The Quantum Software Lifecycle

Benjamin Weder, Johanna Barzen, Frank Leymann,
Marie Salm, Daniel Vietz

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

Benjamin Weder, Johanna Barzen, Frank Leymann, Marie Salm, and Daniel Vietz. 2020. The Quantum Software Lifecycle. In Proceedings of the 1st ACM SIGSOFT International Workshop on Architectures and Paradigms for Engineering Quantum Software (APEQS 2020). Association for Computing Machinery, New York, NY, USA, 2–9. DOI: https://doi.org/10.1145/3412451.3428497

BIBTEX

```
@inproceedings{Weder2020_QuantumSoftwareLifecycle,
  author    = {Weder, Benjamin; Barzen, Johanna; Leymann, Frank;
               Salm, Marie; Vietz, Daniel},
  title     = {{The Quantum Software Lifecycle}},
  booktitle = {Proceedings of the 1st ACM SIGSOFT International
               Workshop on Architectures and Paradigms for
               Engineering Quantum Software (APEQS 2020)},
  publisher = {ACM},
  year      = 2020,
  month     = nov,
  pages     = {2--9},
  doi       = {10.1145/3412451.3428497}
}
```

The Quantum Software Lifecycle

Benjamin Weder
weder@iaas.uni-stuttgart.de
Institute of Architecture of Application Systems,
University of Stuttgart, Germany

Johanna Barzen
barzen@iaas.uni-stuttgart.de
Institute of Architecture of Application Systems,
University of Stuttgart, Germany

Frank Leymann
leymann@iaas.uni-stuttgart.de
Institute of Architecture of Application Systems,
University of Stuttgart, Germany

Marie Salm
salm@iaas.uni-stuttgart.de
Institute of Architecture of Application Systems,
University of Stuttgart, Germany

Daniel Vietz
vietz@iaas.uni-stuttgart.de
Institute of Architecture of Application Systems,
University of Stuttgart, Germany

ABSTRACT

Quantum computing is an emerging paradigm that enables to solve a variety of problems more efficiently than it is possible on classical computers. As the first quantum computers are available, quantum algorithms can be implemented and executed on real quantum hardware. However, the capabilities of today's quantum computers are very limited and quantum computations are always disturbed by some error. Thus, further research is needed to develop or improve quantum algorithms, quantum computers, or required software tooling support. Due to the interdisciplinary nature of quantum computing, a common understanding of how to develop and execute a quantum software application is needed. However, there is currently no methodology or lifecycle comprising all relevant phases that can occur during the development and execution process. Hence, in this paper, we introduce the quantum software lifecycle consisting of ten phases a gate-based quantum software application should go through. We analyze the purpose of each phase, the available methods and tools that can be applied, and the open problems or research questions. Therefore, the lifecycle can be used as a baseline for discussions and future research.

CCS CONCEPTS

• **Computer systems organization → Quantum computing**; • **Software and its engineering → Software creation and management**; **Designing software**; **Software development methods**; **Software post-development issues**.

KEYWORDS

Quantum Software Development, Quantum Computing, Quantum Applications, NISQ, Software Engineering, Software Lifecycle

ACM Reference Format:
Benjamin Weder, Johanna Barzen, Frank Leymann, Marie Salm, and Daniel Vietz. 2020. The Quantum Software Lifecycle. In *Proceedings of the 1st ACM SIGSOFT International Workshop on Architectures and Paradigms for Engineering Quantum Software (APEQS '20), November 13, 2020, Virtual, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3412451.3428497

1 INTRODUCTION

Quantum computing is a promising research area, which may enable to solve a variety of problems more efficiently than it is possible on classical computers [55, 59]. Different quantum algorithms, such as *Shor's algorithm* [65] for factorizing numbers, *Grover's algorithm* [23] for unstructured search, or the *HHL algorithm* [25] for solving linear equations, provide a speedup over the best known classical algorithms for these problems. Furthermore, different vendors, such as IBM or Rigetti, developed quantum computers in recent years, and offer access to them, e.g., via the cloud [39, 44].

However, today's quantum computers are error-prone and have only limited capabilities [42, 59]. One restriction is the small number of qubits that they provide [39]. Hence, the size of the input data that can be represented within the quantum computer is limited [55]. Another problem is the noise that affects calculations on quantum computers [35, 59]. For example, due to unintended interactions between the qubits and their environment, their state is only stable for a certain amount of time, which is referred to as *decoherence* [44, 55]. Therefore, today's quantum computers are often also called *Noisy Intermediate-Scale Quantum (NISQ)* [59] computers.

Thus, a lot of research has to be done to design new or improve existing quantum algorithms, to increase the capabilities of available quantum computers, and to develop required software tooling support, such as suitable modeling tools, quantum compilers, or a platform to share and discuss developed quantum algorithms and their implementations [43]. Hence, quantum computing is a very interdisciplinary research area, which requires the knowledge of experts from many different fields, such as physics, mathematics, or computer science [55]. To ensure the successful cooperation between the various experts, a common understanding of how a typical *quantum software application* is developed, executed, and possibly adapted afterward is required. Thereby, a quantum software application comprises all required software artifacts to execute a quantum algorithm. This means, besides the software artifacts implementing the quantum algorithm, all related classical code, e.g., to initialize the quantum algorithm with the input data. However, there is currently no methodology or lifecycle comprising all relevant phases a quantum software application should go through.

Thus, the goal of this paper is to introduce the *quantum software lifecycle* consisting of ten phases that should be covered during the development and execution of quantum software applications. Thereby, the purpose of each phase and the available tooling support that can be used to conduct them are discussed. Hence, the lifecycle enables a unified view of the development and usage process of quantum applications. Furthermore, it shows the different phases in which future research has to be conducted to improve the processes.

There exists a variety of different quantum computing models, e.g., *gate-based* [49], *measurement-based* [31], and *adiabatic* quantum computing [2]. The diverse models represent quantum algorithms in various ways. However, it can be shown that the different models are formally equivalent [2, 31]. In this paper, we restrict our considerations to the gate-based quantum computing model, as many available quantum computers rely on it [39]. However, some phases of the quantum software lifecycle also apply to the other quantum computing models or need only small adjustments.

The remainder of this paper is structured as follows: Section 2 describes fundamentals and the problem statement that underlies our work. In Section 3, the quantum software lifecycle and its different phases are presented. Then, Section 4 describes the assumptions on which the lifecycle is based and the limitations of our work. The related work is discussed in Section 5, and we conclude in Section 6.

2 FUNDAMENTALS & PROBLEM STATEMENT

In this section, we introduce fundamentals about noisy intermediate-scale quantum computers and motivate why they pose special challenges for the development of new quantum software applications. Then, we analyze how hybrid algorithms can help to circumvent these problems and why the selection of suitable quantum hardware is important. Furthermore, fundamentals about provenance and how it can be used to improve applications are presented. Finally, the purpose of software lifecycles in the research area of software engineering and the problem statement of our work are described.

