Institute of Architecture of Application Systems

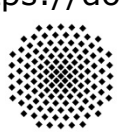
The NISQ Analyzer: Automating the Selection of Quantum Computers for Quantum Algorithms

Marie Salm, Johanna Barzen, Uwe Breitenbücher, Frank Leymann, Benjamin Weder, Karoline Wild

Institute of Architecture of Application Systems,
University of Stuttgart, Germany
{salm, barzen, breitenbuecher, leymann, weder, wild}@iaas.uni-stuttgart.de

BibTeX:

```
@inproceedings{Salm2020_NISQAnalyzer,
  author    = {Salm, Marie and Barzen, Johanna and Breitenb{\"u}cher, Uwe and
               Leymann, Frank and Weder, Benjamin and Wild, Karoline},
  title     = {{The NISQ Analyzer: Automating the Selection of Quantum
               Computers for Quantum Algorithms}},
  booktitle = {Proceedings of the 14th Symposium and Summer School on
               Service-Oriented Computing (SummerSOC 2020)},
  pages     = {66--85},
  publisher = {Springer International Publishing},
  month     = dec,
  year      = 2020,
  doi       = {10.1007/978-3-030-64846-6_5}
}
```

University of Stuttgart
Germany

The NISQ Analyzer: Automating the Selection of Quantum Computers for Quantum Algorithms

Marie Salm[0000−0002−2180−250X], Johanna Barzen[0000−0001−8397−7973], Uwe Breitenbücher[0000−0002−8816−5541], Frank Leymann[0000−0002−9123−259X], Benjamin Weder[0000−0002−6761−6243], and Karoline Wild[0000−0001−7803−6386]

Institute of Architecture of Application Systems, University of Stuttgart, Universitätsstraße 38, Stuttgart, Germany
{salm,barzen,breitenbuecher,leymann,weder,wild}@iaas.uni-stuttgart.de

Abstract. Quantum computing can enable a variety of breakthroughs in research and industry in the future. Although some quantum algorithms already exist that show a theoretical speedup compared to the best known classical algorithms, the implementation and execution of these algorithms come with several challenges. The input data determines, for example, the required number of qubits and gates of a quantum algorithm. A quantum algorithm implementation also depends on the used Software Development Kit which restricts the set of usable quantum computers. Because of the limited capabilities of current quantum computers, choosing an appropriate one to execute a certain implementation for a given input is a difficult challenge that requires immense mathematical knowledge about the implemented quantum algorithm as well as technical knowledge about the used Software Development Kits. In this paper, we present a concept for the automated analysis and selection of implementations of quantum algorithms and appropriate quantum computers that can execute a selected implementation with a certain input data. The practical feasibility of the concept is demonstrated by the prototypical implementation of a tool that we call NISQ Analyzer.

Keywords: Quantum Computing · Quantum Algorithms · Hardware Selection · Implementation Selection · Decision Support · NISQ Analyzer.

1 Introduction

Quantum computing is a promising field that may enable breakthroughs in various areas such as computer science, physics, and chemistry [25]. The unique characteristics of quantum mechanics, such as *superposition* and *entanglement*, are the reasons quantum computing is more powerful than classical computing for specific problems [2,29,32]. In fact, some quantum algorithms already exist that show a theoretical speedup over their best known classical counterparts. For example, the *Shor* algorithm provides an exponential speedup in factorizing numbers [34]. With a large enough quantum computer, this algorithm could break cryptosystems such as the commonly used RSA [29].

However, there are several challenges regarding the execution of quantum algorithms. There is a multitude of different implementations for quantum algorithms that are only applicable to certain input data, e.g., in terms of the number of qubits required for its encoding. These implementations differ from each other in various aspects, e.g., the required number of qubits and operations [11]. Both numbers often depend on the input data. Thus, the input data influences whether a particular quantum algorithm implementation is executable on a certain quantum computer: If the number of required qubits or operations is higher than the number of qubits or the decoherence time, i.e. the time the states of qubits are stable, of the quantum computer, the implementation with the given input cannot be executed successfully. Error rates, fidelity, and qubit connectivity of current so-called *Noisy Intermediate-Scale Quantum (NISQ)* computers also play an important role in the decision [29,17].

Moreover, there is no accepted common quantum programming language [19]. As a result, most quantum computer vendors have their proprietary Software Development Kit (SDK) for developing and executing implementations on their quantum computers [16]. However, this tightly couples the implementation of a quantum algorithm to a certain quantum computer. As a result, choosing an implementation for a quantum algorithm to be executed for given input data and selecting an appropriate quantum computer is a multi-dimensional challenge. It requires immense mathematical knowledge about the implemented algorithm as well as technical knowledge about the used SDKs. Hence, *(i) the selection of a suitable implementation of a quantum algorithm for a specific input* and *(ii) the selection of a quantum computer with, e.g., sufficient qubits and decoherence time* is currently one of the main problems of quantum computing in practice.

In this paper, we present the concept of the *NISQ Analyzer* for analyzing and selecting *(i) an appropriate implementation* and *(ii) suitable quantum computers based on the input data for a chosen quantum algorithm*. The approach is based on defined selection criteria for each implementation described as first-order logic rules. Rule-based selection mechanisms have been established as proven principles and concepts [21]. We consider the number of required qubits of the implementation and the number of qubits of eligible quantum computers, while vendor-specific SDKs are also heeded. In addition, the number of operations of an implementation is determined and the corresponding decoherence times of the different quantum computers are considered. To determine the number of qubits and operations of an implementation, hardware-specific transpilers, e.g., provided by the vendors, are used. For demonstrating the practical feasibility of the proposed NISQ Analyzer, a prototypical implementation is presented. It is designed as a plug-in based system, such that additional criteria, e.g., error rates, fidelity, or qubit connectivity, can be added.

The paper is structured as follows: Section 2 introduces the fundamentals, current challenges, and the problem statement. Section 3 shows an overview of our approach. Section 4 presents the overall system architecture. Section 5 presents the prototypical implementation and our validation. Section 6 discusses related work. Section 7 concludes the paper and presents future work.

2 Background, Challenges and Problem Statement

In this section, we introduce the fundamentals and current challenges when using quantum computers during the Noisy Intermediate-Scale Quantum (NISQ) era [29]. Afterward, quantum algorithms and the current state of their implementations are presented. Finally, we formulate the problem statement and the resulting research question of this paper.

