Pauli noise for easier correction [93].

2.2 Quantum Cloud Computing

Quantum cloud and scarcity of quantum resources. Major cloud providers such as IBM, Microsoft Azure, Google Cloud, and AWS currently offer access to quantum processing units (QPUs) [2, 3, 5, 7]. Globally, access is offered to fewer than 100 QPUs, while IBM alone recently celebrated three *three trillion* circuit executions on their cloud [12]. Notably, the gap between the growing demand for quantum resources and the current supply cannot be closed by building more QPUs, which are expensive to build and operate.

Hybrid computational model. In practice, quantum applications are inherently hybrid, relying on both quantum and classical computing. This is because QPUs are not standalone processors: classical computers are needed to compile quantum programs [16, 49] and to handle noise mitigation or error correction (§ 2.1). The typical workflow and resources required for a quantum application are shown in Figure 1. First, the quantum circuit is pre-processed to prepare error mitigation techniques. Then, the circuit is compiled to a target QPU to match the QPU’s constraints, e.g., basis gate set, and the circuit is executed on one or more QPUs. Lastly, the execution results are typically post-processed to reconstruct the less noisy result, typically through inference [56, 87].

Heterogeneous hybrid cloud resources. As shown in Figure 1, the classical processes include CPUs and specialized accelerators (xPUs, FPGAs, etc.). For instance, GPUs and Tensor Processing Units

(TPUs) can be used for circuit knitting [85, 90], while FPGAs are used for qubit readout classification [58].

At the same time, the quantum cluster is heterogeneous in three dimensions: (1) There exist multiple QPU technologies, such as superconducting [20], trapped ions [27], and neutral atoms [45]. These technologies involve trade-offs between performance metrics, manufacturing complexity, and operational requirements [43]. (2) Different architectures of same-technology QPUs vary in qubit topologies, basis gate sets, and noise models. (3) In fact, QPUs even of the *same* model have different noise models, which vary across calibration cycles, leading to spatiotemporal performance variance [40, 69, 77], as we detail in § 3.

3 Why is Hybrid Orchestration Challenging?

Managing the quantum cloud faces distinct challenges compared to classical orchestration and resource management: primitive programming and execution models, lack of hybrid resource estimation, and conflicting optimization objectives.

#1: Primitive programming and execution models. Developing and running quantum applications today involves hybrid quantum-classical code and resources (§ 2.2). Unfortunately, the prevailing programming model remains primitive: developers must *manually* stitch together classical and quantum logic—typically in Python—and *explicitly* manage low-level execution details. This includes selecting quantum devices, configuring classical control logic, and managing execution workflows. Such manual composition introduces unnecessary complexity, increases the chance of user error, and typically is tightly coupled to specific backends.

Key idea #1: We need hardware-agnostic APIs to enable transparent development and execution of hybrid workflows on heterogeneous hybrid resources.

#2: Hybrid resource estimation. Quantum execution fidelity is closely tied to classical compilation and pre- and post-processing steps, which often introduce additional runtime overhead to improve fidelity. This is shown in Figure 2 (a), where we use the circuit knitting error mitigation technique [60, 89] to cut 12-qubit and 24-qubit circuits in half and execute them sequentially on the same QPU.

Table 1: IBM Cloud Pricing.

Resource Type	Price/Task	Price/Hour
Standard VM	< 1\$	1-5\$
High-end VM	1-10\$	10-40\$
QPU	30-200\$	3000-6000\$

In the 24-qubit case, although the average classical and quantum runtimes increase by 2.5× and 12×, respectively, the average fidelity increases by ~450×.

Notably, classical resources are significantly cheaper and more accessible; e.g., while the IBM Cloud offers less than 20 QPUs, it offers thousands of classical servers [6]. Table 1 shows the price (in \$) per classical/quantum task/hour for different resource types. Standard VMs comprise 4-32 vCPUs and 16-64 GB RAM, while high-end VMs comprise 64+ vCPUs and up to 6 TB RAM. Notably, even the high-end VM-hours cost two orders of magnitude less than QPU-hours.

Key idea #2: We can increase quantum execution fidelity by leveraging error mitigation techniques (§ 2.1), which require using the widely cheaper and more abundant classical resources.

#3: Quantum cloud design trade-offs. The quantum cloud is characterized by conflicting objectives between the users’ Quality-of-Service requirements (high fidelity and low JCTs) and the cloud operator’s requirements (resource efficiency), caused by QPU spatiotemporal heterogeneity and the scarcity of QPUs.

First, QPU noise characteristics differ significantly across space and time, in contrast to classical processors. Specifically, execution fidelity can fluctuate across different QPUs and different calibration cycles [40, 77, 83, 95] as shown in Figure 2 (b), where we run a 12-qubit GHZ circuit on six IBM 27-qubit QPUs on 08-11-23. Fidelity varies across them, with up to 38% higher fidelity in *auckland* than *algiers*.

This performance variance naturally motivates users to select the highest-fidelity QPUs. Figure 2 (c) shows the number of pending jobs for every QPU and every day of a week in November 2023. QPUs face up to two orders of magnitude load difference, e.g., on 26-11-23, *mumbai* faces ~100× more pending jobs than *kolkata*.

Evidently, there is an inherent fidelity-JCT tradeoff. Ideally, users want the fidelity of the *highest-fidelity* QPU with the waiting time of the *least-busy* QPU. However, to maximize fidelity, all incoming jobs must be scheduled on the highest-fidelity QPU(s), forming hotspots, increasing the average JCTs, and decreasing average QPU utilization. Conversely, to minimize JCTs (and increase utilization), the jobs must be evenly distributed across all QPUs, decreasing average fidelity.

Key idea #3: We trade *minimal* fidelity penalties for *significant* JCT reduction. Combined with error mitigation (key idea #2), the fidelity loss can even be compensated.

4 Overview

We propose Qonductor, a scalable cloud orchestrator for developing and deploying hybrid applications on heterogeneous resources. Our system design is based on the key ideas presented in § 3 to address the challenges of quantum orchestration. The system comprises the *data plane*, used to deploy, invoke, and store hybrid workflow images; the *control plane*, which manages worker nodes and performs hybrid resource estimation and scheduling; *worker nodes* that manage the underlying classical accelerators and QPUs, and the *system monitor* that persists Qonductor’s state.

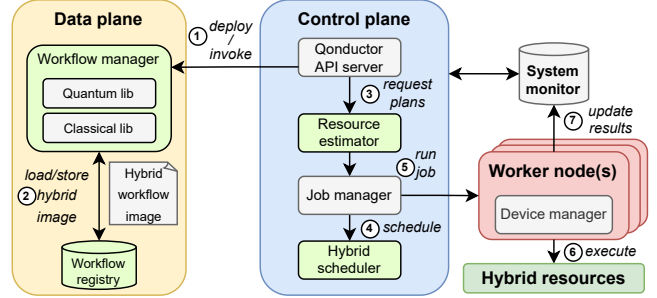


Figure 3: Qonductor overview (§ 4). Qonductor comprises the control plane, data plane, worker node(s), and the system monitor. Core components are highlighted as light green boxes. The control plane performs resource estimation, job management, and hybrid scheduling. The data plane is used to deploy and invoke hybrid images. Workers manage hybrid resources.

4.1 Qonductor Architecture

We detail the architecture and core components of Qonductor, as shown at a high level in Figure 3.

Data plane. The data plane provides functionality for configurable, programmable, and reusable hybrid application development and execution. To achieve this, we implement the *workflow manager* that offers libraries of commonly used quantum algorithms (e.g., QAOA [37]) and classical functions (e.g., error mitigation [53]). To minimize repetitive and manual hybrid application development, the workflow manager packages hybrid applications and user configuration (e.g., accelerator or QPU preferences) into *hybrid workflow images* that are persisted in the *workflow registry*. This enables users to reuse existing hybrid workflows out of the box with minimal effort and distribute them. We elaborate on the Qonductor programming model and the data plane’s components in § 5.

Control plane. The control plane is the core component of Qonductor and is responsible for managing hybrid workflow execution while achieving resource efficiency and improving quality of service by leveraging hybrid resource estimation and scheduling. In more detail, the *Qonductor API server* is the interface between the users and Qonductor. When users invoke hybrid workflow images, the API server calls the *resource estimator* to request resource plans for the users. Given the generated resource plan(s), the *job manager* iterates over the workflow’s jobs and, for each job, requests a resource allocation from the *hybrid scheduler*. The scheduler allocates resources while balancing fidelity and JCTs, guided by the resource plan and user’s execution configuration. Lastly, the job manager runs each workflow job on the worker node(s) assigned by the scheduler. We elaborate on the resource estimator in § 6 and the hybrid scheduler in § 7.

Worker nodes. The worker nodes serve two roles: (1) Execute jobs on their underlying devices (classical accelerators or QPUs) and (2) monitor and update the device status. For the first role, the *device manager* spawns containers that run the job on the node. For the second, the device manager periodically queries the classical nodes and QPUs to get static (e.g., number of cores/qubits) and the current dynamic information (e.g., queue sizes, utilization, calibration data),

Table 2: Qonductor programming API. CP and DP stand for control and data plane, respectively.

Operation	Caller	Callee
Create a workflow with hybrid code.	User	CP
Deploy a workflow.	User	CP
Invoke a workflow.	User	CP
Get the workflow results.	User	CP
Register a workflow image in the registry.	DP	DP
List available hybrid workflow images.	CP	DP
Estimate the hybrid resources required.	CP	CP
Generate a schedule for hybrid tasks.	CP	CP

and updates the system monitor accordingly. For the QPU calibration data specifically, it fetches the new calibration data after each calibration cycle and updates the system monitor accordingly.

