Otso Kinanen

Implementing Software Containers in DevOps Practices for Quantum Computing

Master's Thesis in Information Technology

July 29, 2024

University of Jyväskylä

Faculty of Information Technology

Author: Otso Kinanen

Contact information: Otso.Kinanen@jyu.fi

Supervisors: Stribu, Vlad, and Haghparast, Majid

Title: Implementing Software Containers in DevOps Practices for Quantum Computing

Työn nimi: Konttiteknologiat DevOps menetelmissä Kvanttiohjelmistokehityksessä

Project: Master's Thesis

Study line: Mathematical Information Technology, Software and Telecommunication Technology

Page count: 65+0

Abstract: Quantum computing is an alternative computing paradigm gaining popularity in research, investment, and publicity. Quantum computers could provide exponential speedups for certain problems that are very hard to solve with current classical algorithms. As the development of quantum computing hardware has been accelerating, the need for specialized quantum software engineering is on the rise. This thesis researches the use of containerized application and demonstrate usecases with quantum hardware execution and with the use of kubernetes cluster as a application executing quantum code.

Keywords: Quantum programming, Quantum software engineering, DevOps, Container technologies, Containers, Kubernetes

Suomenkielinen tiivistelmä: Kvanttilaskenta ja -tietokoneet ovat saaneet lähivuosina paljon julkisuutta, keräävät lisääntyvissä määrin rahoitusta ja tutkimus on hyvin aktiivista. Teorissa kvanttitietokoneella voidaan nopeuttaa huomattavasti tiettyjen, perinteiselle tietokoneelle laskennallisesti vaikeiden ongelmien ratkaisemista. Kehitys kvanttikoneiden ympärillä on ollut nopeaa, josta johtuen myös kvanttisovellusten kehitysmenetelmät vaativat huomiota. Tässä työssä selvitän konttiteknologioiden käyttöä kvanttisovelluskehityksessä.

Avainsanat: Kvanttiohjelmointi, Kvanttiohjelmistosuunnittelu, DevOps, Konttiteknologiat, Ohjelmistokontit, Kubernetes

Glossary

Quantum computer	Computer based in quantum mechanical effects.
Qubit	Smallest basic unit used in quantum computing
HPC	High performance computing
CPU	Central processing unit
GPU	Graphics processing unit
QPU	Quantum processing unit
QC	Quantum computer or Quantum Computing
NISQ	Noisy intermediate-scale quantum
DevOps	Methodology combining used in software development combining processes from development (Dev) and operations (Ops)
DS	Design science

List of Figures

List of Tables

Contents

1 Introduction

In this thesis, we are researching the possibilities of applying software containers to quantum computing and quantum software development. Software container technologies play an important role in modern classical development methods, such as DevOps and CI/CD methods. This work introduces the key concepts of quantum computing, classical DevOps, and the technologies used in our solutions, which demonstrate the possible use of containers in quantum development.

Quantum computing is a computational paradigm based on quantum mechanical effects, like superposition and entanglement. Leveraging these effects, quantum computers may offer advantages by allowing parallelism in computation and allowing the different types of algorithm designs to be implemented in classically hard problems where they are hoped for, and has shown promise to outperform classical algorithms. Quantum computing may be simulated with a classical computer, but it seems to be impossible to do it efficiently on a large scale (Nielsen and Chuang 2010, p. 5-7).

The algorithm design for quantum computer is complex, and for an algorithm to be considered useful it should also be better than it's classical counterparts (Nielsen and Chuang 2010, p. 9). Currently, the best known quantum algorithms are being applied to and developed for cryptography, sorting, and optimization problems. The computational advantage has shown promise to bring advantage in domains, such as chemistry, medical and drug research, energy and storing, and basically in all other computational sciences (Preskill 2023; Gill et al. 2022). Among these fields of science, likely one of the most important future application for quantum computer will remain to be simulating quantum mechanical systems, beyond the scope of classical computers capabilities (Nielsen and Chuang 2010, p. 9).

To have an understanding of what should be considered to be included in Quantum DevOps methods, we'll have a look into classical definition of DevOps methodology and tools. DevOps is a methodology of combining previously separate parts of software development process, development, and operations, but goes further as it is often considered to include certain work ways and tools to be used (Ebert et al. 2016).

2 Methodology

2.1 Structure

This thesis is structured as following:

Introduction chapter introduces this work's subject briefly. Then following chapter Methodology describes, the research question and the basis considering current situation considering DevOps in quantum software development and states the reasons for need of clear methodologies in Quantum development, explains the limitations affecting and the use of literature in this work, and introduces the Design Science Research methodology used in this work followed by shortly describing storing and accessing of the designed and implemented artefact.

The third chapter, Background, gives a more detailed introduction to quantum computing and quantum software development, DevOps methodologies in classical computing and a more detailed explanation of software containers as part of the software development process.

From there we go through the theoretical background for Quantum DevOps and explain the approach taken in this thesis to create the solution to enable use of containers in scope of quantum software development, followed by the detailed description of the artefact and the technologies used in the implementation. Finally, in the last chapters we evaluate the artefact, and it's use against the research questions and objectives, followed by conclusions.

2.2 Research question

While quantum computing technology is still taking first steps and yet it hasn't been proven to provide any computational advantage with current hardware, there are some suggestions in the research how to implement software development methodologies for quantum computing. In these suggested models there are still left open the actual technical implementation, into which we are taking a dive in here.

A research paper *Full-stack quantum software in practice: ecosystem, stakeholders and chal-*

lenges proposes a model for software development life cycle for quantum full-stack, presented in Figure 1. This model adds two phases to the software development cycle enabling addition of quantum parts to a quantum hybrid software and to develop a full stack program (Stirbu et al. 2023).

