Future Generation Computer Systems 154 (2024) 281–300

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

QFaaS: A serverless function-as-a-service framework for quantum computing

Hoa T. Nguyen [a,*], Muhammad Usman [b,c], Rajkumar Buyya [a]

[a] Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne, Parkville, 3052, Victoria, Australia

[b] School of Physics, The University of Melbourne, Parkville, 3052, Victoria, Australia

[c] Data61, CSIRO, Clayton, 3168, Victoria, Australia

ARTICLE INFO

Keywords:
Quantum serverless
Quantum function-as-a-service
Quantum software engineering
Hybrid quantum-classical computing
Quantum DevOps
Quantum cloud computing

ABSTRACT

Quantum computing is rapidly reaching a point in which its application design and engineering aspects must be seriously considered. However, quantum software engineering is still in its infancy, with numerous challenges, especially in dealing with the diversity of quantum programming languages and noisy intermediate-scale quantum (NISQ) systems. To alleviate these challenges, we propose QFaaS, a holistic **Q**uantum **F**unction-**a**s-**a**-**S**ervice framework, which leverages the advantages of the serverless model, DevOps lifecycle, and the state-of-the-art software techniques to advance practical quantum computing for next-generation application development in the NISQ era. Our framework provides essential elements of a serverless quantum system to streamline service-oriented quantum application development in cloud environments, such as combining hybrid quantum–classical computation, automating the backend selection, cold start mitigation, and adapting DevOps techniques. QFaaS offers a full-stack and unified quantum serverless platform by integrating multiple well-known quantum software development kits (Qiskit, Q#, Cirq, and Braket), quantum simulators, and cloud providers (IBM Quantum and Amazon Braket). This paper proposes the concept of quantum function-as-a-service, system design, operation workflows, implementation of QFaaS, and lessons learned on the benefits and limitations of quantum serverless computing. We also present practical use cases with various quantum applications on today's quantum computers and simulators to demonstrate our framework capability to facilitate the ongoing quantum software transition.

1. Introduction

Recent breakthroughs in quantum hardware development are creating opportunities for its use in many applications, making it becoming a critical future technology attracting significant investment at the global level [1]. The rapid advancements in quantum hardware trigger more investments in quantum software engineering and quantum algorithms development to maximize the practical use of quantum computers. Quantum computers have demonstrated their abilities to solve many complex problems which are challenging to tackle with classical supercomputers, such as molecule simulations [2], machine learning [3], cryptography [4], and finances [5]. Some notable algorithms have been proposed in the last few decades, such as Deutsch-Jozsa's [6], Shor's [7], and Grover's [8]. Some of these algorithms have been directly applied to problems of practical relevance, albeit at the proof-of-concept level, due to quantum hardware limitations.

Despite the inevitable prospect of quantum computing for future-generation computation, quantum software engineering is an early-emerging domain with numerous open challenges. First, the development of quantum applications is complicated and time-consuming for software engineers, mainly because of the requirement for prior quantum knowledge. Indeed, quantum programming is underpinned by the principles of quantum mechanics, which are quite different from the traditional models. For example, the basic difference between quantum and classical computing comes from their fundamental unit: a classical bit has one state, either 0 or 1, whereas a quantum bit (or qubit) could also be placed in a *superposition* state, i.e., a combination state of 0 and 1 [9]. A software engineer must overcome the hurdle of learning quantum mechanics to develop quantum applications.

Second, quantum computing services are still heavily relying on classical servers for circuit compilation due to the lack of quantum data storage methods in a quantum computer. In classical computing, the compiled binaries of applications can be installed or deployed on persistent storage mediums, allowing them to remain available for re-execution without the need to recompile the code for each use. By contrast, quantum computing services currently do not possess an analogous capability for persistent storage of quantum programs. Consequently, when quantum computation is invoked, it necessitates

* Corresponding author.
E-mail addresses: thanhhoan@student.unimelb.edu.au (H.T. Nguyen), muhammad.usman@data61.csiro.au (M. Usman), rbuyya@unimelb.edu.au (R. Buyya).

https://doi.org/10.1016/j.future.2024.01.018
Received 21 June 2023; Received in revised form 12 January 2024; Accepted 14 January 2024
Available online 15 January 2024

a classical driver to compile a quantum circuit tailored to the specific quantum processor [10]. This circuit is then loaded into the quantum processing unit (QPU) for execution, with the outcome derived from one or more iterations (shots). Additionally, classical computing resources can be requisitioned for post-processing and storing the results of the quantum execution.

Besides, the existence of many quantum software development kits (SDKs) tightly coupled with specific vendor platforms presents challenges such as data lock-in and limitations in migrating software deployments across various platforms. Each SDK and programming language has distinct requirements for environment configuration, syntax, and interaction methods with their respective quantum simulators and computers. Additionally, there is no well-known standard or lifecycle in quantum software engineering similar to practices like Agile and DevOps in the traditional realm [11]. Although some efforts have been made to deal with this issue, such as preliminary approximations based on Model-Driven Engineering (MDE) [12], a comprehensive solution to establish a unified quantum software platform capable of seamlessly working with multiple quantum SDKs and providers remains necessary.

Presently, quantum computing resources remain constrained within the noisy intermediate-scale quantum (NISQ) era [13], characterized by limitations in both the quantity and quality of available qubits. Also, access to quantum computing services is exclusively facilitated through cloud-based platforms, which often incur substantial costs. Indeed, the most widely adopted way to access today's quantum computers is through a cloud service from external vendors, such as IBM Quantum [14], Amazon Braket [15], and Azure Quantum [16]. To ensure a mutually advantageous relationship between quantum cloud providers and clients, it is crucial to establish a win-win paradigm that maximizes the benefits of quantum computing while optimizing both budgetary and quantum resource considerations. In this context, the current pay-per-use pricing model offered by cloud vendors must be complemented by an appropriate computing model that effectively balances the advantages for both parties involved.

Furthermore, the ongoing evolution of new computational paradigms raises a strategic decision in the transition towards incorporating quantum computing: determining the extent of integration within existing classical systems. It is currently impractical to envision an entire replacement of quantum systems for all computational tasks due to efficiency considerations. For example, basic arithmetic operations, such as summing two integers, are far more efficiently executed on classical computers. Thus, the foreseeable future points to a hybrid approach, where quantum computing is employed for specific, complex problems that are beyond the capabilities of classical computers, while routine tasks remain with classical solutions. This integration strategy is particularly applicable during the NISQ era, where leveraging both technologies' strengths is essential for optimizing performance [17].

1.1. Our contributions

To address the research challenges highlighted above, we propose **QFaaS** - a novel and versatile ***Q**uantum **F**unction-**a**s-**a**-**S**ervice* framework without the vendor lock-in problem. The major contributions and novelty of our proposed work are:

- QFaaS framework represents a holistic serverless framework for quantum computing, enabling the seamless integration of quantum computation within established classical systems. We tackle the challenges associated with platform and data lock-in in serverless quantum computing by incorporating multiple quantum SDKs, namely Qiskit, Cirq, Q#, and Braket, to perform the hybrid computation on classical computers, quantum simulators, and quantum computers provided by multiple cloud vendors, including IBM Quantum and Amazon Braket.
- We introduce the concept of quantum functions, quantum function-as-a-service, operation workflows, which involve both classical and quantum computation, and discuss their benefits for quantum software development. Our framework provides a comprehensive reference for quantum software engineers and industries to design and develop their service-oriented quantum computing platforms.
- We propose the practical quantum serverless system architecture with six extendable system layers, a core API set, and a quantum function programming library. In addition to the main framework architecture, we propose an adaptive quantum backend selection policy that determines the most appropriate quantum computation system for executing the quantum function. Besides, we present a caching-based policy to mitigate the cold start problem, which helps to reduce the latency of quantum function invocation. We utilize the state-of-the-art software and system techniques for quantum software development, such as containerization and DevOps lifecycle. By leveraging Kubernetes as the underlying orchestration, our framework is portable and scalable for further advancement.
- We empirically validate the proposed framework through the practical implementation of all its components, following an open-source-oriented approach. Additionally, we showcase two sample operational workflows within the system, catering to both quantum software engineers and end-users. These workflows demonstrate how our framework can effectively support the development and utilization of hybrid quantum–classical applications.
- We conduct thorough experiments using various quantum algorithms on different quantum simulators and quantum computers to evaluate the performance of our framework. Through this evaluation, we provide practical insights into the current state of NISQ devices and discuss the limitations and lessons learned from the serverless quantum computing model.

The rest of the paper is organized as follows: Section 2 introduces the current state-of-the-art in quantum software engineering, the quantum computing as a service (QCaaS) model, and serverless computing. Section 3 proposes the concept of quantum functions and quantum function-as-a-service. Then, Section 4 presents the details of the QFaaS framework, including system architecture and main components. Section 5 describes the design and implementation of the QFaaS framework. Then, Section 6 demonstrates the operation and validation of QFaaS with practical use cases. We discuss the benefits of using QFaaS for quantum service-oriented application development and lessons learned on the limitations of the quantum serverless approach in Section 7. Section 8 discusses the related work and compares our framework's advantages with existing work. Finally, we conclude and present our plan for future work in Section 9.

2. Background

This section outlines the state-of-the-art development of quantum software engineering, quantum computing service model, and serverless quantum computing. A brief introduction to quantum computing and gate-based quantum model for broad readers can be found in Appendix A and other well-known books such as [9,18]. It is important to note that our focus in this work is on gate-based quantum computing SDKs and platforms due to their broad applicability and active development by many well-known quantum cloud providers, such as IBM Quantum [14] and Amazon Braket [15].

2.1. Quantum SDKs and programming languages

Some popular quantum software development kits (SDKs) and programming languages that originated from well-known companies are:

- **Qiskit** [19] (by IBM) is one of the most popular Python-based open-source SDKs for developing gate-based quantum programs. It offers a wide range of additional libraries and support tools, particularly tailored to the IBM Quantum platform [14].
- **Cirq** [20] (by Google) is a prevalent open-source SDK for quantum programming. This SDK supports writing, manipulating, and optimizing quantum gate-based circuits. Cirq programs can run on built-in simulators and Google's quantum processors.
- **Q#** [21] (by Microsoft) is a new programming language from Microsoft for developing and executing quantum algorithms. It comes along with Microsoft's Quantum Development Kit, which includes a set of toolkits and libraries for quantum software development.
- **Braket** [15] (by Amazon) is an emerging Python-based SDK to interact with Amazon Braket service [15]. This SDK provides multiple ways to prototype and develop hybrid quantum applications, then run them on simulators or quantum computers.

Besides, there are numerous quantum languages and SDKs proposed by research groups over the world, such as Forest and pyQuil by Rigetti [22], Strawberry Fields [23] and PennyLane [24] by Xanadu, Quingo [25], QIRO [26], and qcor [27].

2.2. Current state of quantum computing: The NISQ era

John Preskill proposed the *"Noisy Intermediate-Scale Quantum (NISQ)"* term in 2018 [13] to describe the current state of quantum computers. This term indicates two characteristics of today's quantum devices, including *"noisy"*, i.e., unstable and error-prone quantum state due to the affection of various environmental actions, and *"intermediate scale"*, i.e., the quantum volume is at the intermediate level, with about a few tens of qubits [11]. Due to the NISQ nature, the typical pattern for developing today's quantum programs combines quantum and classical parts [11]. In this hybrid model, the classical components are mainly used to pre-process and post-process the data. In contrast, the remaining part is sent to quantum computers for computation. The quantum execution parts are repeated many times and measure the average values to mitigate the error caused by the noisy quantum environment. An example of the hybrid quantum–classical model is Shor's algorithm [7] to find prime factors of integer numbers. In this algorithm, we execute the period-finding part, leveraging the Quantum Fourier Transform on quantum computers and then performing the classical post-process to measure the prime factors based on the outcome of the quantum part.

2.3. Quantum computing as a service (QCaaS)

Today's quantum computers are made available to the industry and research community as a cloud service by a quantum cloud provider [10]. This scheme is well known as Quantum Computing as a Service (QCaaS or QaaS) [28], which corresponds with well-known paradigms in cloud computing such as Platform as a Service (PaaS) or Infrastructure as a Service (IaaS). In terms of QCaaS, software engineers can develop quantum programs and send them to quantum cloud providers to execute those programs on appropriate hardware. After finishing the computation, the users only need to pay for the actual execution time of the quantum program (pay-per-use model). In this way, QCaaS is an efficient way that optimizes the user's budget for using quantum services and the provider's resources. Many popular cloud providers nowadays offer quantum computing services using their quantum hardware, such as IBM Quantum [14], which is publicly accessible for everyone in their early phase. Besides, other quantum computing services (such as Amazon Braket [15], and Azure Quantum [16]) collaborate with other hardware companies such as D-Wave, Rigetti, and IonQ to provide commercial services. However, this paradigm still faces many challenges before solving real-world applications due to the limitations of today's NISQ computers [13]. These devices have a small number of qubits that are error-prone and limited in capabilities. Therefore, improving the quality and quantity of qubits for quantum computers will accelerate the QCaaS model and quantum software development.

2.4. Serverless computing and function as a service (FaaS)

In the classical computing domain, serverless is the state-of-the-art computing model, which can be considered as a second phase for cloud computing [29]. This computing model fits with modern software architecture, especially the microservice applications, where the overall application is decomposed into multiple small and independent modules [30]. The serverless computing concept generally incorporates both Function-as-a-Service (FaaS) and Backend-as-a-Service (BaaS) models. FaaS refers to the stateless ephemeral function model where a function is a small and single-purpose artifact with few lines of programming code. BaaS is a concept to describe serverless-based file storage, database, streaming, and authentication services.

As FaaS is a subset of the serverless model, its main objective is to provide a concrete and straightforward way to implement software compared with traditional monolith architecture. FaaS allows the software engineer to focus only on coding rather than environmental setup and infrastructure deployment. A function can be triggered by a database, object storage, or deployed as a REST API and accessed via an HTTP connection. Functions also need to be scalable, i.e., automatically scaling in when idle and scaling out when the request demand increases. In this way, a FaaS platform can be an efficient way to optimize the resources for providers and reduce costs for customers. There are numerous open-source FaaS platforms in the cloud-native landscape, such as OpenFaaS, OpenWhisk, Kubeless, Knative, and many commercial platforms such as AWS Lambda, Azure Functions, Google Cloud Functions [31].

3. Quantum serverless and quantum function-as-a-service

3.1. Serverless quantum computing

A serverless quantum computing model is a viable solution for effectively utilizing contemporary quantum computers. Each quantum device, characterized by inherently limited resources, is made accessible globally via quantum cloud services. Indeed, by decomposing a monolith application into multiple single-purpose functions, we can distribute them to various backend devices. Furthermore, we can implement a hybrid quantum–classical model by combining quantum functions and classical functions in a unified application. This approach can leverage the power of existing quantum computers to facilitate new promising techniques, such as hybrid quantum–classical machine learning [32].