System monitor. The system monitor is a datastore where the complete system state is persisted. Specifically, the datastore maintains the list of available worker nodes and their resources, i.e., the number of cores, memory, accelerators, etc. (static information), and their current utilization, job queues, live status, etc. (dynamic information). Specifically for QPUs, we store the QPUs’ architectures, coupling maps, number of qubits, etc. (static information), and the current job queues and calibration data (dynamic information). The datastore also stores workflow information, specifically execution status (e.g., failed, completed, running, etc.), resource allocations, and their intermediate or final results.

Fault tolerance. The system relies on the control plane and the system monitor; therefore, it is crucial to make both fault-tolerant. We use a quorum of $2f+1$ nodes to replicate the components of the control plane, with $f=1$ by default. The backup replicas detect the component’s failures through heartbeat messages that experience delays greater than Δ , since we assume a partially synchronous message model [35]. In case of failure, the backups elect a new leader using Raft [65]. The same setup applies to the system monitor datastore.

4.2 System Workflow

The system workflow is shown in Figure 3. Users call invoke/deploy through the Qonductor API server to deploy hybrid workflows or invoke them, respectively (1). The workflow manager loads/stores hybrid workflow images from the workflow registry, depending on the user’s call (2). Then, the API server requests the resource estimator to generate resource plans that trade fidelity for runtime cost (3). The job manager invokes the scheduler to allocate worker nodes for the workflow’s jobs that comply with the resource plan and the user’s preferences (4). Finally, the job manager runs the job on the selected worker nodes (5), which execute it (6) and update the execution results (7).

5 Qonductor Programming Model

We introduce the Qonductor programming model designed to abstract away the complexity of programming and executing hybrid workflows. Clients can either reuse existing images from the workflow registry or create new ones with the help of (1) libraries of quantum and classical routines, (2) automated workflow generation and image packaging, and (3) hardware-agnostic deployment.

Classical and quantum libraries. The extensible libraries of commonly used quantum and classical functions aid programmability. Specifically, the classical library contains error mitigation techniques

[13, 53, 89] and simulation libraries [4, 11]. The quantum library includes state-of-the-art quantum algorithms such as the Variational Quantum Eigensolver (VQE) [72], the Quantum Approximate Optimization Algorithm (QAOA) [37], and the Quantum Fourier Transform (QFT) [97], among others.

Workflow image generation. The workflow manager automatically splits a Python file into quantum and classical code files while maintaining library dependencies and keeping track of input/output data between the files. Then, the manager creates a directed acyclic graph (DAG) $G = (V, E)$ where V is the set of classical and quantum steps and $E = \{(E_i, E_j) \in V \times V\}$ are the control and data flow dependencies between them. The leader node’s job manager later leverages this graph representation to handle workflow scheduling and execution. Lastly, the workflow graph model, the hybrid code files, and the execution configuration files are packed into a hybrid workflow image and stored in the workflow registry.

Workflow registry. Users typically write the same hybrid applications repeatedly, which becomes tedious for complex workflows. To streamline the deployment of such applications, the workflow registry is a repository for ready-to-execute workflow images. Users can leverage the registry to distribute or execute these images by providing input and customizing the execution to suit their unique requirements. Listing 1 shows two example images (L4 and L9), one for error mitigation using CUDA and one for a QAOA algorithm.

```

1 spec:
2   containers:
3     - name: qaoa-error-mitigated
4       image: nvidia/cuda:11.0-base
5       resources:
6         limits:
7           nvidia.com/gpu: 1 # Request one GPU
8     - name: qaoa-algorithm
9       image: qaoa:latest
10      resources:
11        limits:
12          quantum.ibm.com/qpu: 1 # Request one QPU
13        qubits: 20 # Request QPU size >= 20

```

Listing 1: Example YAML deployment configuration file.

Hybrid execution configuration. Users can customize computational resources in Qonductor by requesting specific QPUs or classical accelerators. Listing 1 shows an example YAML execution configuration file where the user requests at least one GPU (L7) and a QPU with at least 20 qubits (L12-L13).

```

1 from qonductor.lib.quantum import QAOA
2 from qonductor.lib.classical import ZNE, REM, DD
3 from qonductor.api import createWorkflow, deploy, workflowResults
4 from functools import partial
5
6 # Define the QAOA circuit and error mitigation techniques
7 qaoa = QAOA(qubits=10, optimizer='COBYLA')
8 zne_circuit = ZNE.apply(qaoa, noise_factors=(1, 3, 5))
9 pre_process = DD.apply(zne_circuit, sequence_type = "XpXm"))
10 rem_corrected = partial(REM.post_select(counts))
11 post_process = ZNE.inference(rem_corrected, "LinearFactory")
12
13 #Read deployment configuration file
14 with open('deployment.yaml', 'r') as file:
15     config = yaml.safe_load(file)
16
17 # Package a hybrid workflow image
18 hwi = createWorkflow([pre_process, qaoa, post_process], config)
19
20 # Deploy the hybrid image
21 workflowID = deploy(hwi)

```

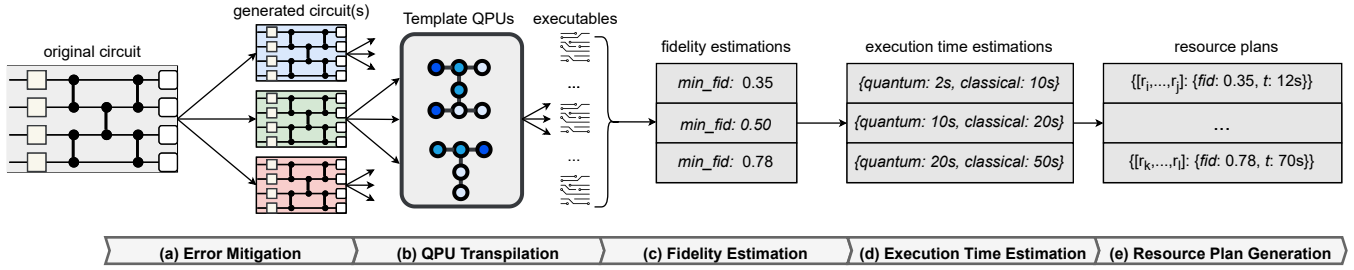


Figure 4: Resource estimator workflow (§ 6). (a) Error mitigation is applied to the original circuit, which generates one or more circuits. (b) The generated circuits are compiled for template QPUs, generating executable circuits. (c) Fidelity is estimated for all generated circuits where the minimum fidelity binds the solution fidelity. (d) The classical and quantum task execution times are estimated. (e) Resource plans are generated based on the estimated fidelities and total execution times.

```

22 # Query the workflow execution status and get the results
23 while workflowStatus(workflowID) is not 'finished':
24     pass
25 results = workflowResults(workflowID)

```

Listing 2: Example usage of the Qonductor APIs.

Qonductor APIs. In contrast to the current standard practice, the Qonductor APIs are hardware-agnostic and delegate hybrid resource allocation to the Qonductor leader node. Table 2 shows Qonductor’s APIs, where “CP” and “DP” stand for control and data plane, respectively. From the user’s point of view, there are only four functions. To create workflow images through the workflow manager, clients call `createWorkflow` with the hybrid code file or a list of classical and quantum functions. To deploy it on Qonductor, clients call `deploy` with the image ID and the deployment configuration file. Similarly, users call `invoke` with the image ID to run it. Lastly, to retrieve the execution results, clients call `workflowResults`. A toy example using the Qonductor APIs is shown in Listing 2.

6 Qonductor Resource Estimator

The resource estimator generates resource plans that serve two purposes: (1) clients can choose the plan that suits their cost-fidelity tradeoff preferences, and (2) the plans contain meta-information, such as fidelity and execution time estimations, that aid the scheduler in performing the final resource allocation. In this work, we focus on the error mitigation techniques provided by Qiskit [13, 21] and Mitq [53] due to being readily available and easy to integrate.

The resource estimator workflow is shown in Figure 4. First, we apply the error mitigation, which generates one or more circuits (a). Then, we transpile the circuit fragments for template QPU models filtered by the client’s preferences (b), to estimate the fidelity (c) and execution time (d) of the mitigated circuits for those models. Finally, we generate resource plans based on the estimated fidelities and aggregate execution times (e). We detail each of the steps below.

Error mitigation. The first stage integrates complementary error mitigation techniques in a stacked manner to enhance execution fidelity (§ 2.1). Specifically, we combine methods that reduce gate, measurement, and decoherence-induced errors at the same time. For instance, REM with Pauli twirling and dynamical decoupling address all major sources of errors. In general, we focus on out-of-the-box techniques provided by Qiskit [13] and the Mitq framework

[53], which include: ZNE, PEC, readout error mitigation, dynamic decoupling, Pauli twirling, twirled readout error extinction, probabilistic error amplification, and quasi-probability decomposition implemented as circuit knitting [21]. Notably, the core error mitigation techniques (e.g., ZNE, PCE, and circuit knitting) generate multiple circuit instances per input circuit with varying noise characteristics.

QPU transpilation. To estimate fidelity and quantum execution time, this step transpiles the generated circuits to template QPUs, after filtering them based on the client’s execution preferences. Template QPUs adopt the basis gate set and qubit coupling map of a specific QPU model (e.g., IBM Falcon r5.11 [8]), but their calibration data are the average of all available QPUs of that model. Thus, we have as many template QPUs as available QPU models in the system. This coarse-grained approach is scalable since quantum cloud providers typically offer a few models (e.g., up to three in IBM [7]).