2.1 Quantum Computers and NISQ Era

Instead of working with classical bits, quantum computers, or more precisely *Quantum Processing Units (QPUs)*, use so-called *qubits* [26]. As classical bits can only be in one of the two states 0 or 1, qubits can be in both states at the same time [26,32]: A unit vector in a two-dimensional complex vector space represents the state of a qubit [26]. Operators applied to these vectors are *unitary* matrices. Qubits interact with their environment, and thus, their states are only stable for a certain time, called *decoherence time* [7,26,32]. The required operations have to be applied in this time frame to get proper results from computations. Furthermore, different quantum computing models exist, e.g., *one-way* [30], *adiabatic* [1], and *gate-based* [26]. In this paper, we only consider the gate-base quantum computing model, as many of the existing quantum computers, e.g., from IBM[1] and Rigetti[2], base on this model [16]. Thereby, *gates* represent the unitary operations. Combined with qubits and measurements they form a *quantum circuit* [26]. Such quantum circuits are gate-based representations of *quantum algorithms*. The number of gate collections to be sequentially executed defines the *depth* of a quantum circuit. Within such a collection, called *layer*, gates are performed in parallel. The number of qubits defines the *width* of the circuit. Both properties determine the required number of qubits and the minimum decoherence time a suitable quantum computer has to provide.

Each quantum computer has a set of physically implemented gates [32]. However, the sets of implemented gates differ from quantum computer to quantum computer. Thus, to create quantum circuits for specific problems, non-implemented gates must be realized by a subroutine of available gates of the specific quantum computer [19]. The substitution of non-implemented gates by subroutines is done by the hardware-specific transpiler of the vendor. Therefore, the transpiler maps the gates and qubits of the circuit to the gate sets and qubits of the regarded quantum computers. The resulting transpiled circuit may have a different depth and width from the circuit. Especially the resulting depth of the transpilied circuit can differ greatly between different quantum computers [17]. The mapping process is known to be NP-hard [8,36]. Thus, transpiling the same circuit several times can lead to slightly different values of width and depth and depends on the mapping algorithm of the transpiler.

Today's quantum computers only have a few qubits and short decoherence times [41]. Further, high error rates limit the number of operations that can

[1] https://www.ibm.com
[2] https://www.rigetti.com

be executed before the propagated error makes the computation too erroneous on these quantum computers. However, it is assumed that quantum computers will have up to a few hundred qubits and can perform thousands of operations reliably soon[3,4] [29]. But these qubits will be error-prone as the correction of such errors requires many more qubits [25,29].

Challenge I: *There are a variety of quantum computers that are different regarding their number of qubits, their decoherence time, and their set of physically implemented gates. Therefore, there is serious heterogeneity of available quantum computers, and not every quantum algorithm implementation can be executed on every quantum computer.*

2.2 Quantum Algorithms and Implementations

Many quantum algorithms show a theoretical speedup over their best known classical counterparts. The number of required qubits and operations for the execution of quantum algorithms often depends on the input data. For example, the Shor algorithm requires $2n$ qubits for factorizing the integer N with a binary size of n [11]. Some implementations require additional qubits for executing the algorithm. For example, *Quantum Phase Estimation (QPE)* [27] for computing the eigenvalues of a unitary matrix, needs in many of the existing implementations additional qubits to define the precision of the result. There are also implementations that can only process limited input data. Thus, selecting an appropriate quantum computer to execute a certain quantum algorithm not only depends on the mathematics of the algorithm itself, but also on the physical requirements of its implementations.

In addition, current implementations of quantum algorithms are tightly coupled to their used SDKs. Vendors like IBM and Rigetti offer their proprietary SDKs for their quantum computers, called *Qiskit*[5] and *Forest*[6], respectively. There are also SDKs that support quantum computers of multiple vendors, e.g., *ProjectQ* [38] or *XACC* [22]. Nonetheless, most of the SDKs only support quantum computers of a single vendor for executing quantum circuits [16]. Furthermore, implementations are not interchangeable between different SDKs because of their different programming languages and syntax. As a result, most of the developed implementations are only executable on a certain set of quantum computers provided by a specific vendor.

Challenge II: *An implementation of a quantum algorithm implies physical requirements on a quantum computer. In addition, an implementation usually depends on the used SDK.*

[3] https://www.ibm.com/blogs/research/2020/09/ibm-quantum-roadmap/
[4] https://ionq.com/news/october-01-2020-most-powerful-quantum-computer
[5] https://qiskit.org
[6] http://docs.rigetti.com/en/stable/

2.3 Problem Statement

In this section, we summarize the challenges presented before and formulate the research question regarding the selection of quantum computers capable of executing a quantum algorithm for certain input data. For executing a certain quantum algorithm for given input data, the user has to consider different aspects regarding available quantum algorithm implementations and quantum computers. First, the user has to find manually a suitable implementation of the quantum algorithm that can process the desired input. With the chosen quantum algorithm implementation, the user has to select a suitable quantum computer that can execute the implementation. Thereby, the heterogeneity of quantum computers, with their different qubit counts, decoherence times, and available gate sets, has to be taken into account (***Challenge I***). Additionally, the mathematical, physical, and technical requirements on the quantum computer and the used SDK of the implementation have to be considered for the quantum computer selection (***Challenge II***). Thus, the selection of quantum algorithm implementations and suitable quantum computers requires an immense manual effort and sufficient knowledge on the user side. Hence, the resulting research question can be formulated as follows:

Problem Statement: *How can the selection of the quantum algorithm implementation and the suitable quantum computer be automated based on a certain input data of the chosen quantum algorithm?*

3 Analysis and Selection Approach

In this section, we introduce our concept of a *NISQ Analyzer*, which enables an automated analysis and selection of quantum algorithm implementations and quantum computers depending on the chosen quantum algorithm and input data. Fig. 1 depicts an overview of the approach. First, a quantum algorithm is selected. Second, implementations of the algorithm are analyzed and selected. Third, quantum computers are analyzed and selected. Finally, the selected implementation is executed on the suitable quantum computer. In the following, the individual phases are described in detail.

3.1 Algorithm Selection

In the *(1) Algorithm Selection* phase, the user selects one of the provided quantum algorithms for solving a particular problem, e.g., the Shor algorithm for factorization as shown in Fig. 1. Thus, a repository with a set of descriptors of different quantum algorithms is provided, as proposed by [18]. The selected quantum algorithm and the input data serves as input for the analysis and selection phase. In the example, the user wants to factorize 9, thus, $N = 9$.