2.1 Noisy Intermediate-Scale Quantum

The term *Noisy Intermediate-Scale Quantum (NISQ)* was coined by John Preskill [59] to illustrate the capabilities of today's quantum computers and to describe the current state of quantum computing research. Thereby, "noisy" means that the gates and qubits of existing quantum computers are affected by noise from various sources, such as measurement and gate errors or qubit decoherence due to unintended interactions between the qubits and their environment [35, 72]. The noise leads to severe restrictions on the capabilities of today's quantum computers, as it limits the number of gates that can be successfully executed consecutively on a qubit before the result gets too inaccurate to be usable [55]. This maximum number of gates is referred to as the *maximum circuit depth*. Therefore, quantum algorithms that require a larger circuit depth cannot be executed. Due to this problem, different error correction codes [36, 38, 60] were proposed for quantum computing to correct occurring errors, and hence, to extend the maximum circuit depth. The application of such codes implies a high overhead, which means additional qubits and gates have to be added to the quantum circuit [59]. However, this is impractical for NISQ machines, as they are also limited in the number of available qubits, which is summarized by the term "intermediate-scale". Thereby, Preskill defines a number between 50 and a few hundred qubits as intermediate-scale. Therefore, quantum computers in the NISQ era are only capable to execute quantum circuits comprising a limited number of qubits and gates. This leads to challenges when developing and implementing quantum algorithms or selecting a suitable quantum computer for the execution of a given quantum algorithm, which will be covered in the following subsections in more detail.

2.2 Hybrid Algorithms

The hardware limitations of NISQ machines lead to the problem that existing quantum algorithms that provide an exponential speed-up compared to their best known classical counterparts can often not be executed on practically useful problems [59]. For example, one difficulty is to initialize the register of the quantum computer with the input data for the problem that needs to be solved, as the number of provided qubits can be too small to encode the data [42].

To reduce the problems of limited amounts of qubits and the restricted circuit depth, algorithms can be split into multiple parts and distributed over classical and quantum hardware [41]. Algorithms utilizing this approach are often referred to as *quantum-classical*, *variational*, or *hybrid algorithms* [48]. Thereby, the idea is to perform pre- or post-processing for a quantum computation on a classical computer [65]. In contrast, the part of the computation that can only be done inefficiently on classical computers, and for which quantum computers can provide an exponential speed-up, are executed on a quantum computer. Examples for hybrid algorithms are Shor's [65] and Simon's [68] algorithms, which use classical post-processing after the quantum computation. Another approach is to perform multiple iterations of quantum and classical computations. Thereby, the input for the quantum computation is improved in each iteration until the result reaches the required accuracy. Examples using this approach are the *variational quantum eigensolver (VQE)* [32] or the *quantum approximate optimization algorithm (QAOA)* [18, 19].

Therefore, hybrid algorithms can be used to solve problems that are not solvable on today's NISQ machines. That means, they can be utilized to already profit from the advantages of quantum computers, even in the current early research and development stage. However, most splits into quantum and classical parts are problem-specific and have to be done manually for each problem when designing the algorithms [41]. Hence, documented best practices and patterns could help to develop new hybrid algorithms. Further, an automated recommendation system could suggest which part of a problem to execute on classical and which part on quantum hardware, and therefore, ease the development of hybrid algorithms.

2.3 Quantum Hardware Selection

Quantum computers can be based on different physical qubit realizations, such as electron spins [58], trapped ions [56], or superconducting qubits [13], and the kind of realization leads to different characteristics when executing quantum algorithms [46]. However, even quantum computers with the same kind of physical realization

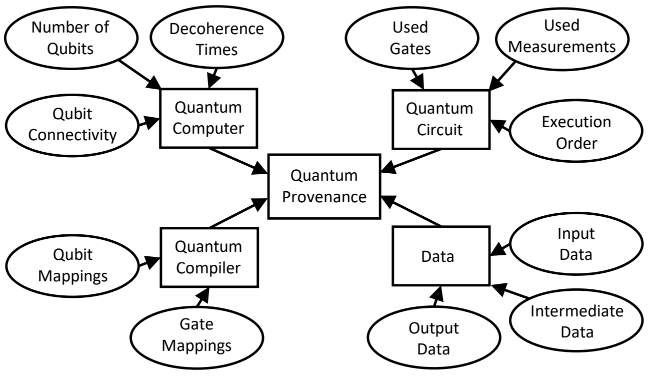

Figure 1: Excerpt of relevant quantum provenance data types

can differ significantly concerning properties, such as qubit count, qubit connectivity, or fidelity of the implemented gates [73]. This results in the fact that not all quantum computers can execute the same set of quantum circuits successfully [44, 62]. Instead, some circuits are better executed on one quantum computer and other circuits on another quantum computer, if they, e.g., use divergent kinds of gates, that are implemented with different fidelities in various quantum computers. Thus, the selection of suitable quantum hardware is a difficult task and an important constituent when developing and executing quantum algorithms. In previous work [62], we outlined an approach to analyze a given quantum circuit, extract its important characteristics, and select suitable quantum hardware based on these characteristics and the properties of the available quantum computers. However, the hardware selection approach needs to be integrated into the quantum software lifecycle.

2.4 Quantum Provenance

Provenance comprises any information that describes the manufacturing process of a product [16, 27]. Thereby, the product can be, e.g., a piece of digital data or a physical object. By collecting provenance data, the reproducibility of the production process can be achieved [57]. Further, it can improve the understandability and quality of the process, e.g., by analyzing past executions and improving the process based on the results [27]. Thus, to enable detailed insights, it is important to capture all relevant information systematically. Provenance data is collected in different areas and with diverse granularities, and hence, existing approaches can be classified into different kinds of provenance, such as *data provenance* [67] or *workflow provenance* [5]. However, there are currently no provenance approaches for quantum computing, that cover all steps from the identification of relevant data, over the collection, to its analysis.

During the NISQ era, provenance approaches are especially important for quantum computing [42]. One reason is that the different realizations of NISQ machines lead to diverse characteristics, which have to be collected to enable the later analysis of quantum computations. Additionally, the noise can lead to errors in the computations, and the provenance data can be used to analyze their origins. Furthermore, provenance data can also be used to select suitable quantum hardware or to improve quantum circuits [62].