The adaptation of the serverless model to quantum computing must account for key differences in deployment and execution when compared to traditional computing services. In classical computing, a service can be deployed once to a server, whether physical or virtual, and then it can be repeatedly invoked by end-users. This permanent deployment is not yet feasible with current quantum computing technology., i.e., a quantum program cannot be deployed persistently in a specific quantum computer [10]. Instead, an appropriate quantum circuit needs to be built every time we execute a specific task. Then, that circuit will be transpiled to corresponding quantum system-level languages (such as QASM [33]) before being sent to a quantum cloud service for execution. Therefore, an adaptable serverless model for executing quantum tasks is needed to address this challenge. By leveraging the ideas of the serverless model and combining quantum and classical parts in a single service, we can adapt to the current nature of quantum cloud services, accelerate the software development process, and optimize quantum resource consumption. This kind of computing model could be a potential approach to enable software engineers to realize the advantages of quantum computing and explore more complicated quantum computation in the future.

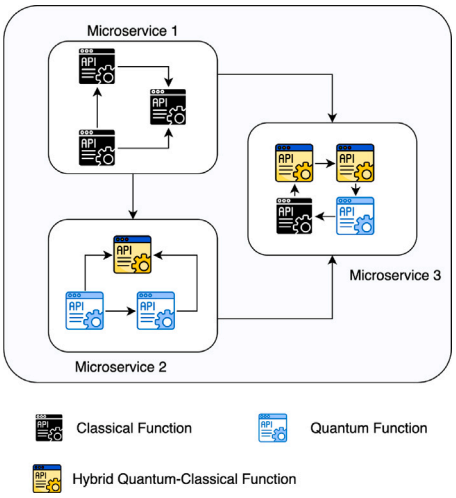


Fig. 1. Service-Oriented Quantum-Classical Application Model.

3.2. Quantum function-as-a-service (QFaaS)

In serverless computing, the Function-as-a-Service (FaaS) indicates a service-oriented cloud computing model in that the software engineer only needs to focus on coding without worrying about server configuration. A function is made by small pieces of programming code and is deployed as a service, which can be triggered and executed on demand [34].

By bringing the FaaS model to quantum computing, we propose the concepts of Quantum FaaS. A *quantum function* can be considered an ephemeral, event-triggered, and single-purpose quantum program with few pieces of quantum code. Due to the limitation of the NISQ devices and the difference in the software deployment model, we need to leverage the classical resources and techniques for developing and executing the quantum function. Specifically, a software engineer can still focus solely on coding quantum functions with high-level quantum SDKs (such as Qiskit, Cirq, Q#, or Braket) without needing to care about the quantum programming environmental setup or server deployment. The function code is automatically deployed in a containerized environment and is published as a service with an API endpoint for invoking. Whenever that service is triggered, the programming logic defined in the function will be executed, where both classical and quantum computation is involved. The classical parts include the pre-processing of input data, quantum circuit generating, and post-processing, where the quantum part indicates the circuit execution on quantum backends.

By adopting the FaaS approach and classical resources for creating a quantum function, we can seamlessly integrate multiple quantum functions together or with classical ones to construct a service or microservice in a large application (see Fig. 1). The serverless and service-oriented application model is a potential approach to bring the advantages of quantum computation to solve intractable problems of classical computing without replacing the whole system. For example, we can replace a classical random number generation function in a finance microservices application with the quantum counterpart to yield truly random numbers [35], which we cannot do in classical programming. However, the application of serverless computing models to quantum applications needs to be carefully considered and adjusted, especially in the current NISQ era, in which quantum hardware limitations can hinder quantum execution. Ultimately, the potential benefits and challenges of the emerging quantum serverless computing model motivate us to explore and empirically evaluate in this study.

4. QFaaS architecture and main components

4.1. Software requirements and design principles

The design of the QFaaS framework is guided by several key requirements and design principles, which contribute to its benefits for streamlining service-oriented quantum application development and help software engineers to plan, develop, and improve their quantum software applications:

- **Serverless**: Quantum software engineers only need to focus on developing and improving their functions, while the framework automatically carries out the rest of other procedures, including the environmental setup, function deployment, selecting the appropriate quantum computation system for the function execution (backend selection), managing the function operation and scaling.
- **Service-Oriented**: Each quantum function can be deployed as a service, which can be accessed through the cloud-based API gateway in multiple methods. This approach simplifies further expansion and maintenance, drawing inspiration from the microservices architecture and the "everything-as-a-service" (XaaS) paradigm. Consequently, new functionalities can be easily integrated into the existing application without disrupting other services.
- **Flexibility**: Users can choose their preferred quantum programming languages, libraries, and cloud providers to avoid potential vendor lock-in situations. The framework needs to support the current NISQ computers and quantum simulators. Its architecture provides the flexibility to implement possible extensions to support other quantum technologies as they emerge in the future.
- **Seamlessness**: The framework needs to support continuous integration and continuous deployment, which are two of the essential characteristics of DevOps to continuously deliver value to end-users. Utilization of this model boosts application development and becomes more reliable when compared with the traditional paradigm [36].
- **Reliability**: The framework implementation should use the state-of-the-art software technologies to ensure high availability, security, fault tolerance, and trustworthiness of the overall system. The execution results are stored in the database for comparison purposes and to optimize the execution parameters of the quantum function.
- **Service Scalability**: As one of the critical characteristics of the serverless model, the quantum service is scalable and adapts to the actual user requests. However, the scalability of current quantum devices is limited by the number of qubits, and NISQ devices cannot execute practical-scale networking and computational instructions beyond small instances of a few specific problems [37]. Therefore, it is important to note that within the context of our framework, this requirement is manifested in the size scalability (i.e., the ability for a system to effectively expand its size as more resources or users are added) [38] of the classical resources, wherein the quantum function is deployed. We primarily focus on horizontally scaling in/out function deployment by adjusting the replication of function instances, in response to user requests. This approach provides a practical and adaptable solution to handle varying workloads depending on the number of concurrent function invocations.
- **Transparency**: The operation workflow of the framework needs to be transparent to both the software engineers and end-users. The information provided by the framework is sufficient for troubleshooting, logging, and monitoring purposes, and can be used for further investigations if needed.

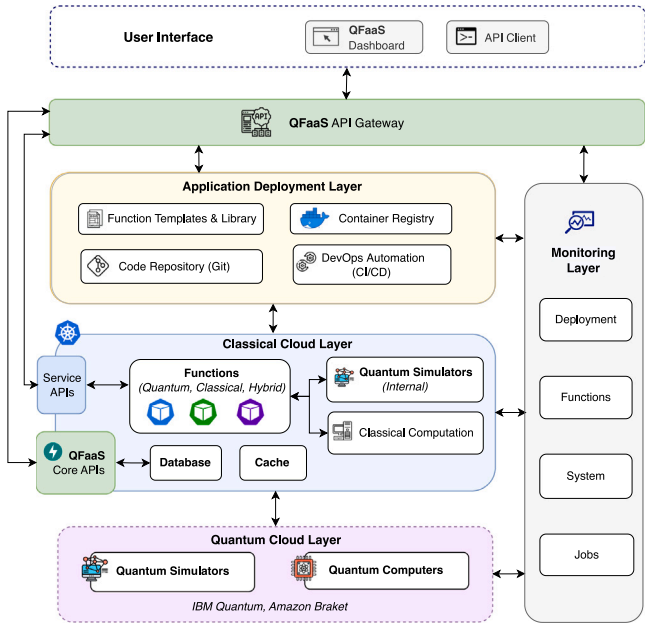


Fig. 2. Overview of QFaaS Architecture Design and Main Components.

4.2. Main components

The architecture design of QFaaS comprises six extendable components: the QFaaS APIs and API Gateway, the Application Deployment Layer, the Classical Cloud Layer, the Quantum Cloud Layer, the Monitoring Layer, and the User Interface. Fig. 2 illustrates the overall design, including the architecture and principal components of our framework.

4.2.1. QFaaS APIs and API gateway

We design two types of APIs that expose to the authenticated user through a secure HTTPS connection:

- **Service APIs** are the set of APIs corresponding to the deployed functions. Each function running on the classical cloud layer has a unique API endpoint accessible to an authorized end-user. These APIs can be integrated seamlessly into existing software workflow.
- **Core APIs** set is one of the most essential components in the QFaaS framework. It comprises a set of secure REST APIs, which provide principal operations and interactions among all components of the whole system. These APIs facilitate function development, invocation, job monitoring, and interaction with the external quantum providers and backend management. Core APIs also facilitate the main functionalities of the QFaaS UI. We explain the detailed design and implementation of the Core APIs in Section 5.1.

The *API gateway* serves as a centralized entrance where users can interact with other components. This API gateway routes users' requests to suitable components for processing and delivers the result back to the users with a common data format after completing the execution.

4.2.2. Application deployment layer

This layer serves as a bridge between quantum software engineers and the cloud layers to deploy and expose each function as a service with an API endpoint. It takes the principal responsibility for code version control, containerizing, and deploying functions by incorporating four key components:

- **Code Repository** is a Git-based platform to manage function codes with version control, which is essential in software development for collaboration, issue tracking, and further workflow automation integration.
- **Function Templates and Library** provides container-based quantum software environment configuration, support library, and common function templates for well-known quantum SDKs and languages, including Qiskit, Cirq, Q#, and Braket. We developed *QFaaS Library*,[1] a Python-based programming library that supports essential interactions to the Core APIs and provides common pre-built data pre-processing, and output post-processing for the function development.
- **DevOps Automation** employs Continuous Integration and Continuous Deployment (CI/CD) integration following DevOps manner to automate the function deploying and updating, ensuring the continuous delivery of reliable quantum functions.
- **Container Registry** stores immutable container images of functions and environmental setup for function deploying, migrating and scaling.

4.2.3. Classical cloud layer

This layer is a cluster of cloud-based classical computers (physical servers or virtual machines), where the quantum functions are deployed and triggered. All the classical computation tasks are executed here, including backend selection, data pre-processing, and post-processing.

We employed Kubernetes to orchestrate all the *pods* (the container-based unit of Kubernetes) for the deployed function across all cluster nodes. Each function will be run on a pod and can be scaled up horizontally by replicating the original pod to serve multiple incoming requests simultaneously. Following the proposed architecture, we can use all built-in quantum simulators of employed SDKs directly inside a pod at the Kubernetes cluster. We call this kind of simulator the *internal quantum simulator*, while the term *external quantum simulator* denotes simulators offered by quantum cloud providers.

We also deployed a NoSQL database on this layer to permanently store the processed job result data and information of users, functions, and backends. In a production deployment, it is recommended that the database be placed externally to ensure the high availability for permanent data storage. Besides, the cached data for quantum circuits can be stored at this layer.

4.2.4. Quantum cloud layer

This layer is an external part, indicating the quantum cloud providers, such as IBM Quantum and Amazon Braket, where the quantum job can be executed in a physical quantum backend. Quantum providers can provide either quantum simulators or actual quantum computers through their cloud services, which can be accessed from the Classical Cloud layer. We develop the corresponding APIs in *QFaaS Library* and *Core APIs* to interact with each external quantum cloud platform. The result processing data from all cloud providers is standardized in a common JSON format, ensuring data consistency and preventing data lock-in issues within the serverless-based platform.

4.2.5. Monitoring layer

This layer incorporates monitoring techniques to check the status of other QFaaS components, including quantum backends, quantum providers, function execution (job), and function deployment. As the classical cloud layer employs Kubernetes as the container orchestration, we can also seamlessly integrate additional open-source monitoring techniques, such as Prometheus,[2] Grafana,[3] and Lens[4] for observing other system aspects of the classical cloud layer such as resource consumption, networks, and system logs.

[1] https://pypi.org/project/qfaas/
[2] https://prometheus.io/
[3] https://grafana.com/
[4] https://k8slens.dev/

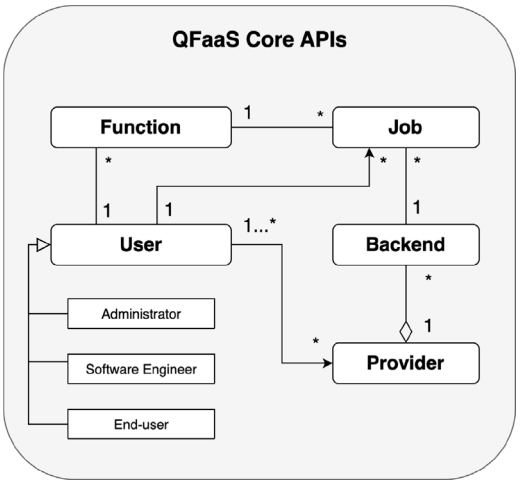


Fig. 3. Class Diagram of QFaaS Core APIs.

4.2.6. User interface

We created a user-friendly web application (QFaaS Dashboard) using React[5] to interact with the QFaaS system (examples can be found in Section 6). This user interface visualizes the essential functionalities of QFaaS, such as function developing, deploying, invoking, monitoring job results, and backend connection.

5. Design and implementation

This section provides the technical design, procedures, and implementation of the QFaaS Core APIs and quantum function development, development, invocation, backend selection, and cold start mitigation in the QFaaS framework.

5.1. QFaaS core APIs

QFaaS Core APIs take responsibility for primary functionalities in the QFaaS framework. We have developed this API set using Python 3.10 with FastAPI,[6] a high-performance Python-based framework supporting the Asynchronous Server Gateway Interface (ASGI) for concurrent execution. We used the MongoDB database to store the persistent data in JSON format. Fig. 3 depicts the overall class diagram, with attributes and methods of each object in QFaaS Core APIs.

- **User:** This class defines user attributes and methods to facilitate access control and role management features. We categorized three different users: *administrator, software engineer,* and *end-user* with different privileges in the system. Administrators control all components; software engineers can develop and deploy functions, while end-users can only use their appropriate functions. Each active user is assigned a unique token (using OAuth2 Bearer[7]), which is used for authentication, authorization, and dependency check for each interaction with the core components of the QFaaS. This implementation enhances security for the whole framework and provides a multiple-user environment for taking advantage of the framework.
- **Function:** This class defines each function's properties and supported methods. Each function belongs to a software engineer (author) and its access can be granted to a specific end user. The *CRUD*, *invoke()* and *scale()* methods of this object interacts directly with other architectural components such as Code Repository, Container Registry and the Classical Cloud layer to handle the function deployment, management, and invocation.
- **Job:** A job in QFaaS is a computation task submitted to a quantum backend for execution. All properties and methods of a job are defined in the Job class. Each Job has a unique *Job ID* assigned by QFaaS and can be associated with a *providerJobID* given by an external provider. The function invocation initializes the job object. After finishing the execution, job results can be post-processed and stored in the database for further retrieval.
- **Provider:** The provider class handles a user's authorization to external quantum providers, including IBM Quantum and Amazon Braket. The design of this class ensures that each user has the specific privilege to access their quantum providers only.
- **Backend:** A backend is a quantum computation node, such as a quantum simulator, or a quantum computer, which takes responsibility for the quantum execution. The Backend class defines the attributes and methods to interact with the backend provided by the classical cloud layer or external quantum cloud layer. We also implement the *Backend Selection* policy in this class for automating selecting the most appropriate quantum backend for each quantum task execution (Section 5.3).