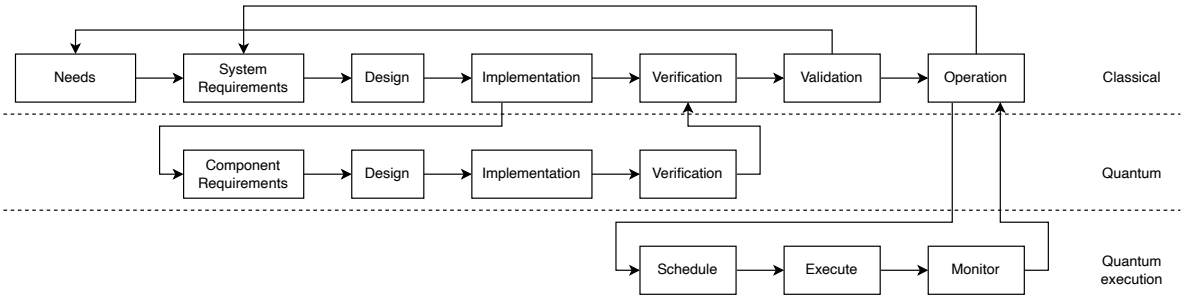


Figure 1. Software development lifecycle of a quantum hybrid system (Stirbu et al. 2023)

A similar model to describe quantum development workflows is presented in Quantum Software Development Lifecycle by Weder et al. Going into more detail the paper introduces inner cycles for quantum workflow including steps: *deployment* and *observability* and for quantum circuit development including steps: *hardware independent implementation*, *testing and verification*, *quantum hardware selection* and *execution* (Weder et al. 2020).

To implement the DevOps with deployment as suggested by these development models, there will is a need to have suitable tools to deploy and execute the quantum parts of these programs. In this thesis I am researching possible use of containerised apps for quantum development.

Research question : How may a container technology be implemented in quantum software development to work as enabler in Quantum DevOps?

2.3 Literature

Literature for this thesis has been acquired using databases and search engines Google Scholar, JYKDOK-Finna, IEEExlore and Google. Literature used consists from books, scientific articles, blog posts, company and organisation websites and software documentation. The references have been chosen by weighing their relevance in multiple factors: recentness,

recognition in the field, publisher's reputation and in other than articles or books, company's or organisation's reliability. In theory chapters Quantum computers and computation the references are aimed to be chosen, if possible to refer to establishing publications and original sources referring to theories in question.

2.4 Limitations

Limiting factor considering this thesis are related to the access to current QC systems. Quantum computing hardware is still quite limited resource and it is not easily accessible as classical local computer that any developer has and can play with without restrictions, or even classical high performance computer which are offered by many vendors in different shapes and sizes. We later discuss in more detail the quantum hardware vendors and their accessibility, development environments and pricing, which are the limiting factors as well as partial motivation for this thesis. One major restricting factor in current quantum hardware is that the systems are closed in multiple ways, they are not offering open access points, or explain details of how the systems software stacks are built inside. This makes it hard to compare the solution presented in this thesis to the current commercial and publicly available systems by the vendor providing quantum computing platforms or development environments.

2.5 Design science research method

This thesis follows the Design Science Research method by Peffers et al. (Peffers et al. 2007) to address the need of research in DevOps for quantum computing, set objectives for a solution designed as part of this thesis, demonstrate the solution and evaluate it against the objectives. The method has a framework for a 6 part process, explained in figure 2 and below.

1. Problem identification and motivation. In this thesis, we are developing a solution to provide the benefits of containers to the reach of quantum developers. Container technologies offer benefits in classical development, in both development process and in the execution of production level code. The characteristics of containers could be suitable for many occasions in quantum development to improve the software develop-

ment offering easier development time executions and operations by streamlining the deployment process. Research question introduced in section 2.2 Research question with more detail.

2. Objectives of a solution. The containers are created to improve portability and help deploy code more easily, with the solution developed here the aim is to bring these benefits to Quantum development. Chapter 4 explains more closely what may be achieved by implying DevOps to Quantum software development and how the developed artefact is helping the progress.
3. Design and development. Develop a containerized application and container image to transfer and execute quantum code on different hardware. More on the implementation and usage in chapter 5 Solution
4. Demonstration. We demonstrate the solution by executing the containerized in quantum hardware, and using the created container as part of Jupyter notebook backend. Demonstrations described in the section 5.4 Demonstrations scenarios.
5. Evaluation. This will involve comparing the characteristics of the DS research process model with the objectives described above. The evaluation of the artefact is done with a framework introduced by Hevner et al in DS in Information Systems Research (Hevner et al. 2004).
6. Communication. The communication is this thesis, which explains the need and implementation of the designed artefact in detail.

2.6 Storing the artefact

The project's source code will be stored in Github under the TORQS (Towards Reliable Quantum Software Development: Approaches and Use Cases) project's account as public repository. TORQS is a research project by University of Jyväskylä and university of Oulu. Examples and key components in suitable parts presented in thesis. https://github.com/torqs-project

Docker image files will be stored in a public repository in DockerHub under the author's personal account, and examples in suitable parts presented in this thesis. https://hub.docker.com/repository/d