Hence, we introduce the research area of *quantum provenance* and perform a first analysis, which kind of provenance data should be collected for quantum computing. Thereby, four different categories of provenance data can be distinguished, as sketched in Figure 1. First, information about the quantum computer executing a quantum circuit, such as the number of provided qubits or decoherence times, has to be gathered. Second, provenance data about the quantum circuit, like used gates and measurements, are important. Third, the input data, output data, and possible intermediate results are required for a successful analysis of the execution. Such intermediate results can, e.g., be retrieved after the different iterations of a variational algorithm and may comprise the parameterization used for the current iteration and the corresponding measurement results. Finally, the *quantum compiler* [9, 28] is in charge of mapping the abstract quantum circuit to the physical qubits and hardware provided gates, and these mappings have a strong influence on the execution time and error probability. Therefore, the details about the mappings have to be collected too. However, further analysis of important data types, as well as new approaches to collect and analyze the data, are required. In this work, we incorporate the quantum provenance approach into the quantum software lifecycle and show in which phases provenance data can be gathered and which phases can benefit from the collected provenance data.

2.5 Software Lifecycles

In the field of software engineering, software lifecycles are often used to document the different phases and the order of their occurrence during the development and execution of certain software artifacts [12, 37, 45, 53]. Thus, they provide a baseline for the discussion about methods and best practices that are applied in the various phases and open problems that should be solved to improve the development and execution process. Moreover, software lifecycles can be used to educate developers or system administrators by providing an overview of all phases and enable them to deepen their understanding of the phases that are most relevant for them.

2.6 Problem Statement

As outlined in the previous sections, the development and execution of quantum software applications comprise a lot of complex tasks, e.g., the selection of suitable quantum hardware. However, there exists no methodology or lifecycle that covers all relevant phases, which can be used as a baseline for further discussions and research. Therefore, the resulting research question for this work can be formulated as follows: *"What phases should a typical quantum software application go through, how do these phases relate to each other, and what are open problems for the different phases?"*

3 QUANTUM SOFTWARE LIFECYCLE

In this section, we present the *quantum software lifecycle*, which is depicted in Figure 2. In the following subsections, we describe its different phases, starting from the *quantum-classical splitting* phase, which is the phase where the user typically enters the lifecycle with the problem description. Then, the other phases are followed by their usual execution order. Thereby, the purpose of each phase, existing methods or tooling support that can be used to accomplish it, and possible input and output data of the phase are discussed.

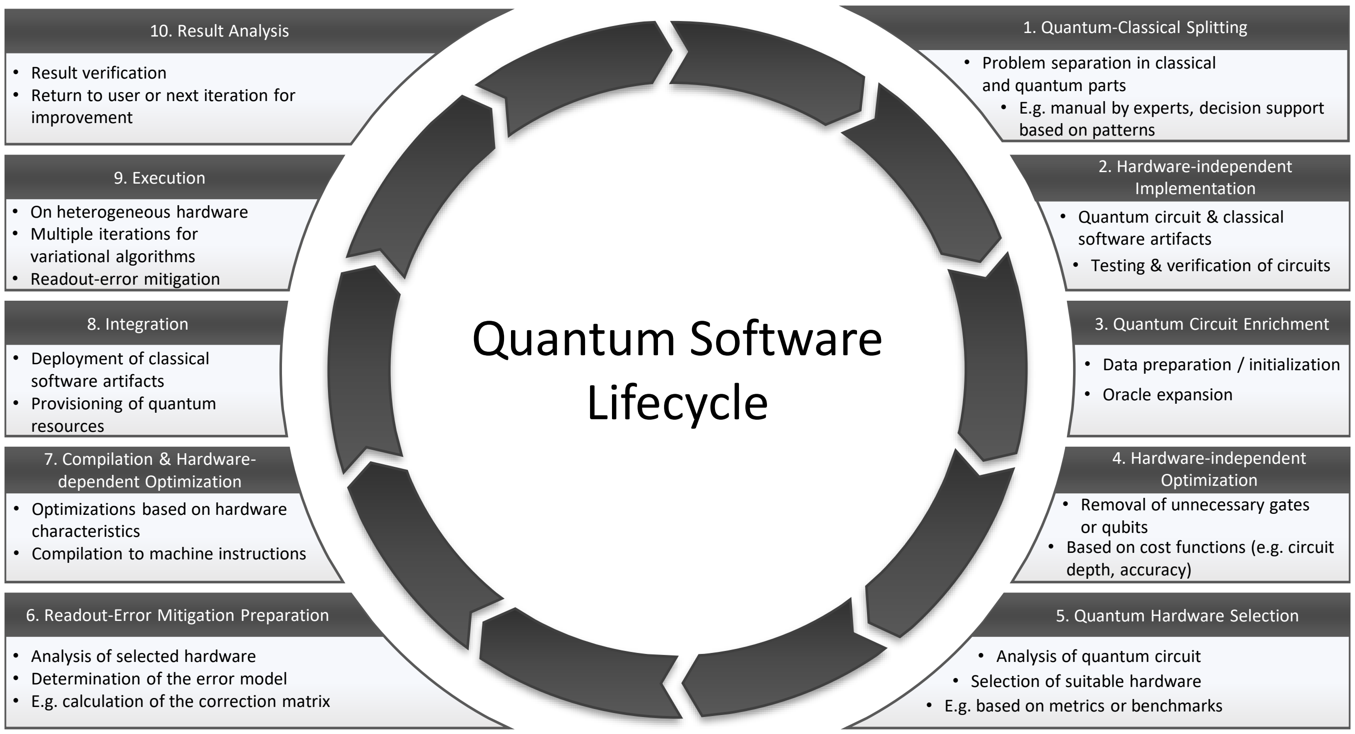
Quantum Software Lifecycle
1. Quantum-Classical Splitting
• Problem separation in classical and quantum parts
• E.g. manual by experts, decision support based on patterns
2. Hardware-independent Implementation
• Quantum circuit & classical software artifacts
• Testing & verification of circuits
3. Quantum Circuit Enrichment
• Data preparation / initialization
• Oracle expansion
4. Hardware-independent Optimization
• Removal of unnecessary gates or qubits
• Based on cost functions (e.g. circuit depth, accuracy)
5. Quantum Hardware Selection
• Analysis of quantum circuit
• Selection of suitable hardware
• E.g. based on metrics or benchmarks
6. Readout-Error Mitigation Preparation
• Analysis of selected hardware
• Determination of the error model
• E.g. calculation of the correction matrix
7. Compilation & Hardware-dependent Optimization
• Optimizations based on hardware characteristics
• Compilation to machine instructions
8. Integration
• Deployment of classical software artifacts
• Provisioning of quantum resources
9. Execution
• On heterogeneous hardware
• Multiple iterations for variational algorithms
• Readout-error mitigation
10. Result Analysis
• Result verification
• Return to user or next iteration for improvement

Figure 2: Overview of the quantum software lifecycle

3.1 Quantum-Classical Splitting

The first phase of the quantum software lifecycle is entered by the user with the new problem description or an updated problem description based on the results of previous iterations. In this phase, it is decided which parts of the problem to solve on a quantum computer and which on a classical computer depending on the requirements of the problem description. Due to the restricted capabilities of NISQ machines, most problems have to be solved in a hybrid manner on quantum and classical hardware (see subsection 2.2) [48, 59]. The separation into quantum and classical parts can be done by experts based on their knowledge and experience. However, this is a difficult and error-prone task, which requires immense knowledge from various fields, such as physics, mathematics, and computer science [55]. Therefore, this process should be supported by an automated recommender, which can, e.g., be based on documented best practices or patterns [41]. Additionally, the recommender can analyze the provenance data of passed executions, e.g., by using machine learning techniques and utilize the insights to improve future recommendations for this phase [3].

