A Digital Twin of Scalable Quantum Clouds

Waylon Luo
Kent State University
Kent, OH, USA
wluo1@kent.edu

Betis Baheri
Kent State University
Kent, OH, USA
bbaheri@kent.edu

Travis Humble
Oak Ridge National Lab
Oak Ridge, TN, USA
humblets@ornl.gov

Jiapeng Zhao
Cisco
San Jose, CA, USA
penzhao2@cisco.com

Tong Zhan
Meta
Menlo Park, CA, USA
tongzhan@meta.com

Rajan Maharjan
Kent State University
Kent, OH, USA
maharj2@kent.edu

Qiang Guan
Kent State University
Kent, OH, USA
qguan@kent.edu

ABSTRACT

Quantum computing has emerged as a transformative technology capable of solving complex problems beyond the limit of classical systems. The rapid development of quantum processors has led to the proliferation of cloud-based quantum computing services offered by platforms such as IBM, Google, and Amazon. These platforms introduce unique challenges in resource allocation, job scheduling, and multi-device orchestration as quantum workloads become increasingly complex. In this work, we present a digital twin of quantum cloud infrastructures: a framework designed to model and simulate the behavior of real quantum cloud systems. Developed in Python using the `SimPy` discrete-event simulation library, the framework replicates key aspects of quantum cloud environments, including detailed quantum device modeling, job lifecycle management, and job fidelity. It incorporates noise-aware fidelity estimation, making it the first of its kind to simulate superconducting gate-based quantum cloud systems at an administrative level with job fidelity. We present use cases as proof of concept, demonstrating that our quantum cloud simulation framework can act as a digital twin of a quantum cloud and support the modeling and implementation of practical systems.

KEYWORDS

Quantum cloud computing, digital twins, quantum job scheduling, simulation frameworks

1 INTRODUCTION

Quantum computing harnesses the unique properties of quantum mechanics to solve computational challenges that are infeasible for classical computers [10, 33]. With the rise of cloud-based quantum services from IBM, Google, and Amazon, researchers and developers now have access to quantum hardware. A quantum cloud provides remote access to quantum computing resources, allowing users to perform quantum computations without needing on-premise quantum hardware. However, with increasing demand, quantum cloud platforms encounter various challenges, including resource allocation, device-specific constraints, scheduling inefficiencies, scalability limitations, and quantum job fidelity. Implementing a digital twin alongside a quantum cloud provides a cost-effective solution to address these challenges.

A digital twin is a virtual representation of a physical object or a set of events that functions as its real-time digital equivalent. Digital twin promotes safety, affordability, and feasibility in managing complex or expensive processes for some products [41]. One can simulate design changes, process modifications, and environmental variations with digital twins. In production, digital twins incorporate real-time feedback from equipment and operators to fine-tune processes. The digital twin of a product can be utilized to optimize operations, guide decisions on ideal operating conditions, and provide potential design updates or alternative configurations.

A digital twin of a quantum cloud allows users to create, test, and optimize virtual prototypes so that researchers can analyze system behavior under various conditions before deploying a physical quantum cloud. By simulating quantum job execution, resource allocation, and device interactions, it provides insights into performance bottlenecks, scheduling strategies, and error mitigation techniques. This approach supports scalability planning and improves system design through controlled experimentation in a risk-free environment. Additionally, it aids in benchmarking quantum workloads and refining orchestration strategies to accelerate the practical adoption of quantum cloud computing.

In this work, we introduce **QCloudSim**, the first-of-its-kind implementation of a digital twin specifically tailored for quantum clouds. QCloudSim allows researchers to experiment with job management, resource allocation, and performance optimization in distributed quantum computing systems. Our framework's primary features include:

- **Orchestration:** QCloudSim is an *orchestration layer* that manages distributed quantum computing resources. It provides tools to model and simulate how quantum jobs are distributed across different quantum devices to optimize resource utilization and job throughput.
- **Flexibility:** QCloudSim provides a *flexible framework* that supports both built-in scheduling packages and user-defined scheduling and resource management strategies.
- **Noise Awareness:** QCloudSim integrates *calibration data* (e.g., from IBM Quantum [1]), which contains quantum gate errors, allowing researchers to analyze their impact on job execution and develop noise-tolerant algorithms.
- **Traceability and Logging:** The framework supports *detailed logging* and lifecycle management for quantum jobs. Users can trace jobs from arrival to completion for benchmarking and debugging purposes.
- **Performance Analysis:** QCloudSim is designed for users to test and benchmark resource allocation policies, device

configurations, and scheduling algorithms in a controlled environment.

- **Scalability:** Designed to handle *large-scale simulations*, QCloudSim supports realistic modeling of quantum cloud systems with a significant number of devices and jobs.
- **Extensibility:** The modular design of QCloudSim makes it highly extensible. Users can add new features, devices, and resource allocation strategies as quantum computing technology evolves.

The QCloudSim framework bridges the gap between theoretical quantum cloud models and practical implementations, allowing researchers to address key challenges in quantum cloud computing and resource management, explore system-level trade-offs, and develop optimized strategies for quantum cloud operations.

The remainder of the paper is structured as follows: Section 2 provides an overview of quantum cloud computing and differentiates QCloudSim from existing quantum simulation frameworks. Section 3 outlines the framework's architecture, while Section 4 details its individual components. Section 5 discusses design choices and implementation details. Section 6 presents potential use cases and experimental results. Lastly, Section 7 summarizes the key contributions of this work.

2 BACKGROUND

In quantum computing, information is encoded in quantum bits (qubits), which can exist in a superposition of basis states $|0\rangle$ and $|1\rangle$, allowing them to represent both classical values 0 and 1 simultaneously [36]. Additionally, qubits exhibit entanglement—a phenomenon in which the state of one qubit becomes intrinsically linked to the state of another, regardless of the physical distance between them. These unique properties allow quantum computers to solve certain computational problems with remarkable efficiency, surpassing the capabilities of classical systems in specific domains.

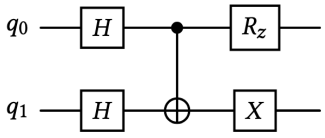


Figure 1: A simple quantum circuit consisting of two qubits with a depth of three.

A quantum cloud comprises quantum devices that process quantum jobs consisting of quantum circuits executed on quantum hardware. Fig. 1 illustrates a simple quantum circuit composed of two qubits (q_0, q_1) with a circuit depth of three. The circuit includes fundamental quantum gates such as the Hadamard gate (H), which creates superposition; the controlled-X (CNOT) gate, which establishes entanglement; the phase rotation gate (R_z), which applies a phase shift; and the Pauli-X gate, which performs a quantum bit flip. Since this work focuses on quantum cloud management rather than quantum circuit design, a detailed discussion of quantum gate operations [46] is beyond the scope of this paper.

Quantum noise is one of the most significant challenges in quantum computing. Achieving high fidelity in processing submitted jobs is crucial for customer satisfaction with quantum cloud services. Quantum devices are inherently susceptible to quantum noise arising from decoherence, gate errors, and measurement imperfections [18, 33]. To mitigate these errors, quantum cloud providers provide *calibration data*, which includes metrics such as qubit coherence times, gate fidelities, and readout errors [26, 28]. Companies such as *IBM Quantum*, *Rigetti Computing*, and *Quantinuum* publicly provide calibration data for their quantum processors. These calibration data can be used to optimize circuit execution and develop noise-aware algorithms.

Quantum computing applications span a wide range of domains, including pharmacological discovery [47], financial modeling [20], optimization, and machine learning [12]. The concept of using quantum computers to simulate complex systems, first proposed by Richard Feynman in his pioneering work [17], has made remarkable progress, bringing us closer to realizing his vision. However, quantum computing faces significant challenges, including operational complexity, strict environmental requirements, the necessity for specialized expertise, and considerable hardware costs [9].

As quantum computing technologies advance, Quantum Cloud Computing (QCC) has emerged as a practical solution for providing remote access to quantum resources. QCC allows users to run quantum algorithms on powerful quantum processors over the cloud without owning expensive physical hardware. Companies such as IBM, Amazon Braket, and Microsoft offer public quantum platforms [13, 27, 35], although these systems remain less mature compared to traditional cloud platforms such as AWS and Azure. Simulations play a critical role in bridging this gap by allowing researchers to study quantum workloads, resource allocation, and noise impacts in a cost-effective, flexible, and scalable environment.