Fidelity and execution time estimation. To predict both the fidelity and execution time of circuits executed with Qonductor using error mitigation, we employ regression-based prediction models. We first construct a dataset comprising over 7,000 job executions collected from our experiments on the IBM quantum cloud. For either type of estimation, we have to use the type of error mitigation applied as a feature. For execution time estimation, we use circuit features such as the number of qubits (width), the number of shots, circuit depth, and the number of two-qubit operations. For fidelity estimation, we incorporate additional features, including the qubit topology and error rates of the target QPU. We train and evaluate multiple models through K -fold cross-validation, using the R^2 score as the primary evaluation metric [39]. Among the models considered, *Polynomial Regression* yields the highest accuracy, achieving an R^2 score of 0.998 for execution time and 0.976 for fidelity prediction.

Resource plan generation. Finally, the resource estimator generates a configurable number of resource plans, by default, three. For this, the estimator stores the estimated fidelity and execution time—which is the sum of quantum and classical execution times—of the workflow, along with the accelerators used (if applicable) for the error mitigation post-processing step.

7 Qonductor Hybrid Scheduler

The Qonductor hybrid scheduler allocates classical and quantum resources to jobs to balance the conflicting objectives of quantum computing, as stated in § 3. The scheduler supports pluggable policies

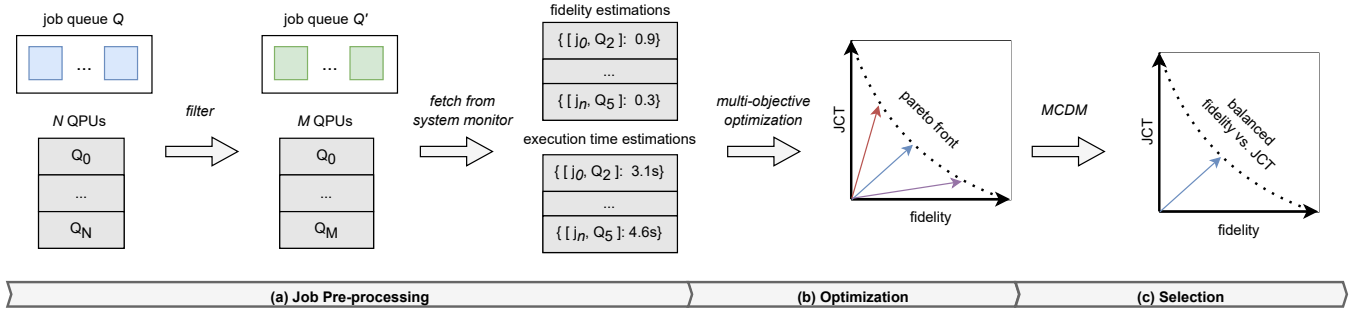


Figure 5: Quantum scheduler workflow (§ 7). (a) Job pre-processing: the jobs and QPUs are filtered based on the configuration options. Then, the fidelity and execution time estimations are fetched from the system monitor datastore. (b) Multi-objective optimization: we use the NSGA-II genetic algorithm to create a Pareto front of solutions. (c) Selection: we select one of the solutions based on multiple-criteria decision-making (MCDM) that uses pseudo-weights for fidelity and JCT.

for heterogeneity and load-aware resource allocation. In Qconductor, we provide two example policies for both types of resources, but mainly focus on quantum job scheduling where prior work is limited.

The scheduling algorithm for classical jobs follows the standard two-stage *filtering-scoring* algorithm of Kubernetes [9]. For each classical job, the first stage filters the available classical nodes based on the user’s configuration file to eliminate incompatible nodes. Based on pluggable scoring policies, the remaining nodes are then scored to find the most suitable nodes. In this work, our default filtering and scoring policies are based on the Kubernetes scheduler [10], but any heterogeneous-aware resource allocation policy is sufficient.

The scheduling algorithm for quantum jobs consists of three stages that support configurable policies: (1) the job pre-processing, (2) the optimization, and (3) the selection, as shown in Figure 5.

Job pre-processing. The first step is to pre-process the jobs to aid the scheduling optimization procedure (Figure 5 (a)). The scheduler filters the job queue and the QPU list to limit the exploration space and reduce the scheduling overheads. Specifically, it filters out the jobs that cannot run on the cluster given their configuration options (e.g., the system cannot accommodate the client’s QPU size requirements). Secondly, the scheduler fetches the fidelity and execution time estimations generated by the resource estimator (§ 6), which are stored in the system monitor. The optimization stage leverages the estimations to generate schedules with fidelity-JCT tradeoffs.

Optimization. The optimization stage creates a Pareto front of solutions for the scheduling problem, where the conflicting objectives are fidelity and JCTs (Figure 5 (b)). In Qconductor, we aim to minimize the mean JCT and maximize the mean fidelity among the scheduled jobs per scheduling cycle, and to do so, (1) we formulate the optimization problem and (2) employ an optimization algorithm to solve it.

Formally, we formulate the trade-off between fidelity and JCT as follows: $f_1(x)$ is the function capturing the mean JCTs, and $f_2(x)$ is the function capturing the mean error (1–mean fidelity), and we aim to minimize both:

$$\begin{aligned} \min \quad & f_1(x) = \frac{1}{N} \sum_{i=1}^N \left(w_{x_i} + \sum_{k=1}^N t_{kx_k} [x_i = x_k] \right), \quad f_2(x) = \frac{1}{N} \sum_{i=1}^N (1 - f_{ix_i}) \\ \text{s.t.} \quad & q_i - s_{x_i} \leq 0, \quad 1 \leq x_i \leq Q, \quad \forall i = 1, \dots, N \end{aligned} \quad (1)$$

where x_i is a discrete variable encoding the assignment of job i to QPU x_i ; N is the number of jobs to be scheduled; Q is the number of available QPUs; w_{x_i} represents the approximate waiting time of the job queue of QPU x_i ; t_{kx_k} is the estimated execution time of job k on QPU x_k ; f_{ix_i} is the estimated fidelity of job i on QPU x_i ; q_i stands for the maximum number of qubits in job i ; s_{x_i} is the size of QPU x_i . This problem formulation scales independently of the number of QPUs, with a complexity of $\mathcal{O}(N)$ for N jobs to be scheduled.

The formulated multi-objective optimization problem is Pareto-efficient by definition, and the potential solutions can be explored in parallel; this makes it a good candidate for genetic algorithms. Therefore, we use the NSGA-II genetic algorithm [34] that is robust against local optima and highly parallelizable. We customize the algorithm’s genetic operators to thoroughly explore the solution space by initializing the population with random integers, simulating the crossover operation on real values using an exponential probability distribution, and perturbing solutions within a parent’s vicinity using a polynomial probability distribution. Lastly, to avoid prolonged execution, we set maximum thresholds for generations and function evaluations and use a sliding window approach for tolerance termination, evaluating a sequence of generations rather than just the latest one.

Selection. The solutions of the Pareto front differ in mean fidelity and JCTs of the scheduled jobs, covering the full range between their maximum and minimum values. To select a single solution based on priority on fidelity, JCT, or balanced, we use Multiple-Criteria Decision-Making (MCDM) with pseudo-weights (Figure 5, (c)). Calculating pseudo weights involves normalizing the distance to the worst solution concerning each objective, which indicates the solution’s location in the objective space. Formally, the pseudo-weight equation is:

$$w_i(x) = \frac{(f_i^{\max} - f_i(x)) / (f_i^{\max} - f_i^{\min})}{\sum_{m=1}^M (f_m^{\max} - f_m(x)) / (f_m^{\max} - f_m^{\min})} \quad (2)$$

The pseudo-weight $w_i(x)$ measures the relative importance of the i -th objective for solution x within the entire Pareto front, and $f_i^{\min}$ and $f_i^{\max}$ are the minimum and maximum objective values of objective i over all solutions in the Pareto front. We select the solution x with a vector closest to a desired preference vector $P = (p_1, p_2)$; here p_1 is mean fidelity, and p_2 is mean JCTs and $p_1 + p_2 = 1$.

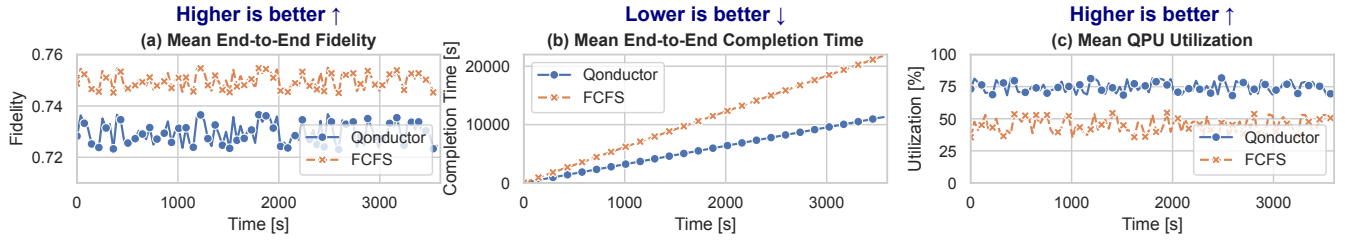


Figure 6: Qonductor end-to-end performance (§ 8.3). The experiments run for one simulated hour and 1500 applications/hour. (a) *Mean end-to-end fidelity: Qonductor’s fidelity is < 3% lower than FCFS.* (b) *Mean end-to-end completion time: Qonductor’s completion times are ~ 48% lower than FCFS.* (c) *Mean QPU utilization: Qonductor’s utilization is ~ 66% higher than FCFS.*

Scheduling triggers. Scheduling is triggered in two ways: (1) job queue size and (2) time-based trigger. In the former case, if the job queue size reaches a specified limit (100 by default), scheduling is invoked. In the latter case, if a pre-defined time interval elapses (120s by default), scheduling is invoked regardless of the queue size.