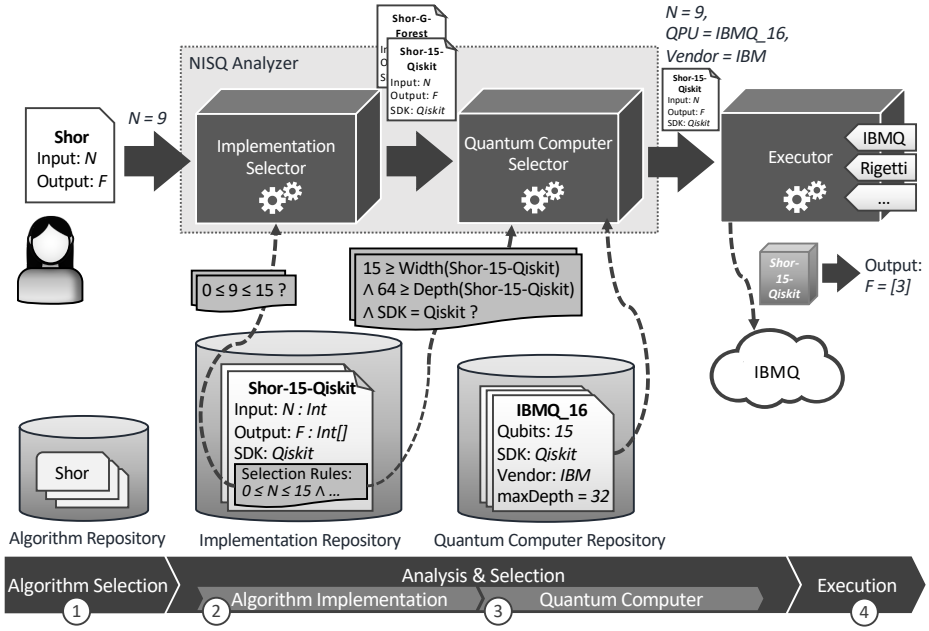


Fig. 1. Approach for automated selection of implementations and quantum computers.

3.2 Algorithm Implementation Analysis and Selection

In the *(2) Algorithm Implementation Analysis and Selection* phase, available implementations of the selected quantum algorithm are browsed to identify applicable implementations that can process the input. Therefore, a repository containing descriptors of different implementations is provided. The descriptors include metadata, such as input and output parameters, and the required SDK. For the identification, selection rules described by first-order logic are attached to the implementations. A rule describes the restrictions for the input data of the respective implementation. The selection rules of the considered implementations are evaluated based on the input data. As exemplary shown in Fig. 1, the selection rule for the implementation *Shor-15-Qiskit* is defined as follows:

$$y \in \mathcal{I} \forall n \in \mathbb{N} \exists l_0, l_1 \in \mathbb{N} : (\mathit{InputRange}(l_0, l_1, y) \wedge \mathit{GreaterEquals}(n, l_0) \\ \wedge \mathit{SmallerEquals}(n, l_1)) \Leftrightarrow \mathit{Processable}(n, y) \tag{1}$$

Thereby, $y = \mathit{Shor\text{-}15\text{-}Qiskit}$ is in the set of implementations $\mathcal{I}$ and n is the input data, e.g. $n = N = 9$. $\mathit{InputRange}(l_0, l_1, y)$ describes the range of processable input data of y, e.g., $l_0 = 0$ and $l_1 = 15$ for lower and upper bound. $\mathit{GreaterEquals}(n, l_0)$ defines "$n \geq l_0$" and $\mathit{SmallerEquals}(n, l_1)$ defines "$n \leq l_1$", such that n has to be between 0 and 15. This is all true if and only if $\mathit{Processable}(n, y)$ is true and, thus, *Shor-15-Qiskit* can process $n = N = 9$. The selection rule is implementation-specific and has to be defined by the developer. All implementations that can process n are considered in the next phase.

3.3 Quantum Computer Analysis and Selection

In the *(3) Quantum Computer Analysis and Selection* phase, appropriate quantum computers are identified for the considered implementations. Therefore, the width and depth of the implementations are analyzed and compared with the number of qubits and the estimated maximum depths provided by the available quantum computers. Thus, a corresponding repository with the properties of given quantum computers is provided, as presented in Fig. 1. The estimated maximum depth of a specific quantum computer is determined by dividing the average decoherence time of the supported qubits through the maximum gate time [33]. The width and depth of the implementation with the input data on the considered quantum computer are determined by using the hardware-specific transpiler. The supported transpilers are wrapped as a service. Additionally, the SDKs used by the implementations and the SDKs supporting the quantum computers are considered. The general rule for selecting a suitable quantum computer for a particular implementation is defined as follows:

$$\begin{aligned} \forall x \in \mathcal{Q} \forall y \in \mathcal{R} \subseteq \mathcal{I} \exists s \in \mathcal{S} \exists q_0, q_1, d_0, d_1 \in \mathbb{N} : \\ (\mathit{Qubits}(q_0, x) \wedge \mathit{Qubits}(q_1, y) \wedge \mathit{GreaterEquals}(q_0, q_1) \\ \wedge \mathit{Depth}(d_0, x) \wedge \mathit{Depth}(d_1, y) \wedge \mathit{GreaterEquals}(d_0, d_1) \\ \wedge \mathit{Sdk}(s, x) \wedge \mathit{Sdk}(s, y)) \Leftrightarrow \mathit{Executable}(y, x) \end{aligned} \tag{2}$$

Thereby, x is a quantum computer of the set of available quantum computers $\mathcal{Q}$, e.g., *IBMQ_16*. y is an implementation of the set of remaining implementations $\mathcal{R} \subseteq \mathcal{I}$. $\mathit{Qubits}(q_0, x)$ defines the provided number of qubits q_0 of x. $\mathit{Qubits}(q_1, y)$ defines the required number of qubits, or the width, q_1 of y. $\mathit{GreaterEquals}(q_0, q_1)$ defines that "$q_0 \geq q_1$" to ensure that the quantum computer x does not have less qubits than required by y. $\mathit{Depth}(d_0, x)$ defines the maximum depth d_0 executable by x. $\mathit{Depth}(d_1, y)$ defines the depth d_1 of the transpiled circuit of y. $\mathit{GreaterEquals}(d_0, d_1)$ defines that "$d_0 \geq d_1$", such that the maximum executable depth of the quantum computer x is not smaller than required by the implementation y. Furthermore, the SDK $s \in \mathcal{S}$, e.g. *Qiskit*, used by the implementation, defined by $\mathit{Sdk}(y, s)$, must also support the selected quantum computer, defined by $\mathit{Sdk}(x, s)$, to ensure their compatibility. This all is true, if and only if $\mathit{Executable}(y, x)$ is true. In the example in Fig. 1, *IBMQ_16* can execute *Shor-15-Qiskit*. If more than one executable implementation remains, the user decides which one to execute. Furthermore, the user also decides, in case, more than one quantum computer can execute the chosen implementation.

3.4 Execution

In the *(4) Execution* phase, the selected implementation is executed by the selected quantum computer, as seen in Fig. 1. The *Executor* supports the different SDKs and can be extended by further plug-ins. Thereby, the required SDK, e.g. *Qiskit*, is used to deliver the quantum circuit to the specific vendor via the cloud. Eventually, the result is returned and displayed to the user.