Evgeny Mozgunov et al. explore the scientific and industrial applications of open-system quantum simulators, providing resource estimates for their implementation on superconducting qubit hardware [29]. Modeling and simulation in this context address challenges related to resource allocation and operational constraints unique to quantum cloud environments. Accurate simulation frameworks are particularly critical for addressing the high cost and limited availability of quantum hardware, providing researchers with tools to explore workload behavior, evaluate system performance, and design robust quantum cloud infrastructures.

The development of QCloudSim addresses significant gaps in existing quantum and cloud-edge simulation frameworks [38] by introducing noise-aware capabilities essential for realistic quantum cloud modeling. Earlier research has focused primarily on simulating quantum network protocols [14] and quantum states within networks [11, 24, 39]. Table 1 provides a detailed comparison of these frameworks, highlighting their capabilities and limitations. While tools such as QuNetSim and NetSquid [8, 15] emphasize network-level quantum simulation, and others like QuEST, PAS, and QXTools [5, 6, 25] focus on quantum circuit operations, these frameworks lack the features necessary for system-wide orchestration and resource management in cloud-edge quantum environments. In contrast, QCloudSim surpasses frameworks such as iQuantum [31] and QSimPy [30], which emphasize workload simulation and resource management by incorporating noise-aware simulation and real-world calibration data. This approach allows

Table 1: Comparison of QCloudSim with existing frameworks

Framework	Focus	Discrete Event	Noise Aware	Digital Twin
QuNetSim [15]	Network simulation	-	✓	✗
NetSquid [8]	Network simulation	✓	✓	✗
QuEST[25]	Circuit operation simulation	-	✗	✗
PAS [5]	Circuit operation simulation	-	✗	✗
QXTools[6]	Circuit operation simulation	-	✗	✗
iQuantum [31]	System & workload simulation	✓	✗	✗
DRAS-CQSim [16]	RL-based HPC cluster scheduling	✓	-	✗
Jawaddi et al. [23]	Classical cloud simulation	✓	-	-
QSimPy [30]	RL-based quantum cloud management	✓	✗	✗
QCloudSim (this work)	**Quantum cloud simulation**	✓	✓	✓

practical modeling of quantum devices and job fidelities. Furthermore, unlike discrete-event frameworks such as DRAS-CQSim [16] and the work by Jawaddi et al. [23], tailored specifically for classical high-performance computing, QCloudSim is explicitly designed as a digital twin for superconducting gate-based quantum clouds.

3 ARCHITECTURE

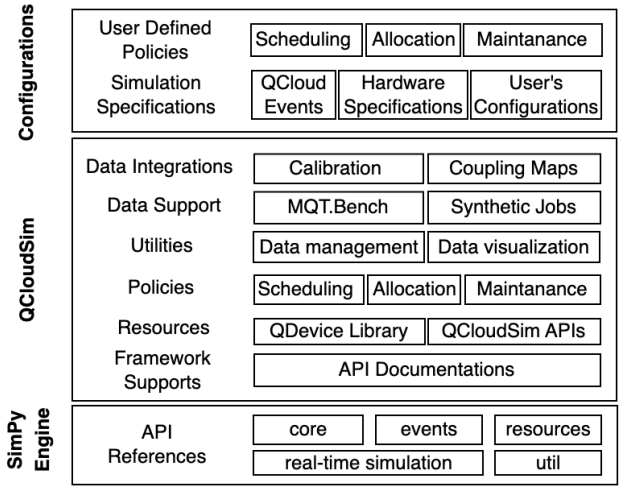


Figure 2: Layered architecture of QCloudSim, consisting of three layers: the SimPy Engine, QCloudSim for core functionalities, and Configurations for user-defined simulation policies.

The architecture of QCloudSim consists of multiple layers, as depicted in Fig. 2, each providing essential functionalities for the simulation framework. A detailed discussion of each architectural layer follows.

SimPy Engine Layer: The foundation of the architecture, the SimPy [37] layer, provides Application Programming Interfaces (APIs) for real-time simulation, concurrency, event handling, and resource management. It coordinates discrete events such as job arrivals, processing, and maintenance, supporting efficient parallel execution and scalability. SimPy's capability to synchronize simulation time with real-time allows QCloudSim to function as a digital twin.

QCloudSim Layer: The QCloudSim layer forms the core of the simulation framework, providing fundamental components and functionalities for quantum cloud simulations. It incorporates various features designed to model and analyze quantum cloud systems. QCloudSim provides user support through detailed API documentation and extensibility options. Framework resources include the `QDevice` library, which contains predefined quantum device profiles with coupling maps and other properties of superconducting quantum devices from major providers such as IBM, Google, and D-Wave. It also offers APIs for simulating core quantum cloud entities such as `QCloud`, `QDevice`, `Broker`, and `QJob`, which are discussed in Section 4. The framework includes customizable policies for scheduling, resource allocation, and maintenance, as detailed in Section 5. This flexibility allows users to simulate and test various strategies to optimize job execution across heterogeneous quantum devices. Users can extend the framework by implementing customized scheduling and resource allocation policies tailored to their specific requirements. Utilities for management and visualization support the tracking and analysis of simulation outcomes. These tools help users interpret results and refine their models based on observed behaviors.

Configurations Layer: The user configurations layer is at the top of the QCloudSim architecture. It allows users to define and implement custom policies and settings. This layer exposes user-defined scheduling and allocation policies, supporting flexible experimentation without modifying the core framework. Users can set simulation parameters such as hardware specifications, cloud events, and task allocation strategies. That makes the QCloudSim framework adaptable to various quantum cloud configurations and research requirements.

4 COMPONENTS

This section discusses the primary components and their roles within the framework layer. Key components of the architecture handle the simulation of quantum cloud functionalities, including job scheduling, resource allocation, and fidelity estimation.

`QCloudSimEnv` is central to the framework, integrating quantum devices, job generators, and brokers to manage job distribution and

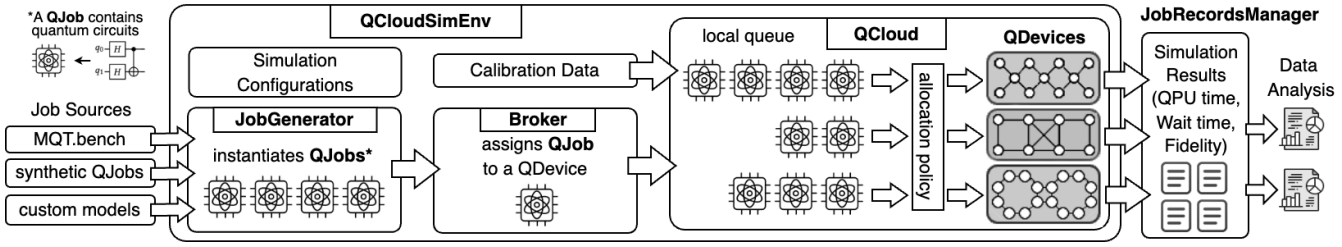


Figure 3: The architecture of the QCloudSim ecosystem. The system simulates the end-to-end orchestration of quantum jobs, beginning from job sources (e.g., MQT.Bench [34], CSV files, or custom models), through the QCloudSim environment—comprising the JobGenerator, Broker, and local queue management—to execution on quantum devices (QDevices). Simulation results are processed and visualized by the JobRecordsManager, providing insights into system performance and execution metrics.

execution. `QCloudSimEnv` is a simulation environment designed to model and test quantum cloud operations. Extended from SimPy's `Environment` class, it combines key components such as quantum devices, a job generator, and a broker to dynamically manage job scheduling and resource allocation. Upon initialization, the environment accepts a `QCloud`, a specified `Broker`, a `JobRecordsManager` for tracking job statuses, and job feed methods—supporting either random job generation using a custom inter-arrival model or predefined jobs from a CSV file. It configures a `JobGenerator` to manage job creation and distribution based on the selected feed method.

`QCloud` acts as a core component for managing quantum devices and simulating job allocation, communication, and execution in a quantum cloud environment. It is initialized within the simulation environment `QCloudSimEnv`. A `QCloud` instance maintains a list of quantum devices (`QDevices`) and uses a `JobRecordsManager` to track job lifecycles.