5.2. QFaaS quantum function structure

This section describes the structure of a quantum function in the QFaaS framework for the development process. Our framework provides a set of pre-configured function templates, each encapsulated in a Docker image with the necessary quantum software development kit (SDK) environment, to streamline the development of quantum functions. Each function has a single working directory, including main components following the common pattern of the serverless platform (such as Lambda [39]). Function handler code includes classical parts (using Python) and quantum parts. When end-users invoke the function, QFaaS executes the function handler and starts the computation as defined. Handler for Qiskit, Cirq, and Braket function can be defined at `handler.py` file while Q# function requires an additional Q# code at `handler.qs` file and then to import it to the main `handler.py` file. The sample structure for the function handler with classical pre-processing and post-processing is described in Code 1. In the function handler code, we import `qfaas` library[8] and all additional libraries (including compiled binary for Q# function). Then, we define the function handling procedure (as shown in Code 1) as follows:

```
import qfaas, [additional_libraries]

def handle(event, context):
    # 1. Pre-processing (optional)
    data = pre_process(event.data)
    # 2. Generate Quantum Circuit
    qc = generate_circuit(data.input)
    # 3. Verify/select quantum
    backend
    backend = Backend(data, qc)
    # 4. Submit job for execution
    job = backend.submit_job(qc)
    # 5. Post-process (optional)
    result = post_process(job.result)
  return result
```

Code 1: Sample structure of a hybrid quantum-classical function

[5] https://reactjs.org/
[6] https://fastapi.tiangolo.com/
[7] https://datatracker.ietf.org/doc/html/rfc6750
[8] https://pypi.org/project/qfaas/

1. **Input Pre-processing**: Users can optionally define the pre-processing for input data handling in `event` object. The `event` is JSON-based data that contains user input raw data, followed by the QFaaS JSON format, while `context` provides HTTP methods (such as GET/POST), HTTP headers, and other properties, which are optional for the request. This pre-processing is executed on classical computers and the processed data is used for the circuit generation process.
2. **Quantum Circuit Generating and Compilation**: The engineer can define quantum circuit code that takes `input` parameters as input data. Based on the given parameters, an appropriate quantum circuit is built. As aforementioned in Section 3.2, each quantum function is a single-purpose and event-triggered quantum program to solve a specific problem that produces the output based on the user's input. To ensure the versatility of the quantum function, the quantum function code needs to be designed to generate the quantum circuit dynamically based on the user's input (so we called *variable circuits* for short). Otherwise, if the quantum circuit is fixed regardless of different input data, the same circuit will be generated every time the quantum function is invoked. The characteristic of the quantum circuit is used for the backend selection in the next stage. After that, the initial circuit may need to be transpiled to be compatible with the supported gates and qubit topology of the selected backend. The conversion is also required in the case of the quantum cross-platform execution model.
3. **Quantum Backend Selection**: QFaaS provides a built-in quantum backend verification and selection in `Backend` class (detailed implementation in Section 5.3) to ensure the most appropriate backend is selected for the execution. It is worth noting that our proposed backend selection strategy is a best-effort approach as some quantum jobs take longer to execute than others, and no quantum provider discloses any information about the pending jobs' characteristics at the moment. However, our framework also allows users to define their customized backend selection strategy, which paves the way for more advanced techniques to be designed when more information is available in the future.
4. **Quantum Job Submission and Execution**: We also provide the `submit_job` method in the `Backend` class to perform the job submission to the selected quantum backend in the previous step. This method involves the quantum circuit transpilation to ensure the submitted circuit is compatible with all supported gates of the quantum backend. The outcome of this method is a `job` with a unique ID for result retrieval and further inspection. As today's quantum computers are NISQ devices [13], each quantum execution should be run many times (shots) to mitigate quantum errors. Besides, due to the limited number of available quantum computers, a quantum task (job) needs to be queued at the cloud provider (from seconds to hours) before execution. After the quantum computation is finished, the raw result is retrieved from the provider to the function handler for the post-processing step.
5. **Output Post-Processing**: Users can define the customized post-processing method for further analyzing the raw result from the quantum computer before generating the final result for end-users. We provided several sample post-processing methods based on result counts, which will be discussed in Section 6.

After finishing all the processing, the function handler returns the result to end-users, including the HTTP Status Code and response data in common JSON format, regardless of the difference of targeted backends. The job result is also kept in the MongoDB database for further retrieval.

Algorithm 1: QFaaS Quantum Backend Selection

```
Input : qc: quantum circuit, sdk: quantum SDK,
        a: autoselect option, t: preferred backend type,
        bName: backend name (for manual selection),
        γ: list of all parameter weights, u: current user
Output: be: quantum backend instance
procedure BackendSelection:
    be ← null
    q ← qc.getNumQubit()
    if t is internal then
        be ← getInternalSimulator (sdk, q, u)
    else
        if a is True then
            # Auto Backend Selection
            Pre-select backends and get all transpiled qc
            B ← getBackendList (u)
            B ← preSelect (B, q, sdk, t)
            T ← getTranspile (qc, B)
            Scoring all backend in B
            for b ∈ B do
                τ ← get transpiled circuit for b (τ ∈ T)
                Normalize depth, QV, workload & CLOPS
                v ← norm (getQV(b), B)
                d ← norm (getDepth(τ), T)
                c ← norm (getCLOPS(b), B)
                w ← norm (getWorkload(b), B)
                Compute precision, speed & overall score
                p ← v·γ_v + d·γ_d
                s ← w·γ_w + c·γ_c
                ε ← p·γ_p + s·γ_s
                Update all backend scores
                B ← updateScore (b, ε)
            end for
            be ← getMaxScore (B)
        else
            # Manual Backend Selection
            be ← verifyBackend (bName, u, qc)
        end if
    end if
```

5.3. Quantum backend selection

Quantum Backend Selection is an essential procedure in the function invocation process to determine which quantum backend is suitable for the quantum circuit execution. In the initial version of QFaaS, we have incorporated a scoring-based policy, as outlined in Algorithm 1. This policy empowers users to prioritize their selection of the quantum backend based on factors such as result precision or the speed of function execution.

In our backend selection logic, users can specify their preferred backend type as `internal` to use the internal quantum simulator for testing and prototyping purposes. They can also manually select a specific quantum backend when invoking a function. Otherwise, the QFaaS framework will process the automatic backend selection strategy as follows:

1. **Backend Pre-selection and Circuit Transpilation:**
   QFaaS filters a list of backends based on key requirements to execute the quantum task. These requirements include (1) scale of the quantum backend (i.e., the number of qubits must be sufficient), (2) availability (i.e., the quantum backend must be operational), and (3) and compatibility (i.e., the quantum backend must support the quantum SDKs used by quantum circuit) [40]. After pre-selecting the list of appropriate quantum backends ($\mathcal{B}$), the circuit will be transpiled to adapt to all quantum backends in $\mathcal{B}$, and the characteristics of each corresponding

transpiled circuit can be used for determining the most suitable quantum backend.

2. **Backend Scoring based on execution priority**
Considering the current state of NISQ devices and the quantum computing service model of quantum providers, we determine two priorities to select a quantum backend when invoking a function, i.e., precision and speed.
*(a) The **precision** of the execution results* depends on the quality of qubits in a quantum system, which can be determined by the quantum volume (QV, denoted by v). Quantum volume [41] is a holistic metric that indicates how well a quantum circuit can be executed in a quantum system, measured by the largest random square circuit that can be successfully run. We also consider the depth (denoted by d) of the circuit as a shallower circuit has a higher chance of being executed faithfully inside a quantum system [42,43]. In other words, a finer quantum backend for generating better result precision has a higher quantum volume and requires a quantum circuit to be transpiled with a smaller circuit depth. For equally weighting two metrics with different scales in our backend scoring algorithm, we normalize v and d of the ith backend in $\mathcal{B}$ to be between 0 and 1 (using min–max normalization) by the following formulas:
$\bar{v}_i = \frac{v_i - \min(v)}{\max(v) - \min(v)}$ and $\bar{d}_i = \frac{\max(d) - d_i}{\max(d) - \min(d)}$
where $\bar{v}_i$ and $\bar{d}_i$ are normalized values of quantum volume and transpiled circuit depth.
The precision score (p_i) of the ith quantum backend can be calculated by $p_i = \bar{v}_i\gamma_v + \bar{d}_i\gamma_d$ where $\gamma_v + \gamma_d = 1$ and γ_v, γ_d are the weight of quantum volume and circuit depth, respectively.
*(b) The **speed** of execution* relies on how fast a quantum system can execute quantum circuits, which can be measured by Circuit Layer Operations Per Second [43] (CLOPS, denoted by c). We also consider the current workload (i.e., the total number of pending jobs to be executed, denoted by w) of a quantum backend to determine the speed score, as the waiting time can be typically shorter. Similar to the calculation of precision score, we normalize c and w of the ith backend in $\mathcal{B}$ to be between 0 and 1 by the following formulas:
$\bar{c}_i = \frac{c_i - \min(c)}{\max(c) - \min(c)}$ and $\bar{w}_i = \frac{\max(w) - w_i}{\max(w) - \min(w)}$
where $\bar{c}_i$ and $\bar{w}_i$ are normalized values of CLOPS and current workload of the quantum backend.
The speed score (s) of an ith quantum backend can be determined by $s_i = \bar{c}_i\gamma_c + \bar{w}_i\gamma_w$, where $\gamma_c + \gamma_w = 1; \gamma_c$ and γ_w are the weight of CLOPS and workload of the quantum backend, respectively.

After scoring the precision (p_i) and speed (s_i), we calculate the overall score (ϵ_i) of the ith backend by using the following formula: $\epsilon_i = p_i\gamma_p + s_i\gamma_s$ where $\gamma_p + \gamma_s = 1$; γ_p and γ_s are the weight of precision and speed priority, respectively. This approach can dynamically select the most suitable quantum backend based on user priority, the characteristics of current function invocation, and the status of all available quantum backends. It is important to note that although all weight parameters can be adjusted, users can only need to adjust two primary weights γ_p and γ_s based on their priority on either the precision of the execution result or how fast the quantum circuit will be executed. For example, if a user prioritizes the precision of the result rather than the speed, they can set γ_p close to 1, in which the quantum can be queued in a specific backend (which can be executed with the highest accuracy) even if the availability of the least busy or highest quantum backend is not the best one. Otherwise, the default value of all weight parameters can be set to 0.5 to maintain the balanced contribution of all factors to the backend selection decision. To validate the operation of this policy, we provide examples of backend selection for three consecutive quantum function invocations in Section 6.3.

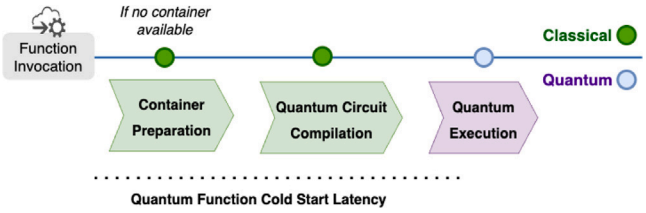


Fig. 4. Quantum function cold start latency process during the invocation.

It is important to note that we have already utilized the most accessible circuit information and key benchmarking metrics of quantum computers (qubit number, quantum volume, CLOPS) [43] for implementing the backend selection strategy. Due to the limitations to accessing the public information of pending jobs at the other cloud provider (e.g., Amazon Braket through Strangeworks), this backend selection procedure is only supported for IBM Quantum providers at the time of its initial development. Supporting the backend selection for Amazon Braket and other quantum cloud vendors and considering other aspects, such as costs, is our future plan.

5.4. Quantum function cold start mitigation

Cold start is a typical problem of serverless computing [31]. It is referred to when the framework needs to initialize a new container instance and prepare the function execution when handling a new invocation [44,45]. This latency is even more significant in the context of quantum functions, as each function typically requires a specific environment setup and circuit compilation optimization to adapt to the targeted quantum backend.

The quantum function cold start latency can be illustrated in Fig. 4. If there is no container of a specific function is available at the time of its invocation, a new instance needs to be initialized, which can cause a notable delay. After setting up the environment for the quantum SDK, a corresponding quantum circuit needs to be generated and compiled. As most available quantum computers do not have fully connected qubit topology and do not support all quantum gates, quantum circuit transpilation is typically required to tailor it to specific qubit topology and native gate set of targeted quantum backend. This process can also result in substantial delays in the quantum circuit compilation phase before its execution. Thus, mitigating cold start latency in a quantum serverless platform, particularly in the NISQ era, is unavoidably essential.

Focusing on the nature of current quantum execution, we design an agile and practical strategy to mitigate the cold start issue and enhance quantum function execution as a complement to the QFaaS framework (see Fig. 5). To reduce the container preparation latency, we keep the function's container "warm" (i.e., keep at least one instance up and running) after its creation to avoid the long container initialization latency. To mitigate the quantum circuit compilation latency, we utilize the cache-based approach, which is a popular strategy in classical serverless computing [45]. To enhance the reusability and flexibility of cached data, we use QASM (quantum assembly language) to store a copy of the pre-transpiled quantum circuits. Open QASM is a lightweight assembly language to represent universal quantum circuits [33], which can be imported to or generated from different quantum SDKs (e.g., Qiskit). During the circuit compilation, QFaaS will check the availability of the corresponding transpiled circuit and load it into the current function instance for execution. Otherwise, the quantum circuit has to be transpiled, but a copy of the transpiled circuit will be stored for further reuse. Besides, during the idle time period of the function, we can proactively pre-transpile quantum circuits of that function to optimize with the qubit topology and native gate set of available quantum backends. Another source to enrich the transpiled

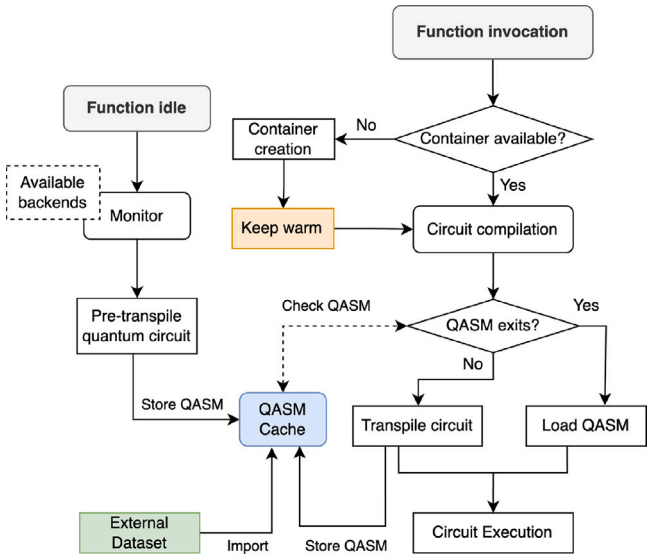


Fig. 5. QFaaS Transpilation Caching approach for mitigating the cold start latency of container preparation and quantum circuit compilation.