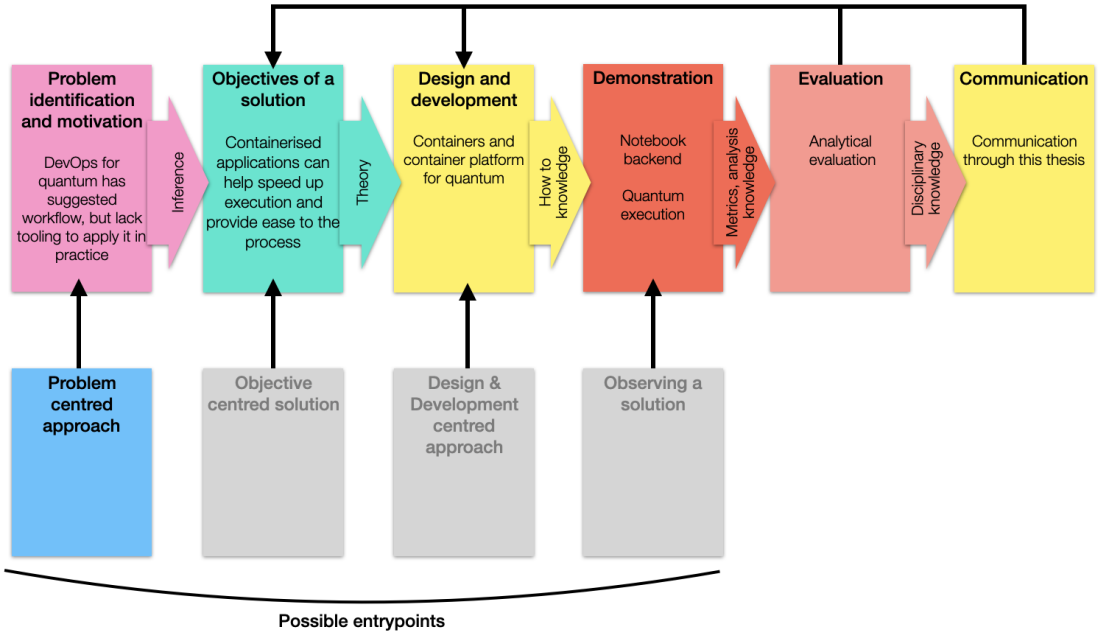


Figure 2. DSRP method presented in sequence diagram, adapted from A Design Science Research Methodology for Information Systems Research by Peffers et al. (Peffers et al. 2007)

3 Background

3.1 Quantum computing fundamentals

Quantum computer is a computational device of which computation is based on quantum logic and quantum bits instead of I/O bits and Boolean logic as a classical computer (Vedral and Plenio 1998). The first, author to introduce the idea of quantum computing was Richard Feynman in 1981 in a conference speech "Simulating Physics with Computers" pointing out the limits of current computing methods when working with quantum mechanics (Preskill 2023). In the speech Feynman suggested that if we aim to simulate probabilities to understand and predict quantum mechanics we are not able to do it with our current machines, or by a machine using similar technology, as the demands on predicting probabilities will rise exponentially. In his speech, he rather suggested that we simulate quantum mechanics by using a quantum computer (Feynman et al. 2018). Others simultaneously but independently suggesting theories of possibilities of using quantum mechanics in computing were mathematician Yuri Manin and physicists Paul Benioff (Preskill 2023).

From there on, the often mentioned steps towards the current state of quantum computing and quantum programming are the findings of useful quantum algorithms outside the research of quantum mechanics, of which most well known, being Shor's factorisation algorithm and Grover's search algorithm. Simple example of Grover's in figure 3 where the algorithm is presented in Qiskit code (Steane 1998; Preskill 2023; Rieffel and Polak 2000). Both these algorithms demonstrated superiority of quantum over classical computing in theory (Steane 1998). The algorithm introduced by Peter Shor in 1994 would be able to factorize prime numbers in a way and in time that would not be achievable by any classical computer and Grover's search algorithm 1996 which enables quantum computer to find item from dataset in $O(\sqrt{n})$ time where n is the size of given dataset (Ugwuishiwu et al. 2020; Giri and Korepin 2017). The simple quantum circuit presentation based on Grover's algorithm is presented in Figure 3.

Now, four decades later, we have several potential technologies to implement quantum mechanics and to build a working quantum computer. These technologies are based on quantum

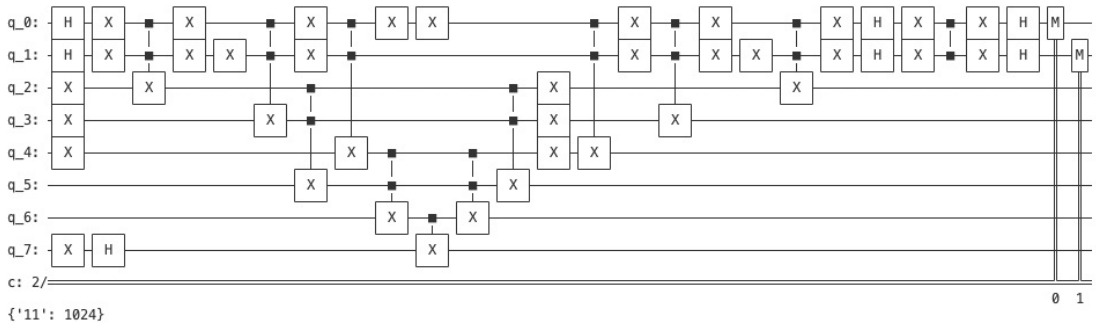


Figure 3. Grover's algorithm by Qiskit (Qiskit contributors 2023)

systems of trapped ions, photons, silicon and superconducting qubits. Of which the superconducting considered to have been showing the most potential so far and has seen the much progress since its first introduction as an option to act as a qubit in a quantum computer in 1999 (Huang et al. 2020). To describe current and near future quantum computers, John Preskill has introduced us a term Noisy Intermediate-Scale Quantum (NISQ). In his definition, this covers machines from 50 qubit to few hundreds qubits (Preskill August 2018).

3.1.1 Quantum bits

In a classical computing, the data is being divided into information units up to a single bit, which can hold the value of 0 or 1. In quantum computer, the smallest unit of information is a qubit. A qubit is the term used for both the physical quantum bit used in quantum computer and the computational abstraction of a quantum bit, and regardless of the physical implementation of the system it is based on (Preskill 2023). While in classical computation the bit's state is exactly one of two possibilities, qubit's state may not always be definite but more of a probabilistic, and in some states the value of a qubit may be considered to be in several states in same time, based on a phenomena called superposition. Because of this qubit's unique nature the state of a qubit is often presented mathematically as vector in complex Hilbert space (David P DiVincenzo 1995; Preskill 2023).

Even as a qubit may hold a value other than 0 or 1 it may only be measured and read in these exact states 0 and 1 like a classical computer. Before measurement, qubit may be set to superposition, and that state can be manipulated by quantum gates to effect probability of outcome when the qubit's value is read.