`QDevice` class hierarchy provides a simulation framework for quantum devices, encapsulating details about qubit topology, operational characteristics, and resource management. The `BaseQDevice` is an abstract class defining methods for job processing and performance calculations. The `QuantumDevice` subclass implements these functionalities by introducing graph-based topologies loaded from JSON files to represent qubit connectivity or coupling maps derived from reliable sources. The class supports maintenance cycles [1], customizable through intervals and durations, which simulates real-world constraints on device availability. Specialized subclasses, such as `IBM_QuantumDevice`, extend the framework to model specific IBM quantum hardware, incorporating attributes such as CLOPS (Circuit Layer Operations Per Second) [45], coherence times (`T1` and `T2`), and error rates extracted from calibration data [1].

The **`Broker`** serves as the intermediary between job requests and available quantum devices, managing device selection, resource allocation, and job execution strategies. For example, `SerialBroker` and `ParallelBroker` implementations handle these tasks by evaluating device states and coordinating the distribution of jobs across the system. Additionally, users can define their own `CustomBroker` by extending the abstract `Broker` class, allowing experimentation with custom algorithms to optimize job scheduling for specific requirements.

A **`QJob`** encapsulates the key attributes and behaviors of a quantum job, representing a unit of work submitted to the quantum computing infrastructure. Each job is defined by several parameters that determine its resource requirements and execution characteristics for a quantum circuit, as illustrated in Fig. 1. A `QJob` may include multiple quantum circuits. However, to simplify the simulation, we assume that these are abstracted into a single representative circuit per `QJob`. This abstraction supports efficient scheduling and resource allocation while preserving the core execution properties of quantum workloads.

A `QJob` instance includes the following parameters:

- `job_id`: A unique identifier for the `QJob`.
- `num_qubits`: The maximum number of qubits required.
- `depth`: The maximum depth of the quantum circuit.
- `num_shots`: The number of repetitions for the `QJob`.
- `priority`: The priority level assigned to the `QJob`.
- `arrival_time`: The time at which the `QJob` arrives.
- `gates`: A list of quantum gates used in the circuit.

By incorporating parameters such as circuit depth, qubit requirements, and priority levels, `QJob` functions as a foundational abstraction for simulating job scheduling and resource allocation in QCloudSim environments.

The **`JobGenerator`** component creates `QJobs` through three different modes:

Dynamic Generation: In the dynamic generation mode, `QJobs` are created using a user-defined job model based on selected statistical distributions. For example, the inter-arrival times between `QJobs` are modeled using an exponential distribution, as described in Section 6, and controlled by a `QJob` generation model passed as a parameter to the `JobGenerator`. This approach allows the framework to simulate varying levels of workload intensity, reflecting real-world quantum computing scenarios where `QJob` arrivals are stochastic. Each dynamically generated `QJob` is assigned attributes such as the number of shots, circuit depth, and required number of qubits, sampled from predefined ranges. By incorporating randomization, this mode supports performance evaluations under diverse operational conditions.

Predefined Dispatching: For deterministic simulation, the predefined dispatching mode allows `QJobs` to be loaded from external CSV files. This method allows researchers to replicate specific workflows or test scenarios using curated datasets. The CSV file includes fields such as job ID, number of shots, arrival time, circuit depth, and qubit requirements. The `JobGenerator` reads the data and

schedules QJobs to arrive at their specified times. If no arrival time is provided, the current timestamp is used by default. This mode is particularly useful for benchmarking, debugging, and comparing system performance under controlled, reproducible conditions. The JobGenerator also supports QJob creation from MQT.Bench datasets [34], which provide quantum circuits for benchmarking. QJobs can be derived from either a JSON file or a CSV file that contains metadata of quantum jobs.

JobRecordsManager is responsible for tracking the lifecycle of quantum jobs, logging key events, and maintaining a record of system activity. It captures job-related events, including *arrival*, *start*, *finish*, and *fidelity*. Each event is associated with a timestamp and stored in a structured format. This logging provides a granular view of job progression, supporting the analysis of system performance metrics such as waiting times, execution durations, and throughput. Researchers can use this data to identify bottlenecks, evaluate resource utilization, and assess the effectiveness of scheduling algorithms.

Calibration data offers real-time information about the performance of a quantum processor, including qubit coherence times, gate fidelities, readout errors, and other hardware metrics. These parameters are essential for assessing the reliability of quantum computations.

These components collectively define the ecosystem of the framework. As shown in Fig. 3, the framework integrates multiple components to manage the generation, allocation, execution, and analysis of quantum jobs (QJobs). Jobs can originate from various sources, including standardized benchmarks such as MQT.Bench [34], synthetic job datasets, or user-defined models. MQT Bench is a benchmarking framework for quantum circuit compilation that evaluates the performance of quantum compilers based on metrics such as fidelity, gate count, and circuit depth. These sources provide the framework users with flexibility for diverse research scenarios. The central component, QCloudSimEnv, coordinates job flow through its modular subcomponents. The JobGenerator creates QJobs with defined circuits and metadata, such as resource requirements and execution parameters. These QJobs are then passed to the Broker, which schedules them based on customizable allocation policies and forwards them to quantum devices within the QCloud. Each QDevice is configured with unique characteristics, including qubit connectivity, gate fidelity, and noise properties, and maintains a local queue for job execution. Allocation policies determine how QJobs are distributed across devices to optimize performance. The JobRecordsManager collects simulation results, including execution metrics such as QPU time, wait time, and fidelity. These results are analyzed to extract insights into system performance, helping researchers evaluate and improve quantum cloud resource management strategies.

5 DESIGN AND IMPLEMENTATION

This section outlines the primary components and methodologies implemented within the QCloudSim framework, beginning with scheduling and resource allocation strategies. It then discusses qubit allocation policies and algorithms, along with their respective computational efficiencies. The model for fidelity estimation based on calibration data is also presented. Next, the definition of QPU time is introduced. The distinction between simulation time and actual execution time is addressed in the subsequent subsection. Additionally, the integration of maintenance cycles into the framework is described. Finally, the usability of QCloudSim is illustrated through its concise and straightforward simulation setup.

5.1 Scheduling and Allocation

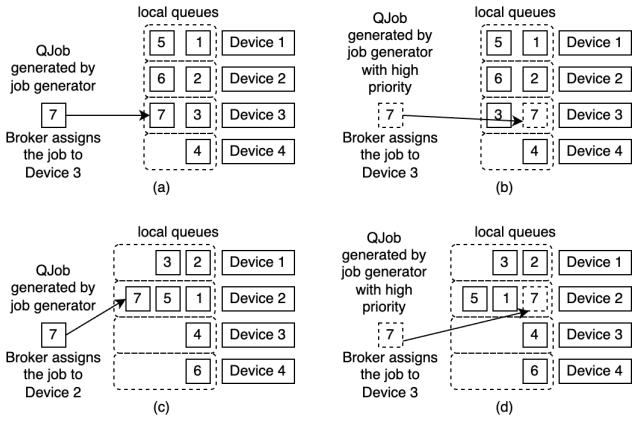


Figure 4: Different scheduling algorithms: (a) fair-share scheduling for device assignment and FIFO scheduling for local queues; (b) fair-share scheduling for device assignment and priority scheduling for local queues; (c) random scheduling for device assignment and FIFO scheduling for local queues; and (d) random scheduling for device assignment and priority scheduling for local queues.

Our framework supports two standard load-balancing methods: fair-share and randomized scheduling. In fair-share scheduling, as used by IBM Quantum Cloud [21], QJobs are assigned sequentially to devices in a round-robin manner. Randomized scheduling assigns QJobs to devices randomly, simulating a user's preferred device selection.

When a QJob is generated, the broker allocates it to a device based on the selected scheduling policy. In Fig. 4(a), fair-share scheduling assigns QJob 7 to Device 3, as it is next in the round-robin sequence. In Fig. 4(c), randomized scheduling assigns QJob 7 to Device 2, selected randomly.