Calibration crossovers. If a generated schedule spans a calibration cycle, we automatically re-evaluate and adjust the jobs scheduled to run after the calibration update. Specifically, our scheduler partitions the schedule at the calibration boundary. Then, it invokes the resource estimator to re-compute fidelity and runtime predictions for jobs in the post-calibration window using the latest calibration data and then reassigns or delays those jobs as necessary.

Priority access. In our current implementation, Qonductor does not inherently support provider reservations, as such reservations tend to exacerbate QPU load imbalances by encouraging users to repeatedly select the highest-fidelity QPUs. Instead, when deployed in a cloud environment that implements reservations, Qonductor treats the reserved QPUs as temporarily offline, effectively removing them from the available resource pool during the reservation period.

8 Evaluation

8.1 Evaluation Setup

Implementation. We implement Qonductor on top of Kubernetes [9] in Python v.3.11 [91] and Go v.1.21 [19]. In the resource estimator (§ 6), we use the sci-kit-learn v.1.4.0 [70] library for estimating fidelity and quantum execution times. For the optimization and MCDM scheduler stages (§ 7), we employ the pymoo v.0.6.1 [23] framework. Lastly, as our quantum cloud provider, we select IBM Quantum [7] due to its open-access model.

Experimental setup. We conduct two types of experiments: (1) real QPU runs to collect the dataset of the resource estimator (§ 6), i.e., the job execution times and fidelities, and (2) classical simulations of the hybrid cloud. For (1), we utilize the IBM Quantum open access plan [7] and run jobs on all freely available QPUs. For (2), we run on AMD EPYC 7713P 64-Core servers with 0.5 TB of RAM and use Qiskit’s FakeBackends for noisy simulations.

Benchmarks. We use the MQT Benchmark library [75] to generate over 70,000 benchmark circuits, 2 to 130 qubits in size. The library covers all standard quantum algorithms, including VQE [72], Grover’s [41], Shor’s [81] algorithms, QAOA [37], and Quantum Fourier Transform (QFT) [97].

Metrics. For evaluating Qonductor’s performance, we use the following metrics: (1) **(Job) Completion Time:** Time a job/application requires to complete (Qonductor processing + waiting + executing). (2) **Fidelity:** We use *Hellinger fidelity* as a measure of the quality of the execution on noisy QPUs [14, 44]. Fidelity ranges in [0,1] and higher is better. (3) **Execution Time:** Time the job runs on a classical or quantum resource, excluding processing and waiting times.

Baselines. We use the First-Come-First-Serve (FCFS) scheduling algorithm and different configurations of our system as baselines unless otherwise stated. Qoncord [95] specifically addresses the scheduling challenges inherent to VQAs, focusing on dividing training iterations between exploratory and fine-tuning phases. In contrast, Qonductor provides a generalized orchestration framework for a broad spectrum of hybrid quantum-classical applications. As such, a direct empirical comparison between Qoncord and Qonductor would not yield meaningful insights, as they are tailored to address distinct challenges within the quantum computing landscape.

8.2 Quantum Cloud Simulation

To evaluate Qonductor, we set up a cloud simulation environment replicating the real conditions of the IBM Quantum platform [7].

Dataset collection. We monitor all available QPUs on the IBM Quantum platform for ten days in November 2023 to gather the QPUs’ queue sizes. We then aggregate and analyze the differences in queue sizes for each QPU to measure the job arrival rates. We identify a notable variance in rates across during the day ranging from 1100 to 2050 jobs per hour. The total average of all hours is 1500 jobs per hour and is the baseline system load for our evaluation.

Load generator. The load generator creates synthetic workloads that mirror the real-world hybrid application patterns. It generates hybrid applications with random quantum circuits, number of shots, and circuit sizes, following a normal distribution. All applications are transpiled on Qonductor, and a random number of them (50% on average) use error mitigation, hence utilizing hybrid resources. These applications are then submitted to Qonductor with a fixed frequency, simulating real-world arrival rates.

Metrics collection. We patch Qiskit’s FakeBackends with the ability to maintain their own queue of scheduled jobs, job waiting and execution times, and the notion of time flow, reflecting the real-world job flow. After each scheduling cycle, the job manager receives the results and assigns the new jobs to the queues of the chosen backends.

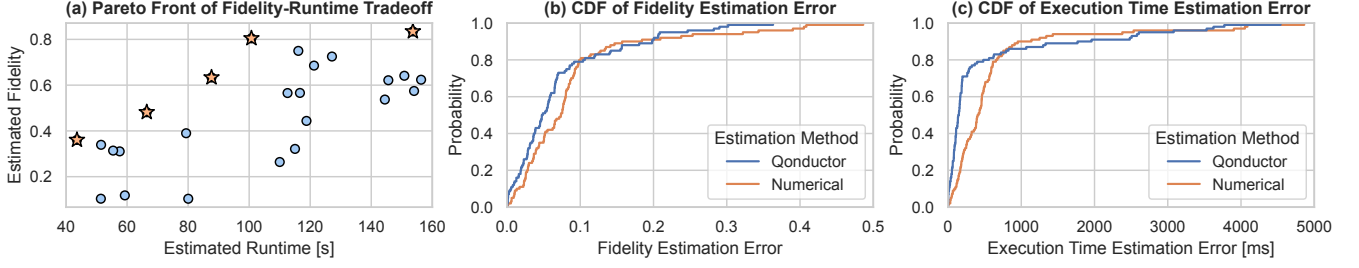


Figure 7: Resource estimator’s performance (§ 8.4). (a) *Pareto front (star points) of the tradeoff between estimated fidelity and hybrid application runtime.* (b) *CDF of the fidelity estimation error. ~75% of estimations have an error of less than 0.1.* (c) *CDF of the execution time estimation error, in milliseconds. 80% of estimations have an error of less than 500ms.*

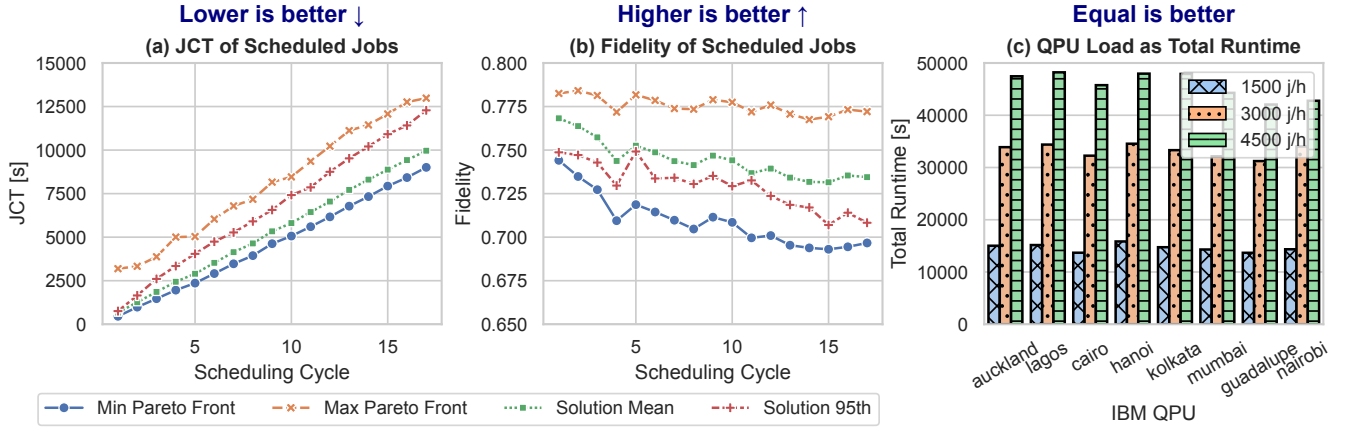


Figure 8: Qonductor’s scheduler performance (§ 8.5). (a) *JCTs for scheduled jobs. The chosen solution (mean) incurs 34% lower and 15.1% higher JCTs compared to the maximum and minimum Pareto fronts, respectively.* (b) *Fidelities of scheduled jobs. The chosen solution incurs 4% lower fidelity than the maximum possible.* (c) *QPU load as the total active runtime for increasing workloads. Qonductor distributes the load almost evenly, with a maximum load difference of 15.8% between QPUs.*

8.3 Qonductor’s End-to-End Performance

RQ1: *What is the end-to-end performance of Qonductor w.r.t. mean fidelity, completion times, and utilization?* We evaluate Qonductor’s performance by simulating synthetic cloud workloads as stated in § 8.2 and measuring the mean hybrid application fidelity, completion time, and QPU utilization. As a baseline, we use the standard practice in the current quantum cloud, FCFS scheduling.