QASM files is to integrate the external quantum circuit dataset, such as MQTBench [46], which comprises over 70,000 quantum circuits in QASM format of different applications. As Open QASM is a lightweight and platform-agnostic quantum assembly language [33], this caching approach also brings its potential to future expansion of QFaaS to support other quantum computing systems. Also, the circuit loading time from a pre-transpiled QASM file is by far faster than transpile the circuit; this approach can significantly reduce the overhead for the function repeated execution, hence reducing the cold start latency for the quantum function execution. The empirical result to validate this strategy can be found in Section 6.4.

5.5. QFaaS sample operation workflows

This section provides two sample operation workflows, including developing/deploying and invoking quantum functions using the QFaaS framework.

5.5.1. Developing and deploying quantum functions

QFaaS simplifies the function development process for quantum software engineers. They can utilize the following workflows to create new functions, update existing functions, and troubleshoot issues during the development process. Fig. 6(a) depicts the function development process, which consists of seven key steps as follows:

1. Create a new function by using the QFaaS UI. The engineers specify which quantum SDK will be used (Qiskit, Cirq, Q#, or Braket), include the required library, and write their quantum function code or use pre-defined circuit library (0).
2. Push function codes to the Application Deployment layer through the QFaaS API Gateway.
   After these steps, QFaaS automatically takes responsibility for the rest of the deployment procedure by performing the following steps:
3. The API gateway forwards the function code and pushes it to the Code Repository.
4. After the function code is pushed, it triggers the Automation components to start the continuous deployment.
5. Pull the function template and combine it with function code to build up and containerize it into a Docker image. Then, those images will be pushed to a Container Registry to be stored for further utilization (such as migrating or scaling in a function).
6. Deploy the function as a container-based service into the Kubernetes cluster at the classical cloud layer.
7. Expose the service API URL endpoint corresponding to the deployed function. After this stage, the function serves as a service and is ready for invoking from end-users.

During the development stage of the quantum function, engineers can test the deployed services with different backends several times and analyze the results, which are stored in the database for comparison and further improve function configurations (e.g., shots) to achieve optimal results. QFaaS also monitors the operation of quantum functions and performs the cold start mitigation strategy as described in Section 5.4

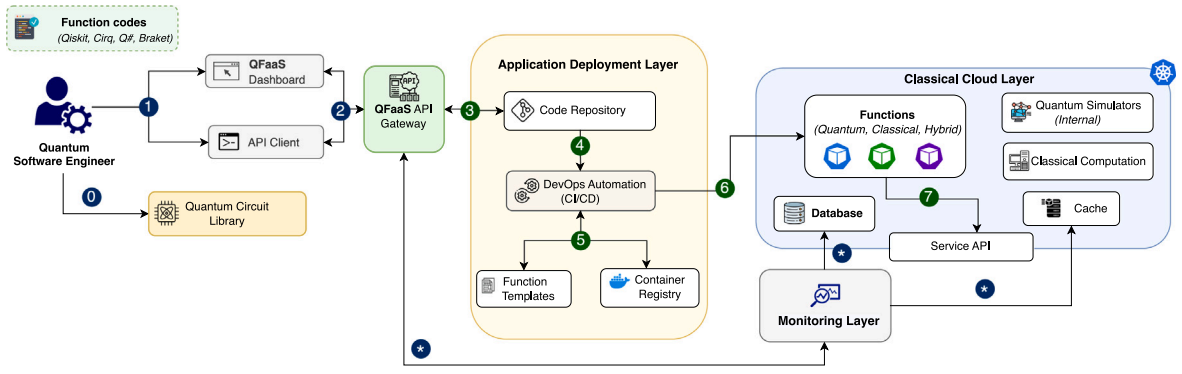
5.5.2. Invoking quantum functions

The users can invoke the deployed function through the QFaaS API gateway. Fig. 6(b) demonstrates the overall workflow for the function invocation, including seven steps as follows:

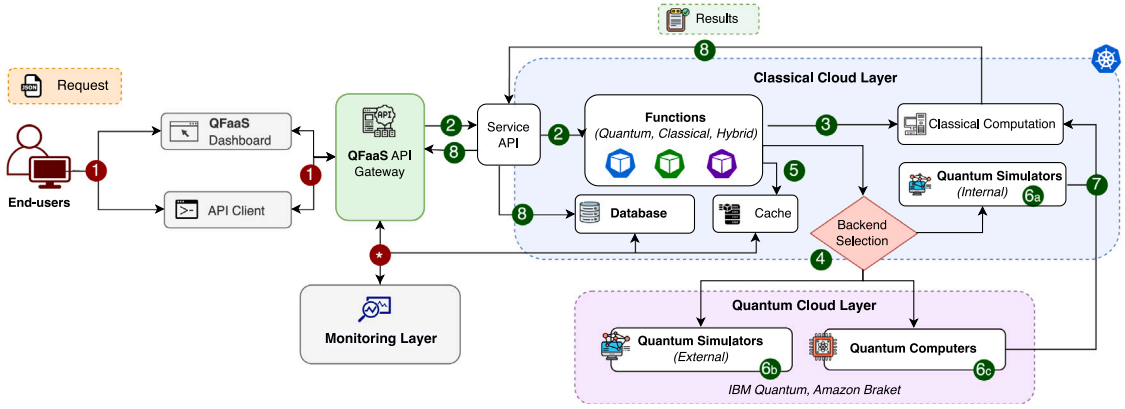
1. *Sending request*: In the requested data, the user can clarify their preferred backend or let the framework automatically select the suitable backend, the result retrieval method, and the number of shots they want to repeat the quantum task.
   After receiving the user's requested data, QFaaS automatically accomplishes the rest of the process.
2. *Routing the requests*: The API Gateway routes user requests to appropriate available functions. In the event that a function is not yet initiated or undergoing scaling in to zero, QFaaS takes charge of initializing and activating the function to handle the incoming user request. This scenario is anointed as a *cold start* in serverless jargon.
3. *Input Pre-Processing and Quantum Circuit Compilation*: The user's input data undergoes pre-processing at the classical computation node. Then, a corresponding quantum circuit is generated based on the provided input.
4. *Backend Selection*: An appropriate backend is selected based on user requests and availability at the quantum provider, using our decision policy (Algorithm 1).
5. *Executing the quantum job*: The quantum circuit is transpiled and dispatched to the chosen backend, which can be an internal quantum simulator (6a), an external quantum simulator (6b), or a quantum computer (6c). Once the backend completes the execution, the outcome is transmitted back to the function handler for post-processing on classical resources in the same invocation. If users want to check the response later or when the waiting time at the provider is longer than a predefined timeout (1 min by default), the function will send back the QFaaS Job ID and backend information after successfully submitting the quantum circuit to the selected backend. This delayed scenario is expected to happen often when submitting a job to an external quantum cloud provider during peak hours due to the current shared nature of available quantum resources.
6. *Output Post-Processing*: The outcome from quantum backends could be analyzed and post-processed before being sent back to end-users and stored in the database.
7. *Returning the results*: Following the previous step, the final result is returned to end-users through the API Gateway, following the same approach as when they initially submitted the request. End-users can obtain the final result data and information regarding the quantum backend device utilized during the quantum execution.

6. QFaaS example of operation and evaluation

This section provides explanatory examples of the operation and performance evaluation of QFaaS in function deployment, resource consumption, and scalability. Besides, we validate the proposed quantum backend selection and cold start mitigation strategy with various



(a) Function development and deployment for Quantum Software Engineers.



(b) Function invocation for End-users.

Fig. 6. Overview of two main operation workflows in QFaaS: (a) Function development and deployment, (b) Function invocation.

quantum algorithms. We also use QFaaS to evaluate the performance of popular simulators and computers to give practical insights into the limitations and challenges of quantum software engineering in the NISQ era (see Appendix B).

6.1. Environment setup

We deployed the core components of QFaaS on a set of four virtual machines (VMs) offered by the Melbourne Research Cloud.[9] We set up the Kubernetes cluster using microk8s[10] with *containerd* as the underlying containerization technology on a three-VM cloud cluster (one master node with 4 vCPU, 16 GB RAM, and two worker nodes with 8vCPU, 32 GB RAM each). The function deployment component is built on top of OpenFaaS [47]. The QFaaS Code Repository and Automation components are deployed on the last VM (4 vCPU, 16 GB RAM) with Gitlab as the underlying Git-based platform. For the quantum computation, we have tested the Qiskit functions with the built-in QASM simulator on the classical computers at the classical cloud layer and quantum backends provided by IBM Quantum [14]. For Q# and Cirq functions, we used their built-in quantum simulators and executed them on the classical cloud layer. For Braket functions, we used their local simulator and external backends at Amazon Braket through the support of Strangeworks Backstage program [48]. The circuits and transplied QASM files for other quantum algorithms in the backend selection and cold start mitigation validation are adopted from the MQT Bench dataset [46].

[9] https://cloud.unimelb.edu.au/
[10] https://microk8s.io/

6.2. Example of operation and performance evaluation

To demonstrate the practical operation of QFaaS, we utilize an explanatory quantum circuit to generate truly random numbers, utilizing the superposition characteristic of qubits. It is evident that random numbers play an essential role in cryptography, finances, and many other fundamental scientific fields [49]. By leveraging quantum principles, Quantum Random Number Generation (QRNG) is a reliable way to provide true randomness, which cannot be achieved by classical computers [50]

We deployed the QRNG circuits in four popular quantum SDKs (Qiskit, Cirq, Q#, and Braket). The main idea of this circuit is to leverage the Hadamard gate to create the superposition state of each qubit and then measure to get a random value (0 or 1) with the same possibility (50%). To validate the hybrid quantum–classical integration feature, we implemented sample post-processing by analyzing all possible outcomes when the function is executed multiple times (shots) and returning the most frequent result to the user.

The request for invoking the QRNG function using all supported SDKs and languages follows the QFaaS format. Upon completion of the processing, the sample response, as illustrated in Fig. 7, indicates that a 10-qubit random number, specifically 367 (0101101111 in binary), has been generated. This particular random number occurs most frequently, appearing twice, during the execution of the function using the Amazon Braket Simulator (aws.SV1).

6.2.1. Function deployment evaluation

In this evaluation, we measured the image size, average image building time, and the total deploying time in the first and later update at the Application Deployment layer (using Gitlab) to provide insight into QFaaS system performance. Table 1 records the detailed result of this evaluation.

Table 1
QFaaS functions deploying and re-deploying (updating) time.

Function	Function Image Size (MB)	1st Building time (s)	1st Deploying time (s)	Re-building time (s)	Re-Deploying time (s)
Qiskit QRNG	755.85	180.62	262	8.35	31
Cirq QRNG	**561.19**	**105.57**	**168**	8.53	**30**
Braket QRNG	1010.07	354.04	496	6.36	**30**
Q# QRNG	2200	267.75	466	**5.77**	34

Table 2
QFaaS function resource consumption during the idle time and busy time with 1 and 10 concurrent users.

Function	Idle time		1 user		10 concurrent users	
	CPU (vCore)	RAM (MB)	CPU (vCore)	RAM (MB)	CPU (vCore)	RAM (MB)
Qiskit QRNG	**0.001**	**87.9**	0.052	**87.93**	0.082	**88.04**
Cirq QRNG	**0.001**	124.56	**0.024**	128.605	**0.053**	129.07
Braket QRNG	**0.001**	96.063	0.042	99.586	0.078	99.96
Q# QRNG	0.037	688.77	0.056	735.16	–	–

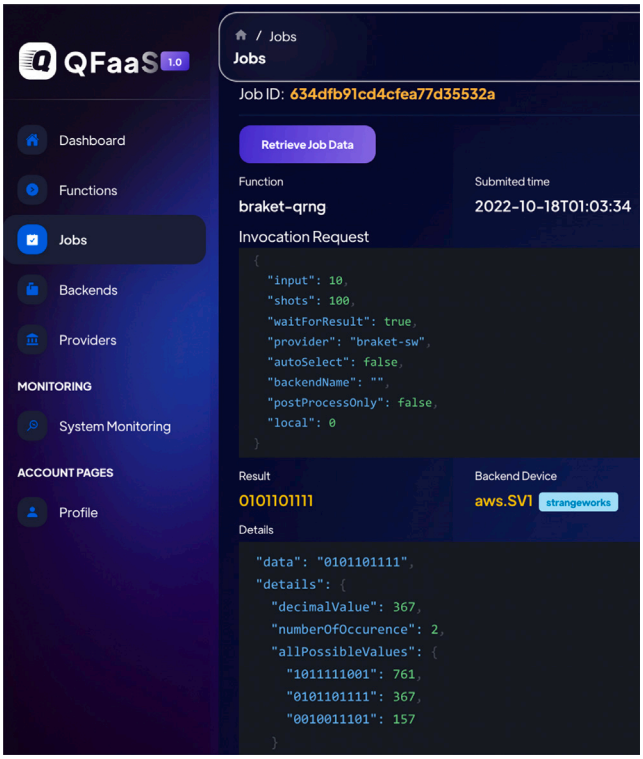


Fig. 7. Sample QRNG Function Invocation Result on QFaaS Dashboard interface using Amazon Braket backend (aws.SV1).

As all function's essential components are compressed in the container images, its size is varied from 561 MB (Cirq) up to 2.2 GB (QSharp). The total deploying time includes the function image-building time and image-deploying time, which is below ten minutes for creating the function deployments the first time. Deploying the Cirq QRNG took the shortest time (below 3 min) whereas Braket and Q# functions required 7-8 min to complete. However, when we update the function handler code, the re-deploying times are significantly faster (around 30 s) for all functions. It is mainly because a container image comprises multiple layers and the updated image can inherit multiple layers from the previous images. We also utilized caching technique to optimize the continuous integration and deployment process. These function deploying and updating times are reasonable in practice as the software engineer can focus on coding and offload the configuration and deployment to the Application Deployment layer. The corresponding service endpoint of the quantum function is then ready for invoking after several minutes in the first deployment and below a minute for the following update.

6.2.2. Function resource consumption

We monitored the resource consumption, including the CPU and memory (RAM) used by the pod associated with the deployed function in QFaaS. We measured the average value of maximum CPU and RAM usage, provided by Kubernetes Dashboard and K8sLens 6[11] monitoring tool. In this evaluation, we kept a single pod for each function and considered 3 scenarios: the idle time (i.e., keep the pod running without any invocation), 1 user, and 10 concurrent users.