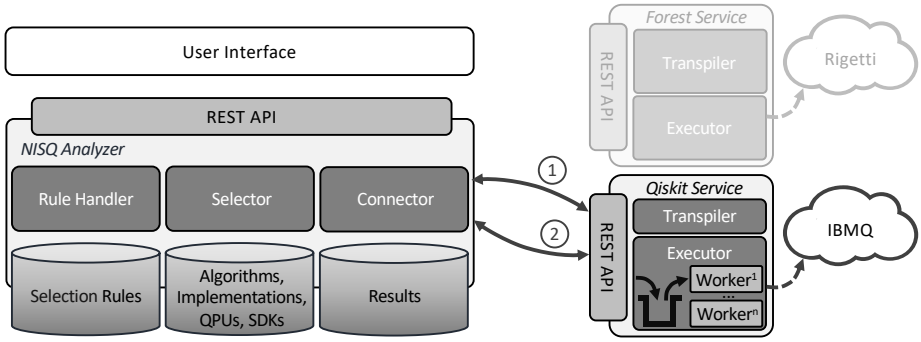


Fig. 2. System architecture for automated analysis and selection.

4 System Architecture

In this section, we introduce the overall system architecture, as shown in Fig. 2. It comprises a *User Interface (UI)*, the *NISQ Analyzer*, and *Services* wrapping the transpilation and execution logic of, e.g., vendor-specific SDKs. The NISQ Analyzer provides a HTTP REST API used by the UI. The *Rule Handler* component, which is part of the NISQ Analyzer, generates, adds, updates, and accesses the selection rules defined in Sec. 3. The *Selector* component identifies suitable implementations and quantum computers dependent on the selected quantum algorithm and input data by invoking the Rule Handler and the Services. The descriptors containing the metadata about algorithms, implementations, quantum computers, and SDKs are stored in a respective repository. A Service invokes the hardware-specific *Transpiler* and *Executor* of a specific SDK. For example, the *Qiskit Service* transpiles and executes Qiskit implementations on quantum computers provided via the cloud on IBMQ[7]. Each Service provides its own HTTP REST API. A *Results* repository stores analysis and execution results.

To start the analysis and selection, the user selects the desired quantum algorithm and provides the input data using the UI. The Selector invokes the Rule Handler to evaluate which implementations of the quantum algorithm can process the input. Therefore, the selection rules defined by the developers are stored in a repository. For determining the width and depth of an implementation on a certain quantum computer, the Selector calls the Transpiler of the required Service using the *Connector* (see (1) in Fig. 2). The resulting values are returned to the Selector and passed to the Rule Handler for evaluating the quantum computer selection rule defined in Sec. 3. The invocation of the transpiler and the evaluation of the general selection rule is performed for each supported quantum computer. With the analysis results presented by the UI, the user selects an implementation and a recommended quantum computer for execution. Next, the NISQ Analyzer invokes the Executor of the specific Service to deliver the implementation to the specific vendor, e.g. IBMQ (see (2) in Fig. 2). Finally, the

[7] https://quantum-computing.ibm.com

Service returns the result to the user. Since multiple transpiler and execution frameworks exist, further Services providing a HTTP REST API and the defined interface can be implemented and invoked by the NISQ Analyzer. On the side of the NISQ Analyzer, the Uniform Resource Locators (URLs) of further Services have to be passed as configuration parameters.

5 Prototype and Validation

In this section, we present our prototypical implementation of the NISQ Analyzer, the UI, and, as a proof of concept, the Qiskit Service. For the validation of our approach, three use cases are presented. Afterward, we discuss the limitations of our prototype.

5.1 Prototype

The prototypical UI of the NISQ Analyzer, as shown in Fig. 2, is implemented in TypeScript. The NISQ Analyzer[8] is implemented in Java using the Spring Boot framework. The descriptors of available quantum computers, SDKs, quantum algorithms, and implementations, as well as the analysis and execution results, are stored in a relational database.

Rule Handling & Selection with Prolog For the implementation of the first-order logic rules presented in Sec. 3, we use the logic programming language Prolog. For handling Prolog programs, the NISQ Analyzer uses the library and interpreter of SWI-Prolog[9]. The program logic in Prolog is expressed by facts and rules. For computations, facts and rules are queried and evaluate to true or false. A fact is, e.g., "`providesQubits(ibmq_16, 15).`". It defines that the number of qubits provided by *ibmq_16* is 15. Querying facts always evaluates to true. For the implementation selection, the required rules are defined by the developers and queried with the input data of the user. For the selection of suitable quantum computers, the general rule is defined as follows:

```
executable(CircuitWidth, CircuitDepth, Implementation, Qpu) :-
    providesQubits(Qpu, ProvidedQubits),
    ProvidedQubits >= CircuitWidth,
    t1Time(Qpu, T1Time),
    maxGateTime(Qpu, GateTime),
    CircuitDepth =< T1Time/GateTime,
    requiredSdk(Implementation, Sdk),
    usedSdk(Qpu, Sdk).
```

[8] https://github.com/UST-QuAntiL/nisq-analyzer
[9] https://www.swi-prolog.org

The general rule is applied on the facts that are automatically generated based on the metadata of implementations and quantum computers. The fact `providesQubits` defines the number of qubits supported by the considered quantum computer. The number of qubits is compared to the width of the considered implementation by `ProvidedQubits >= CircuitWidth`. The facts `t1Time` and `maxGateTime` define the decoherence time and the maximum gate time of the quantum computer to compare the estimated maximum depth with the depth of the implementation: `CircuitDepth =< T1Time/GateTime`. `requiredSdk` defines the SDK of the implementation. This has to match with `usedSdk` which specifies the SDK supporting the specific quantum computer. The general rule evaluates to true, if the properties of the implementation match the properties of the considered quantum computer.

Transpilation & Execution Service The Qiskit Service[10] is implemented in the programming language Python. For transpiling and executing an implementation on a specific quantum computer, the Python module of the Qiskit SDK[11] is used. Qiskit supports the quantum computers of IBMQ. For transpilation, the NISQ Analyzer examines if the implementation and the quantum computer is supported by Qiskit. This reduces the number of invocations of the Qiskit Service and improves the performance of the overall system. Then, the NISQ Analyzer sends a HTTP request to the Qiskit Service. The request contains the source code location of the implementation, the name of the quantum computer, and the input data of the user. The Qiskit Service retrieves the source code, passes the input data, and transpiles the resulting quantum circuit of the implementation. Thereby, the transpilation process is done locally. The corresponding HTTP response of the Qiskit Service contains the width and depth of the transpiled quantum circuit. The transpiler supports several optimization levels[12]. For our prototype, the light optimization level is used. Higher optimization levels require more classical compute resources and, therefore, more computing time. However, they may further reduce the depth and width of a quantum circuit.