A qubit state is often presented as a vector, with ket notation $|0\rangle$ and $|1\rangle$ and the superposition of the qubit with the same notation is often presented as $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ where α and β are complex numbers. Which may be presented also as $z = x + iy$ and $i = \sqrt{-1}$, where z is presenting a complex number,x and y, real numbers and i an imaginary number .(McMahon 2007).

When one computational qubit offers us measurable states 0 and 1, the number of possible states grow exponentially when adding qubits to the system. A system of n qubits has a state space of 2^n (Rieffel and Polak 2000). As where on qubit may be measured to be in the state of $|0\rangle$ or $|1\rangle$ a four qubit system may be measured in any of the following 16 states $|0000\rangle$, $|0001\rangle$, $|0010\rangle$, $|0011\rangle$, $|0100\rangle$, $|0101\rangle$, $|0110\rangle$, $|0111\rangle$, $|1000\rangle$, $|1001\rangle$, $|1010\rangle$, $|1011\rangle$, $|1100\rangle$, $|1101\rangle$, $|1110\rangle$ or $|1111\rangle$.

Physical qubit implementations To implement a quantum computer, theoretical physicist David DiVincenzio has created a 5 condition list, known as DiVincenzo's criteria (David P. DiVincenzo September 2000). The criteria state that a quantum computer must have:

- A scalable physical system, with well characterized qubits
- The ability to initialize the state of the qubits to a simple fiducial state, such as $|000..\rangle$
- Long relevant decoherence times, much longer than the gate operation time
- A "universal" set of gates
- A qubit-specific measurement capability

Following these criteria, there are currently several competing technologies for a quantum computer architecture and qubit implementation. None of the technologies has passed NISQ era, or may be declared as winner of the race. And even within the different approaches there is variation in implementation by different manufacturers, for example superconducting qubits has been implemented, with Cooper-pair or charge qubit and flux qubit on superconducting circuit (Kjaergaard et al. 2020).

Sometimes may be talked about logical qubits, which refer to a qubit that works perfectly, with no errors common with current hardware. With current quantum computers, we may use several physical noisy qubits to build one logical qubit (Jones et al. 2018).

Superconducting qubits are currently considered to be the leading candidate as a choice of qubit implementation in large scale general quantum computer. It is used by many providers and for example in Google's Sycamore processor used in their quantum supremacy proving experiment. Current devices have reached gate coherence times in the order of tens of nanoseconds, and in most advanced devices the gate fidelity is reaching 99,5% - 99,9%. Superconducting qubits must be operated in very low temperatures and lose coherence when temperatures rise around the QPU.

Trapped ion, is one candidate for a qubit. Based on the use of electromagnetic fields and laser cooling, controlling the positions of ions. It was suggested soon after the finding of Shor's algorithm in 1995. Trapped ion technology fulfils all DiVincenzo Criteria, but has problems when scaling the number of qubits up.

Photonic qubits. The use of photons as well as every other qubit has ups and downs, they are not easily reactive, so they may be more coherent than other options for qubit, but on the other side this makes it harder to have several photonic qubits to interact in multi qubit gates. As well as trapped ion, photonic quantum computers have the abilities needed for quantum computing, they are extremely hard to scale up.
(Lau et al. 2022)

Among these suitable candidates, Intel and SemiQon are developing hardware based on semiconducting spin qubits, with manufacturing process closer to the classical semiconducting silicon processors (Intel Corporation 2023). These examples are currently most noticeable but does not cover all the devices in currently in development.

3.1.2 Quantum gates

In quantum computing, the equivalent to logic ports are quantum logic gates, also often referred to as quantum gates or simply gates. Quantum gates are used to interact and manipulate the qubits state, to perform computations with them. In difference to classical logic gates, all quantum gates are reversible. One of the DiVincenzio's criteria for a quantum computer, as mentioned, is *a universal set of gates*, with which any quantum operation or circuit may be approximated (David P DiVincenzo 1995). A commonly implemented set is called

The Clifford set, and may be formed as following

One qubit gates

- Hadamard's - creating a super position
- S gate - a phase gate

The S and H gates may be combined in different ways to create a Pauli group, which is formed as

- X - Flips state, equivalent to classical NOT gate, turns $|0\rangle$ into $|1\rangle$ and $|1\rangle$ to $|0\rangle$
- Y - Flips $|0\rangle$ into $i|1\rangle$, and $|1\rangle$ into $-i|0\rangle$
- Z - Flips phase, does not affect if assigned to $|0\rangle$ and turns $|1\rangle$ into $-|1\rangle$

Two qubit gate

- Controlled Not, or C Not, assigned to two qubits and works as X gate for other qubit, but only if the reference gate is in state 1.

3.1.3 Current state of quantum hardware

As one of the most notable milestone considering current state of quantum computing may be considered the Google's experiment on Sycamore processor, creating quantum state on 53 superconducting qubits and performing a computational task in 200 seconds, which allegedly would have taken 10.000 years with current state of the art classical supercomputer (Arute et al. 2019). Shortly after that, IBM – provider of the classical computer used in the comparison – claimed that Googles numbers are highly exaggerated and claimed a theory that the calculations on classical super computer would not take 10.000 years, but approximately 2 and half days (Pednault et al. 2019). This is just one experiment on the way to quantum supremacy with still long way to go for current hardware. Either way it is fair to say that the quantum computers have already reached some real milestones towards it.

Preskill states that 50 qubits barrier to be a significant step as we are entering area of which is beyond the power of current classical supercomputers simulations (Preskill August 2018). Companies such as IBM, are pushing out larger machines, with 433 qubits available at 2022

and jumping to 1121 qubits at 2023 (IBM 2023), but even as quantum chips are having more qubits available noise and variable quality of qubits remains to be a problem in current systems and in near future (Byrd and Ding 2023). To reach the quantum advantage in an for an actual problem, with known algorithms we will need to increase the reliability of qubits and the number of them by a lot. For an example of the magnitude often mentioned use case for quantum computer is breaking the RSA encryption using Shor's factoring algorithm. To perform this to a 2048-bit number, it is estimated that we would need 10 thousand logical qubits or 10 million physical qubits (Preskill 2012).