As the results are shown in Table 2, the required resource to keep Qiskit, Cirq, and Braket functions running are kept low, which are 0.001 CPU vCore and roughly 100 MB of memory. In contrast, the Q# function requires more additional resources to keep its pod running, even during idle time. Then, we used JMeter 5[12] to continually generate 1000 requests to obtain 10-qubit random numbers (using the internal quantum simulators) with 100 shots for each invocation. The maximum CPU and RAM used are slightly increased in cases of Qiskit, Cirq, and Braket while the corresponding increment is higher with the Q# function. We also note that the figures for the Q# function in the last columns are disregarded as its pod frequently crashed when we constantly send requests from 10 concurrent connections. Therefore, we suggest using Qiskit, Cirq, or Braket for prototyping a quantum function to achieve better performance and maintain proper resource consumption.

6.2.3. Function scalability evaluation

As the underlying orchestration technique is based on Kubernetes, we can enable the auto-scaling feature to scale out the function deployment horizontally (i.e., increase the number of function replications), dealing with the scenario when the request workload grows significantly from multiple concurrent connections. To evaluate the effectiveness of different scalability levels, we perform a set of evaluations on the 10-qubit Qiskit QRNG function. In this evaluation, we increase N - the number of concurrent users from 8 to 64, using JMeter 5. In each case, we consider a set of three different scenarios: *non-scale* (1 pod/function), scale out to N/2 pods, and scale out to N pods (we note that the number of pods is fixed for evaluation purposes only). For example, suppose there are 64 users (N) invoking the function simultaneously; we will conduct three test cases: 1 pod, 32 pods (N/2), and 64 pods (N), and record the average response time and the standard deviation.

[11] https://k8slens.dev/
[12] https://jmeter.apache.org/

Table 3
Backend Selection for Job 1 (Deutsch-Jozsa's algorithm - using 5 qubits. ibmq_kolkata backend is chosen with the highest $\epsilon = 0.61$.

Backend	Qubits	QV (v)	$\bar{v}$	QC depth (d)	$\bar{d}$	CLOPS (c)	$\bar{c}$	Workload (w)	$\bar{w}$	p	s	ϵ
ibm_washington	127	64	0.33	20	0	850	0	2	1	0.17	0.5	0.34
ibmq_kolkata	27	128	1	19	0.17	2000	0.56	40	0.68	0.59	0.62	**0.61**
ibm_hanoi	27	64	0.33	17	0.5	2300	0.71	88	0.27	0.42	0.49	0.46
ibmq_guadalupe	16	32	0	14	1	2400	0.76	103	0.14	0.5	0.45	0.48
ibm_perth	7	32	0	15	0.83	2900	1	120	0	0.42	0.5	0.46

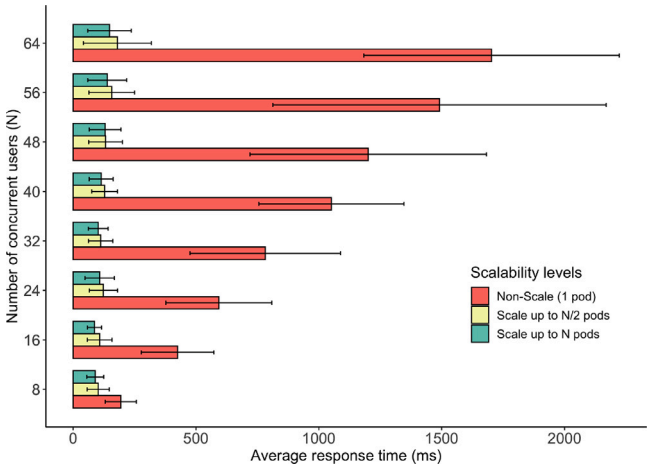


Fig. 8. Scalability evaluation on 10-qubit Qiskit QRNG function.

Fig. 8 demonstrates the result of our benchmarking. Overall, it is clear that if the function is non-scalable, the average response times for high-demand scenarios significantly increase. The previous section shows that the average response time for the 10-qubit Qiskit QRNG function is 81 ms. This figure jumps dramatically, up to 1703 ms, if 64 users use the function simultaneously. However, thanks to the containerization approach in our framework, we can quickly scale out deployment in seconds to ensure the response time is maintained. We can see that the average response time fluctuates between 87 to 148 ms if we scale out to N pods or from 102 to 180 ms when the number of pods is N/2.

It is important to notice that by scaling a quantum function, our approach is to replicate its classical instance deployment (i.e., Kubernetes pod), which is comparable to any existing serverless system. Its main objective is to handle concurrent requests from multiple connections efficiently by reducing total response time. As shown in Fig. 8, this scaling approach shows significant performance improvement in the case of using *internal quantum simulators* (for function testing or prototyping purposes) as the corresponding quantum simulator is incorporated into each function instance.

This scaling approach has no impact on the general performance in a special situation when all concurrent users manually select the same quantum backend for execution, as all quantum circuits will be forwarded to the same backend. However, this limitation can be addressed by designing an automatic backend selection algorithm based on the availability of quantum resources at the provider as our proposed algorithm (see Section 5.3). This way, each time the new instance of the same function is triggered, it will execute the backend selection individually to determine the best-suited quantum backend for execution without overflowing the same quantum backends. Considering the limitations of NISQ devices and available information about quantum jobs provided by the quantum cloud vendor, our scaling and backend selection strategy can still offer a best-effort and viable approach to facilitate the service-oriented quantum application requirements. We have planned to advance these techniques in future releases of QFaaS when more information about waiting jobs is publicly accessible from the provider.

6.3. QFaaS backend selection validation

To illustrate and validate the operation of the proposed Backend Selection policy, we present a case involving three invocations of different quantum functions (Deutsch-Jozsa's algorithm [6] using 5 qubits, the GHZ state [51] using 10 qubits, and Shor's algorithm [7] using 18 qubits) that need to be executed on the most suitable quantum backend. We assign equal weight to all factors, with $\gamma = 0.5$, thereby ensuring a balanced contribution of all aspects (including quantum volume, circuit depth, CLOPS, and workload) to the final backend decision. It is important to note that users can adjust these weight parameters according to their preferences, prioritizing either precision or execution speed. We assume that five quantum backends are available for the backend selection, each supporting a different number of qubits ranging from 7 to 127. Quantum volume and CLOPS are fixed values associated with each quantum backend, sourced from IBM Quantum [14], while the quantum circuit depth and workload are variable. Therefore, the quantum backend can be dynamically selected based on the current quantum circuits to be executed and the status of all available quantum backends (once per function invocation).

Table 3 illustrates the backend selection process for the first quantum job. While the *ibm_washington* backend has the largest number of qubits and is less busy, its quantum volume and CLOPS are significantly lower compared to other 27-qubit quantum computers. Considering all the factors, the *ibm_kolkata* backend is chosen as it has a higher chance of achieving both higher precision and faster execution. For the second quantum job (see Table 4), only four backends with more than 10 qubits are considered. Since all the transpiled quantum circuits have the same depth, the depth scores are normalized to 1. The *ibm_hanoi* backend is eventually selected as it has the highest score of 0.8. Lastly, the third quantum job involves Shor's algorithm to factorize 9, which requires 18 qubits and a significant number of layers in the transpiled circuit (Table 5). Only three backends meet the key requirement for the number of qubits, and among them, the *ibmq_kolkata* backend with the highest score is chosen.

It is also essential to highlight that the backend selection policy presented here is an approximation approach, as certain metrics, such as the workload of a quantum backend, can be tricky. For example, a backend with a smaller number of pending jobs does not necessarily guarantee faster execution, as those pending jobs could be large tasks requiring a longer processing time. However, users have the flexibility to disregard or assign a lower priority (close to 0) to this metric when determining the speed score of a backend. In our future plans for advancing the QFaaS framework, we aim to incorporate more advanced techniques, such as machine learning, which automatically adjust these metrics to enhance the backend selection process. This will further refine the effectiveness of the policy, offering improved decision-making capabilities within the QFaaS framework.

6.4. QFaaS cold start mitigation evaluation

We evaluate the cold start mitigation strategy with Grover's algorithm to diverse the use cases of QFaaS. Grover's algorithm [8] is a quantum search algorithm that provides a quadratic speedup over classical counterparts. Its complexity grows with the number of qubits, which in turn affects the depth of the quantum circuit and the qubit connectivity.

Table 4
Backend Selection for Job 2 (GHZ State - using 10 qubits). ibmq_hanoi backend is chosen with the highest $\epsilon = 0.8$.

Backend	Qubits	QV (v)	$\bar{v}$	QC depth (d)	$\bar{d}$	CLOPS (c)	$\bar{c}$	Workload (w)	$\bar{w}$	p	s	ϵ
ibm_washington	127	64	0.33	13	1	850	0	5	1	0.67	0.5	0.59
ibmq_kolkata	27	128	1	13	1	2000	0.74	212	0	1	0.37	0.69
ibm_hanoi	27	64	0.33	13	1	2300	0.94	22	0.92	0.67	0.93	**0.8**
ibmq_guadalupe	16	32	0	13	1	2400	1	103	0.53	0.5	0.77	0.64
ibm_perth	7	–	–	–	–	–	–	–	–	–	–	–

Table 5
Backend Selection for Job 3 (Shor's algorithm to factorize number 9 - using 18 qubits). ibmq_kolkata backend is chosen with the highest $\epsilon = 0.7$.

Backend	Qubits	QV (v)	$\bar{v}$	QC depth (d)	$\bar{d}$	CLOPS (c)	$\bar{c}$	Workload (w)	$\bar{w}$	p	s	ϵ
ibm_washington	127	64	0.33	38 381	0	850	0	4	1	0.17	0.5	0.34
ibmq_kolkata	27	128	1	37 776	1	2000	0.79	130	0	1	0.4	**0.7**
ibm_hanoi	27	64	0.33	38 190	0.32	2300	1	57	0.58	0.33	0.79	0.56
ibmq_guadalupe	16	–	–	–	–	–	–	–	–	–	–	–
ibm_perth	7	–	–	–	–	–	–	–	–	–	–	–

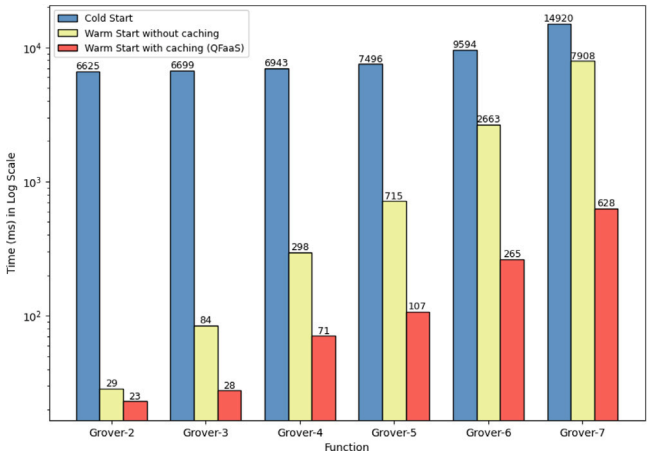


Fig. 9. Quantum function cold start latency mitigation evaluation with Grover-n function, where n is the number of qubits. Grover's algorithm circuits and transpiled QASM files adopted from the MQT Bench dataset [46], transpilation optimization level 3 to IBM Quantum 27-qubit devices.

We deploy different variations of Grover's algorithm from 2 qubits to 7 qubits, using the quantum circuits and transpiled QASM files of the MQT Bench dataset [46] (no ancilla qubit version, transpilation optimization level 3 to IBM Quantum 27-qubit quantum computers). We measure the average start-up latency from the function invocation until the function is ready for the quantum execution (i.e., finish the circuit compilation and transpilation) with the results as shown in Fig. 9.

It is obvious that our transpilation caching approach (warm start with caching) significantly reduces the latency compared to both the cold start and the warm start without caching scenarios. For example, in the Grover-7 function (7 qubits), the cold start latency is around 15 s, the warm start without transpilation caching is around 8s, while the QFaaS transpilation caching approach brings it down to under 0.628s, demonstrating a substantial improvement in execution preparation time. This pattern of reduction holds consistently across all evaluated Grover-n functions, with the caching approach offering a significant decrease in function preparation latency. This evaluation suggests that the QFaaS's transpilation caching strategy can effectively handle the classical cold start problem for quantum function execution, thereby enhancing the system's responsiveness and performance for end-users.

6.5. Industry use cases of QFaaS

It is imperative for industries to start investing in quantum computing services to stay competitive in the cloud-based computing market. QFaaS can serve as a proof-of-concept case study and reference system architecture design for further practical development. For example, Citynow Asia, a Japan-based quantum-driven company, has already started using QFaaS to develop their quantum serverless platform (QuaO) [52]. We anticipate a growing interest in quantum serverless computing and the adoption of QFaaS in the future.

7. Discussion and lessons learned

This section presents key lessons learned regarding both the opportunities and limitations of the QFaaS framework, providing a roadmap for future advancements in QFaaS and the broader quantum serverless domain. Throughout the empirical development and rigorous validation of our QFaaS framework, we have gained pivotal insights into the prospects of the serverless approach to quantum computing, including:

- ***L1:*** *Serverless computing holds great potential to accelerate quantum software development.*
  From the software developers' perspective, serverless approaches can help to obviate the necessity for quantum software environment setup, service deployment, and quantum infrastructure configuration, enabling them to concentrate on application development and experimentation without worrying about the underlying system. Quantum functions can be developed and deployed in a manner analogous to classical functions, ensuring seamless integration into existing application workflows. For quantum cloud providers, serverless models can provide an efficient, cost-effective means for allocating resources, optimizing utilization, and reducing idle time. By effectively implementing serverless models and offering a competitive pay-per-use cost model, providers can enhance user engagement and encourage long-term commitment to their services.
- ***L2:*** *The state-of-the-art software techniques and workflows can be effectively leveraged to expedite the quantum serverless paradigm.*
  Our work not only theorizes but also empirically demonstrates the adaptation of the DevOps methodology and techniques within the QFaaS framework, such as containerization, continuous integration, and continuous deployment. The development process of a quantum function can further be split into multiple stages. Quantum simulators can be used in the initial prototyping and testing phase, while quantum computers can be used later on during the production stage.
- ***L3:*** *The hybrid architecture of QFaaS represents one of the adaptive and practical approaches to facilitate quantum serverless systems,*

reflecting the current reliance of quantum execution on classical runtimes.

This architecture highlights the crucial cooperation between classical and quantum computation resources. In this setup, the classical counterpart acts as the runtime server, responsible for storing and compiling the quantum code prior to its execution on the designated quantum computer. The abstraction for decomposing a complex application into multiple smaller quantum functions adapts to the NISQ devices. The development and deployment of quantum functions are simplified and streamlined by the employment of DevOps techniques. The quantum function performance can also be optimized based on the needs of users thanks to the lightweight yet adaptive backend selection strategy.