Figure 6 (a) shows the mean fidelity across simulation time. Fluctuations in fidelity are random and depend on (1) the workloads executed at each time point and (2) the application of error mitigation. Qonductor’s mean fidelity is 2 – 3% lower than that of FCFS since the scheduler selects QPUs with sub-optimal fidelity in favor of completion times. Figure 6 (b) shows the mean end-to-end completion times across simulation time. Qonductor’s mean completion times are ~48% lower than FCFS since Qonductor balances the load across QPUs, reducing the quantum waiting times, while the classical waiting times are practically zero. Notably, both approaches face linearly increasing completion times as a function of time since QPUs are still scarce and, even with load balancing, face long queues. Lastly, Figure 6 (c) shows the mean QPU utilization across simulation time. Due to load-balancing scheduling and the aforementioned

tradeoff exploration between fidelity and completion times, Qonductor achieves 66% higher utilization than FCFS, on average, by distributing the quantum job load across QPUs more evenly.

RQ1 takeaway. Qonductor achieves 48% lower hybrid application completion times and 66% higher QPU utilization for up to < 3% lower fidelity, compared to FCFS scheduling, on average.

8.4 Resource Estimator’s Performance

RQ2: *How systematically and accurately does the Qonductor resource estimator explore the fidelity-runtime tradeoff?* We evaluate the resource estimator’s performance by visualizing the generated resource plans’ fidelity vs runtime and the accuracy of its estimations.

Figure 7 (a) shows the fidelity-runtime Pareto front of resource plans, where star points highlight the Pareto-optimal plans. Here, we are using a 20-qubit QAOA max-cut circuit. Each point is a unique resource plan w.r.t. the configuration of error mitigation (e.g. sampling overheads), QPUs used, and classical accelerators for post-processing. Notably, the second-highest fidelity solution incurs 34.6% lower runtime than the highest, for only 3.6% lower fidelity.

To measure the resource estimator’s estimation accuracy, we plot the absolute difference between the estimated fidelities and quantum execution times with the real, post-execution values, $|est - real|$.

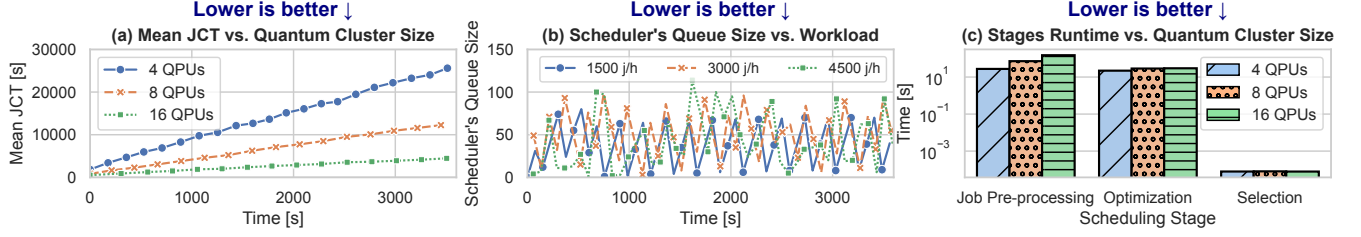


Figure 9: Qonductor’s scheduler scalability analysis (§ 8.5). (a) *Mean JCT as the quantum cluster scales w.r.t. the number of QPUs. As the number of QPUs increases, the mean JCT decreases.* (b) *Qonductor scheduler scalability w.r.t. system load. The scheduler successfully handles workloads up to 3× the current IBM load.* (c) *Qonductor scheduler’s stages scalability w.r.t. the cluster size. All stages runtimes are relatively constant as the cluster size increases.*

The baseline is the numerical approach followed by state-of-the-art work [62, 95, 101], where fidelity and execution times are computed based on the calibration data of the QPU and the operations applied in the circuit, e.g., by traversing the circuit DAG and multiplying the noise errors or summing the gate execution times, respectively. Figure 7 (b) shows the CDF of the fidelity estimation errors. Our regression model is more accurate than the numerical method by including the effects of error mitigation, although the difference is noticeable only for errors less than 0.1 fidelity. Figure 7 (c) shows the CDF of quantum execution time estimation error. Here, the regression model outperforms the numerical approach, and 80% of the estimations are off by less than 500ms.

RQ2 takeaway. Our resource estimator accurately estimates execution fidelity and runtime with minimal error in 80% of cases and generates resource plans with Pareto-optimal fidelity-runtime tradeoffs.

8.5 Qonductor Scheduler’s Performance

RQ3: *How well does the Qonductor scheduler balance quantum job fidelity and JCTs, and the load across QPUs?* We use our simulation environment to evaluate the balance between quantum job fidelity and JCTs, the load difference across QPUs, and the mean quantum job execution time.

Figures 8 (a) and (b) show the minimum and maximum values of the Pareto front for each scheduling cycle, and our chosen solution. Here, the workload is 1500 jobs/hour and we use equal weights between fidelity and JCTs. The chosen solutions consistently gravitate towards the minimum Pareto front for JCT. Specifically, the mean and 95th percentile JCTs are 34% and 17.4% lower compared to the maximum, respectively. The mean and 95th percentile fidelities are only 4% and 6% lower than the maximum, respectively.

To evaluate load balancing across QPUs, we track the total execution time allocated to each QPU over one hour. The resulting distribution for the eight simulated QPUs is presented in Figure 8 (c), where the load distribution across QPUs is nearly uniform. The maximum load difference between any two QPUs is 15.8% in the workload case of 1500 jobs/hour.

Lastly, Figure 10 (a) shows the mean execution time of the scheduled quantum jobs. The minimum and maximum Pareto fronts are the lower and upper bounds on the mean execution time and act as a proxy to even QPU utilization. The chosen solution achieves 63.4% lower execution time compared to the maximum Pareto front.

RQ3 takeaway. The Qonductor scheduler successfully manages the tradeoff between fidelity and JCT. The chosen solutions incur 34% lower JCT for a 4% drop in fidelity. Also, the scheduler balances the load across all available QPUs with minimal load difference (< 16%).

RQ4: *How systematically does the Qonductor scheduler create and explore the Pareto front of solutions?* To evaluate this, we generate a synthetic workload of 100 randomly generated quantum jobs and visualize the Pareto front of the generated schedules.

Figure 10 (b) shows three different priorities on objectives: JCT, fidelity, and balanced. By prioritizing JCT over fidelity, the scheduler chooses the solution with the lowest mean JCT, i.e., 67% lower JCT than priority on fidelity. Inversely, the scheduler chooses the solution with the highest mean fidelity, i.e., 16% higher than the case of prioritizing JCT. Lastly, assigning equal weights selects a balanced solution, where 6% lower fidelity leads to 54% lower JCT.

RQ4 takeaway. The Qonductor scheduler systematically explores the tradeoff between fidelity and JCTs. Notably, users can experience 54% lower JCTs for 6% lower fidelity, on average.

RQ5: *How well does the Qonductor scheduler scale with the cluster size (number of QPUs) and the workload (jobs per hour)?* We measure mean JCT improvement with increasing QPU cluster size, the scheduler’s pending queue size as the workload increases, and the internal scheduling stages’ runtimes with increasing QPU cluster size.

Figure 9 (a) shows the mean JCT as the QPU cluster size increases from 4 to 16 QPUs. Qonductor adapts to the growing number of QPUs by utilizing them to evenly distribute the workload. Doubling the system size from 4 to 8 QPUs improves JCTs by 52.8% and making it four times larger (16 QPUs) improves JCT by 81% (4.35× lower).

Figure 9 (b) shows the pending job queue size as the workload increases from 1500 j/h to 3000 and 4500 j/h, respectively. The scheduler remains stable even with 3× higher workload than the current, or ~2.2× the current IBM *peak* workload (~ 2000 j/h). The oscillation of the three lines reflects the time or window-based triggers of our scheduler. Specifically, each drop in time means that scheduling was invoked, which empties the scheduling queue. Over time, it increases again, until the next scheduling trigger.

Finally, Figure 9 (c) shows the runtime of the three scheduling stages as the QPU cluster size increases. Notably, only the job pre-processing’s runtime increases since the fidelity and execution time estimations are performed for more QPUs. We do not compare against an increasing workload (j/h) since the scheduling stages’ overheads only scale with the number of QPUs.

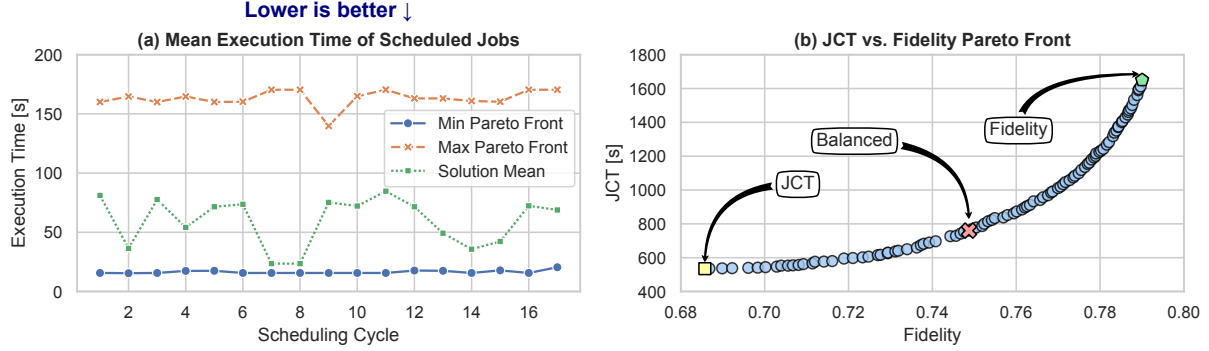


Figure 10: Qonductor scheduler’s performance (§ 8.5). (a) *Mean execution time of the quantum jobs. The chosen solution achieves 63.4% lower execution time than the maximum Pareto front.* (b) *The scheduler’s selection stage (MCDM) selects solutions that match the priorities on conflicting objectives. Balanced: 6% lower fidelity gives 54% lower JCT.*

RQ5 takeaway. The Qonductor scheduler successfully utilizes newly available QPU resources, handles increasing system loads up to 3× the current IBM system load, and indicates scalable performance for its internal stages.