The difference between the mentioned number of necessary logical qubits and physical qubits is explained by the noise and errors in the current hardware. In the current state of quantum computing, noise is an important factor to consider and part of the very nature of hardware currently available (Resch and Karpuzcu 2021).

3.2 Quantum software engineering

3.2.1 Computing model/implementation

Current quantum systems are mostly based on batch model computing, where the quantum task is part of a classical program. In this structure the classical part is in control of scheduling, and communication between different services and the quantum part is running a quantum algorithm taking advantage of the different mechanism in computation executed in the QPU (Stirbu et al. 2023).

In all current quantum computers, by any provider, there is a classical computer working as a control plane for the quantum hardware. Using the batch model may cause some delays and demands for the program architecture, when executions may end up queuing rather than executing in real-time. In current full stack ecosystems this is done under the hood, but in smaller providers' hardware there is a need for an effective and capable orchestration platform with efficient scheduler and workload manager (Stirbu et al. 2023; Gill et al. 2022).

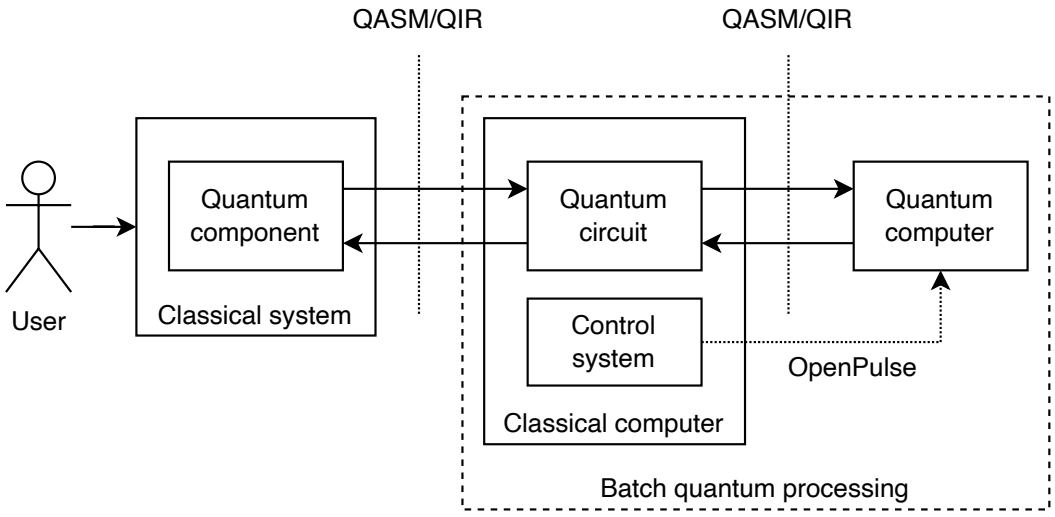


Figure 4. Components and interfaces of quantum computers; adapted from (Gill et al. 2022)

3.2.2 Current state of quantum computing ecosystems

There are currently several vendors providing software development kits for quantum computing. To have some view on the state of the art at 2024 we have a look at the most used and well known tools provided by the leading commercial providers and currently otherwise notable in the field. To be noted is that, the landscape is changing fast and info presented in this chapter may change quickly, as organisations are presenting new software and hardware.

Qiskit is a toolkit and python library for quantum computing. It is developed and managed by IBM and used in their quantum cloud platform offering access to IBM's hardware, 17 quantum computers with qubits ranging from 27 to 133 currently publicly available, among several simulator backends. Qiskit is also supported by several other providers of hardware in multiple different architecture (IMB Quantum 2024a). Qiskit toolkit is used by a majority, 68,8% of quantum developers. IBM quantum platform includes a wide selection of learning material for quantum development and for qiskit library (IMB Quantum 2024b). Executing quantum code on IBM's cloud with quantum hardware is priced in a pay-as-you-go plan for $1,60 USD per second [1]. For their QC ecosystem, IBM offers also a more recent service called Quantum serverless to provide platform to use their quantum hardware for hybrid

1. https://www.ibm.com/quantum/pricing

and full stack quantum computing. [2]. With integrated quantum-classical cloud service the performance is improving, as the time of queuing is cut and the computation is based on integrated model rather than batch, as it would if sending quantum job to IBM quantum service from external system.

Qiskit is designed to accommodate different type of Quantum Computers in NISQ era, **algorithm designers** researching and developing applications leveraging quantum computing, **circuit designers** optimizing the circuits for a QC and exploring its properties like error correction or verification and validation, and **quantum physicists** to research and optimize gates, with precise control and ability to explore noise, apply dynamical decoupling and perform optimized control theory (McKay et al. 2018).

Qiskit is an open source project and currently offers dozens of additional libraries, plugins, simulator backends, application packages for multiple domains such as machine learning, physics, chemistry, and finance and other related projects available. In Qiskit there is also several transpiler plugins available for users to optimize and interact with the transpiling process (IMB Quantum 2024c). Among Qiskit IBM offers OpenQAMS and OpenPulse. OpenQAMS is an imperative language which main purpose is to act as intermediate representation for high-level compilers for QC hardware. It offers precise control over gates, measurement, and conditionals (Ajallooiean et al. 2024; McKay et al. 2018). OpenPulse is a specification for pulse-level control, for general purpose QC, and it is designed to be hardware architecture agnostic (McKay et al. 2018).