3.2 Hardware-Independent Implementation

In the second phase, the quantum circuits and the classical software artifacts implementing the quantum and classical problem parts resulting from the previous phase are created. Thereby, for the implementation of the quantum circuits, a *hardware-independent high-level quantum programming language* should be used to enable a later vendor-agnostic hardware selection (see subsection 3.5) [24, 44, 74]. Otherwise, the selected quantum programming language can reduce the set of compatible quantum computers already significantly. The quantum circuit should also be defined independently of certain input data to enable its reusability for different instances of the tackled problem. Therefore, the quantum circuit does not yet contain the initialization steps, which are used to pass input data to the quantum computer (see subsection 3.3) [41]. Hence, phases one and two of the quantum software lifecycle can be skipped if the same problem should be solved for different input data, and the lifecycle can be (re)started in phase three. After the implementation, the quantum circuits are verified by experts or using automated approaches to prove their correct functionality [4, 50, 76]. Further, the classical software artifacts should be tested accordingly [54].

3.3 Quantum Circuit Enrichment

During this phase, the quantum circuit is enriched with the required details to solve a particular instance of the problem that is tackled by the quantum software application. Thus, the quantum circuit is initialized with the input data in a *data preparation* step [15, 41, 55]. This means, an initialization circuit is added to the beginning of the original circuit, which prepares the required state in the quantum register. This step is necessary as many quantum computers only allow to initialize their register in the *all-zero state*, which means all qubits set to zero [41]. Thereby, different schemes exist to encode the input data into the initializing circuit, such as *basis* or *amplitude encoding* [42, 51]. Another important step is the expansion of the *oracles* that may be used in the quantum circuit. Many quantum algorithms, such as Simon's or Grover's algorithm, utilize oracles, e.g., to decide if an element in a collection is the searched one or not [23, 52, 68]. However, these oracles depend on a specific problem instance and have to be implemented using a collection of gates during this phase of the quantum software lifecycle [33].

3.4 Hardware-Independent Optimization

After the quantum circuit enrichment, a *hardware-independent optimization* phase is performed for the resulting quantum circuits. Hence, the optimizations of this phase are based on the hardware-independent high-level quantum programming language [24, 74]. Thereby, an equivalent representation of the quantum circuit is created, which is optimized with respect to a certain cost function, such as the circuit size, circuit depth, or accuracy [74]. These optimizations can be, for example, based on best practices, patterns and anti-patterns, or collected provenance data about passed executions including the performed optimizations and the resulting outcomes.

3.5 Quantum Hardware Selection

In the fifth phase, suitable quantum hardware for the execution of the implemented and optimized quantum circuits has to be selected. Thereby, a suitable quantum computer must be capable of executing a quantum circuit with a given maximum error probability [62]. Additionally, other optimization goals, such as the incurred monetary costs for the execution or the set of vendors that are classified as confidential by the user, have to be taken into account. For the quantum hardware selection, the quantum circuits must be analyzed first, and important characteristics, such as the width and depth of the circuit, have to be retrieved (see subsection 2.3) [44, 73]. Afterward, the hardware selection can be performed based on the obtained circuit characteristics and the capabilities of the available quantum computers. Thereby, the capabilities of quantum computers can be assessed using different metrics, such as *quantum volume* [8] and the *total quantum factor* [64]. Another approach is the usage of benchmarks, which can, e.g., be based on the sampling of pseudo-random quantum circuits [6] or error correction codes [36]. Further, provenance information about past executions of quantum circuits with similar characteristics on different quantum computers can be utilized to improve the quantum hardware selection process.

3.6 Readout-Error Mitigation Preparation

Due to the noisy devices during the NISQ era, results of quantum computations are always disturbed by some errors [59]. One reason for errors are gates that can not be executed exactly, which can be solved by error-correction codes if enough qubits are available [36, 38, 60]. However, also the measurements are noisy and can add errors to the results, which are referred to as *readout-errors* [47]. Thus, it is important to apply *readout-error mitigation* to the results to reduce the influence of these errors (see subsection 3.9) [47, 71]. Such readout-error mitigation approaches are based on so-called *unfolding techniques* and depend on the error model of the used quantum computer [11, 42, 47]. However, the error model may change over time, e.g., due to a re-calibration of the quantum computer [75]. Hence, the current error model has to be analyzed periodically and stored as provenance data during this phase.

3.7 Compilation & Hardware-Dependent Optimization

After the selection of suitable quantum hardware, the quantum circuits have to be compiled to the machine instructions that are required for the execution by the selected quantum computer [9, 74]. If the quantum circuits are implemented using a hardware-independent high-level quantum programming language (see subsection 3.2), the compilation process is usually performed in two separate steps [30, 44, 74]. First, (i) the quantum circuits are compiled to a *quantum intermediate representation*. This intermediate representation can be, e.g., the quantum programming language of an SDK, such as *OpenQASM* for *Qiskit* [29] or *Quil* for *Forest* [61], that supports the execution on the selected quantum computer. Then, (ii) the intermediate representation has to be compiled to the machine instructions utilized by the selected quantum computer in the second compilation step [9]. For this compilation, the hardware-dependent compilers provided by the quantum hardware vendors, such as IBM or Rigetti, can be used. Thereby, a hardware-dependent optimization is performed during the compilation [28, 69]. This means, the specific characteristics of the selected quantum computer, such as the decoherence times of different qubits or the qubit connectivity, are taken into account. For example, qubits on which many two-qubit gates are executed in the quantum circuit are mapped to physical qubits of the quantum computer that are directly connected if possible to avoid additional *SWAP* gates. As described in subsection 2.4, the collection of the qubit and gate mappings performed by the hardware-dependent compilers as provenance data is important as they can have a significant impact on the quantum circuit execution and the returned results [9, 28, 30].