For executing an implementation on a quantum computer, e.g. at IBMQ, jobs are put in queues. Thus, the response of IBMQ containing the execution result is delivered asynchronously. Therefore, execution requests of the NISQ Analyzer are received by the Qiskit Service via HTTP and then placed in a queue using Redis Queue[13]. With several *Workers* listening on the queue, multiple implementations can be executed in parallel, see Fig. 2. The corresponding HTTP response contains the content location of the long-running task, where the execution result is later provided by a relational database of the Qiskit Service.

[10] https://github.com/UST-QuAntiL/qiskit-service
[11] https://github.com/Qiskit
[12] https://qiskit.org/documentation/stubs/qiskit.compiler.transpile.html
[13] https://python-rq.org

5.2 Case Studies for Validation

For validating our approach, three quantum algorithms and exemplary implementations are considered as use cases. The implementation selection rules evaluated by the NISQ Analyzer, the results of the transpiler, and the resulting recommendations of the NISQ Analyzer are presented in the following. The first considered quantum algorithm is *Simon's* algorithm for the distinction of two function classes [35]. The second algorithm is the *Grover* algorithm for searching an item in an unsorted list [9,31]. The third is the *Shor* algorithm for factorizing an integer with exponential speedup [34]. For each algorithm, implementations using the Qiskit SDK are provided in our GitHub repository[14]. Thereby, the considered general implementations use functions provided by Qiskit[15] for automatically generating quantum circuits dependent on the input data. For validation, the quantum computers *ibmq_16_melbourne* [12], supporting 15 qubits and a calculated maximum depth of 32 levels, and the *ibmq_5_yorktown* [13], supporting 5 qubits and a calculated maximum depth of 96 levels, of IBMQ are considered. In addition, the IBMQ quantum computer simulator *ibmq_qasm_simulator*, simulating 32 qubits, is considered. The simulator is not restricted in its maximum depth. The properties of the quantum computers are accessible by the Qiskit SDK[16]. We group the simulator and the quantum computers as backends.

Simon's Algorithm A function $f : \{0,1\}^n \rightarrow \{0,1\}^n$ exists with $f(x_1) = f(x_2)$ if $x_1 = x_2 \oplus s$, whereby $s \in \{0,1\}^n$ [24,35]. If $s = 0^n$, the function f is a one-to-one mapping from source to target set, otherwise it is a one-to-two mapping [24]. The secret string s has to be found with a minimum number of function calls by a quantum computer. Such black box functions are often called *oracles* [32]. A sample general implementation for Simon's Algorithm is *simon-general-qiskit*. The input data is s which defines the oracle for the resulting quantum circuit. As required by the specific oracle-generating function[17] used, the length of s has to be a power of two. Thus, the implementation selection rule is defined as follows:

```
processable(S, simon-general-qiskit) :-
    S =~ '^[01]+$',
    atom_length(S, X), X is X /\ (-X).
```

`simon-general-qiskit` is the name of the respective implementation. First, the rule body specifies a regular expression for the non-empty secret string `S` such that it only contains the characters 0 or 1. The second line counts the length of `S` and evaluates if it is a power of two. Therefore, the bit-wise AND operation with the two's complement of the length of `S` is used. Valid input data for `S` are shown in Table 1. To determine if `processable` evaluates to true for a certain input, e.g. '0110', the NISQ Analyzer evaluates such Prolog rules as follows:

[14] https://github.com/UST-QuAntiL/nisq-analyzer-content

[15] https://qiskit.org/documentation/apidoc/qiskit.aqua.algorithms.html

[16] https://quantum-computing.ibm.com/docs/manage/account/ibmq

[17] https://qiskit.org/documentation/stubs/qiskit.aqua.components.oracles.TruthTableOracle.html

```
?- processable('0110', simon-general-qiskit).
```

Depending on the backend, Table 1 shows the width and depth of the transpiled `simon-general-qiskit` implementation. The resulting values show a correlation to the length of s. Quantum computers marked with * in Table 1 do not support the required number of qubits or maximum depth for execution. Thus, they are excluded by the general selection rule defined in 5.1 and are not recommended by the NISQ Analyzer. For the input '1111111100000000', the required number of qubits is higher than the provided number of *ibmq_5_yorktown*. The transpiler denies the transpilation and provides no result.

Input	Width	Depth	Backend
01	2	4	ibmq_qasm_simulator
	2	4	ibmq_5_yorktown
	2	4	ibmq_16_melbourne
0110	3	8	ibmq_qasm_simulator
	3	26	ibmq_5_yorktown
	3	39	*ibmq_16_melbourne**
1000	3	6	ibmq_qasm_simulator
	3	15	ibmq_5_yorktown
	3	23	ibmq_16_melbourne
1111	3	12	ibmq_qasm_simulator
	3	48	ibmq_5_yorktown
	3	74	*ibmq_16_melbourne**
10110010	5	52	ibmq_qasm_simulator
	5	81	ibmq_5_yorktown
	5	132	*ibmq_16_melbourne**
11111111 00000000	7	196	ibmq_qasm_simulator
	–	–	*ibmq_5_yorktown**
	7	471	*ibmq_16_melbourne**

Table 1. Analysis results of the Qiskit transpiler for *simon-general-qiskit*.

Grover Algorithm The Grover algorithm searches an item in an unsorted list of M items with a quadratic speedup compared to classical search algorithms [9,14]. The input for the algorithm is a Boolean function that defines the searched item [32]. For the computation, briefly, superposition of all M items and *amplitude amplification* is used to measure the searched item on a quantum computer. The Grover algorithm can also be used to solve the *Boolean satisfiability problem (SAT)*. The input for the sample implementation *grover-general-sat-qiskit* are Boolean formulas, as shown in Table 2. Therefore the selection rule used by the NISQ Analyzer can be defined as follows:

```
processable(Formula, grover-general-sat-qiskit) :-
    Formula =~ '^[0-9A-Za-z|&()~^ ]+$'.
```

The regular expression matches all alphanumerical characters, round brackets, and logical operators in the required format. "`|`" defines a logical OR. "`&`" specifies a logical AND. "`~`" is a NOT and "`^`" a XOR. Additionally, `Formula` has to be non-empty. Width and depth of the transpiled *grover-general-sat-qiskit* implementation for valid formulas are presented in Table 2. It shows that more complex Boolean formulas require more resources. For each input, *ibmq_16_melbourne* is excluded (marked with *). It has enough qubits, but the required depth is already too high and, therefore, the NISQ Analyzer does not recommend the quantum computer.