Once a QJob reaches a device's local queue, it follows the default First-In-First-Out (FIFO) rule, processing tasks in their order of arrival, as shown in Fig. 4(a) and (c). Users may prioritize QJobs based on criteria such as fidelity, error rate, or circuit size. In priority scheduling, the QJob with the highest priority (i.e., the smallest assigned score) is executed first, as illustrated in Fig. 4(b) and (d). If two QJobs share the same priority, the FIFO order applies. A QJob's priority score may be determined based on circuit size, user and service agreements [22], or its wait time, as shown in Eq. (1).

$$P = (k_S \cdot S) + (k_B \cdot B) + (k_W \cdot W) + (k_C \cdot C) \tag{1}$$

where:

- $P \in \mathbb{R}$: the priority score (lower value indicates higher priority),

- $k_S \in \mathbb{R}$: weight for circuit size, where $S \in \mathbb{R}_{\geq 0}$,
- $k_B \in \mathbb{R}$: weight for business importance, where $B \in \mathbb{R}$,
- $k_W \in \mathbb{R}$: weight for wait time, where $W \in \mathbb{R}_{\geq 0}$,
- $k_C \in \mathbb{R}$: weight for adjustment factor, where $C \in \mathbb{R}$.

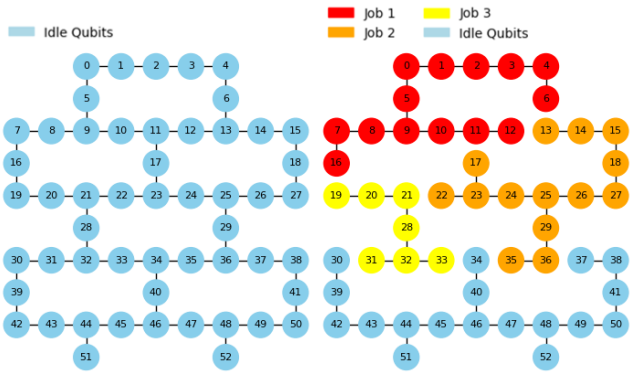
The equation is shown as an example of how a priority score can be calculated based on key factors. Users may customize or redefine the equation to suit their specific requirements by adjusting the variables, weights, or structure to align with their prioritization criteria.

5.2 Qubit Allocation Policy

Our framework supports two resource allocation strategies: serial programming and multi-programming. In the serial programming model, only one QJob is executed at a time, with all quantum resources dedicated to it until completion before proceeding to the next job. While quantum cloud platforms predominantly adopt serial programming, the concept of multi-programming in quantum computing has been widely explored in academic research [7, 43, 44] and is anticipated to become feasible in the near future.

In contrast, multi-programming supports the concurrent execution of multiple QJobs on a single quantum processor. This strategy divides resources, such as qubits, across tasks to improve resource utilization and reduce wait times. However, each QJob must be assigned qubits that form a connected subgraph. This constraint allows all required two-qubit operations to be executed without additional routing or SWAP operations [36]. Managing concurrent tasks introduces complexity, particularly in preventing interference between QJobs.

Multiprogramming on a quantum device is illustrated in Fig. 5. Part (a) shows the qubit topology of IBM Rochester [40] in its idle state, while part (b) demonstrates multiprogramming, where the red, orange, and yellow nodes represent qubits assigned to different QJobs. This partitioning preserves qubit connectivity while maintaining operational independence among tasks.



(a) Qubit topology of the device in the idle state. **(b) Qubit topology during multi-programming.**

Figure 5: Qubit topology, or coupling map, of IBM Rochester [40] in two different states.

By experimenting with scheduling models, researchers can study the impact of allocation policies on performance before deploying them on real quantum hardware. Certain restrictions apply when allocating QJobs to QDevices. The resource allocation in the simulation must satisfy two primary constraints: the number of required qubits and the connectivity of the allocated qubits.

Qubit Count Constraint: The number of qubits required by a QJob must not exceed the total number of available qubits on the selected quantum machine. Let q_i represent the number of qubits required for QJob i, and let Q represent the total number of available qubits on the quantum machine. The condition that ensures the qubit requirement for QJob i does not exceed the available capacity is given by $q_i \leq Q$.

In the case of multi-programming, this restriction is extended to accommodate multiple QJobs. Let k denote the total number of QJobs deployed on the quantum machine, and let $q_1, q_2, \ldots, q_k$ represent the number of qubits required by each respective allocated QJob on a single device. The total number of required qubits must not exceed the available qubits Q, expressed as $\sum_{i=1}^{k} q_i \leq Q$. This condition ensures that the aggregate qubit requirements of all deployed QJobs remain within the quantum machine's capacity. Consequently, each QJob is allocated only if the combined qubit demands fit within the available resources.

Qubit Connectivity Constraint. The qubits allocated to a QJob must form a connected subgraph. To model this, the qubit connectivity is represented as a graph $G(V, E)$, where V denotes the set of qubits (vertices) and E represents the set of edges corresponding to potential interactions between qubits. Let $S \subseteq V$ be the subset of qubits allocated to a specific QJob. The subgraph induced by these qubits, denoted as $G(S, E')$, must form a connected graph, where $E' \subseteq E$ includes only the edges connecting qubits in S.

5.3 Qubit Allocation Algorithm

Based on specified connectivity criteria, the qubit allocation algorithm identifies a connected subgraph of N qubits. For example, consider a quantum circuit requiring 5 qubits to be allocated on an IBM quantum device with 53 available qubits. The optimal solution minimizes the number of disconnected subgraphs remaining after qubit selection. As shown in Fig. 5(a), valid candidate subgraphs include (0-1-2-3-4), (5-0-1-2-3), and (1-2-3-4-6), among others. However, selecting (9-10-11-12-13) is not optimal, as it results in two disconnected subgraphs: (5-0-1-2-3-4-6) in one and the remaining nodes in the other.

To find the optimal qubit partition using a brute-force algorithm, one must iterate through all possible combinations of nodes. The number of such combinations is computed using the binomial coefficient, $C(n,k) = \frac{n!}{k!(n-k)!}$. For instance, selecting 10 qubits out of 53 results in $C(53, 10) = 19{,}499{,}099{,}620$, which is over 19 billion combinations. Selecting 20 qubits out of 53 results in $C(53, 20) \approx 202$ trillion combinations. This combinatorial explosion renders the brute-force approach computationally impractical for larger problems.

In our work, we trade off optimality for faster qubit selection. The function `select_vertices`, outlined in Algorithm 1, takes a qubit network graph, a color map, the required qubit count N, and a QJob identifier as input. The algorithm first filters nodes based on a specified color criterion ("skyblue") to generate a list of candidate qubits without iterating through all possible combinations. If there

are fewer than N candidates, the algorithm terminates without returning a solution. Otherwise, it performs a breadth-first search (BFS) from each candidate node to build a connected subgraph of N qubits, ensuring that the selected qubits satisfy both color and connectivity requirements. If a valid subgraph is found, the function returns the selected qubits; otherwise, it returns None. The algorithm has an average-case time complexity of $O(V^2)$, where V is the number of vertices—significantly more efficient than the brute-force approach, which has a time complexity of $O(V^N)$.

Algorithm 1 `select_vertices`: Select Connected Subgraph of N Vertices Based on Color Criteria

Require: Graph $G = (V, E)$, Color Map C, Integer N, String *name*
Ensure: A connected subgraph S of N vertices or None

```
candidate_nodes ← {v ∈ V | C(v) = 'skyblue'}
if |candidate_nodes| < N then
    return None
end if
for each start_node in candidate_nodes do
    connected_nodes ← BFS from start_node in G
    subgraph_nodes ← {start_node}
    for each (u, v) in connected_nodes do
        if |subgraph_nodes| ≥ N then
            break
        end if
        if v ∈ candidate_nodes then
            subgraph_nodes ← subgraph_nodes ∪ {v}
        end if
    end for
    if |subgraph_nodes| = N then
        return subgraph_nodes
    end if
end for
return None
```

5.4 Fidelity Estimation

In quantum computing, fidelity quantifies how closely an implemented quantum state or operation matches the intended ideal state or operation. A fidelity of 1 indicates a perfect match, while lower values reflect deviations caused by errors or noise. Noise in quantum devices is characterized using calibration data, which includes gate error rates, readout error rates, and other device-specific parameters. For IBM quantum devices, calibration data [1] includes single-qubit gate error rates, two-qubit gate error rates, and readout error probabilities of incorrectly measuring a qubit's state.