3.8 Integration

In this phase, the compiled quantum circuits and classical software artifacts have to be deployed and integrated to execute the quantum software application in the next phase. Thereby, for the classical software artifacts, suited *deployment models* should be created to automate their deployment [34, 79]. Such deployment models describe all required components and information for the deployment of an application in a reusable and maintainable manner [78]. Applications defined by deployment models can be automatically deployed by a *deployment system*, such as *Terraform* [26] or *Kubernetes* [14]. Alternatively, the deployment of the classical software artifacts can be done manually. However, this process is time-consuming, error-prone, and requires immense technical knowledge [10]. For the quantum circuits, the utilized SDKs of the quantum hardware providers, such as Qiskit [29] or Forest [61], usually handle the deployment to the supported quantum hardware [39, 70]. However, the quantum computers that are currently available over the cloud, for example, from IBM, are mostly job-based. Therefore, the deployment of quantum circuits equals their execution at the moment, and hence, is done in the next phase of the quantum software lifecycle. However, some providers enable reserving time slices for the execution on their quantum computers, and therefore, this reservation can be done during the integration phase [39]. Finally, the classical software artifacts have to be configured to enable the invocation of the quantum parts, for example, by updating the endpoint information with the details about the selected quantum computer.

3.9 Execution

In the *execution* phase, the quantum software application is conducted on the heterogeneous quantum and classical hardware. First, pre-processing is performed if required, and then, the quantum circuit is executed. After the classical post-processing, the results are sent back to the user. Additional to algorithm-specific post-processing, readout-error mitigation should be performed to reduce the noise in the results [11, 42, 71]. For this, the hardware-dependent error model that is analyzed and stored in the *readout-error mitigation preparation* phase can be used (see subsection 3.6). For example, some approaches store the error model in the form of a *correction matrix* and apply this matrix to the results [47]. However, the error model can not be determined exactly and different unfolding techniques can lead to different qualities of mitigation for various quantum circuits and quantum computers [11]. In the case of a variational algorithm, multiple iterations between classical and quantum processing may occur [48]. Thereby, these algorithms enable to improve the result by computing new input data or changing the parametrization of some parameterized gates in each iteration [17]. Hence, it is not always required to enter a new quantum software lifecycle iteration to improve the result. During the execution, provenance data about the used hardware, their current state, and possible intermediate results should be collected to enable the later successful analysis of the execution and the final results.

3.10 Result Analysis

In the last phase of the quantum software lifecycle, the results of the execution phase are analyzed. If the results can be automatically validated, e.g., for the factorization of numbers by multiplying the factors and verifying the equality to the input number, the next iteration of the quantum software lifecycle can be entered if the results are faulty. For other problems, the assessment has to be done by the user to decide whether an additional iteration is required to improve the results of the quantum software application or not.

4 DISCUSSION

In this section, we summarize the assumptions on which the quantum software lifecycle is based and discuss the potential limitations of our work. The quantum software lifecycle is intended as a baseline for discussions and future research about the development and execution of gate-based quantum software applications during the NISQ era. Therefore, some of the lifecycle phases have to be adapted, for example, to represent the particularities of the adiabatic [2] or measurement-based [31] quantum computing model. Furthermore, some of the presented phases are only required due to the limited capabilities of NISQ machines [59]. For example, the readout-error mitigation preparation phase, and the corresponding mitigation step in the execution phase, are only needed as long as the influence of readout-errors is significant. In the same way, the development of an efficient *quantum random access memory (QRAM)* [22] can change the data preparation step in the quantum circuit enrichment phase. However, there is no efficient implementation of QRAM available today [42]. Therefore, the quantum software lifecycle is no fixed construct and may be refined with new advancements in quantum computing research, especially when fully-fault tolerant quantum computers are available after the NISQ era [59].

5 RELATED WORK

Zhao [80] presents a comprehensive survey of the research area of quantum software engineering. Thereby, he summarizes, e.g., available quantum programming languages, software tools, or methods to test and maintain quantum software applications. Furthermore, he also proposes a lifecycle consisting of five phases: (i) *quantum software requirements analysis*, (ii) *quantum software design*, (iii) *quantum software implementation*, (iv) *quantum software testing*, and (v) *quantum software maintenance*. However, the proposed lifecycle is very abstract and does not incorporate important phases for quantum software applications during the NISQ era, e.g., the data preparation, the oracle expansion, or the mitigation of readout-errors. Thus, our introduced quantum software lifecycle is a refinement of this lifecycle for applications during the NISQ era.

In different research areas of computer science, lifecycles are described to document the diverse phases a software artifact goes through [12, 37, 45, 53]. Kohlborn et al. [37] propose a *business and software service lifecycle*, which covers the various phases of web service development and execution from the requirement analysis, over the implementation and operation, to the retirement of the web service. Leymann et al. [45] and Canós et al. [12] introduce *workflow lifecycles*, in which they define different phases a workflow must pass through during development and runtime, as well as their order and important input and output data. Furthermore, they integrate an *exec log* to their lifecycle, which collects information during the various phases and which can be used for later analysis of executions [1]. This log is often also referred to as an *audit log* or *audit trail* [45, 77] and can be compared to the provenance component used in different phases of the quantum software lifecycle. Munassar and Govardhan [53] compare various well-known software development lifecycles, such as the *waterfall model*, the *V-model*, or the *spiral model*. Gabor et al. [20] introduce an engineering process for machine learning, the so-called *machine learning pipeline*, which summarizes the necessary tasks to successfully apply machine learning in the lifecycle of self-adaptive systems. Furthermore, they show the applicability of quantum artificial intelligence (QAI) by describing the tasks of the machine learning pipeline that can be executed on a quantum computer [21].

In addition to software lifecycles, there are also data lifecycles proposed in the literature. Such lifecycles describe the relevant data processing steps from the data gathering, over its storage and analysis, to the release of the collected data [7, 40].

Some existing research works present relevant phases that have to be taken into account when developing and executing quantum software applications without describing an entire lifecycle. Additionally, software tooling support for different phases is developed in various works. In the following, we present research works covering one or multiple phases of the quantum software lifecycle, that were not already discussed in the previous sections.

Svore et al. [74] and Häner et al. [24] propose a software design flow for the compilation and optimization of quantum software applications. Thereby, they introduce a two-level compilation process, that first compiles the quantum algorithm definition from a high-level quantum programming language to an intermediate representation and in the second step to the quantum hardware instructions for a certain quantum computer. During that process, they first optimize the quantum algorithm hardware-independent and afterward hardware-dependent while compiling to the machine instructions as presented in the quantum software lifecycle.

Sim et al. [66] introduce *algo2qpu*, a framework to deploy hybrid algorithms to cloud-based quantum computers or simulators. They propose a workflow with steps that should be performed during the development and execution of quantum software applications, such as quantum algorithm selection, quantum circuit implementation, compilation, and execution. However, they do, e.g., not consider the splitting into quantum and classical parts using a recommendation system or the hardware selection based on the implemented quantum circuit and the properties of the available quantum computers.