Input	Width	Depth	Backend
$(A \vee B) \wedge (A \vee \neg B)$	7	29	ibmq_qasm_simulator
$\wedge(\neg A \vee B)$	7	150	*ibmq_16_melbourne**
$(A \vee B \vee \neg C) \wedge (\neg A \vee B \vee C)$	8	103	ibmq_qasm_simulator
$\wedge(\neg A \vee \neg B \vee \neg C)$	15	296	*ibmq_16_melbourne**
$(\neg A \vee \neg B \vee \neg C)$	12	173	ibmq_qasm_simulator
$\wedge(\neg A \vee B \vee C) \wedge (A \vee \neg B \vee C)$			
$\wedge(A \vee B \vee \neg C) \wedge (A \vee \neg B \vee \neg C)$	15	526	*ibmq_16_melbourne**

Table 2. Analysis results of the Qiskit transpiler for *grover-general-sat-qiskit*.

Shor Algorithm The input for the Shor algorithm is an odd integer $N = pq$, where p and q are the searched prime factors [14]. Part of the algorithm is the computation of the period of a function by a quantum computer [31]. Therefore, superposition and the *Quantum Fourier Transform (QFT)*, a quantum variation of the classical Discrete Fourier Transform (DFT), is used to measure the period. By further post-processing with the Euclidean algorithm on classical computers, the searched prime factors can be obtained [32]. The first considered implementation of the Shor algorithm is *shor-general-qiskit* [3]. It can process all odd integers greater than 2. Therefore, the rule for the implementation selection of the NISQ Analyzer is defined as follows:

```
processable(N, shor-general-qiskit) :- N > 2, 1 is mod(N, 2).
```

Transpiling the *shor-general-qiskit* implementation with several demonstrative valid input data results in extremely high depths, as presented in Table 3. As none of the considered quantum computers can compute such quantum circuits, only the *ibmq_qasm_simulator* is recommended by the NISQ Analyzer.

Input	Width	Depth	Backend
3	10	2401	ibmq_qasm_simulator
9	18	15829	ibmq_qasm_simulator
15	18	14314	ibmq_qasm_simulator
21	22	27515	ibmq_qasm_simulator
33	26	48503	ibmq_qasm_simulator
35	26	49139	ibmq_qasm_simulator
39	26	48379	ibmq_qasm_simulator

Table 3. Analysis results of the Qiskit transpiler for *shor-general-qiskit*.

In comparison, an exemplary fix implementation of the Shor algorithm is *shor-fix-15-qiskit*[18]. It only factorizes $N = 15$, as it implements a concrete quantum circuit. Thus, the implementation itself does not assume any input data. The attached rule for the implementation selection is defined as follows:

```
processable(N, shor-fix-15-qiskit) :- N is 15.
```

Transpiling *shor-fix-15-qiskit* for all considered backends results in small values of width and depth, as shown in Table 4. In contrast to the general implementation, each of the backends can execute *shor-fix-15-qiskit* according to the NISQ Analyzer with the limitation that only the input $N = 15$ can be processed.

Input	Width	Depth	Backend
15	3	5	ibmq_5_yorktown
15	4	11	ibmq_16_melbourne
15	5	5	ibmq_qasm_simulator

Table 4. Analysis results of the Qiskit transpiler for *shor-fix-15-qiskit*.

5.3 Discussion and Limitations

Currently, the prototype of our approach only supports implementations for Qiskit and quantum computers of IBMQ. Thus, only implementations using Qiskit can be transpiled and considered for the quantum computer selection. However, our plug-in based system supports extensibility for further SDKs. For transpiling and executing given implementations by the Qiskit Service, the resulting quantum circuits are returned from the source code. Furthermore, the response time of the NISQ Analyzer depends on the response time of the transpilation process. Especially general implementations, such as *shor-general-qiskit*,

[18] https://quantum-circuit.com/app_details/HYLMtcuK6b7uaphC7

are computationally intensive as the circuit is constructed after passing the required input data. As each implementation that can process a given input is transpiled for each backend, this could result in performance issues - especially with an increasing set of different quantum computers. Therefore, one approach is to define lower bounds for width and depth. For example, the lower bounds for the *shor-fix-15-qiskit* implementation could be determined by counting the number of qubits, which is 5, and the number of gate layers, which is 7, of the original circuit. But, as shown in Table 4, both values can be smaller depending on the target backend. This results from the hardware-specific optimization functionalities of the transpiler. Thus, pre-filtering the set of quantum computers by defined lower bounds of width and depth could result in excluding suitable quantum computers. Dividing the average decoherence time of all qubits by the maximum gate time to determine the maximum depth of a quantum computer can only be considered as a rough estimate [33]. Currently, no further functional or non-functional requirements, such as costs, execution time, and quality of qubits, are considered. Thus, the user has to select the desired solution if more than one suitable implementation or backend is recommended. If no implementation or backend suits the input data, no recommendation is given.

As the definition of Prolog rules for the implementation selection have to be provided by the developer, knowledge in logic programming and the implementation itself is required. Nevertheless, the defined rules enable the automated selection of suitable implementations for other users. Furthermore, Prolog does only support Horn clauses, which are formulas in conjunctive normal form (CNF). Horn clauses contain at most one positive literal. For example, a formula of type $(A \wedge B \Leftrightarrow C)$, as concretely defined in Sec. 3, is equivalent to the Horn clause $(\neg A \vee \neg B \vee C) \wedge (A \vee \neg C) \wedge (B \vee \neg C)$. Negating A or B leads to a formula with more than one positive literal which is not a Horn clause. To prevent this limitation, the closed-world assumption and the negation as failure inference rule are assumed. Since the system generates or provides all required data, we can expect that the closed-world assumption holds true.

6 Related Work

For the comparison of different quantum computers, several metrics were developed, such as *quantum volume* [5] or *total quantum factor (TQF)* [33]. Additionally, several benchmarks for the quantification of the capabilities of quantum computers were proposed [2,4]. However, these metrics only give a rough comparison of the capabilities of the regarded quantum computers. They do not consider the aspects of specific quantum algorithms. Hence, selecting the quantum computer with the highest score independent of the quantum algorithm and the input data does not always lead to a suitable decision.

Suchara et al. [40] introduce the *QuRE Toolbox*, a framework to estimate required resources, such as qubits or gates, to execute a quantum algorithm on quantum computers of different physical technologies. Thereby, the quantum algorithm description is used as input for resource estimation. Additionally, they

consider error-correction. Thus, they approximate the number of additional gates and qubits required to compare the efficiency of different error-correction codes in diverse setups. However, their focus is on building a suitable quantum computer, not on selecting an existing one. Therefore, they do not consider the current set of different quantum computers, their supporting SDKs, and their limitations.