Using this calibration data, we define the *estimated fidelity* of a QJob as the product of F_{single} (single-qubit gate fidelity), F_{two} (two-qubit gate fidelity), and F_{readout} (readout fidelity):

$$F_{\text{estimated}} = F_{\text{single}} \cdot F_{\text{two}} \cdot F_{\text{readout}} \tag{2}$$

where the individual components are defined as follows:

$$F_{\text{single}} = (1 - \epsilon_{\text{single}})^d,$$
$$F_{\text{two}} = \prod_{(q_i,q_j)\in G_{\text{two}}} (1 - \epsilon_{q_i,q_j}),$$
$$F_{\text{readout}} = (1 - \epsilon_{\text{readout}})^n.$$

The term ϵ_{single} represents the error rate for single-qubit gates (e.g., rotation or Pauli gates), while ϵ_{q_i,q_j} denotes the error rate for two-qubit gates (e.g., CNOT or controlled-Z gates) applied to qubit pairs q_i and q_j. The term $\epsilon_{\text{readout}}$ represents the error rate associated with measuring individual qubits. These error rates are derived from calibration data. Detailed discussions of these error rates for single-qubit gates, two-qubit gates, and readout operations are outside the scope of this work. The set G_{two} includes all qubit pairs (q_i, q_j) involved in two-qubit gate operations within the quantum circuit. For example, if two CNOT gates are applied between qubits q_1 and q_2, and q_3 and q_4, then $G_{\text{two}} = \{(q_1, q_2), (q_3, q_4)\}$. Each q_i refers to a specific physical qubit in the quantum processor. For multi-qubit operations, the fidelity depends on both the individual qubits and their interaction characteristics. The circuit parameters include: d, the total number of single-qubit gate layers (i.e., the depth of the single-qubit operations), and n, the number of qubits involved in the readout operation. The circuit shown in Fig. 1 has a depth of three.

The expression in Eq. (2) is derived from approaches for noise modeling and error propagation in quantum circuits, as discussed in [19, 42]. For simplicity, we use the average error rate for single-qubit gates (ϵ_{single}) and readout errors ($\epsilon_{\text{readout}}$). The model assumes that gate and readout errors are independent and accumulate multiplicatively across the circuit. Two-qubit gate errors (ϵ_{q_i,q_j}) are modeled individually for each pair of qubits, based on their specific error rates from the calibration data. The calibration data represents a snapshot of the device's performance at a particular point in time. Since error rates can fluctuate, the fidelity estimate may vary in actual quantum cloud systems depending on when the QJob is executed [4].

5.5 QPU Time

The QJob's QPU time (τ), defined as the time required to execute a quantum job, is calculated based on the *CLOPS* (Circuit Layer Operations Per Second) and Quantum Volume (QV) [32] metrics used by IBM [45]. CLOPS is a performance benchmark introduced to quantify the speed of quantum computers in executing quantum circuits. QV is a metric that measures the overall capability of a quantum computer, taking into account factors such as the number of qubits, error rates, connectivity, and gate fidelity to determine the largest circuit size the device can reliably execute. The QPU time is determined using the following formula:

$$\tau = \frac{M \cdot K \cdot S \cdot D}{CLOPS} \tag{3}$$

where:

- M: The number of templates.
- K: The number of parameter updates.
- S: The number of shots required for statistical reliability.
- D: The number of QV layers, defined as $\log_2(\text{QV})$.

The numerator ($M \cdot K \cdot S \cdot D$) in Eq. (3) represents the total computational load, accounting for the number of circuits, parameter updates, repetitions, and the depth of operations. The denominator (*CLOPS*) provides a normalization factor that reflects the quantum processor's computational speed.

Consider a QJob with the following properties: $M = 100$, $K = 10$, $S = 50{,}000$, and $D = \log_2(127) \approx 7$. The values for M and K are

```
from QCloud import *

ibm_kawasaki = IBM_Kawasaki(env=None, name="ibm_kawasaki", printlog=True)
qcloudsimenv = QCloudSimEnv(devices=[ibm_kawasaki], broker_class=ParallelBroker, job_feed_method="generator",
                            job_generation_model=lambda: random.expovariate(lambd=0.1))
qcloudsimenv.run(until=100)
```

Listing 1: Sample code demonstrating the setup and execution of QCloudSim with ibm_kawasaki.

derived from [45]. The quantum processor used is *ibm_strasbourg*, which has a *CLOPS* value of 220,000 and a quantum volume of 127. Using Eq. 3, it would take approximately 26 minutes of QPU time to complete the given job.

5.6 Simulation Time versus Execution Time

We discuss two key aspects of time: *execution time* and *simulation time* (sim-time). Execution time refers to the actual time a process or task takes to complete on physical hardware. It is influenced by factors such as hardware speed, task complexity, and algorithm efficiency, and is typically measured to evaluate the performance of programs and systems. In contrast, sim-time represents the virtual timeline within a simulation environment, which advances in discrete steps as events occur. Sim-time is configurable to various units, such as seconds, minutes, or hours, depending on the simulation's scope. For example, sim-time can be scaled to seconds for a day-long job flow or to minutes and hours for a month-long simulation.

In a digital twin, where the goal is to replicate real-world processes in a virtual environment, SimPy's `RealtimeEnvironment` is particularly useful. Unlike the standard simulation environment, `RealtimeEnvironment` synchronizes simulation time with wall-clock time to advance simulation steps in real time. This feature is especially valuable in scenarios requiring interaction between the simulation and external systems or users. For example, a digital twin of a quantum cloud platform could use `RealtimeEnvironment` to mimic real-time scheduling, resource allocation, or job execution, allowing the simulation to respond dynamically to live inputs and events. By setting `RealtimeEnvironment(factor=1)`, framework users can align simulation time with physical time.

5.7 Maintenance Cycles

Maintenance cycles are supported in the QCloudSim framework to reflect the periodic downtime observed in real quantum devices. Each device can be optionally configured with parameters such as maintenance intervals and durations. The `maintenance` method in the `QuantumDevice` class schedules these as recurring events in the simulation environment. During maintenance, the device is temporarily unavailable, and the broker cannot assign new `QJobs`. Although the effects of maintenance on job throughput and scheduling are not studied in this work, this feature is included to enhance the realism of the framework and enable future exploration of such operational behaviors.

5.8 Ease of Use and Setup in QCloudSim

The QCloudSim framework is designed with simplicity and usability in mind. It allows users to set up and execute quantum-cloud simulations with minimal effort. With as few as three lines of code, users can define a quantum device, initialize the simulation environment, and run the simulation, as demonstrated in Listing 1.

6 EVALUATION

In this section, we present potential use cases and experimental scenarios that showcase the capabilities of the QCloudSim framework. To demonstrate the practicality of a digital twin for the IBM quantum cloud, we configured IBM's QPUs (`QDevices`) that were online as of January 26, 2025. One instance was initialized for each machine listed in Table 2 [2], totaling eleven machines. Calibration data for each `QDevice` was collected from the IBM Quantum Platform [2] as CSV files and imported at the beginning of the simulation to calculate gate errors for job fidelity, as discussed in Section 5.4. The simulation spans one sim-week, equivalent to 10,080 sim-minutes, providing sufficient data to observe trends in system performance and utilization.

Table 2: IBM quantum cloud and the specifications of its quantum processing units (QPUs).

QPU (QDevice)	Qubits	CLOPS	QV
ibm_kawasaki	127	29,000	127
ibm_kyiv	127	30,000	127
ibm_sherbrooke	127	30,000	127
ibm_quebec	127	32,000	127
ibm_rensselaer	127	32,000	127
ibm_brisbane	127	180,000	127
ibm_brussels	127	220,000	127
ibm_strasbourg	127	220,000	127
ibm_marrakesh	156	195,000	512
ibm_fez	156	195,000	512
ibm_torino	133	210,000	512

The evaluation employs the multiprogramming allocation policy discussed in Section 5.2, which optimizes the distribution of `QJobs` across available `QDevices`.