Scherer et al. [63] propose an approach to analyze quantum circuits and estimate their resource requirements by using a combination of manual analysis and automated estimates based on the *Quipper* quantum programming language. Therefore, this approach can be utilized in the quantum hardware selection phase.

6 CONCLUSION

Quantum computing is a promising research area that can enable breakthroughs in different fields in the future. While there is lots of ongoing research and effort to develop required tooling support or new quantum algorithms, a holistic overview of the relevant phases, a typical quantum software application goes through, is missing. However, due to the interdisciplinary nature of quantum computing, a common understanding of the relevant phases is required as a basis for discussions and future research. Additionally, this can serve as a starting point to deepen the understanding of methods and related tools that are used in the various phases. In this paper, we presented the quantum software lifecycle, which consists of ten identified phases that may occur during the development of quantum software applications, their execution, and possible adaptations based on the results of previous executions. Thereby, the goals of the phases, available methods and tools for their accomplishment, and possible input and output data are outlined. Furthermore, we motivated the need for a comprehensive provenance approach that covers the gathering of all relevant data and analyzed in which phases provenance data can be collected or utilized.

In future work, we plan to develop new methods and tooling support for the various phases of the quantum software lifecycle. Thereby, we want to focus specifically on the deployment of hybrid algorithms, the question of how to provide them as a service, and the selection of suitable quantum hardware. Additionally, a provenance system that collects all relevant data about quantum computations and provides it for the different phases will be developed. Finally, we plan to extend our pattern language on quantum computing [41] with further patterns, which can, e.g., be used to decide what parts of a problem to run on quantum or classical hardware.

ACKNOWLEDGEMENTS

The authors would like to thank the German Research Foundation (DFG) for financial support of the project within the Cluster of Excellence in *Simulation Technology* (EXC 2075 – 390740016) at the University of Stuttgart. This work was partially funded by the BMWi project *PlanQK* (01MK20005N).

REFERENCES

[1] Rakesh Agrawal, Dimitrios Gunopulos, and Frank Leymann. 1998. Mining Process Models from Workflow Logs. In *International Conference on Extending Database Technology*. Springer, 467–483. https://doi.org/10.1007/BFb0101003
[2] Dorit Aharonov, Wim Van Dam, Julia Kempe, Zeph Landau, et al. 2008. Adiabatic Quantum Computation Is Equivalent to Standard Quantum Computation. *SIAM review* 50, 4 (2008), 755–787. https://doi.org/10.1137/080734479
[3] Sunita B Aher and LMRJ Lobo. 2013. Combination of machine learning algorithms for recommendation of courses in E-Learning System based on historical data. *Knowledge-Based Systems* 51 (2013), 1–14. https://doi.org/10.1016/j.knosys.2013.04.015
[4] Matthew Amy. 2018. Towards Large-scale Functional Verification of Universal Quantum Circuits. *arXiv preprint arXiv:1805.06908* (2018). https://doi.org/10.4204/EPTCS.287.1
[5] Manish Kumar Anand, Shawn Bowers, and Bertram Ludäscher. 2010. Techniques for Efficiently Querying Scientific Workflow Provenance Graphs. In *EDBT*, Vol. 10. 287–298. https://doi.org/10.1145/1739041.1739078
[6] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, et al. 2019. Quantum supremacy using a programmable superconducting processor. *Nature* 574, 7779 (2019), 505–510. https://doi.org/10.1038/s41586-019-1666-5
[7] Alex Ball. 2012. *Review of Data Management Lifecycle Models.* University of Bath, IDMRC.
[8] Lev S Bishop, Sergey Bravyi, Andrew Cross, Jay M Gambetta, et al. 2017. Quantum volume. *Technical Report* (2017).
[9] Jeffrey Booth Jr. 2012. Quantum Compiler Optimizations. *arXiv preprint arXiv:1206.3348* (2012).
[10] Uwe Breitenbücher, Tobias Binz, Kálmán Képes, Oliver Kopp, et al. 2014. Combining Declarative and Imperative Cloud Application Provisioning based on TOSCA. In *International Conference on Cloud Engineering (IC2E)*. IEEE, 87–96. https://doi.org/10.1109/IC2E.2014.56
[11] Lydia Brenner, Pim Verschuuren, Rahul Balasubramanian, Carsten Burgard, Vincent Croft, Glen Cowan, and Wouter Verkerke. 2019. Comparison of unfolding methods using RooFitUnfold. *arXiv:1910.14654* (2019).
[12] José H Canós, Mª Carmen Penadés, and José Á Carsí. 1999. From Software Process to Workflow Process: the Workflow Lifecycle. In *Proceedings of the International Process Technology Workshop, Grenoble, France.*
[13] John Clarke and Frank K Wilhelm. 2008. Superconducting quantum bits. *Nature* 453, 7198 (2008), 1031. https://doi.org/10.1038/nature07128
[14] CNCF. 2020. *Kubernetes. [online].* https://kubernetes.io/
[15] John A Cortese and Timothy M Braje. 2018. Loading Classical Data into a Quantum Computer. *arXiv preprint arXiv:1807.02500* (2018).
[16] Susan B Davidson and Juliana Freire. 2008. Provenance and Scientific Workflows: Challenges and Opportunities. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 1345–1350. https://doi.org/10.1145/1376616.1376772
[17] Yuxuan Du, Min-Hsiu Hsieh, Tongliang Liu, and Dacheng Tao. 2020. Expressive power of parametrized quantum circuits. *Phys. Rev. Research* 2 (2020), 033125. Issue 3. https://doi.org/10.1103/PhysRevResearch.2.033125
[18] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. 2014. A Quantum Approximate Optimization Algorithm. *arXiv preprint arXiv:1411.4028* (2014).
[19] Edward Farhi and Aram W Harrow. 2016. Quantum Supremacy through the Quantum Approximate Optimization Algorithm. *arXiv:1602.07674* (2016).
[20] Thomas Gabor, Andreas Sedlmeier, Thomy Phan, Fabian Ritz, Marie Kiermeier, et al. 2020. The scenario coevolution paradigm: adaptive quality assurance for adaptive systems. *International Journal on Software Tools for Technology Transfer* (2020), 1–20. https://doi.org/10.1007/s10009-020-00560-5
[21] Thomas Gabor, Leo Sünkel, Fabian Ritz, Thomy Phan, Lenz Belzner, et al. 2020. The Holy Grail of Quantum Artificial Intelligence: Major Challenges in Accelerating the Machine Learning Pipeline. *arXiv preprint arXiv:2004.14035* (2020).
[22] Vittorio Giovannetti, Seth Lloyd, and Lorenzo Maccone. 2008. Quantum random access memory. *Physical review letters* 100, 16 (2008), 160501. https://doi.org/10.1103/PhysRevLett.100.160501
[23] Lov K Grover. 1996. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*. 212–219. https://doi.org/10.1145/237814.237866
[24] Thomas Häner, Damian S Steiger, Krysta Svore, and Matthias Troyer. 2018. A software methodology for compiling quantum programs. *Quantum Science and Technology* 3, 2 (2018), 020501. https://doi.org/10.1088/2058-9565/aaa5cc
[25] Aram W Harrow, Avinatan Hassidim, and Seth Lloyd. 2009. Quantum Algorithm for Linear Systems of Equations. *Physical review letters* 103, 15 (2009), 150502. https://doi.org/10.1103/PhysRevLett.103.150502
[26] HashiCorp. 2020. *Terraform. [online].* https://www.terraform.io/
[27] Melanie Herschel, Ralf Diestelkämper, and Houssem Ben Lahmar. 2017. A Survey on Provenance: What for? What Form? What from? *The VLDB Journal* 26, 6 (2017), 881–906. https://doi.org/10.1007/s00778-017-0486-1
[28] Luke E Heyfron and Earl T Campbell. 2018. An efficient quantum compiler that reduces T count. *Quantum Science and Technology* 4, 1 (2018), 015004.