Azure Quantum, Q# and QDK. Azure quantum is a quantum cloud computing service and a part of azure cloud. It offers a development environment with a hardware access to several vendors quantum machinery. Currently, available are vendors like IonQ, Pascal, Quantinuum and Rigetti. Microsoft has research and development going on for their own quantum computer. For an example of current pricing on Azure Quantum, executing on PASQAL Fresnel1 QPU is priced $3000 USD/QPU hour.[3]. Azure quantum offers a learning environment for quantum computing and Q# programming language in Azure Quantum has specialized copilot AI in the learning environment. Q# is a stand-alone programming language, unlike the

2. https://www.ibm.com/quantum/blog/quantum-serverless-programmin
3. https://learn.microsoft.com/en-us/azure/quantum/pricing

other most well quantum programming libraries. Like Qiskit, Q# is open source (Microsoft 2024). In relatively similar manner other cloud giants Google and Amazon are offering currently quantum sdk's and access to 3rd party quantum hardware through their cloud services, **Cirq** python library and Google Ai cloud platform, and **Amazon Braket** cloud platform of which architecture may be seen in figure 5. Currently, none of these three companies are offering public access to their own quantum hardware (Google Quantum AI 2024; Amazon Web Services Inc. 2023).

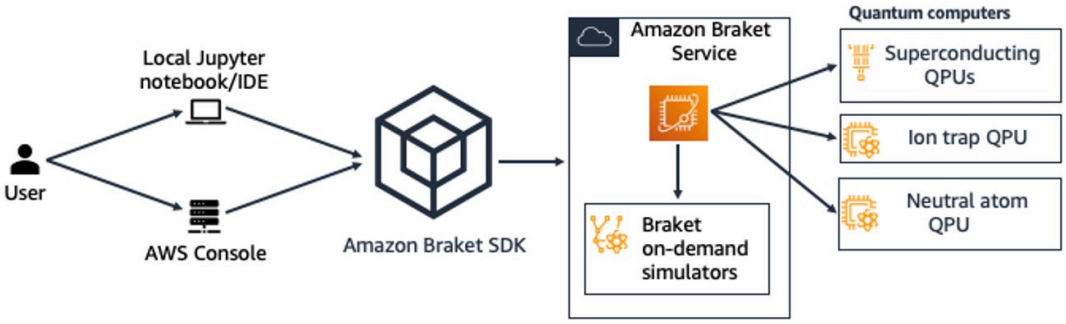


Figure 5. Presentation of Amazon Braket cloud service architecture at December 2023 from (Amazon Web Services Inc. 2023).

Pennylane is a Python library focusing in machine learning for quantum computing by enabling the use of popular, commonly used classical machine learning frameworks. PennyLane is designed to support various execution with variable QC simulators and actual QC hardware, handling the communication with device and compiling the circuits (Bergholm et al. 2022). Like Qiskit and Q#, PennyLane is open source. For PennyLane, there is Lightining simulator backend enabling execution in HPCs with multithread CPUs, different multithreaded GPUs with AMD ROCm, NVIDIA CUDA and MPI-distributed environments (Asadi et al. 2024).

Cuda and cuQuantum. To enable use of their GPUs for quantum, Nvidia has a specialized platform to run quantum simulators in their hardware. Nvidia CUDA is a computing platform developed for GPUs, for computationally demanding tasks suitable for parallel computing.

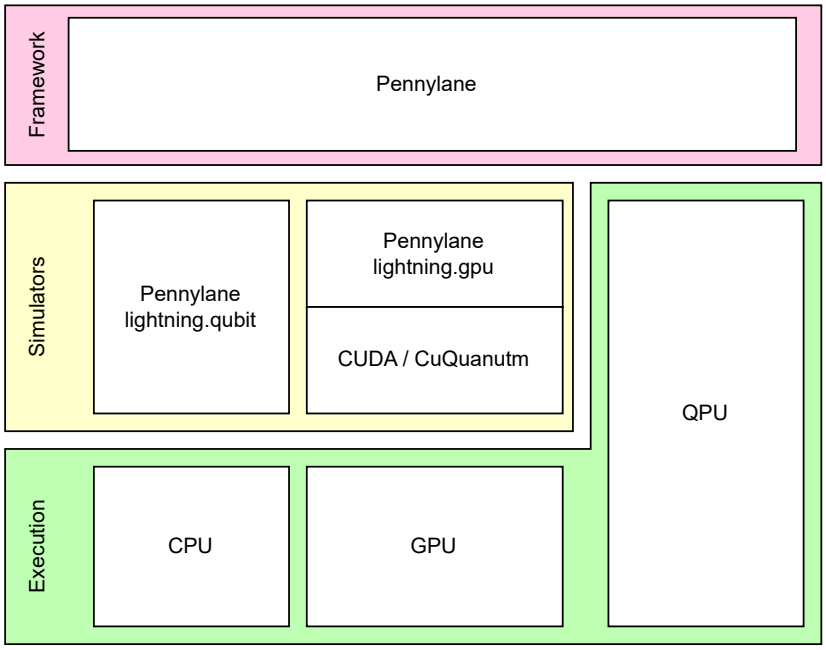


Figure 6. One example of execution stack using PennyLane framework and it's lightning qubit plugins for different target backends. These backend plugins enables the code to be executed in HPC clusters with CPUs or GPUs

cuQuantum is an SDK based on CUDA, offering libraries for Quantum computing, with two libraries, cuStateVec for state vector computation and cuTensorNet for tensor network computation. CuStateVec is used by gate-based general quantum computer simulators, providing measurement, gate application, expectation value, sampler, and state vector movement. CuStateVec library is available for Cuda versions from 11 and 12. Nvidia cuQuantum is used by Cirq, PennyLane Lightning and Qiskit Aer, for their GPU-powered quantum simulator backends (Nvidia 2024). Example of the relations of these software packages presented as a stack in figure 6.

As a bit simplified conclusion on software development kits and development tools for quantum could be said that most of the wide spread tools are currently offered by hardware providers like IBM and Xanadu to provide users a complete stack of software and algorithm development platform and hardware to use their solutions, and on the other hand by cloud computing giants like Amazon, Microsoft, and Google offering variety in hardware selection by several providers like Rigetti or IonQ, powerful cloud-based simulators and tight

integration to their classical cloud platforms. These structures offer easy way of building applications but may easily leads to a vendor lock-in situation for the developer.