`QJobs` are modeled as follows: the `job_id` is incremented by one for each incoming job. The number of shots (`num_shots`) is randomly generated between 10,000 and 100,000. The circuit `depth` is randomly selected between 5 and 20, and the number of qubits (`num_qubits`) is also randomly chosen from the same range. To mimic user behavior, incoming `QJobs` are randomly assigned to the configured `QDevices`, simulating user-driven machine selection.

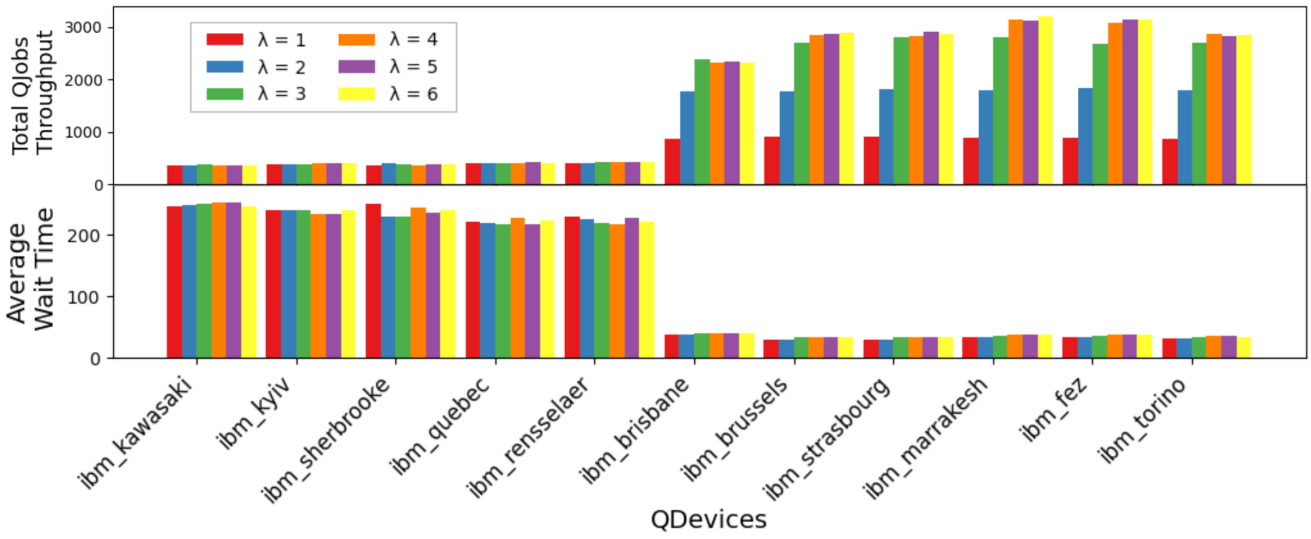


Figure 6: Comparison of total QJob throughput (top panel) and average wait time in sim-minutes (bottom panel) across different quantum devices (QDevices) over one sim-week, under varying arrival rates ($\lambda = 1$ to $\lambda = 6$).

The arrival time between QJobs is modeled using the exponential distribution function in Python, `random.expovariate(λ)`. The exponential distribution is defined by its probability density function (PDF): $f(x;\lambda) = \lambda e^{-\lambda x}$ for $x \geq 0$, where λ is the rate parameter representing the average frequency of events per unit time. A higher λ corresponds to shorter intervals between events, while a lower λ results in longer intervals. In this experiment, simulations are conducted for λ values ranging from 1 to 6. When $\lambda = 1$, it is most likely that a new QJob will be generated every sim-minute, resulting in an expected total of 10,080 jobs over one simulated week (10,080 sim-minutes). Similarly, when $\lambda = 6$, it is likely that six QJobs will be generated per sim-minute, yielding an expected total of 60,480 jobs. We assume equal priority for all QJobs in this evaluation.

Two bar plots displaying the total QJob throughput (top panel) and the average wait time in sim-minutes (bottom panel) across various QDevices are illustrated in Fig. 6. Each bar color represents a different arrival rate (λ), ranging from $\lambda = 1$ (red) to $\lambda = 6$ (yellow). The top panel of Fig. 6 shows that QDevices such as *ibm_brisbane*, *ibm_brussels*, *ibm_fez*, and *ibm_torino* achieve significantly higher QJob throughput compared to others. These devices exhibit higher CLOPS values, particularly under increased arrival rates ($\lambda \geq 3$), as indicated by their consistent gains in throughput. In contrast, devices such as *ibm_kawasaki*, *ibm_kyiv*, and *ibm_sherbrooke* display limited throughput, suggesting that lower CLOPS values may constrain resources and reduce QJob processing capacity.

The bottom panel reveals that devices such as *ibm_brisbane*, *ibm_brussels*, and *ibm_torino*, which have higher CLOPS values (180K, 220K, and 210K, respectively), exhibit consistently lower wait times across all arrival rates. This reflects their ability to process jobs more efficiently under high workloads. In contrast, devices such as *ibm_kawasaki*, *ibm_kyiv*, and *ibm_sherbrooke*, with significantly lower CLOPS values (29K, 30K, and 30K, respectively), experience higher and relatively stable wait times, indicating their limited capacity to handle increased demand.

Interestingly, *ibm_marrakesh* and *ibm_fez*, despite having a higher number of qubits (156) and moderate CLOPS values (195K), exhibit slightly higher wait times compared to *ibm_brussels* and *ibm_torino*. This suggests that CLOPS plays a more critical role than qubit count in determining job throughput and queuing efficiency. These results highlight the importance of both CLOPS and the overall resource allocation policy in optimizing performance and minimizing wait times for quantum jobs across heterogeneous devices.

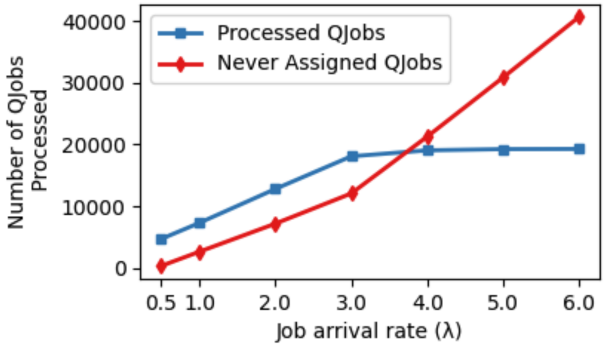


Figure 7: Number of QJobs processed and never assigned under varying job arrival rates (λ). The blue line with square markers represents the total number of processed QJobs, while the red line with diamond markers indicates the number of QJobs that were never assigned to a quantum device.

The relationship between the job arrival rate (λ) and the number of QJobs either processed or never assigned within the quantum cloud system is illustrated in Fig. 7. The blue line with square markers represents the number of processed QJobs, which increases steadily as λ rises but eventually plateaus, indicating that the system reaches its processing capacity beyond $\lambda = 4$. In contrast, the red

line with diamond markers shows the number of never-assigned QJobs, which grows gradually as the arrival rate surpasses the system's capacity.

The average fidelity across all devices was computed using a weighted approach that accounts for the standard deviation (σ) of each QJob's fidelity. First, all fidelity values and their corresponding standard deviations were aggregated across devices. The variance for each measurement was computed as the square of its standard deviation, σ^2, and the corresponding weight was defined as the inverse of the variance, $w = 1/\sigma^2$. The weighted average fidelity was then calculated using the following formula:

$$\text{Weighted Average Fidelity} = \frac{\sum_i w_i \cdot x_i}{\sum_i w_i},$$

where x_i is the fidelity value and w_i is its corresponding weight. Finally, the overall standard deviation, σ, was calculated as:

$$\sigma = \sqrt{\frac{1}{\sum_i w_i}}.$$

This standard deviation reflects the combined uncertainty of the weighted average. Using this method, a weighted average fidelity of 0.7508 was obtained, with an overall standard deviation of 0.0098.

Each experiment was run 10 times to ensure statistical reliability. The largest relative standard deviation (RSD) for total job throughput was 1.67%, and the largest RSD for wait time was 3.87%. These RSD values indicate consistent performance across simulation results with minimal variability.