9 Related Work

Quantum cloud and serverless. Research work in quantum cloud computing either analyzes the existing quantum cloud characteristics [77] or proposes quantum cloud architectures, e.g., serverless [38, 48, 82]. However, these approaches do not tackle all three challenges we identify in § 3 in a single system.

Quantum resource estimation (QRE). Research on QRE is limited to predicting the number of physical qubits required to run certain quantum algorithms given a set of assumptions, e.g., the quantum error correction scheme used, if any, and the QPU’s calibration data. To our knowledge, Microsoft’s Azure Quantum Resource Estimator is the only automated and systematic QRE approach [22]. However, it does not account for the fidelity and execution runtime impact of classical resources, in contrast to our approach.

Quantum Resource Estimation (QRE). Azure’s Quantum Resource Estimator (QRE) predicts the number of physical qubits needed to run fault-tolerant quantum algorithms under error correction [22]. It supports only a limited set of predefined noise models (typically 3–4), making it difficult to extend or adapt to arbitrary hardware. In contrast, Qonductor’s estimator predicts both fidelity and execution time for hybrid algorithms on current noisy devices, incorporating the impact of classical resources.

Quantum job scheduling. While the field of scheduling has received significant attention in the realm of classical computing, its application to quantum computing remains in its early stages due to the relative infancy of the technology. As a consequence, the area of quantum scheduling is still relatively underdeveloped. To the best of our knowledge, quantum job scheduling work is limited to [47, 78, 79, 84, 95]. However, these systems face one or more of the following limitations: (1) they implement only single-to-many scheduling, (2) they do not offer fine-grained control over the balance between JCTs and fidelity, (3) they delegate the final scheduling decision to the user, or (4) are limited to VQA algorithms only. In contrast, our many-to-many scheduling policy addresses the needs of both

users and cloud providers. It automatically schedules quantum jobs by trading fidelity for JCTs, while balancing the load across QPUs.

Quantum Resource-sharing. Existing work on quantum resource sharing focuses almost exclusively on multi-programming [33, 57, 63, 64]. Specifically, existing work aims to allocate high-quality regions of the QPU to the bundled programs. Integrating multi-programming is left as future work for our system.

Classical resource management and scheduling. Resource allocation and task scheduling on the classical cloud are active areas of research and have been extensively studied. Specifically, a non-exhaustive list of work includes task scheduling [42, 55, 67, 76, 102], resource allocation [25, 29, 59, 94, 96, 100], and container orchestration [46, 80, 92, 98]. Moreover, there exists work on heterogeneous scheduling [54, 66, 68, 96] and application-specific scheduling, e.g. for deep learning workloads [52, 61, 71, 99]. However, the classical domain does not face the unique challenges of the quantum § 3. As such, it is not trivial to adapt this work for the quantum cloud.

10 Conclusion

We introduced Qonductor, a cloud orchestrator for developing and deploying hybrid quantum-classical applications on hybrid heterogeneous clouds. To our knowledge, Qonductor is the first holistic approach for hybrid orchestration and improves upon the state-of-the-art in three dimensions: **First**, it exposes hardware-agnostic APIs that abstract the underlying complexity away (§ 5), **Second**, it offers the first approach to hybrid resource estimation that systemizes tradeoff management in hybrid resource allocation (§ 6), and **Third**, it is the first approach to hybrid scheduling, where for quantum jobs we balance the tradeoff between fidelity and JCTs (§ 7).

Acknowledgments

We sincerely thank the anonymous reviewers for their feedback. We thank Karl Jansen and Stefan Kühn from the Center for Quantum Technology and Applications (CQTA)- Zeuthen for supporting this work by providing access to IBM quantum resources. Funded by the Bavarian State Ministry of Science and the Arts as part of the Munich Quantum Valley (MQV), grant number 6090181.