Sivarajah et al. [37] present the quantum software development platform $t|ket\rangle$. It supports implementations of several quantum programming languages and their execution on quantum computers of different vendors. For cross-compilation an intermediate representation is used. Their internal compiler optimizes given implementations in several phases and maps them to the architecture of the desired quantum computer. Before execution, it is validated if an implementation is executable on the selected quantum computer. Thereby, the used gate set, the required number of qubits and the required connection between the qubits is compared with the properties of the quantum computer. However, they do not estimate the maximum depth of the supported quantum computers and, therefore, do not compare it to the depth of an implementation. Additionally, they do not support the recommendation of suitable quantum computers as their scope is the cross-compilation and optimization of implementations.

Furthermore, JavadiAbhari et al. [15] estimate and analyze required resources of implementations in their compilation framework *ScaffCC*. Thereby, they track width and depth, the number of gates, and the interaction between qubits. However, their work focuses on hardware agnostic compilation and optimization and, therefore, does not recommend suitable quantum computers.

Also in other domains approaches for decision support exist. In cloud computing, different approaches for automating the service and provider selection are presented. Zhang et al. [43] propose an approach to map user requirements automatically to different cloud services and their suited configuration using a declarative language. Han et al. [10] introduce a cloud service recommender system based on quality of service requirements. Strauch et al. [39] provide a decision support system for the migration of applications to the cloud. For *Service Oriented Architecture (SOA)*, decision models are introduced to support the design of application architectures [44,45]. Manikrao et al. [20] propose a service selection framework for web services. It semantically matches functional and non-functional requirements of the service providers and the user and recommends based on previous user feedback. Brahimi et al. [6] present a proposal for recommending *Database Management Systems (DBMSs)* based on the requirements of the user. However, none of these systems include quantum technologies and their special characteristics, such as the limited resources of quantum computers or the varying requirements of quantum algorithms dependent on the input data.

7 Conclusion and Future Work

In this paper, we presented the concept of a NISQ Analyzer. It analyzes and selects (i) an appropriate algorithm implementation and (ii) a suitable quantum computer for executing a given quantum algorithm and input data by means

of defined selection rules. Thereby, the width and depth of the implementations are dynamically determined using hardware-specific transpilers and are compared with the properties of available quantum computers. Implementations of quantum algorithms are tightly coupled to the used SDKs. Thus, the compatibility between the SDK used by the implementation and the SDK supporting the quantum computer has to be considered. The selected implementation is then sent to the corresponding quantum computer for execution.

The implemented NISQ Analyzer will be part of a platform for sharing and executing quantum software as proposed in [18,19]. It is currently realized by the project PlanQK[19]. As part of PlanQK, we plan to use established deployment automation technologies to automate the deployment of quantum algorithms [42]. In the future, we want to analyze the source code of implementations to consider further properties, such as error rates, fidelity, and qubit connectivity. We also plan to implement further services to support additional vendors of quantum computers and hardware-specific transpilers. In addition, we plan to determine and develop further metrics for a more precise analysis and selection of implementations and quantum computers. Thereby, we plan to integrate further functional and non-functional requirements. We also want to support variational quantum algorithms [23,28]. This approach uses quantum computers and classical computers alternately for optimization problems in a hybrid manner: This allows to limit the time spend on a quantum computer, i.e. it avoids problems resulting from decoherence and the lack of gate fidelity. Thus, it overcomes the current limitations of NISQ computers to a certain extent.

Acknowledgements

This work was partially funded by the BMWi project *PlanQK* (01MK20005N) and the DFG's Excellence Initiative project *SimTech* (EXC 2075 - 390740016).

References

1. Aharonov, D., Van Dam, W., Kempe, J., Landau, Z., Lloyd, S., Regev, O.: Adiabatic quantum computation is equivalent to standard quantum computation. SIAM review **50**(4), 755–787 (2008)
2. Arute, F., Arya, K., Babbush, R., Bacon, D., Bardin, J.C., Barends, R., et al.: Quantum supremacy using a programmable superconducting processor. Nature **574**(7779), 505–510 (2019)
3. Beauregard, S.: Circuit for Shor's algorithm using 2n+3 qubits. Quantum Information and Computation **3**(2), 175–185 (2003)
4. Benedetti, M., Garcia-Pintos, D., Perdomo, O., Leyton-Ortega, V., Nam, Y., Perdomo-Ortiz, A.: A generative modeling approach for benchmarking and training shallow quantum circuits. npj Quantum Information **5**(1), 45 (2019)
5. Bishop, L.S., Bravyi, S., Cross, A., Gambetta, J.M., Smolin, J.: Quantum volume. Technical Report (2017)

[19] https://planqk.de/en/

6. Brahimi, L., Bellatreche, L., Ouhammou, Y.: A recommender system for dbms selection based on a test data repository. In: Pokorný, J., Ivanović, M., Thalheim, B., Šaloun, P. (eds.) Advances in Databases and Information Systems. pp. 166–180. Springer International Publishing, Cham (2016)
7. Chuang, I.L., Yamamoto, Y.: Creation of a persistent quantum bit using error correction. Phys. Rev. A **55**, 114–127 (1997)
8. Cowtan, A., Dilkes, S., Duncan, R., Krajenbrink, A., Simmons, W., Sivarajah, S.: On the qubit routing problem (2019)
9. Grover, L.K.: A fast quantum mechanical algorithm for database search. In: Proceedings of the twenty-eighth annual ACM symposium on Theory of computing. pp. 212–219 (1996)
10. Han, S.M., Hassan, M.M., Yoon, C.W., Huh, E.N.: Efficient Service Recommendation System for Cloud Computing Market. In: Proceedings of the 2nd international conference on interaction sciences: information technology, culture and human. pp. 839–845 (2009)
11. Häner, T., Roetteler, M., Svore, K.M.: Factoring using 2n+2 qubits with toffoli based modular multiplication. Quantum Information and Computation **18**(7-8), 673–684 (2017)
12. IBMQ team: 15-qubit backend: IBM Q 16 Melbourne backend specification V2.3.1 (2020), `https://quantum-computing.ibm.com`
13. IBMQ team: 5-qubit backend: IBM Q 5 Yorktown backend specification V2.1.0 (2020), `https://quantum-computing.ibm.com`
14. J., A., Adedoyin, A., Ambrosiano, J., Anisimov, P., Bärtschi, A., Casper, W., Chennupati, G., Coffrin, C., Djidjev, H., Gunter, D., Karra, S., Lemons, N., Lin, S., Malyzhenkov, A., Mascarenas, D., Mniszewski, S., Nadiga, B., O'Malley, D., Oyen, D., Pakin, S., Prasad, L., Roberts, R., Romero, P., Santhi, N., Sinitsyn, N., Swart, P.J., Wendelberger, J.G., Yoon, B., Zamora, R., Zhu, W., Eidenbenz, S., Coles, P.J., Vuffray, M., Lokhov, A.Y.: Quantum algorithm implementations for beginners (2018)
15. JavadiAbhari, A., Patil, S., Kudrow, D., Heckey, J., Lvov, A., Chong, F.T., Martonosi, M.: Scaffcc: A framework for compilation and analysis of quantum computing programs. In: Proceedings of the 11th ACM Conference on Computing Frontiers. CF '14, Association for Computing Machinery, New York, NY, USA (2014)
16. LaRose, R.: Overview and Comparison of Gate Level Quantum Software Platforms. Quantum **3**, 130 (2019)
17. Leymann, F., Barzen, J.: The bitter truth about gate-based quantum algorithms in the NISQ era. Quantum Science and Technology pp. 1–28 (Sep 2020)
18. Leymann, F., Barzen, J., Falkenthal, M.: Towards a Platform for Sharing Quantum Software. In: Proceedings of the 13th Advanced Summer School on Service Oriented Computing. pp. 70–74. IBM Technical Report, IBM Research Division (2019)
19. Leymann, F., Barzen, J., Falkenthal, M., Vietz, D., Weder, B., Wild, K.: Quantum in the Cloud: Application Potentials and Research Opportunities. In: Proceedings of the 10th International Conference on Cloud Computing and Services Science. SciTePress (2020)
20. Manikrao, U.S., Prabhakar, T.V.: Dynamic selection of web services with recommendation system. In: International Conference on Next Generation Web Services Practices (NWeSP'05). pp. 5 pp.– (2005)
21. Masood, S., Soo, A.: A rule based expert system for rapid prototyping system selection. Robotics and Computer-Integrated Manufacturing **18**(3-4), 267–274 (2002)