In the following experiment, we evaluate the performance of a deterministic job batch executed on varying numbers of devices. The jobs are dispatched from a predefined .csv file containing 50,000 QJobs, each modeled using the parameters discussed earlier in this section. The job batch is distributed across a range of *ibm_strasbourg* devices, with the number of devices varying from 1 to 7. The *ibm_strasbourg* device is known for its high CLOPS performance and features 127 qubits. Using the same device model across all configurations ensures consistency, control, and fairness in the experiment by eliminating variations caused by hardware differences. This approach allows the focus to remain on scalability and parallelism. It simplifies resource allocation and provides a controlled environment to evaluate the performance of a specific machine. However, in practice, creating completely identical quantum processing units is infeasible due to manufacturing variability and differences in hardware calibration. Thus, this setup serves as a theoretical simplification for experimental purposes. The relationship between the number of machines and the corresponding simulation time (in hours) is illustrated in Fig. 8. As the number of machines increases, the simulation time decreases significantly, demonstrating an inverse relationship. This behavior aligns with the principle of parallel processing, where distributing tasks across multiple machines reduces the computational load per machine and improves overall efficiency.

For instance, adding one machine to an existing setup reduces simulation time markedly—from approximately 2900 hours to 1460 hours—indicating substantial efficiency gains at the early stages. However, as the number of machines increases beyond four or five, the reduction in simulation time becomes less pronounced,

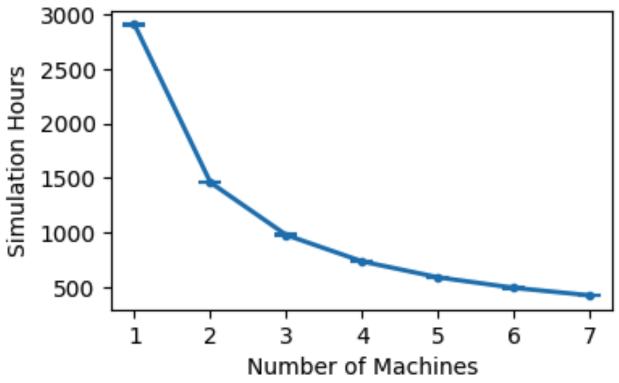


Figure 8: Simulation time to process 50,000 QJobs versus the number of *ibm_strasbourg* machines.

suggesting diminishing returns. This observation highlights that while initial increases in the number of machines enhance efficiency, marginal gains diminish beyond a certain threshold and may not justify the additional cost and complexity.

Each experiment was executed 10 times, and the average results were recorded for consistency. The highest relative standard deviation (RSD) observed for simulation hours was 0.58%. All simulations were conducted on a system equipped with an Apple M1 Max chip, featuring a 10-core CPU and 32 GB of unified memory. The total runtime for all experiments was approximately 132 minutes.

7 CONCLUSION

We presented this work as a proof of concept demonstrating that a quantum cloud can be modeled as a digital twin. By incorporating noise-aware integration and real-world calibration data, the framework supports simulations that capture the impact of noise and hardware imperfections on quantum workloads. By simulating multiple *what-if* scenarios, the QCloudSim digital twin provides researchers and developers with a flexible environment to design, evaluate, and optimize quantum cloud infrastructures with deeper insight. In addition, the framework supports the exploration of service fee models, assisting providers in analyzing pricing strategies based on job execution time, hardware noise characteristics, and scheduling priorities. Overall, it addresses the complex demands of modern quantum technologies while incorporating noise-aware insights, laying a foundation for advancing quantum computing research and shaping the future of quantum cloud systems for next-generation applications and innovations.

CODE AND DATA AVAILABILITY

The source code for the simulations and experiments presented in this study is publicly available at the GitHub repository [3].

ACKNOWLEDGMENTS

This work was supported by NSF under grant #2238734, 2311950, 2230111. We would like to acknowledge the QCUP program at ORNL for providing the computational resources and infrastructure.

REFERENCES

[1] [n. d.]. About Calibration Jobs — docs.quantum.ibm.com. https://docs.quantum.ibm.com/admin/calibration-jobs.

[2] [n. d.]. IBM Quantum Processing Units. https://quantum.ibm.com/services/resources. Accessed: 2025-01-26.

[3] 2025. Quantum Cloud Simulation: A Digital Twin. https://github.com/quantumcloudsim/SigSim2025.git. https://doi.org/10.5281/zenodo.15272987 Submitted to ACM SIGSIM-PADS 2025.

[4] Héctor Abraham, Reginald AduOffei, Luciano Bello, Ching-Yun Chen, and Others. [n. d.]. The Qiskit Textbook: Quantum Noise. https://qiskit.org/textbook/ch-quantum-hardware/error-mitigation-recovery.html. Accessed: 2025-01-13.

[5] H. Bian, J. Huang, J. Tang, R. Dong, L. Wu, and X. Wang. 2023. PAS: A new powerful and simple quantum computing simulator. *Software: Practice and Experience* 53, 1 (2023), 142–159.

[6] John Brennan, Lee J O'Riordan, K. G. Hanley, Myles Doyle, Momme Allalen, David Brayford, Luigi Iapichino, and Niall Moran. 2022. QXTools: A Julia framework for distributed quantum circuit simulation. *J. Open Source Softw.* 7 (2022), 3711. https://api.semanticscholar.org/CorpusID:246886257

[7] S. Catrina and A. Băicoianu. 2024. Quantum Tunneling: From Theory to Error-Mitigated Quantum Simulation. *Advanced Quantum Technologies* (2024). https://onlinelibrary.wiley.com/doi/abs/10.1002/qute.202400163

[8] Tim Coopmans, Robert Knegjens, Axel Dahlberg, David Maier, Loek Nijsten, Julio de Oliveira Filho, Martijn Papendrecht, Julian Rabbie, Filip Rozpędek, Matthew Skrzypczyk, Leon Wubben, Walter de Jong, Damian Podareanu, Ariana Torres-Knoop, David Elkouss, and Stephanie Wehner. 2020. NetSquid, a NETwork Simulator for QUantum Information using Discrete events. *Communications Physics* 4 (2020), 1–15. https://api.semanticscholar.org/CorpusID:235967111

[9] A. D. Córcoles, Maika Takita, Ken Inoue, Scott Lekuch, Zlatko K. Minev, Jerry M. Chow, and Jay M. Gambetta. 2021. Exploiting Dynamic Quantum Circuits in a Quantum Algorithm with Superconducting Qubits. *Phys. Rev. Lett.* 127 (Aug 2021), 100501. Issue 10. https://doi.org/10.1103/PhysRevLett.127.100501

[10] Andrew W. Cross, Ali Javadi-Abhari, Thomas Alexander, Niel de Beaudrap, Lev S. Bishop, Steven Heidel, Colm A. Ryan, Prasahnt Sivarajah, John A. Smolin, Jay M. Gambetta, and Blake R. Johnson. 2021. OpenQASM 3: A Broader and Deeper Quantum Assembly Language. *ACM Transactions on Quantum Computing* 3 (2021), 1 – 50. https://api.semanticscholar.org/CorpusID:233476587

[11] Axel Dahlberg and Stephanie Wehner. 2018. SimulaQron—a Simulator For Developing Quantum Internet Software. *Quantum Science and Technology* 4, 1 (sep 2018), 015001. https://doi.org/10.1088/2058-9565/aad56e

[12] Erik P. DeBenedictis. 2018. A Future with Quantum Machine Learning. *Computer* 51, 2 (2018), 68–71. https://doi.org/10.1109/MC.2018.1451646

[13] Simon J. Devitt. 2016. Performing Quantum Computing Experiments In The Cloud. *Phys. Rev. A* 94 (Sep 2016), 032329. Issue 3. https://doi.org/10.1103/PhysRevA.94.032329

[14] Stephen Diadamo, Janis Nötzel, Simon Sekavcnik, Riccardo Bassoli, Roberto Ferrara, Christian Deppe, Frank H. P. Fitzek, and Holger Boche. 2021. Integrating Quantum Simulation for Quantum-Enhanced Classical Network Emulation. *IEEE Communications Letters* 25 (2021), 3922–3926. https://api.semanticscholar.org/CorpusID:238259031

[15] S. Diadamo, J. Notzel, B. Zanger, and M. M. Bese. 2021. QuNetSim: A Doftware Framework For Quantum Networks. *IEEE Transactions on Quantum Engineering* 2 (2021), 1–12.