References

- [1] Anonymous. 2023. Qiskit: An Open-source Framework for Quantum Computing. <https://www.ibm.com/quantum/qiskit>.
- [2] Anonymous. 2025. AWS Bracket. <https://aws.amazon.com/braket/>. Accessed: 2025-08-11.
- [3] Anonymous. 2025. Azure Quantum. <https://azure.microsoft.com/en-us/products/quantum>. Accessed: 2025-08-11.
- [4] Anonymous. 2025. cuQuantum SDK: A High-Performance Library for Accelerating Quantum Science. <https://docs.nvidia.com/cuda/cuquantum/latest/index.html>. Accessed: 2025-08-11.
- [5] Anonymous. 2025. Google Cirq. <https://quantumai.google/>. Accessed: 2025-08-11.
- [6] Anonymous. 2025. IBM Classical Infrastructure. <https://www.ibm.com/products/bare-metal-servers>.
- [7] Anonymous. 2025. IBM Quantum. <https://quantum.ibm.com/services/resources>. Accessed: 2025-08-11.
- [8] Anonymous. 2025. IBM Quantum Processor Types. <https://docs.quantum.ibm.com/run/processor-types>. Accessed: 2025-08-11.
- [9] Anonymous. 2025. Kubernetes. <https://kubernetes.io/>.
- [10] Anonymous. 2025. Kubernetes Scheduler. <https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/>.
- [11] Anonymous. 2025. PennyLane Simulators. <https://pennylane.ai/plugins/>. Accessed: 2025-08-11.
- [12] Anonymous. 2025. Qiskit 3 Trillion Circuit Executions. https://www.linkedin.com/posts/jay-gambetta-a274753a_the-number-of-quantum-circuits-executed-on-activity-7136792322477821952-E3wQ?utm_source=share&utm_medium=member_desktop. Accessed: 2024-02-08.
- [13] Anonymous. 2025. Qiskit Error Mitigation Techniques. <https://docs.quantum.ibm.com/guides/error-mitigation-and-suppression-techniques>. Accessed: 2025-08-11.
- [14] Anonymous. 2025. Qiskit Hellinger Fidelity. https://docs.quantum.ibm.com/api/qiskit/0.31/qiskit.quantum_info.hellinger_fidelity. Accessed: 2025-08-11.
- [15] Anonymous. 2025. Qiskit Least-Busy Policy. https://quantum.cloud.ibm.com/docs/en/api/qiskit-ibm-runtime/qiskit-runtime-service#least_busy. Accessed: 2025-08-11.
- [16] Anonymous. 2025. Qiskit Transpiler. <https://qiskit.org/documentation/apidoc/transpiler.html>. Accessed: 2025-08-11.
- [17] Anonymous. 2025. Qoogle Quantum AI Calibration. <https://quantumai.google/cirq/google/calibration>. Accessed: 2022-04-11.
- [18] Anonymous. 2025. Quantum Computer Datasheet. <https://quantumai.google/hardware/datasheet/weber.pdf>. Accessed: 2025-08-11.
- [19] Anonymous. 2025. The Go Programming Language. <https://go.dev/>. Accessed: 2024-04-08.
- [20] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, et al. 2019. Quantum supremacy using a programmable superconducting processor. *Nature* 574, 7779 (2019), 505–510.
- [21] Luciano Bello, Agata M. Brańczyk, Sergey Bravyi, Almudena Carrera Vazquez, Andrew Eddins, Daniel J. Egger, Bryce Fuller, Julien Gacon, James R. Garrison, Jennifer R. Glick, Tanvi P. Gujarati, Ikko Hamamura, Areeq I. Hasan, Takashi Imamichi, Caleb Johnson, Ieva Liepuoniute, Owen Lockwood, Mario Motta, C. D. Pemmaraju, Pedro Rivero, Max Rossmannek, Travis L. Scholten, Seetharami Seelam, Iskandar Sitdikov, Dharmashankar Subramanian, Wei Tang, and Stefan Woerner. 2023. Circuit Knitting Toolbox. <https://github.com/Qiskit-Extensions/circuit-knitting-toolbox>. <https://doi.org/10.5281/zenodo.7987997>
- [22] Michael E. Beverland, Prakash Murali, Matthias Troyer, Krysta M. Svore, Torsten Hoefler, Vadym Kliuchnikov, Guang Hao Low, Mathias Soeken, Aarthi Sundaram, and Alexander Vashchilo. 2022. Assessing requirements to scale to practical quantum advantage. arXiv:2211.07629 [quant-ph] <https://arxiv.org/abs/2211.07629>
- [23] J. Blank and K. Deb. 2020. pymoo: Multi-Objective Optimization in Python. *IEEE Access* 8 (2020), 89497–89509.
- [24] Sergey Bravyi, Sarah Sheldon, Abhinav Kandala, David C. McKay, and Jay M. Gambetta. 2021. Mitigating measurement errors in multiqubit experiments. *Phys. Rev. A* 103 (Apr 2021), 042605. Issue 4. <https://doi.org/10.1103/PhysRevA.103.042605>
- [25] Fangzhe Chang, Jennifer Ren, and Ramesh Viswanathan. 2010. Optimal resource allocation in clouds. In *2010 IEEE 3rd International Conference on Cloud Computing*. IEEE, 418–425.
- [26] Yulin Chi, Jieshan Huang, Zhanchuan Zhang, Jun Mao, Zinan Zhou, Xiaojiong Chen, Chonghao Zhai, Jueming Bao, Tianxiang Dai, Huihong Yuan, et al. 2022. A programmable qudit-based quantum processor. *Nature communications* 13, 1 (2022), 1166.
- [27] Juan I Cirac and Peter Zoller. 1995. Quantum computations with cold trapped ions. *Physical review letters* 74, 20 (1995), 4091.
- [28] Andrew W. Cross, Lev S. Bishop, Sarah Sheldon, Paul D. Nation, and Jay M. Gambetta. 2019. Validating quantum computers using randomized model circuits. *Phys. Rev. A* 100 (Sep 2019), 032328. Issue 3. <https://doi.org/10.1103/PhysRevA.100.032328>
- [29] Wenyun Dai, Longfei Qiu, Ana Wu, and Meikang Qiu. 2016. Cloud infrastructure resource allocation for big data applications. *IEEE Transactions on Big Data* 4, 3 (2016), 313–324.
- [30] Andrew J Daley, Immanuel Bloch, Christian Kokail, Stuart Flannigan, Natalie Pearson, Matthias Troyer, and Peter Zoller. 2022. Practical quantum advantage in quantum simulation. *Nature* 607, 7920 (2022), 667–676.
- [31] Poulami Das, Swamit Tannu, Siddharth Dangwal, and Moinuddin Qureshi. 2021. ADAPT: Mitigating Idling Errors in Qubits via Adaptive Dynamical Decoupling. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) (MICRO '21). Association for Computing Machinery, New York, NY, USA, 950–962. <https://doi.org/10.1145/3466752.3480059>
- [32] Poulami Das, Swamit Tannu, and Moinuddin Qureshi. 2021. JigSaw: Boosting Fidelity of NISQ Programs via Measurement Subsetting. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) (MICRO '21). Association for Computing Machinery, New York, NY, USA, 937–949. <https://doi.org/10.1145/3466752.3480044>
- [33] Poulami Das, Swamit S. Tannu, Prashant J. Nair, and Moinuddin Qureshi. 2019. A Case for Multi-Programming Quantum Computers. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) (MICRO '52). Association for Computing Machinery, New York, NY, USA, 291–303. <https://doi.org/10.1145/3352460.3358287>
- [34] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6, 2 (2002), 182–197. <https://doi.org/10.1109/4235.996017>
- [35] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the Presence of Partial Synchrony. *J. ACM* 35, 2 (April 1988), 288–323. <https://doi.org/10.1145/42282.42283>
- [36] Suguru Endo, Simon C. Benjamin, and Ying Li. 2018. Practical Quantum Error Mitigation for Near-Future Applications. *Physical Review X* 8, 3 (July 2018). <https://doi.org/10.1103/physrevx.8.031027>
- [37] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. 2014. A Quantum Approximate Optimization Algorithm. arXiv:1411.4028 [quant-ph]
- [38] Jose Garcia-Alonso, Javier Rojo, David Valencia, Enrique Moguel, Javier Berrocal, and Juan Manuel Murillo. 2022. Quantum Software as a Service Through a Quantum API Gateway. *IEEE Internet Computing* 26, 1 (Jan 2022), 34–41. <https://doi.org/10.1109/MIC.2021.3132688>
- [39] James Gareth, Witten Daniela, Hastie Trevor, and Tibshirani Robert. 2013. *An introduction to statistical learning: with applications in R*. Springer.
- [40] Emmanouil Giortamis, Francisco Romão, Nathaniel Tornow, and Pramod Bhatotia. 2025. QOS: Quantum Operating System. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI '25)*. USENIX Association, Boston, MA, 429–447. <https://www.usenix.org/system/files/osdi25-giortamis.pdf>
- [41] Lov K Grover. 1996. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*. 212–219.
- [42] Lizheng Guo, Shuguang Zhao, Shigen Shen, and Changyuan Jiang. 2012. Task scheduling optimization in cloud computing based on heuristic algorithm. *Journal of networks* 7, 3 (2012), 547.
- [43] Laszlo Gyongyosi and Sandor Imre. 2019. A Survey on quantum computing technology. *Computer Science Review* 31 (2019), 51–71. <https://doi.org/10.1016/j.cosrev.2018.11.002>
- [44] Ernst Hellinger. 1909. Neue begründung der theorie quadratischer formen von unendlichvielen veränderlichen. *Journal für die reine und angewandte Mathematik* 1909, 136 (1909), 210–271.
- [45] Loïc Henriet, Lucas Beguin, Adrien Signoles, Thierry Lahaye, Antoine Browaeys, Georges-Olivier Reymond, and Christophe Jurczak. 2020. Quantum computing with neutral atoms. *Quantum* 4 (2020), 327.
- [46] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A platform for {Fine-Grained} resource sharing in the data center. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*.
- [47] Rakpong Kaewpuang, Minrui Xu, Dusit Niyato, Han Yu, Zehui Xiong, and Jiawen Kang. 2023. Stochastic Qubit Resource Allocation for Quantum Cloud Computing. In *NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 1–5.
- [48] Peter J Karalekas, Nikolas A Tezak, Eric C Peterson, Colm A Ryan, Marcus P da Silva, and Robert S Smith. 2020. A quantum-classical cloud platform optimized for variational hybrid algorithms. *Quantum Science and Technology* 5, 2 (mar 2020), 024003. <https://doi.org/10.1088/2058-9565/ab7559>
- [49] Oğuzcan Kirmemiş, Francisco Romão, Emmanouil Giortamis, and Pramod Bhatotia. 2025. Weaver: A Retargetable Compiler Framework for FPQA Quantum Architectures. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization* (Las Vegas, NV, USA) (CGO '25). Association for Computing Machinery, New York, NY, USA, 299–316. <https://doi.org/10.1145/3696443.3708965>
- [50] P. V. Klimov, J. Kelly, Z. Chen, M. Neeley, A. Megrant, B. Burkett, R. Barends, K. Arya, B. Chiaro, Yu Chen, A. Dunsworth, A. Fowler, B. Foxen, C. Gidney, M. Giustina, R. Graff, T. Huang, E. Jeffrey, Erik Lucero, J. Y. Mutus, O. Naaman, C. Neill, C. Quintana, P. Roushan, Daniel Sank, A. Vainsencher, J. Wenner, T. C. White, S.