Despite the promising potential of the quantum serverless approach, the current state of quantum hardware and software presents several significant challenges. These serve as crucial lessons learned for further exploration in our work:

- ***L4:*** *Current NISQ hardware is expensive, unreliable, and constrained, making it significantly challenging to incorporate the quantum serverless paradigm into production environments.*

  The intrinsic noise in these devices poses substantial challenges to the reliability and accuracy of quantum operations, where techniques such as quantum error correction and mitigation can be employed. Furthermore, the execution of tasks on real quantum devices is often associated with long queuing times and inconsistent execution durations, making them less suitable for time-sensitive and real-time applications. Throughout our empirical study with QFaaS, even the execution of Shor's algorithms requires minutes on current platforms, which is unanticipated for a common serverless execution. However, the rapid advances in quantum hardware in recent years hold promise for addressing this problem in the near future.
- ***L5:*** *Quantum resources cannot be scaled in the same manner as their classical counterparts.*

  Contrary to the classical domain, where serverless offers clear scalability and resource management benefits, the quantum realm introduces unique challenges. The scaling constraint of quantum hardware underscores the need for unique approaches to resource management in quantum serverless architectures. Our research recognizes that the advantages seen in classical serverless computing may not translate directly to quantum computing without significant adaptations and optimizations, specifically concerning the scalability of quantum resources. Techniques such as virtualization and containerization for quantum resources are potential directions to improve the utilization and scalability of quantum computing resources in the future.
- ***L6:*** *The quantum serverless model is still in its early stages, and there are numerous open problems that require extensive research and attention.*

  The application of serverless computing for quantum applications is still an emerging area where more knowledge is needed to understand the full extent of its suitability and benefits. A serverless approach may introduce challenges for non-trivial quantum applications that require iterative adjustment and optimization, such as quantum machine learning. Besides, the nature of current NISQ devices can also limit the potential of a quantum serverless approach. Despite this, the serverless approach has gained prominence due to its ability to reduce costs, improve scalability, and eliminate the need for hardware-side management, which can be particularly beneficial for future quantum software development. Indeed, quantum vendors, such as IBM Quantum, have placed quantum serverless as their key priority in the development roadmap [53]. There are numerous open problems that require further extensive research and attention to exploit the potential of a quantum serverless approach. Key areas that necessitate additional exploration include circuit cutting, optimizing the orchestration of hybrid quantum–classical tasks, circuit transpilation caching, reducing quantum cold start, and developing adaptive backend selection mechanisms. Resolving these challenges will play a critical role in facilitating the widespread adoption of the quantum serverless paradigm in the near future.

8. Related work

This section discusses the related work in the context of frameworks for developing service-oriented quantum software. To the best of our knowledge, QFaaS can be considered one of the pioneers in serverless-based function-as-a-service frameworks for quantum computing. Table 6 summarizes the difference between QFaaS and related work in the context of their various capabilities. It is important to note that all related frameworks [54,55], empirical studies [10,56] and minimum viable product (MVP) [57] designs did not provide an open-source or detailed description for reproducible purposes. Besides, two other SDKs [25,58] are related but their classification is not a quantum serverless framework. Therefore, we can only use the feature information reported in these studies to compare with our framework capabilities for reference purposes but cannot fully validate the features of related work in practice.

Existing quantum software platforms lack various features to provide a universal environment for developing service-oriented quantum applications. Hevia et al. proposed QuantumPath (QPath) [54], which is a software development platform aiming to support multiple quantum SDKs for gate-based and annealing quantum applications, and provide multiple design tools for creating a quantum algorithm. However, QPath does not support a serverless computing model and scalability features for further expansion, and it is still in the preliminary phase without providing a performance evaluation to validate the proposed design. Fu et al. [25] proposed the overall framework for developing heterogeneous quantum–classical applications, adapting with NISQ devices. Instead of working with popular quantum languages and SDKs, they also proposed a new programming language for quantum computing, called Quingo. Although this is an exciting direction for further quantum framework development, it can face many challenges when developing a new programming language compared to improving well-known languages. For example, expanding the support for large developer communities, security testing for potential vulnerabilities, and covering all aspects of quantum and classical computation. In another way, Claudino et al. [55] proposed the XACC framework, which extended the C++ programming language to support quantum chemistry simulations. However, this framework focused solely on designing multiple quantum algorithms for chemical problems and simulating quantum computation on GPU backends. Cambridge Quantum Computing proposed $t|ket\rangle$ [58], which is an open-source language-agnostic quantum compiler for NISQ devices. This framework focuses on circuit optimizations, transformation, and qubit mapping features and does not consider the service-oriented quantum application approach. In light of the Quantum Computing as a Service approach, Garcia-Alonso et al. [10] proposed the proof-of-concept about Quantum API Gateway with two simple API endpoint validations (execution and feedback) using Python and Flask platform on the Amazon Braket platform. This paper also presented an execution time forecasting model and quantum computer recommendation. Still, no information about what kind of quantum SDKs, quantum problems, or datasets are used for the forecasting is provided. There is also some Proof-of-Concept (PoC) designs for cloud-based quantum software development that have been proposed in recent years. For instance, Sim et al. [56] proposed *algo2qpu*, a hardware and software agnostic framework that supports designing and testing hybrid quantum–classical algorithms on the Rigetti cloud-based quantum computer. Using their proposed framework, they implemented two applications in quantum chemistry and machine learning. However, similar to the work mentioned above, *algo2qpu* also

Table 6
A summary of related work and their comparison of system and software engineering aspects for quantum computing (✓: Yes, ×: No, -: Not validated).

Criteria		QPath [54]	Quingo [25]	XACC [55]	QAPI [55]	t|ket [58]	algo2qpu [56]	SCQ [57]	QFaaS *(Proposed)*
Type (Main category)		Frame-work	Quantum SDK	Frame-work	Empirical Study	Compiler and SDK	Empirical Study	MVP Design	Frame-work
Systems	1. Serverless without Vendor lock-in	×	×	×	×	×	×	×	✓
	2. Modular/Microservices Components	×	×	×	×	✓	×	✓-	✓
	3. Support multiple Quantum Clouds	✓	×	×	×	✓	×	×	✓
	4. Support multiple Quantum Simulators	✓	×	×	×	✓	×	×	✓
	5. Containerization/Kubernetes Integration	×	×	×	×	×	×	✓-	✓
	6. Distributed and Scalable System	×	×	×	×	×	×	×	✓
	7. Evaluation on NISQ devices	×	×	✓	✓	✓	✓	×	✓
	8. Permanent Data Storage	✓	×	×	×	×	×	×	✓
	9. Security & User Authentication	×	×	×	×	×	×	×	✓
	10. Monitoring Integration	×	×	×	×	×	×	×	✓
Software	11. Hybrid Quantum-Classical Integration	×	✓	✓	×	×	✓	✓-	✓
	12. Support multiple Quantum SDKs	✓	×	×	×	✓	×	×	✓
	13. Built-in Software Library/Templates	✓	✓	✓	×	✓	×	×	✓
	14. Quantum Software Workflows	×	✓	✓	×	×	✓	×	✓
	15. Quantum Backend Selection	×	×	×	×	×	×	×	✓
	16. REST API support with API Gateway	×	×	×	✓	×	×	×	✓
	17. Full-stack Software Framework	✓	×	×	×	✓	×	×	✓
	18. DevOps (CI/CD) Integration	×	×	×	×	×	×	×	✓
	19. Practical Use Cases	✓	✓	✓	×	✓	✓	×	✓
	20. Open-Source Software	×	✓	×	×	✓	×	×	✓

did not apply the serverless quantum computing model but considered the standalone quantum application model instead. Another minimum viable product (MVP) design proposed by Grossi et al. [57] suggested using IBM services such as IBM Cloud Functions and IBM Containers to integrate serverless features for Qiskit programs. However, this work only considered single-SDK and single-vendor environments with no implementation or validation provided. This work did not consider the software workflow or the backend selection strategy and depended on a specific quantum SDK and platform technology. This approach can lead to a data lock-in problem, which is one of the most serious challenges of serverless computing.

As shown in Table 6, we use ten criteria (1–10) to evaluate in terms of system architecture and computing model. The major contribution of QFaaS is one of the first frameworks adopting the serverless function-as-a-service model for quantum computing, which avoids vendor lock-in problems of serverless computing by supporting multiple quantum SDKs/programming languages, quantum cloud providers, and simulators (1). The idea of serverless integration is also proposed in [57] using Qiskit SDK and IBM techniques. Still, no implementation and evaluation are provided to validate the design, and it can also lead to the vendor lock-in problem as that design relies on single vendor techniques. The core elements of the QFaaS system are developed on top of open-source cloud-native technology, such as Docker container, Kubernetes, OpenFaaS, and Gitlab, with flexibility for further expansion and integration (2). This system design leverages the state-of-the-art techniques in classical cloud computing to support the quantum computing as a service (QCaaS) model, which adapts to the limitations of the NISQ computers. We validated the system design and used QFaaS to evaluate the performance of multiple quantum simulators and computers to provide insight into the current state of the NISQ era (3–7). The database integration with a common data scheme for all different SDKs in QFaaS allows users to perform further analyses and avoids data lock-in issues (8). Furthermore, we also incorporate other essential system components, such as monitoring and security for QFaaS, to demonstrate the completed quantum software stack in practice (9–10).

Regarding software engineering aspects to facilitate the Quantum FaaS model, we consider the ten remaining features (11–20). Our framework simplifies the hybrid quantum–classical integration model by providing function templates and a supported software library (11–12). It allows users to write both classical and quantum code in a single file. Besides, we also propose an adaptable quantum software life cycle with seven stages of a quantum function and demonstrate its implementation in practice. This is the first quantum function life cycle, as other proposals in the literature focused on the general, standalone quantum software (14). We design a backend selection strategy and directly apply it to QFaaS to automatically select the most suitable quantum backends for executing the quantum computation parts (15). As several works in the literature only proposed the proofs-of-concept (PoC) [10,56,57], we do not only propose the PoC but also fully implement and validate the proposed PoC with practical use cases (19) of well-known quantum circuits. Apart from adopting open-source cloud-native techniques, we also develop full-stack software components for quantum functions. QFaaS backend included a complete OpenAPI set with a centralized API gateway (16), a Python-based supported library, and multiple Docker-based function templates (13), where its front end is a user-friendly web application. We demonstrate the first integration of continuous integration and continuous deployment (CI/CD) of DevOps into the quantum software workflow (18). Finally, as we incorporated several latest open-source techniques, our framework is also designed to be a part of the quantum open-source software ecosystem to contribute to our effort in the early development of serverless quantum computing (20).

In summary, as no existing quantum function-as-a-service framework in the literature consider the multi-SDK, multi-cloud environment, QFaaS is one of the first practical platforms that seriously investigated and prepared the initial steps towards a universal serverless quantum computing architecture. Our major innovation also includes bringing state-of-the-art system design and software engineering techniques in classical computing to support service-oriented quantum software development and mitigate the challenges in the current NISQ era.

9. Conclusions and future work

We proposed and developed QFaaS - a holistic serverless framework for developing quantum function as a service, enabling traditional

software engineers to leverage their knowledge and experience to adapt to quantum counterparts in the *Noisy Intermediate-Scale Quantum* era quickly. Our framework brings state-of-the-art methods such as containerization, DevOps, and the serverless model to reduce the burden of quantum software development and pave the way towards combining hybrid quantum and classical components. QFaaS provides essential features with multiple quantum software environments, leveraging the well-known quantum SDKs and languages to develop quantum functions running on multiple quantum simulators or cloud-based quantum computers. The current implementation of QFaaS demonstrates the possibility and advantages of our framework in developing quantum function as a service without vendor lock-in issues and can be seamlessly integrated into the current software workflow. Throughout our empirical implementation and evaluation, we also highlight the lessons learned and limitations of the quantum serverless approach that need to be further investigated.

Due to the current restriction around access to quantum cloud services from our region, we are able to demonstrate the experiments with IBM Quantum and Amazon Braket (through Strangeworks) at the moment. We plan to extend QFaaS's ability to connect with other providers and enable the cross-platform execution feature in the future. Besides, we are developing a machine learning-based approach for improving the automatic selection of the quantum backend for hybrid quantum–classical applications and the cold start mitigation policy for quantum function execution. We will also enhance the security and scalability capabilities in QFaaS to support a large number of requests from multiple users. As Quantum Software Engineering is still an emerging area of research with numerous challenges, there is a need for a significant research effort to make it reliable and simultaneously adapt to rapid advances in quantum hardware.

Software availability

The QFaaS Framework with the source code of all components and sample functions code can be accessed from our iQuantum Initiative website http://clouds.cis.unimelb.edu.au/iquantum.

CRediT authorship contribution statement

Hoa T. Nguyen: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Data curation, Conceptualization. **Muhammad Usman:** Writing – review & editing, Validation, Supervision. **Rajkumar Buyya:** Writing – review & editing, Validation, Supervision, Methodology, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

This work is partially supported by the University of Melbourne, Australia by establishing an IBM Quantum Network Hub and the Nectar Research Cloud. We appreciate the support from Strangeworks Backstage Program for providing access to Amazon Braket. Hoa Nguyen acknowledges the support from the Science and Technology Scholarship Program for Overseas Study for Master's and Doctoral Degrees, Vingroup, Vietnam.

Appendix A. A brief introduction of quantum computing

This section summarizes essential characteristics and building blocks of gate-based quantum computing for broad readers to get familiar with the core concepts of the quantum computing model.

A.1. Qubits, superposition, and entanglement

Quantum computing is based on the theory of quantum mechanics and, therefore, is fundamentally different from classical computing [9]. The basic units of classical and quantum computing are strikingly different at the fundamental level: a classical bit and a quantum bit (or qubit). A bit has two states for computation, either 0 or 1. Besides these classical states, a qubit can have a *superposition* state, i.e., a combination of states 0 and 1 simultaneously. Quantum algorithms can achieve exponential speed-up by leveraging this characteristic compared with the classical solution. We can describe the general state of a qubit $|\psi\rangle$ as follows:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

where $\alpha, \beta \in \mathbb{C}$ are complex numbers, $|\rangle$ is Dirac notation to describe quantum states [9]. However, whenever we measure the superposition state, it could collapse to one of the classical states (i.e., 0 or 1):

$$\|\alpha\|^2 + \|\beta\|^2 = 1$$

where $\|\alpha\|^2$ and $\|\beta\|^2$ is the probability of 0 and 1 as a result after measuring qubit $|\psi\rangle$. Hence, it is not straightforward to design a useful quantum algorithm by only utilizing the superposition attribute. Another critical characteristic of qubits that could be leveraged to design quantum algorithms is *entanglement*. Entanglement is a robust correlation between two qubits, i.e., one qubit always knows exactly the state of the other qubit after measurement, even if they are very far away. In other words, if a pure state $|\psi\rangle_{AB}$ on two systems A and B cannot be written as $|\psi\rangle_A \otimes |\phi\rangle_B$, we called it *entangled* [18].