https://doi.org/10.1088/2058-9565/aad604
[29] IBM. 2020. *Qiskit. [online]*. https://qiskit.org/
[30] Ali JavadiAbhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, et al. 2014. ScaffCC: A Framework for Compilation and Analysis of Quantum Computing Programs. In *Proceedings of the 11th ACM Conference on Computing Frontiers*. 1–10. https://doi.org/10.1145/2597917.2597939
[31] Richard Jozsa. 2006. An introduction to measurement based quantum computation. *NATO Science Series, III: Computer and Systems Sciences. Quantum Information Processing-From Theory to Experiment* 199 (2006), 137–158.
[32] Abhinav Kandala, Antonio Mezzacapo, Kristan Temme, et al. 2017. Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets. *Nature* 549, 7671 (2017), 242–246. https://doi.org/10.1038/nature23879
[33] Elham Kashefi, Adrian Kent, Vlatko Vedral, and Konrad Banaszek. 2002. Comparison of quantum oracles. *Physical Review A* 65, 5 (2002), 050304. https://doi.org/10.1103/PhysRevA.65.050304
[34] Kálmán Képes, Uwe Breitenbücher, Frank Leymann, et al. 2019. Deployment of Distributed Applications Across Public and Private Networks. In *Proceedings of the 23rd IEEE International Enterprise Distributed Object Computing Conference (EDOC)*. IEEE, 236–242. https://doi.org/10.1109/EDOC.2019.00036
[35] Emanuel Knill. 2005. Quantum computing with realistically noisy devices. *Nature* 434, 7029 (2005), 39–44. https://doi.org/10.1038/nature03350
[36] Emanuel Knill, Raymond Laflamme, Rudy Martinez, and Camille Negrevergne. 2001. Benchmarking Quantum Computers: The Five-Qubit Error Correcting Code. *Physical Review Letters* 86 (2001), 5811–5814. Issue 25. https://doi.org/10.1103/PhysRevLett.86.5811
[37] Thomas Kohlborn, Axel Korthaus, and Michael Rosemann. 2009. Business and Software Service Lifecycle Management. In *IEEE International Enterprise Distributed Object Computing Conference*. IEEE, 87–96. https://doi.org/10.1109/EDOC.2009.20
[38] Raymond Laflamme, Cesar Miquel, Juan Pablo Paz, and Wojciech Hubert Zurek. 1996. Perfect Quantum Error Correcting Code. *Physical Review Letters* 77, 1 (1996), 198. https://doi.org/10.1103/PhysRevLett.77.198
[39] Ryan LaRose. 2019. Overview and Comparison of Gate Level Quantum Software Platforms. *Quantum* 3 (2019), 130. https://doi.org/10.22331/q-2019-03-25-130
[40] Christopher Lenhardt, Stanley Ahalt, Brian Blanton, Laura Christopherson, et al. 2014. Data Management Lifecycle and Software Lifecycle Management in the Context of Conducting Science. *Journal of Open Research Software* 2, 1 (2014). https://doi.org/10.5334/jors.ax
[41] Frank Leymann. 2019. Towards a Pattern Language for Quantum Algorithms. In *Quantum Technology and Optimization Problems*. Springer International Publishing, 218–230. https://doi.org/10.1007/978-3-030-14082-3_19
[42] Frank Leymann and Johanna Barzen. 2020. The bitter truth about gate-based quantum algorithms in the NISQ era. *Quantum Science and Technology* 5, 4 (2020), 044007. https://doi.org/10.1088/2058-9565/abae7d
[43] Frank Leymann, Johanna Barzen, and Michael Falkenthal. 2019. Towards a Platform for Sharing Quantum Software. In *Proceedings of the 13th Advanced Summer School on Service Oriented Computing (IBM Technical Report (RC25685))*. IBM Research Division, 70–74.
[44] Frank Leymann, Johanna Barzen, Michael Falkenthal, et al. 2020. Quantum in the Cloud: Application Potentials and Research Opportunities. In *Proceedings of the 10th International Conference on Cloud Computing and Services Science*. SciTePress, 9–24. https://doi.org/10.5220/0009819800090024
[45] Frank Leymann and Dieter Roller. 2000. *Production Workflow: Concepts and Techniques*. Prentice Hall PTR.
[46] Norbert M. Linke, Dmitri Maslov, Martin Roetteler, Shantanu Debnath, et al. 2017. Experimental comparison of two quantum computing architectures. *Proceedings of the National Academy of Sciences* 114, 13 (2017), 3305–3310. https://doi.org/10.1073/pnas.1618020114
[47] Filip B Maciejewski, Zoltán Zimborás, and Michał Oszmaniec. 2019. Mitigation of readout noise in near-term quantum devices by classical post-processing based on detector tomography. *arXiv preprint arXiv:1907.08518* (2019). https://doi.org/10.22331/q-2020-04-24-257
[48] Jarrod R McClean, Jonathan Romero, Ryan Babbush, and Alán Aspuru-Guzik. 2016. The theory of variational hybrid quantum-classical algorithms. *New Journal of Physics* 18, 2 (2016), 023023. https://doi.org/10.1088/1367-2630/18/2/023023
[49] Kristel Michielsen, Madita Nocon, Dennis Willsch, Fengping Jin, et al. 2017. Benchmarking gate-based quantum computers. *Computer Physics Communications* 220 (2017), 44 – 55. https://doi.org/10.1016/j.cpc.2017.06.011
[50] Andriy Miranskyy, Lei Zhang, and Javad Doliskani. 2020. Is Your Quantum Program Bug-Free?. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. Association for Computing Machinery, 29–32. https://doi.org/10.1145/3377816.3381731
[51] Kosuke Mitarai, Masahiro Kitagawa, and Keisuke Fujii. 2019. Quantum analog-digital conversion. *Physical Review A* 99, 1 (2019), 012301. https://doi.org/10.1103/PhysRevA.99.012301
[52] Michele Mosca. 2008. Quantum Algorithms. *arXiv preprint arXiv:0808.0369* (2008).