22. McCaskey, A.J., Lyakh, D., Dumitrescu, E., Powers, S., Humble, T.S.: XACC: a system-level software infrastructure for heterogeneous quantum-classical computing. Quantum Science and Technology pp. 1–17 (2020)
23. Moll, N., Barkoutsos, P., Bishop, L.S., Chow, J.M., Cross, A., Egger, D.J., Filipp, S., Fuhrer, A., Gambetta, J.M., Ganzhorn, M., Kandala, A., Mezzacapo, A., Müller, P., Riess, W., Salis, G., Smolin, J., Tavernelli, I., Temme, K.: Quantum optimization using variational algorithms on near-term quantum devices. Quantum Science and Technology **3**(3), 030503 (Jun 2018)
24. Nannicini, G.: An introduction to quantum computing, without the physics (2017)
25. National Academies of Sciences, Engineering, and Medicine: Quantum Computing: Progress and Prospects. The National Academies Press, Washington, DC (2019)
26. Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information. Cambridge University Press, USA, 10th edn. (2011)
27. O'Brien, T.E., Tarasinski, B., Terhal, B.M.: Quantum phase estimation of multiple eigenvalues for small-scale (noisy) experiments. New Journal of Physics **21**(2), 1–43 (2019)
28. Peruzzo, A., McClean, J., Shadbolt, P., Yung, M.H., Zhou, X.Q., Love, P.J., Aspuru-Guzik, A., O'Brien, J.L.: A variational eigenvalue solver on a photonic quantum processor. Nature Communications **5**(1) (Jul 2014)
29. Preskill, J.: Quantum Computing in the NISQ era and beyond. Quantum **2**, 79 (2018)
30. Raussendorf, R., Briegel, H.J.: A one-way quantum computer. Phys. Rev. Lett. **86**, 5188–5191 (2001)
31. Rieffel, E., Polak, W.: An introduction to quantum computing for non-physicists. ACM Comput. Surv. **32**(3), 300–335 (Sep 2000)
32. Rieffel, E., Polak, W.: Quantum Computing: A Gentle Introduction. The MIT Press, 1st edn. (2011)
33. Sete, E.A., Zeng, W.J., Rigetti, C.T.: A Functional Architecture for Scalable Quantum Computing. In: IEEE International Conference on Rebooting Computing. pp. 1–6 (2016)
34. Shor, P.W.: Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. SIAM Journal on Computing **26**(5), 1484–1509 (1997)
35. Simon, D.R.: On the power of quantum computation. In: Proceedings of the 35th Annual Symposium on Foundations of Computer Science. p. 116–123. SFCS '94, IEEE Computer Society, USA (1994)
36. Siraichi, M.Y., Santos, V.F.d., Collange, S., Quintão Pereira, F.M.: Qubit Allocation. In: CGO 2018 - International Symposium on Code Generation and Optimization. pp. 1–12 (2018)
37. Sivarajah, S., Dilkes, S., Cowtan, A., Simmons, W., Edgington, A., Duncan, R.: t|ket⟩: A retargetable compiler for nisq devices. Quantum Science and Technology (2020)
38. Steiger, D.S., Häner, T., Troyer, M.: ProjectQ: an open source software framework for quantum computing. Quantum **2**, 49 (2018)
39. Strauch, S., Andrikopoulos, V., Bachmann, T., Karastoyanova, D., Passow, S., Vukojevic-Haupt, K.: Decision Support for the Migration of the Application Database Layer to the Cloud. In: 2013 IEEE 5th International Conference on Cloud Computing Technology and Science. vol. 1, pp. 639–646. IEEE (2013)
40. Suchara, M., Kubiatowicz, J., Faruque, A., Chong, F.T., Lai, C.Y., Paz, G.: QuRE: The Quantum Resource Estimator Toolbox. In: IEEE 31st International Conference on Computer Design (ICCD). pp. 419–426. IEEE (2013)

41. Tannu, S.S., Qureshi, M.K.: Not all qubits are created equal: A case for variability-aware policies for nisq-era quantum computers. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. p. 987–999. ASPLOS '19, Association for Computing Machinery, New York, NY, USA (2019)
42. Wild, K., Breitenbücher, U., Harzenetter, L., Leymann, F., Vietz, D., Zimmermann, M.: TOSCA4QC: Two Modeling Styles for TOSCA to Automate the Deployment and Orchestration of Quantum Applications. In: 2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC). IEEE Computer Society (2020)
43. Zhang, M., Ranjan, R., Nepal, S., Menzel, M., Haller, A.: A Declarative Recommender System for Cloud Infrastructure Services Selection . In: International Conference on Grid Economics and Business Models. pp. 102–113. Springer (2012)
44. Zimmermann, O., Grundler, J., Tai, S., Leymann, F.: Architectural Decisions and Patterns for Transactional Workflows in SOA. In: Service-Oriented Computing – ICSOC 2007. pp. 81–93. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
45. Zimmermann, O., Koehler, J., Leymann, F., Polley, R., Schuster, N.: Managing architectural decision models with dependency relations, integrity constraints, and production rules. Journal of Systems and Software **82**(8), 1249–1267 (2009)