A.1.1. Quantum gates and quantum circuits

To perform the quantum operations on qubits, we apply quantum gates, which are conceptually similar to how we apply classical gates, such as AND, OR, XOR, and NOT on classical bits to perform classical computation. For example, we can use the Hadamard (H) gate on qubit $|0\rangle$ to create an equal superposition $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle+|1\rangle)$. A logic quantum gate U can be represented by a unitary matrix such that $U^\dagger U = \mathbb{I}$ where I is the identity matrix. The general representation of a single-qubit gate U is as follows:

$$U(\theta, \phi, \lambda) = \begin{bmatrix} \cos(\theta/2) & -e^{i\lambda}\sin(\theta/2) \\ e^{i\phi}\sin(\theta/2) & e^{i\lambda+i\phi}\cos(\theta/2) \end{bmatrix}$$

where θ, ϕ, λ are different parameters for each specific gate [18].

We can categorize quantum gates into two main types: single-qubit and multiple-qubit gates. Some popular single-qubit gates are the Pauli gates (X, Y, Z), the Hadamard (H) gate, and the Phase (P) gate. For example, the H gate can be represented as the following (with $\theta, \phi, \lambda = \frac{\pi}{2}, 0, \pi$, respectively):

$$H = U(\tfrac{\pi}{2}, 0, \pi) = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

We can also apply quantum gates to multiple qubits simultaneously by using multi-qubit gates, such as the Controlled-NOT (CNOT) gate and the Toffoli gate. CNOT gate, for instance, is a controlled two-qubit gate. If the control qubit is $|1\rangle$, the target qubit is flipped. Otherwise, if the control qubit is $|0\rangle$, the target qubit remains unchanged [9]. For example,

$$\text{CNOT}|01\rangle = |01\rangle; \quad \text{CNOT}|10\rangle = |11\rangle;$$

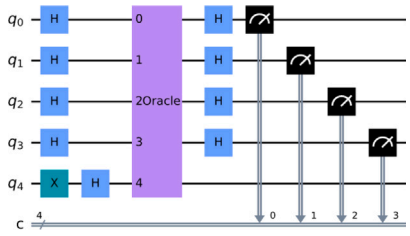


Fig. A.10. An example quantum circuit for the Deutsch-Jozsa algorithm (generated by using Qiskit).

A.1.2. Quantum circuits and quantum algorithms

When implementing a quantum algorithm using the gate-based approach, we need to connect an appropriate combination of quantum gates to build quantum circuits. A quantum circuit's general operation includes three main stages: (1) Initializing the qubits, (2) Applying the quantum gates, and (3) Measuring.

For example, the quantum circuit shown in Fig. A.10 implements Deutsch-Jozsa's algorithm [6]. The main objective of this algorithm is to determine whether the property of the oracle is constant (i.e., always return 0 or 1) or balanced (i.e., return 0 and 1 with the same probability). The oracle in this circuit is a *"black box"* where we do not know which binary value is inside. However, when we query it with arbitrary input data, it will return a binary answer, either 0 or 1. For the traditional approach, we need to interact with the oracle at least two times and at most $2^{n-1}+1$ times, where n is the number of input bits. Using the Deutsch-Jozsa algorithm, we need to query the oracle only once to get the final result. If the measurement outcomes of all the qubits are 0, we can determine that the oracle is constant; otherwise, it is balanced. This algorithm was also the first to demonstrate that quantum computers could outperform the classical computer in 1992 [18].

Appendix B. Using QFaaS to evaluate performance of NISQ devices and simulators

B.1. Sample QRNG circuits for the evaluation

Fig. B.12 shows sample code snippets of different SDKs for generating quantum random numbers. According to the user's request, an appropriate quantum circuit will be generated with the corresponding number of qubits. This circuit can be then being transpiled to adapt to the targeted quantum backend (see Fig. B.11).

B.2. Performance evaluation of different quantum simulators using QFaaS

In this evaluation, we used QFaaS to benchmark the performance of four state-of-the-art quantum simulators, which are associated with four popular quantum SDKs we integrated. For a practically fair comparison, we used the default quantum simulator (QASM simulator for Qiskit, *braket_sv* simulator for Braket, and built-in simulator for Q# and Cirq) of all frameworks for execution. We repeat each experiment 100 times, then measure the average response time and the standard deviation when executing the QRNG function using 1 qubit to 20 qubits in each quantum SDK.

Fig. B.13 illustrates the average response time of four functions when we increase the number of qubits *(n)* from 1 to 20. The Cirq simulator registers the fastest response time in all test cases with a slight increase from 49 ms for generating a 1-qubit random number to 61 ms for a 20-qubit one. A similar trend could also be seen if we look at the Qiskit, Q#, and Braket function figures when *n* increases from 1 to 15. The Qiskit function response time is slightly longer than Cirq and Q# during the 1- to 15-qubit span. However, when *n* reaches 20, the

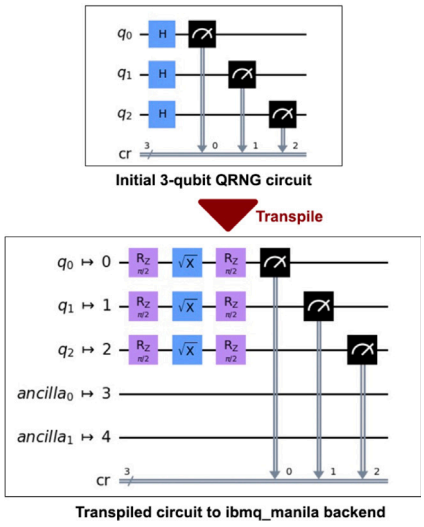


Fig. B.11. A simple quantum circuit for generating 3-qubit random number and the corresponding transpilation to adapt to ibmq_manila backend.

Fig. B.12. Sample code snippets for QRNG functions.

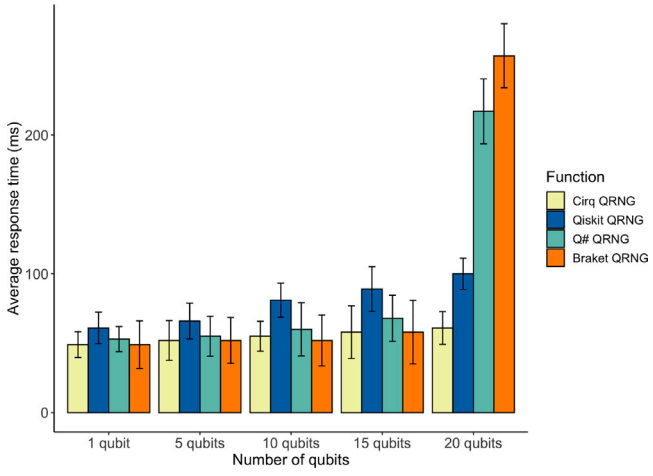


Fig. B.13. Average response time evaluation of QRNG function using the simulator of 4 popular quantum SDKs and languages.

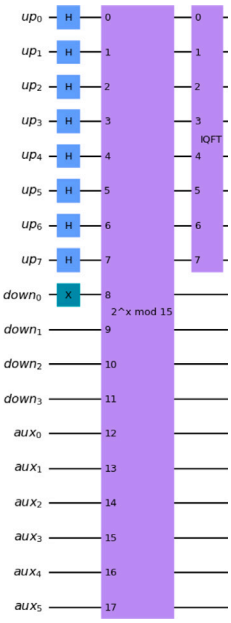


Fig. B.14. A simplified 18-qubit quantum circuit for implementing Shor algorithm to factorize 15 (plotted using Qiskit).

response time of the Q# and Braket functions increases significantly and doubles the Qiskit counterpart. This evaluation demonstrates the current state of several popular quantum simulators, but it depends on specific quantum applications and can be changed with the further development of these SDKs.

B.3. Performance evaluation of NISQ computers and simulator with Shor's algorithm on QFaaS

Shor's algorithm [7] is one of the most prominent quantum algorithms for proving the advantage of quantum computing together with its classical counterpart. It is well-known for finding the prime factors of integers in polynomial time, which raises the severe risk for classical cryptography based on the security of large integers such as RSA. In this case study, we demonstrate the implementation of Shor's algorithm as a QFaaS function (Shor function) by utilizing the Qiskit Terra API [19]. The quantum circuit for Shor's algorithm is also dynamically generated based on the input number that end-users want to factorize. For example, to factorize 15, we need to use 18 qubits for the corresponding quantum circuit (see Fig. B.14).

Using the Shor class in `qiskit.algorithms`,[13] we need to define a simple function code to generate the quantum circuit to factorize the integer N, then execute that circuit at the appropriate backend device selected automatically by QFaaS or manually by the end-users. Fig. B.14 shows a general 18-qubit quantum circuit for factorizing 15 using Shor's Algorithm and a code snippet of Qiskit Shor's function.

As a demonstration illustrated in Fig. B.15, we use the QFaaS Dashboard to invoke the Shor function to factorize 21, using Qiskit QASM Simulator at QFaaS Classical Cloud Layer. The request is converted to JSON format automatically and sent to the QFaaS API gateway to trigger the Shor function. After finishing the execution, the final response data is returned, and we got two expected factors 21, i.e., 3 and 7.

In this evaluation, we compare the actual performance of the Shor function with today's quantum computers provided by IBM Quantum [14]. We pick five *adequate* integer numbers for the test cases,

[13] https://qiskit.org/documentation/stubs/qiskit.algorithms.Shor.html

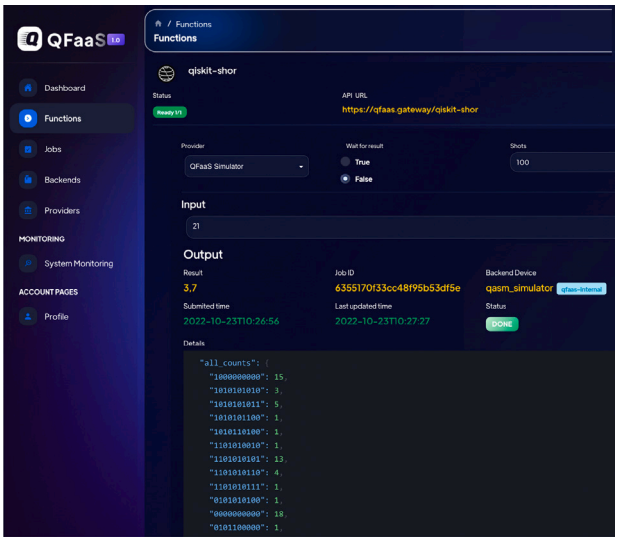


Fig. B.15. Sample Shor's Function Invocation Result (factorizing 21) on QFaaS Dashboard using Qiskit QASM Simulator.

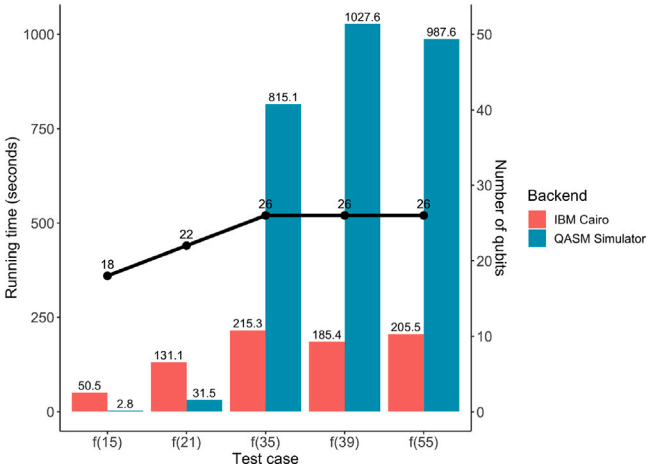


Fig. B.16. Performance evaluation of Shor function on (`ibm_cairo`) quantum computer and simulator provided by IBM Quantum. f(15), f(21), f(35), f(39), and f(55) are five test cases to factorize 15, 21, 35, 39, and 55, respectively.

including 15, 21, 35, 39, and 55. Due to the limitation of the current NISQ devices, we keep these test cases small to fit with the capacity of available quantum backends. All experiments are conducted on a 27-qubit quantum computer (`ibm_cairo`, using Falcon r5.11 quantum processor) and the QASM simulator (`ibmq_qasm_simulator`).

Every time we invoke the Shor function with each test case, an appropriate circuit will be generated and sent to the IBM Quantum. Then, each quantum job will be validated and kept in the queue (from seconds to hours) before being executed in the backend due to the current fair-share policy of IBM Quantum. Therefore, to make a fair comparison, we only measure the actual running time (including the circuit validation and running, without the queuing time). We execute each quantum task 100 times (shots) to ensure that the final result of all factorization is correct. In Fig. B.16, the bar chart shows the actual running time, and the line chart indicates the number of qubits used for each test case in both backends. These input numbers need less than 27 qubits to build a corresponding circuit. Regarding the run time, we can see that the QASM simulator is much faster than the quantum computer when the number of required qubits is small, from 18 to 22 (for factorizing 15 and 21). However, we can see the opposite trend

when executing 26-qubit circuits to factorize 35, 39, and 55. These circuits cost around 3 mins to complete in an IBM Cairo quantum computer, whereas the QASM simulator takes 13.5 to 17 mins to finish the execution. A significant reason for the considerable delay of the QASM Simulator in these test cases can be the complexity of the 26-qubit quantum circuit for the Shor algorithm, which requires a lot of resources to simulate. These results give us insight into the selection order of existing quantum computing services for developing quantum software. We can use the quantum simulator for the prototyping and testing phases before entering the production stage with the quantum computers.

References

[1] S.S. Gill, A. Kumar, H. Singh, M. Singh, K. Kaur, M. Usman, R. Buyya, Quantum computing: A taxonomy, systematic review and future directions, Softw. - Pract. Exp. 52 (1) (2022) 66–114, http://dx.doi.org/10.1002/spe.3039.

[2] A. Kandala, A. Mezzacapo, K. Temme, M. Takita, M. Brink, J.M. Chow, J.M. Gambetta, Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets, Nature 549 (7671) (2017) 242–246, http://dx.doi.org/10.1038/nature23879.

[3] M.T. West, S.-L. Tsang, J.S. Low, C.D. Hill, C. Leckie, L.C.L. Hollenberg, S.M. Erfani, M. Usman, Towards quantum enhanced adversarial robustness in machine learning, Nat. Mach. Intell. (2023) http://dx.doi.org/10.1038/s42256-023-00661-1.

[4] J. Quan, Q. Li, C. Liu, J. Shi, Y. Peng, A simplified verifiable blind quantum computing protocol with quantum input verification, Quantum Eng. 3 (1) (2021) 1–10, http://dx.doi.org/10.1002/que2.58.

[5] P. Griffin, R. Sampat, Quantum computing for supply chain finance, in: Proceedings of 2021 IEEE International Conference on Services Computing, SCC 2021, IEEE, Chicago, IL, USA, 2021, pp. 456–459, http://dx.doi.org/10.1109/SCC53864.2021.00066.

[6] D. Deutsch, R. Jozsa, Rapid solution of problems by quantum computation, Proc. R. Soc. Lond. Ser. A 439 (1907) (1992) 553–558, http://dx.doi.org/10.1098/rspa.1992.0167.