3.2.3 Simulating quantum computers on classical computers

The development of the quantum computing, as introduced in the section3.1 started out of the necessity of different computing paradigm when interacting with quantum objects, as simulating quantum mechanics computational demands grow exponentially with classical computer. Still, atleast for now, as the current quantum computers are very prone to errors and the running costs of an actual quantum hardware is high, we are widely using simulators to run circuits to simulate the action of a quantum computer.

Even if the final program or the algorithm is eventually aimed to be executed on QPU the simulators is important part of testing the algorithms. When simulating a circuit there is no noise or other QC related unwanted errors. This makes it possible to examine the algorithms actions and to ensure it works as intended. Simulator also enable debugging the quantum code with step by step execution, which is impossible with actual quantum hardware. For many simulators there are noise models available, simulating the errors and noise encountered in quantum hardware to provide better understanding of the results that may be expected if moved to actual QC (Gheorghe-Pop et al. 2020).

GPUs are offering a great advance in quantum simulation over CPUs. In quantum simulation, computations may be executed in parallel up to thousands of threads, providing the computational advancement of GPUs over CPUs. In a high performance computer (HPC), such as a super computer or a computer cluster with multiple processing units, simulations may reach the levels of up to 40 qubits circuits with reasonable execution times (Willsch et al. 2022) and especially HPC clusters with GPUs provide advantage not only by expanding the reach of simulations but also by lowering the execution times significantly. In Nvidia experiments, the GPU speed-ups over CPUs were in the scope of 50–90 times.(Morino, Hehn, and Fang 2024) With these speed-ups in execution, and considering the high cost of current quantum hardware, GPUs offer a very useful platform for quantum algorithm development, at least for the NISQ era. In listing 3.1 the code sample presents how in PennyLane

library you choose their GPU simulator, `dev = qml.device('lightning.gpu', wires=wires)` backend for improved performance.

3.3 DevOps and continuous software development methods

3.3.1 Continuous integration

Continuous Integration, CI, has been argued not to have one single explicit definition (Soares et al. 2022). In one common view, continuous integration methods are being considered being applied when a developer integrates his code at least daily, leading up to several integrations daily in software projects worked by several developers (Fowler and Foemmel 2006; Stahl, Martensson, and Bosch 2017). With more frequent builds, test runs and integrations, the development process aims to reach more fluent integration process and time savings with CI automation (Shahin, Ali Babar, and Zhu 2017).

Now to look a bit more closely at the process and key practices to implement CI by Martin Fowler as presented in Continuous Integration. The workflow of **building a feature** described shortly. You start with taking a copy of the latest version of the software under work from the version control system, for example do a `pull` from a git repository. From there you do the edits, or additions to the code, and finish the task, including adding or altering the tests for the edited part. Then you do the local build, compile the program and run the tests. After finishing and having successful build and test runs, you pull a new version from the version control, as some others might have altered some parts clashing your edits. When done with correcting possible fixes to the latest version, do the local build again and run the tests. Then commit the code to the version control and integrate new code to the project.

The headline principalities to follow when practising continuous integration by Fowler, *maintain a single source repository*, ensuring that every change is done into same source code, *automate the build*, ensuring the build is done similarly every time, *make your build self-testing*, with covering automated test suite the testing is simplified and may be standardised for the life cycle of the application, *everyone commits to the mainline every day*, making committed edits to remain small enough to trace and to understand, *keep the build fast*, to provide feedback immediately to the developer, enabling to build and commit constantly, *test*

Listing 3.1. Code by PennyLane Team to run a 20 wire circuit with their GPU powered simulator as a backend (O'Riordan and Team 2022).

```
import pennylane as qml
from timeit import default_timer as timer

wires = 20
layers = 3
num_runs = 5
# Instantiate CPU (lightning.qubit) or GPU (lightning.gpu) device
dev = qml.device('lightning.gpu', wires=wires)

# Create QNode of device and circuit
@qml.qnode(dev, diff_method="adjoint")
def circuit(parameters):
    qml.StronglyEntanglingLayers(weights=parameters,
        wires=range(wires))
    return [qml.expval(qml.PauliZ(i)) for i in range(wires)]
    # For PL >= v0.30.0 the following return should be used
    # return qml.math.hstack([qml.expval(qml.PauliZ(i)) for i in
        range(wires)])
# Set trainable parameters for calculating circuit Jacobian
shape = qml.StronglyEntanglingLayers.shape(n_layers=layers,
    n_wires=wires)
weights = qml.numpy.random.random(size=shape)
# Run, calculate the quantum circuit Jacobian and average the
    timing results
timing = []
for t in range(num_runs):
    start = timer()
    jac = qml.jacobian(circuit)(weights)
    end = timer()
    timing.append(end - start)
print(qml.numpy.mean(timing))
```

in a clone of the production environment, to have the best possible test environment before moving to production, *everyone can see what's happening*, by transparency the communication inside the team is easier, and *automate deployment*, which is already referring to the next sections subject. More into all of these points may be read from Continuous Integration by Martin Fowler (Fowler and Foemmel 2006).

When the suggested CI implementation process has been followed, the IEEE standard states that the following outcomes should be reached and demonstrable. The process should include automated build, packaging, and security, for both software and the used hardware elements, and the system should have automated notifications on any system change, to improve communication and support the organisation's decision-making (IEEE 2021).

3.3.2 Continuous delivery and Continuous deployment

Term Continuous Deployment was first introduced in 2009 as a blog post by Timothy Fitz, while Term Continuous Delivery was introduced in a book going by the same name in 2010 by Jez Humble and David Farley (Humble 2010; Fitz 2009). While the terms are fairly similar and both may be referred to as CD occasionally, they have an important and clear difference, as the terms imply. When in, continuous deployment has a pipeline through automated tests all the way to the production and to the end user, as continuous delivery leaves the last step to b nn nnnnnnh hhhhe manually applied and the decision to the business (Red Hat 2022). In short should be also said that continuous deployment includes all the steps of continuous delivery but not the other way around (Shahin, Ali Babar, and Zhu 2017). As the methods share the vast majority of common phases and methods, we shall go through them. Workflow visualized in figure 7.