- Boixo, R. Babbush, V. N. Smelyanskiy, H. Neven, and John M. Martinis. 2018. Fluctuations of Energy-Relaxation Times in Superconducting Qubits. *Phys. Rev. Lett.* 121 (Aug 2018), 090502. Issue 9. <https://doi.org/10.1103/PhysRevLett.121.090502>
- [51] Sebastian Krinner, Simon Storz, Philipp Kurpiers, Paul Magnard, Johannes Heinsoo, Raphael Keller, Janis Luetolf, Christopher Eichler, and Andreas Wallraff. 2019. Engineering cryogenic setups for 100-qubit scale superconducting circuit systems. *EPJ Quantum Technology* 6, 1 (2019), 2.
- [52] Woosuk Kwon, Gyeong-In Yu, Eunji Jeong, and Byung-Gon Chun. 2020. Nimble: Lightweight and parallel gpu task scheduling for deep learning. *Advances in Neural Information Processing Systems* 33 (2020), 8343–8354.
- [53] Ryan LaRose, Andrea Mari, Sarah Kaiser, Peter J. Karalekas, Andre A. Alves, Piotr Czarnik, Mohamed El Mandouh, Max H. Gordon, Yusef Hindy, Aaron Robertson, Purva Thakre, Misty Wahl, Danny Samuel, Rahul Mistry, Maxime Tremblay, Nick Gardner, Nathaniel T. Stemen, Nathan Shammah, and William J. Zeng. 2022. Mitiq: A software package for error mitigation on noisy quantum computers. *Quantum* 6 (Aug 2022), 774. <https://doi.org/10.22331/q-2022-08-11-774>
- [54] Gunho Lee and Randy H. Katz. 2011. {Heterogeneity-Aware} Resource Allocation and Scheduling in the Cloud. In *3rd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 11)*.
- [55] Kun Li, Gaochao Xu, Guangyu Zhao, Yushuang Dong, and Dan Wang. 2011. Cloud Task Scheduling Based on Load Balancing Ant Colony Optimization. In *2011 Sixth Annual Chinagrid Conference*. 3–9. <https://doi.org/10.1109/ChinaGrid.2011.17>
- [56] Ying Li and Simon C. Benjamin. 2017. Efficient Variational Quantum Simulator Incorporating Active Error Minimization. *Physical Review X* 7, 2 (June 2017). <https://doi.org/10.1103/physrevx.7.021050>
- [57] Lei Liu and Xinglei Dou. 2021. QuCloud: A New Qubit Mapping Mechanism for Multi-programming Quantum Computing in Cloud Environment. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 167–178. <https://doi.org/10.1109/HPCA51647.2021.00024>
- [58] Satvik Maurya, Chaithanya Naik Mude, William D. Oliver, Benjamin Lienhard, and Swamit Tannu. 2023. Scaling Qubit Readout with Hardware Efficient Machine Learning Architectures. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (Orlando, FL, USA) (ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 7, 13 pages. <https://doi.org/10.1145/3579371.3589042>
- [59] Seyedehmehnaz Mireslami, Logan Rakai, Mea Wang, and Behrouz Homayoun Far. 2019. Dynamic cloud resource allocation considering demand uncertainty. *IEEE Transactions on Cloud Computing* 9, 3 (2019), 981–994.
- [60] Kosuke Mitarai and Keisuke Fujii. 2021. Constructing a virtual two-qubit gate by sampling single-qubit operations. *New Journal of Physics* 23, 2 (2021), 023021.
- [61] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. 2020. {Heterogeneity-Aware} cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 481–498.
- [62] Paul D. Nation and Matthew Treinish. 2023. Suppressing Quantum Circuit Errors Due to System Variability. *PRX Quantum* 4 (3 2023), 010327. Issue 1. <https://doi.org/10.1103/PRXQuantum.4.010327>
- [63] Siyuan Niu and Aida Todri-Sanial. 2023. Enabling Multi-programming Mechanism for Quantum Computing in the NISQ Era. *Quantum* 7 (feb 2023), 925. <https://doi.org/10.22331/q-2023-02-16-925>
- [64] Yasuhiro Ohkura, Takahiko Satoh, and Rodney Van Meter. 2022. Simultaneous Execution of Quantum Circuits on Current and Near-Future NISQ Systems. *IEEE Transactions on Quantum Engineering* 3 (2022), 1–10. <https://doi.org/10.1109/TQE.2022.3164716>
- [65] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 USENIX annual technical conference (USENIX ATC 14)*. 305–319.
- [66] Sanjaya K Panda, Indrajeet Gupta, and Prasanta K Jana. 2015. Allocation-aware task scheduling for heterogeneous multi-cloud systems. *Procedia Computer Science* 50 (2015), 176–184.
- [67] Sanjaya K Panda and Prasanta K Jana. 2015. Efficient task scheduling algorithms for heterogeneous multi-cloud environment. *The Journal of Supercomputing* 71 (2015), 1505–1533.
- [68] Sanjaya K Panda and Prasanta K Jana. 2015. A multi-objective task scheduling algorithm for heterogeneous multi-cloud environment. In *2015 International Conference on Electronic Design, Computer Networks & Automated Verification (EDCAV)*. IEEE, 82–87.
- [69] Tirthak Patel, Abhay Potharaju, Baolin Li, Rohan Basu Roy, and Devesh Tiwari. 2020. Experimental Evaluation of NISQ Quantum Computers: Error Measurement, Characterization, and Implications. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15. <https://doi.org/10.1109/SC41405.2020.00050>
- [70] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.
- [71] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. 2018. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*. 1–14.
- [72] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J Love, Alán Aspuru-Guzik, and Jeremy L O'brien. 2014. A variational eigenvalue solver on a photonic quantum processor. *Nature communications* 5, 1 (2014), 4213.
- [73] Bibek Pokharel, Namit Anand, Benjamin Fortman, and Daniel A. Lidar. 2018. Demonstration of Fidelity Improvement Using Dynamical Decoupling with Superconducting Qubits. *Physical Review Letters* 121, 22 (Nov. 2018). <https://doi.org/10.1103/physrevlett.121.220502>
- [74] John Preskill. 2018. Quantum Computing in the NISQ era and beyond. *Quantum* 2 (Aug. 2018), 79. <https://doi.org/10.22331/q-2018-08-06-79>
- [75] Nils Quetschlich, Lukas Burgholzer, and Robert Wille. 2023. MQT Bench: Benchmarking Software and Design Automation Tools for Quantum Computing. *Quantum* (2023). MQT Bench is available at <https://www.cda.cit.tum.de/mqtbench/>.
- [76] Fahimeh Ramezani, Jie Lu, and Farookh Hussain. 2013. Task scheduling optimization in cloud computing applying multi-objective particle swarm optimization. In *Service-Oriented Computing: 11th International Conference, ICSOC 2013, Berlin, Germany, December 2-5, 2013, Proceedings 11*. Springer, 237–251.
- [77] Gokul Subramanian Ravi, Kaitlin N. Smith, Pranav Gokhale, and Frederic T. Chong. 2021. Quantum Computing in the Cloud: Analyzing job and machine characteristics. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*. 39–50. <https://doi.org/10.1109/IISWC53511.2021.00015>
- [78] Gokul Subramanian Ravi, Kaitlin N Smith, Prakash Murali, and Frederic T Chong. 2021. Adaptive job and resource management for the growing quantum cloud. In *2021 IEEE International Conference on Quantum Computing and Engineering (QCE)*. IEEE, 301–312.
- [79] Marie Salm, Johanna Barzen, Frank Leymann, and Benjamin Weder. 2022. Prioritization of compiled quantum circuits for different quantum computers. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 1258–1265.
- [80] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*. 351–364.
- [81] Peter W Shor. 1999. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review* 41, 2 (1999), 303–332.
- [82] Iskandar Sitdikov, David Garcia Valinas, Francisco Martin Fernandez, Iulia Zidaru, Paul Schweigert, and Akihiko Kuroda. 2023. Qiskit Serverless. <https://github.com/Qiskit/qiskit-serverless>
- [83] Kaitlin N. Smith, Joshua Viszlai, Lennart Maximilian Seifert, Jonathan M. Baker, Jakub Szefer, and Frederic T. Chong. 2023. Fast Fingerprinting of Cloud-based NISQ Quantum Computers. In *2023 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 1–12. <https://doi.org/10.1109/HOST55118.2023.10133778>
- [84] Samuel Stein, Nathan Wiebe, Yufei Ding, Peng Bo, Karol Kowalski, Nathan Baker, James Ang, and Ang Li. 2022. EQC: ensemble quantum computing for variational quantum algorithms. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 59–71.
- [85] Wei Tang and Margaret Martonosi. 2022. ScaleQC: A Scalable Framework for Hybrid Computation on Quantum and Classical Processors. arXiv:2207.00933 [cs.ET]
- [86] Swamit S Tannu and Moinuddin K Qureshi. 2019. Mitigating measurement errors in quantum computers by exploiting state-dependent bias. In *Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture*. 279–290.
- [87] Kristan Temme, Sergey Bravyi, and Jay M. Gambetta. 2017. Error Mitigation for Short-Depth Quantum Circuits. *Phys. Rev. Lett.* 119 (Nov 2017), 180509. Issue 18. <https://doi.org/10.1103/PhysRevLett.119.180509>
- [88] Caroline Tornow, Naoki Kanazawa, William E. Shanks, and Daniel J. Egger. 2022. Minimum Quantum Run-Time Characterization and Calibration via Restless Measurements with Dynamic Repetition Rates. *Phys. Rev. Appl.* 17 (Jun 2022), 064061. Issue 6. <https://doi.org/10.1103/PhysRevApplied.17.064061>
- [89] Nathaniel Tornow, Emmanouil Giortamis, and Pramod Bhatotia. 2025. QVM: Quantum Gate Virtualization Machine. *Proc. ACM Program. Lang.* 9, PLDI, Article 187 (June 2025), 26 pages. <https://doi.org/10.1145/3729290>
- [90] Nathaniel Tornow, Christian B. Mendl, and Pramod Bhatotia. 2024. Quantum-Classical Computing via Tensor Networks. arXiv:2410.15080 [quant-ph] <https://arxiv.org/abs/2410.15080>
- [91] Guido Van Rossum and Fred L. Drake. 2009. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA.
- [92] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the tenth european conference on computer systems*. 1–17.
- [93] Joel J. Wallman and Joseph Emerson. 2016. Noise tailoring for scalable quantum computation via randomized compiling. *Physical Review A* 94, 5 (Nov. 2016). <https://doi.org/10.1103/physreva.94.052325>
- [94] Jun-Bo Wang, Junyuan Wang, Yongpeng Wu, Jin-Yuan Wang, Huiling Zhu, Min Lin, and Jiangzhou Wang. 2018. A machine learning framework for resource allocation assisted by cloud computing. *IEEE Network* 32, 2 (2018), 144–151.
- [95] Meng Wang, Poulami Das, and Prashant J. Nair. 2024. Qoncord: A Multi-Device Job Scheduling Framework for Variational Quantum Algorithms. In *2024 57th*

- IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 735–749. <https://doi.org/10.1109/MICRO61859.2024.00060>
- [96] Wei Wang, Ben Liang, and Baochun Li. 2014. Multi-resource fair allocation in heterogeneous cloud computing systems. *IEEE Transactions on Parallel and Distributed Systems* 26, 10 (2014), 2822–2835.
 - [97] Yaakov S Weinstein, MA Pravia, EM Fortunato, Seth Lloyd, and David G Cory. 2001. Implementation of the quantum Fourier transform. *Physical review letters* 86, 9 (2001), 1889.
 - [98] Alexander Wieder, Parmod Bhatotia, Ansley Post, and Rodrigo Rodrigues. 2012. Orchestrating the Deployment of Computations in the Cloud with Conductor. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, San Jose, CA, 367–381. <https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/wieder>
 - [99] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. 2018. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 595–610.
 - [100] Zhen Xiao, Weijia Song, and Qi Chen. 2012. Dynamic resource allocation using virtual machines for cloud computing environment. *IEEE transactions on parallel and distributed systems* 24, 6 (2012), 1107–1117.
 - [101] Hezi Zhang, Keyi Yin, Anbang Wu, Hassan Shapourian, Alireza Shabani, and Yufei Ding. 2024. MECH: Multi-Entry Communication Highway for Superconducting Quantum Chiplets. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (La Jolla, CA, USA) (ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 699–714. <https://doi.org/10.1145/3620665.3640377>
 - [102] PeiYun Zhang and MengChu Zhou. 2018. Dynamic Cloud Task Scheduling Based on a Two-Stage Strategy. *IEEE Transactions on Automation Science and Engineering* 15, 2 (2018), 772–783. <https://doi.org/10.1109/TASE.2017.2693688>