[7] P.W. Shor, Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer, SIAM J. Comput. 26 (5) (1997) 1484–1509, http://dx.doi.org/10.1137/S0097539795293172, URL: http://epubs.siam.org/doi/10.1137/S0097539795293172.

[8] L.K. Grover, A fast quantum mechanical algorithm for database search, in: Proceedings of the 28th ACM Symposium on Theory of Computing, Vol. 75, no. 6, STOC '96, ACM, New York, USA, 1996, pp. 212–219, http://dx.doi.org/10.1145/237814.237866.

[9] M.A. Nielsen, I.L. Chuang, Quantum Computation and Quantum Information, Cambridge University Press, 2012, p. 676, http://dx.doi.org/10.1017/CBO9780511976667.

[10] J. Garcia-Alonso, J. Rojo, D. Valencia, E. Moguel, J. Berrocal, J.M. Murillo, Quantum software as a service through a quantum API gateway, IEEE Internet Comput. 26 (1) (2022) 34–41, http://dx.doi.org/10.1109/MIC.2021.3132688.

[11] B. Weder, J. Barzen, F. Leymann, D. Vietz, Quantum software development lifecycle, in: M.A. Serrano, R. Pérez-Castillo, M. Piattini (Eds.), Quantum Software Engineering, Springer International Publishing, Cham, 2022, pp. 61–83, http://dx.doi.org/10.1007/978-3-031-05324-5.

[12] F. Gemeinhardt, A. Garmendia, M. Wimmer, Towards model-driven quantum software engineering, in: Proceedings of the 2021 IEEE/ACM 2nd International Workshop on Quantum Software Engineering, Q-SE 2021, Institute of Electrical and Electronics Engineers Inc., 2021, pp. 13–15, http://dx.doi.org/10.1109/Q-SE52541.2021.00010.

[13] J. Preskill, Quantum computing in the NISQ era and beyond, Quantum 2 (2018) 79, http://dx.doi.org/10.22331/q-2018-08-06-79, URL: https://quantum-journal.org/papers/q-2018-08-06-79/.

[14] IBM, IBM Quantum, 2021, URL: https://quantum-computing.ibm.com/.

[15] C. Gonzalez, Cloud based QC with Amazon Braket, Digitale Welt 5 (2) (2021) 14–17, http://dx.doi.org/10.1007/s42354-021-0330-z, URL: http://link.springer.com/10.1007/s42354-021-0330-z.

[16] Microsoft, Azure Quantum, 2021, URL: https://azure.microsoft.com/en-us/services/quantum.

[17] A.A. Khan, A. Ahmad, M. Waseem, P. Liang, M. Fahmideh, T. Mikkonen, P. Abrahamsson, Software architecture for quantum computing systems — A systematic review, J. Syst. Softw. 201 (2023) 111682, http://dx.doi.org/10.1016/j.jss.2023.111682.

[18] IBM Quantum, Qiskit Textbook, IBM, Online, 2022, URL: https://qiskit.org/textbook/.

[19] G. Aleksandrowicz, T. Alexander, P. Barkoutsos, L. Bello, Y. Ben-Haim, D. Bucher, F.J. Cabrera-Hernández, et al., Qiskit: An open-source framework for quantum computing, 2019, http://dx.doi.org/10.5281/ZENODO.2562111, URL: https://zenodo.org/record/2562111.

[20] Google, Cirq Framework, 2021, http://dx.doi.org/10.5281/zenedo.5182845, URL: https://zenodo.org/record/5182845.

[21] Microsoft, Q# quantum programming language, 2021, URL: https://github.com/microsoft/qsharp-language.

[22] R.S. Smith, M.J. Curtis, W.J. Zeng, A practical quantum instruction set architecture, 2016, http://dx.doi.org/10.48550/arXiv.1608.03355, URL: http://arxiv.org/abs/1608.03355.

[23] N. Killoran, J. Izaac, N. Quesada, V. Bergholm, M. Amy, C. Weedbrook, Strawberry fields: A software platform for photonic quantum computing, Quantum 3 (2019) 129, http://dx.doi.org/10.22331/q-2019-03-11-129, URL: https://quantum-journal.org/papers/q-2019-03-11-129/.

[24] V. Bergholm, J. Izaac, M. Schuld, et al., PennyLane: Automatic differentiation of hybrid quantum-classical computations, 2018, http://dx.doi.org/10.48550/arXiv.1811.04968, URL: http://arxiv.org/abs/1811.04968.

[25] X. Fu, J. Yu, X. Su, H. Jiang, H. Wu, et al., Quingo: A programming framework for heterogeneous quantum-classical computing with NISQ features, ACM Trans. Quantum Comput. 2 (4) (2021) 1–37, http://dx.doi.org/10.1145/3483528.

[26] D. Ittah, T. Häner, V. Kliuchnikov, T. Hoefler, QIRO: A static single assignment-based quantum program representation for optimization, ACM Trans. Quantum Comput. 3 (3) (2022) 1–32, http://dx.doi.org/10.1145/3491247, URL: https://dl.acm.org/doi/10.1145/3491247.

[27] A. Mccaskey, T. Nguyen, A. Santana, D. Claudino, T. Kharazi, H. Finkel, Extending C++ for heterogeneous quantum-classical computing, ACM Trans. Quantum Comput. 2 (2) (2021) 1–36, http://dx.doi.org/10.1145/3462670.

[28] A. Ahmad, A.B. Altamimi, J. Aqib, A reference architecture for quantum computing as a service, 2023, http://dx.doi.org/10.48550/arXiv.2306.04578, URL: http://arxiv.org/abs/2306.04578.

[29] J. Schleier-Smith, V. Sreekanti, A. Khandelwal, J. Carreira, N.J. Yadwadkar, R.A. Popa, J.E. Gonzalez, I. Stoica, D.A. Patterson, What serverless computing is and should become, Commun. ACM 64 (5) (2021) 76–84, http://dx.doi.org/10.1145/3406011.

[30] S. Eismann, J. Scheuner, E. van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C.L. Abad, A. Iosup, Serverless applications: Why, when, and how? IEEE Softw. 38 (1) (2021) 32–39, http://dx.doi.org/10.1109/MS.2020.3023302, URL: https://ieeexplore.ieee.org/document/9190031/.

[31] J. Scheuner, P. Leitner, Function-as-a-Service performance evaluation: A multivocal literature review, J. Syst. Softw. 170 (2020) 110708, http://dx.doi.org/10.1016/j.jss.2020.110708.

[32] M. Cerezo, G. Verdon, H.-Y. Huang, L. Cincio, P.J. Coles, Challenges and opportunities in quantum machine learning, Nat. Comput. Sci. (2022) http://dx.doi.org/10.1038/s43588-022-00311-3, URL: https://www.nature.com/articles/s43588-022-00311-3.

[33] A.W. Cross, L.S. Bishop, J.A. Smolin, J.M. Gambetta, Open quantum assembly language, 2017, http://dx.doi.org/10.48550/arXiv.1707.03429, URL: http://arxiv.org/abs/1707.03429.

[34] P. Leitner, E. Wittern, J. Spillner, W. Hummer, A mixed-method empirical study of Function-as-a-Service software development in industrial practice, J. Syst. Softw. 149 (2019) 340–359, http://dx.doi.org/10.1016/j.jss.2018.12.013.

[35] Y. Li, Y. Fei, W. Wang, X. Meng, H. Wang, Q. Duan, Z. Ma, Quantum random number generator using a cloud superconducting quantum computer based on source-independent protocol, Sci. Rep. 11 (1) (2021) 23873, http://dx.doi.org/10.1038/s41598-021-03286-9, URL: https://www.nature.com/articles/s41598-021-03286-9.

[36] C. Ebert, G. Gallardo, J. Hernantes, N. Serrano, DevOps, IEEE Softw. 33 (3) (2016) 94–100, http://dx.doi.org/10.1109/MS.2016.68.

[37] P.C. Lotshaw, T. Nguyen, A. Santana, A. McCaskey, R. Herrman, J. Ostrowski, G. Siopsis, T.S. Humble, Scaling quantum approximate optimization on near-term hardware, Sci. Rep. 12 (1) (2022) http://dx.doi.org/10.1038/s41598-022-14767-w.

[38] M. Van Steen, A. Tanenbaum, Distributed Systems, fourth ed., 2023, URL: https://www.distributed-systems.net/.

[39] V. Giménez-Alventosa, G. Moltó, M. Caballer, A framework and a performance assessment for serverless MapReduce on AWS Lambda, Future Gener. Comput. Syst. 97 (2019) 259–274, http://dx.doi.org/10.1016/j.future.2019.02.057.

[40] H.T. Nguyen, M. Usman, R. Buyya, iQuantum: A case for modeling and simulation of quantum computing environments, in: Proceedings of the 2023 IEEE International Conference on Quantum Software, QSW, IEEE, Chicago, USA, 2023, http://dx.doi.org/10.1109/QSW59989.2023.00013.

[41] A.W. Cross, L.S. Bishop, S. Sheldon, P.D. Nation, J.M. Gambetta, Validating quantum computers using randomized model circuits, Phys. Rev. A 100 (3) (2019) 032328, http://dx.doi.org/10.1103/PhysRevA.100.032328.

[42] E. Younis, C. Iancu, Quantum circuit optimization and transpilation via parameterized circuit instantiation, in: Proceedings of 2022 IEEE International Conference on Quantum Computing and Engineering, QCE, IEEE Computer Society, Los Alamitos, CA, USA, 2022, pp. 465–475, http://dx.doi.org/10.1109/QCE53715.2022.00068.

[43] A. Wack, H. Paik, A. Javadi-Abhari, P. Jurcevic, I. Faro, J.M. Gambetta, B.R. Johnson, Quality, Speed, and Scale: Three key attributes to measure the performance of near-term quantum computers, 2021, URL: http://arxiv.org/abs/2110.14108.

[44] P. Castro, V. Ishakian, V. Muthusamy, A. Slominski, The rise of serverless computing, Commun. ACM 62 (12) (2019) 44–54, http://dx.doi.org/10.1145/3368454, URL: https://dl.acm.org/doi/10.1145/3368454.
[45] M. Golec, G.K. Walia, M. Kumar, F. Cuadrado, S.S. Gill, S. Uhlig, Cold start latency in serverless computing: A systematic review, taxonomy, and future directions, 2023, arXiv:2310.08437.
[46] N. Quetschlich, L. Burgholzer, R. Wille, MQT Bench: Benchmarking software and design automation tools for quantum computing, Quantum (2023) http://dx.doi.org/10.22331/q-2023-07-20-1062, MQT Bench is available at https://www.cda.cit.tum.de/mqtbench/.
[47] A. Ellis, OpenFaaS - Serverless Functions Made Simple, 2022, URL: https://github.com/openfaas/faas.
[48] Strangeworks, Strangeworks quantum computing platform, 2022, URL: https://app.quantumcomputing.com/.
[49] M. Herrero-Collantes, J.C. Garcia-Escartin, Quantum random number generators, Rev. Modern Phys. 89 (2017) 015004, http://dx.doi.org/10.1103/RevModPhys.89.015004, URL: https://link.aps.org/doi/10.1103/RevModPhys.89.015004.
[50] L. Huang, H. Zhou, K. Feng, C. Xie, Quantum random number cloud platform, npj Quantum Inf. 7 (1) (2021) 107, http://dx.doi.org/10.1038/s41534-021-00442-x.
[51] D.M. Greenberger, GHZ (greenberger—Horne—Zeilinger) theorem and GHZ states, in: D. Greenberger, K. Hentschel, F. Weinert (Eds.), Compendium of Quantum Physics, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 258–263, http://dx.doi.org/10.1007/978-3-540-70626-7_78.
[52] A. Citynow, Enabling serverless quantum computing with QuaO, 2023, citynow.asia, URL: https://citynow.asia/press/release/Enabling-Serverless-Quantum-Computing-with-QuaO.
[53] IBM Quantum, IBM Quantum Development Roadmap 2022, 2022, URL: https://www.ibm.com/quantum/roadmap.
[54] J.L. Hevia, G. Peterssen, M. Piattini, QuantumPath : A quantum software development platform, Softw. - Pract. Exp. 2021 (December) (2021) 1–14, http://dx.doi.org/10.1002/spe.3064, URL: https://onlinelibrary.wiley.com/doi/10.1002/spe.3064.
[55] D. Claudino, A.J. McCaskey, D.I. Lyakh, A backend-agnostic, quantum-classical framework for simulations of chemistry in C ++ , ACM Trans. Quantum Comput. (2022) http://dx.doi.org/10.1145/3523285.
[56] S. Sim, Y. Cao, J. Romero, P.D. Johnson, A. Aspuru-Guzik, A framework for algorithm deployment on cloud-based quantum computers, 2018, http://dx.doi.org/10.48550/arXiv.1810.10576.
[57] M. Grossi, L. Crippa, A. Aita, G. Bartoli, V. Sammarco, E. Picca, N. Said, F. Tramonto, F. Mattei, A serverless cloud integration for quantum computing, 2021, http://dx.doi.org/10.48550/arXiv.2107.02007, URL: http://arxiv.org/abs/2107.02007.
[58] S. Sivarajah, S. Dilkes, A. Cowtan, W. Simmons, A. Edgington, R. Duncan, t|ket: A retargetable compiler for NISQ devices, Quantum Sci. Technol. 6 (1) (2021) http://dx.doi.org/10.1088/2058-9565/ab8e92.

Hoa T. Nguyen is a Ph.D. Candidate at The Cloud Computing and Distributed Systems(CLOUDS) Laboratory, University of Melbourne, Australia. He received his B.Eng. degree in Computer Networks and Communications and his MSc. degree in Computer Science from the University of Information Technology - Vietnam National University Ho Chi Minh City (UIT – VNU-HCM), in 2016 and 2019, respectively. His research interests include quantum cloud computing, serverless architectures, quantum software engineering, and cybersecurity.

Dr. Muhammad Usman is a Team Leader of Quantum Systems program at CSIRO and an Associate Professor (Honorary) at the University of Melbourne. His research interests and expertise include quantum software engineering, quantum algorithms, quantum physics and quantum processor design. Dr Usman has published over 70 peer-reviewed papers and have delivered several invited talks and seminars at international venues. He is a recipient of several awards and fellowships including USA Fulbright Fellowship, DAAD Research Fellowship, Rising Stars in Computational Materials Science and Best Research Award at the University of Melbourne. Dr Usman is an executive editorial board member at Nano Futures journal.

Dr. Rajkumar Buyya is a Redmond Barry Distinguished Professor and Director of the Cloud Computing and Distributed Systems (CLOUDS) Laboratory at the University of Melbourne, Australia. He has authored over 625 publications and seven text books including “Mastering Cloud Computing” published by McGraw Hill, China Machine Press, and Morgan Kaufmann for Indian, Chinese and international markets respectively. He is one of the highly cited authors in computer science and software engineering worldwide (h-index=154, gindex=322, 124,500+ citations).