[16] Y. Fan and Z. Lan. 2021. DRAS-CQSim: A Reinforcement Learning Based Framework for HPC Cluster Scheduling. *Software Impacts* 8 (may 2021), 100077.

[17] Richard P Feynman. 2018. Simulating physics with computers. In *Feynman and computation*. cRc Press, 133–153.

[18] Jay M. Gambetta, Jerry M. Chow, and Matthias Steffen. 2017. Building Logical Qubits in a Superconducting Quantum Computing System. *Nature Physics* 13 (2017), 1050–1056. https://doi.org/10.1038/nphys4118

[19] Jay M. Gambetta, Antonio D. Córcoles, Seth T. Merkel, John A. Smolin, Jerry M. Chow, Chad Rigetti, Stefano Poletto, Britton L.T. Plourde, Matthias Steffen, and Blake R. Johnson. 2012. Characterization of Addressability by Simultaneous Randomized Benchmarking. *Physical Review Letters* 109, 24 (2012), 240504. https://doi.org/10.1103/PhysRevLett.109.240504

[20] Paul Griffin and Ritesh Sampat. 2021. Quantum Computing for Supply Chain Finance. In *2021 IEEE International Conference on Services Computing (SCC)*. 456–459. https://doi.org/10.1109/SCC53864.2021.00066

[21] IBM Quantum. 2025. Fair-Share Scheduler Documentation. Available online at https://docs.quantum.ibm.com/guides/fair-share-scheduler.

[22] IBM Quantum. 2025. Managing Allocations for Quantum Resources. Available online at https://docs.quantum.ibm.com/admin/manage-allocation.

[23] S. N. Agos Jawaddi and A. Ismail. 2024. Integrating OpenAI Gym and CloudSim Plus: A Simulation Environment for DRL Agent Training in Energy-Driven Cloud Scaling. *Simulation Modelling Practice and Theory* 130 (jan 2024), 102858.

[24] J.R. Johansson, P.D. Nation, and Franco Nori. 2013. QuTiP 2: A Python Framework For The Dynamics Of Open Quantum Systems. *Computer Physics Communications* 184, 4 (2013), 1234–1240. https://doi.org/10.1016/j.cpc.2012.11.019

[25] Tyson Jones, Anna Brown, Ian Bush, and Simon C Benjamin. 2019. QuEST and High Performance Simulation of Quantum Computers. *Sci Rep* 9, 1 (2019), 10736.

[26] Abhinav Kandala, Kristan Temme, Antonio D. Córcoles, Antonio Mezzacapo, Jerry M. Chow, and Jay M. Gambetta. 2019. Error Mitigation Extends the Computational Reach of a Noisy Quantum Processor. *Nature* 567 (2019), 491–495. https://doi.org/10.1038/s41586-019-1040-7

[27] Frank Leymann, Johanna Barzen, Michael Falkenthal, Daniel Vietz, Benjamin Weder, and Karoline Wild. 2020. Quantum in the Cloud: Application Potentials and Research Opportunities. In *International Conference on Cloud Computing and Services Science*. https://api.semanticscholar.org/CorpusID:212717763

[28] David C. McKay, Thomas Alexander, Luciano Bello, Michael J. Biercuk, Lev Bishop, Jiayin Chen, Jerry M. Chow, Antonio D. Córcoles, Daniel Egger, Stefan Filipp, Juan Gomez, Michael Hush, Ali Javadi-Abhari, Diego Moreda, Paul Nation, Brent Paulovicks, Erick Winston, Christopher J. Wood, James Wootton, and Jay M. Gambetta. 2019. Qiskit Backend Specifications for OpenQASM and OpenPulse Experiments. *arXiv* (2019). arXiv:arXiv:1809.03452 https://arxiv.org/abs/1809.03452

[29] Evgeny Mozgunov. 2024. Applications and Resource Estimates for Open System Simulation on a Quantum Computer. *arXiv preprint arXiv:2406.06281* (2024).

[30] Hoa T Nguyen, Muhammad Usman, and Rajkumar Buyya. 2024. QSimPy: A Learning-centric Simulation Framework for Quantum Cloud Resource Management. *arXiv preprint arXiv:2405.01021* (2024).

[31] Nguyen, Hoa T. and Usman, Muhammad and Buyya, Rajkumar. 2023. iQuantum: A Case for Modeling and Simulation of Quantum Computing Environments. In *2023 IEEE International Conference on Quantum Software (QSW)*. 21–30. https://doi.org/10.1109/QSW59989.2023.00013

[32] Elijah Pelofske, Andreas Bärtschi, and Stephan Eidenbenz. 2022. Quantum volume in practice: What users can expect from nisq devices. *IEEE* 3 (Jun 2022), 1–19.

[33] 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

[34] Nils Quetschlich, Lukas Burgholzer, and Robert Wille. 2023. MQT Bench: Benchmarking Software and Design Automation Tools for Quantum Computing. *Quantum* (2023). https://www.cda.cit.tum.de/mqtbench/ MQT Bench is available at https://www.cda.cit.tum.de/mqtbench/.

[35] Gokul Subramanian Ravi, Kaitlin N. Smith, Pranav Gokhale, and Frederic T. Chong. 2022. Quantum Computing in the Cloud: Analyzing Job and Machine Characteristics. arXiv:2203.13121 [quant-ph]

[36] Eleanor G. Rieffel and Wolfgang H. Polak. 2011. *Quantum Computing: A Gentle Introduction*. MIT Press. https://mitpress.mit.edu/9780262015066/quantum-computing/

[37] SimPy. 2024. Discrete event simulation for Python. https://simpy.readthedocs.io/en/latest/index.html.

[38] Cagatay Sonmez, Atay Ozgovde, and Cem Ersoy. 2017. EdgeCloudSim: An Environment for Performance Evaluation of Edge Computing Systems. In *2017 Second International Conference on Fog and Mobile Edge Computing (FMEC)*. 39–44. https://doi.org/10.1109/FMEC.2017.7946405

[39] Damian Steiger, Thomas Häner, and Matthias Troyer. 2016. ProjectQ: An Open Source Software Framework for Quantum Computing. *Quantum* 2 (12 2016). https://doi.org/10.22331/q-2018-01-31-49

[40] Yuki Takeuchi, Yasuhiro Takahashi, Tomoyuki Morimae, and Seiichiro Tani. 2022. Divide-and-conquer Verification Method for Noisy Intermediate-scale Quantum Computation. *Quantum* 6 (2022), 758.

[41] Fei Tao, He Zhang, Ang Liu, and A. Y. C. Nee. 2019. Digital Twin in Industry: State-of-the-Art. *IEEE Transactions on Industrial Informatics* 15, 4 (2019), 2405–2415. https://doi.org/10.1109/TII.2018.2873186

[42] Kristan Temme, Sergey Bravyi, and Jay M. Gambetta. 2017. Error Mitigation for Short-Depth Quantum Circuits. *Physical Review Letters* 119, 18 (2017), 180509. https://doi.org/10.1103/PhysRevLett.119.180509

[43] H. Thapliyal and T. Humble. 2024. *Quantum Computing*. Springer. https://link.springer.com/content/pdf/10.1007/978-3-031-37966-6.pdf

[44] S. Upadhyay and S. Ghosh. 2024. SHARE: Secure Hardware Allocation and Resource Efficiency in Quantum Systems. *arXiv preprint arXiv:2405.00863* (2024). https://arxiv.org/abs/2405.00863

[45] Andrew Wack, Hanhee Paik, Ali Javadi-Abhari, Petar Jurcevic, Ismael Faro, Jay Gambetta, and Blake Johnson. 2021. Quality, Speed, and Scale: Three Key Attributes To Measure The Performance Of Near-Term Quantum Computers. (10 2021).

[46] Colin P. Williams. 2011. *Explorations in Quantum Computing* (2nd ed.). Springer. https://link.springer.com/book/10.1007/978-1-84628-887-6

[47] Maximillian Zinner, Florian Dahlhausen, Philip Boehme, Jan Ehlers, Linn Bieske, and Leonard Fehring. 2021. Quantum Computing's Potential For Drug Discovery: Early Stage Industry Dynamics. *Drug Discovery Today* 26, 7 (2021), 1680–1688. https://doi.org/10.1016/j.drudis.2021.06.003