[53] Nabil Mohammed Ali Munassar and A Govardhan. 2010. A Comparison Between Five Models Of Software Engineering. *International Journal of Computer Science Issues (IJCSI)* 7, 5 (2010), 94.
[54] Glenford J Myers, Corey Sandler, and Tom Badgett. 2011. *The Art of Software Testing*. John Wiley & Sons.
[55] Michael A Nielsen and Isaac Chuang. 2002. Quantum Computation and Quantum Information.
[56] Roee Ozeri. 2011. The trapped-ion qubit tool box. *Contemporary Physics* 52, 6 (2011), 531–550. https://doi.org/10.1080/00107514.2011.603578
[57] Beatriz Pérez, Julio Rubio, and Carlos Sáenz-Adán. 2018. A systematic review of provenance systems. *Knowledge and Information Systems* 57, 3 (2018), 495–543. https://doi.org/10.1007/s10115-018-1164-3
[58] Jarryd J. Pla, Kuan Y. Tan, Juan P. Dehollain, Wee H. Lim, et al. 2012. A single-atom electron spin qubit in silicon. *Nature* 489, 7417 (2012), 541–545. https://doi.org/10.1038/nature11449
[59] John Preskill. 2018. Quantum Computing in the NISQ era and beyond. *Quantum* 2 (2018), 79. https://doi.org/10.22331/q-2018-08-06-79
[60] Michael D. Reed, Leonardo DiCarlo, S. E. Nigg, L. Sun, et al. 2012. Realization of three-qubit quantum error correction with superconducting circuits. *Nature* 482, 7385 (2012), 382–385. https://doi.org/10.1038/nature10786
[61] Rigetti. 2020. *Docs for the Forest SDK. [online]*. http://docs.rigetti.com/en/stable/
[62] Marie Salm, Johanna Barzen, Uwe Breitenbücher, Frank Leymann, et al. 2020. A Roadmap for Automating the Selection of Quantum Computers for Quantum Algorithms. *arXiv preprint arXiv:2003.13409* (2020).
[63] Artur Scherer, Benoît Valiron, Siun-Chuon Mau, Scott Alexander, et al. 2017. Concrete resource analysis of the quantum linear-system algorithm used to compute the electromagnetic scattering cross section of a 2D target. *Quantum Information Processing* 16, 3 (2017), 60. https://doi.org/10.1007/s11128-016-1495-5
[64] Eyob A. Sete, William J. Zeng, and Chad T. Rigetti. 2016. A Functional Architecture for Scalable Quantum Computing. In *IEEE International Conference on Rebooting Computing*. 1–6. https://doi.org/10.1109/ICRC.2016.7738703
[65] Peter W. Shor. 1997. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM J. Comput.* 26, 5 (1997), 1484–1509. https://doi.org/10.1137/S0036144598347011
[66] Sukin Sim, Yudong Cao, Jonathan Romero, Peter D Johnson, et al. 2018. A framework for algorithm deployment on cloud-based quantum computers. *arXiv preprint arXiv:1810.10576* (2018).
[67] Yogesh L. Simmhan, Beth Plale, and Dennis Gannon. 2005. A Survey of Data Provenance in e-Science. *SIGMOD Rec.* 34, 3 (2005), 31–36. https://doi.org/10.1145/1084805.1084812
[68] Daniel R Simon. 1994. On the power of quantum cryptography. In *35th Annual Symposium on Foundations of Computer Science*. 116–123.
[69] Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. 2020. t| ket>: A retargetable compiler for NISQ devices. *Quantum Science and Technology* (2020). https://doi.org/10.1088/2058-9565/ab8e92
[70] Robert S Smith, Michael J Curtis, and William J Zeng. 2016. A Practical Quantum Instruction Set Architecture. *arXiv preprint arXiv:1608.03355* (2016).
[71] Chao Song, Jing Cui, H Wang, J Hao, H Feng, and Ying Li. 2019. Quantum computation with universal error mitigation on a superconducting quantum processor. *Science advances* 5, 9 (2019). https://doi.org/10.1126/sciadv.aaw5686
[72] Andrew Steane. 1998. Quantum computing. *Reports on Progress in Physics* 61, 2 (1998), 117. https://doi.org/10.1088/0034-4885/61/2/002
[73] Martin Suchara, John Kubiatowicz, Arvin Faruque, et al. 2013. QuRE: The Quantum Resource Estimator Toolbox. In *IEEE 31st International Conference on Computer Design (ICCD)*. IEEE, 419–426. https://doi.org/10.1109/ICCD.2013.6657074
[74] Krysta M Svore, Alfred V Aho, Andrew W Cross, Isaac Chuang, et al. 2006. A Layered Software Architecture for Quantum Computing Design Tools. *Computer* 39, 1 (2006), 74–83. https://doi.org/10.1109/MC.2006.4
[75] Swamit S Tannu and Moinuddin K Qureshi. 2019. 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*. 987–999.
[76] Shiou-An Wang, Chin-Yung Lu, I-Ming Tsai, and Sy-Yen Kuo. 2008. An XQDD-Based Verification Method for Quantum Circuits. *IEICE transactions on fundamentals of electronics, communications and computer sciences* 91, 2 (2008), 584–594.
[77] Brent R Waters, Dirk Balfanz, Glenn Durfee, and Diana K Smetters. 2004. Building an Encrypted and Searchable Audit Log. In *NDSS*, Vol. 4. Citeseer, 5–6.
[78] Benjamin Weder, Uwe Breitenbücher, Kálmán Képes, Frank Leymann, and Michael Zimmermann. 2020. Deployable Self-contained Workflow Models. In *Proceedings of the 8th European Conference on Service-Oriented and Cloud Computing (ESOCC)*. Springer, 85–96. https://doi.org/10.1007/978-3-030-44769-4_7
[79] Michael Wurster, Uwe Breitenbücher, Michael Falkenthal, Christoph Krieger, et al. 2019. The Essential Deployment Metamodel: A Systematic Review of Deployment Automation Technologies. *Software-Intensive Cyber-Physical Systems* (2019).
[80] Jianjun Zhao. 2020. Quantum Software Engineering: Landscapes and Horizons. *arXiv preprint arXiv:2007.07047* (2020).

All links were last followed on October 12, 2020.