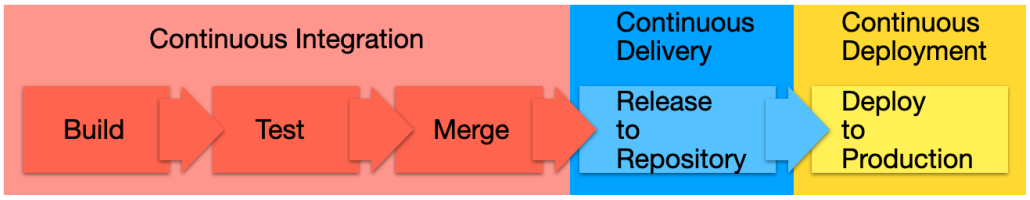


Figure 7. Continuous Integration, Continuous Delivery and Continuous Deployment by Redhat (Red Hat 2022)

Now we go through the deployment pipeline and its important factors in continuous delivery as presented by Humble and Farley in Continuous delivery: reliable software releases through build, test, and deployment automation. **Build Binaries Once**, compile and build the executable files only once, use the same exact build in tests that is on the way to delivery, or the other way around take the exact build that passed the tests when deploying into production environment. When applied correctly this helps and allows you to have correct configurations, do the code separation for different environments and helps structure the build system. **Deploy the Same Way to Every Environment** as the program may be run on multiple platforms in multiple phases, it should be always be deployed the same. As running environments are complex, there will always be differences that may affect the outcome if not taken in consideration. **Smoke-test Your Deployments** to test that the deployment runs, smoke-test may be as simple as starting the main screen and checking it has the correct view and runs, but it is crucial to have to guarantee the application runs. **Deploy into a Copy of Production** to be as sure as possible that the program runs with the data and surroundings it is meant to run. **Each Change Should Propagate through the Pipeline Instantly** as each change is run through separately, is it clear where the possible failures seed from. And finally **If Any Part of the Pipeline Fails, stop the Line** (Humble and Farley 2010). In our take Containers are answering to many key elements here, and I will go through them hem in section Containers.

To revisit terms continuous deployment and continuous delivery in this thesis context. Advantages in continuous deployment is implied to be reached in example in a situation where fatal code passes the automated tests, and is deployed into the production. In these situations, the source of the failure can be retraced quickly to the latest deployment and recovered easily. When deploying manually, you may end up with several changes in the same deployment, and the error tracing may turn to be more difficult (Fitz 2009). Opposing that in continuous delivery method, the deployment is considered to be a business decision and should not be automated. It also gives more control over having different strategies considering releases and versions of given software. Important is to still understand that in continuous delivery, the process aims to end in fully tested and deployable software (Humble 2010).

Choosing the strategy between continuous delivery and continuous deployment, may be

based on many reasons, differing between tools available, chosen or forced testing methods and business decisions (Shahin et al. 2017). When considering the current state of quantum hardware, varying executing surroundings and nature of common quantum or quantum hybrid software in this thesis we will not aim to find solutions to imply automatic deployment to be included in Quantum DevOps.

3.3.3 DevOps in classical computing

The foundation for DevOps lays in Agile manifesto and Agile development methods, and in Lean methods before that. It is even described in The DevOps Handbook: How to create world-class Agility, Reliability, and Security in Technology to have roots in "Lean, Theory of Constraints, the Toyota Production System, resilience engineering, learning organisations, safety culture, human factors and many others" (Kim et al. 2016).

In the earlier years, while already gaining popularity, DevOps was found to lack an exact and definitive definition (Stahl, Martensson, and Bosch 2017). In 2016 systematic mapping of published research on DevOps, by Jabbari et al., showed that when defining DevOps most common definitions beside mentioning the combination of Development and Operations departments, were more precisely defining it to be a *paradigm or set of practices that enables communication and efficient teamwork between developers and operators*, *a paradigm bridging the cap between developers and operators* and *DevOps is a paradigm or set of principles focusing on software delivery by enabling continuous feedback, high responsivity to changes and using automation in delivery pipelines*, some papers in the mapping also included mentions that DevOps included enabling deployment automation directly from version control to production environment (Jabbari et al. 2016). Among these, often mentioned in DevOps literature are earlier methods continuous integration, continuous delivery, continuous deployment, continuous release and continuous testing, Agile, and DevOps Tools or Toolkit.

In 2021 IEEE published IEEE standard for DevOps: Building Reliable and Secure Systems Including Application Build, Package and Deployment. The standard offers precise requirements, definitions, and practices to implement DevOps and describes detailed process to build the organization and to manage the process of a project implying DevOps methods

(IEEE 2021).

The reason DevOps methods are applied is to gain benefits in the development process over more traditional methods, some examples found in study by Riungu-Kalliosaari et al. are *more implemented features and frequent releases, improved quality assurance, enhanced collaboration and communication, maximizing competences improved visibility of implemented features to the customer* and also enabled to *test with real customers and to be able to continuously experiment* (Riungu-Kalliosaari et al. 2016). The tools used to reach these benefits are very often considered to be a part of the DevOps themselves and the way of reaching beneficial results in DevOps, or to enable implementing the method as meant. In table 1, is listed and presented important tools and toolkits found in the survey by Leite et al. in 2019 (Leite et al. 2019).

In this thesis' context, DevOps for QC should take advantage of tools and automation in several stages of development process and deployment. Optimally to have automated tools to cover processes and stages in continuous integration, continuous delivery and continuous testing, and going beyond tools, to find ways to implement the methods from agile and other DevOps ways that are found advantageous in development processes.

3.3.4 Architecture

In DevOps and Continuous development methods, it is recommended to adopt microservice architecture rather than traditional monolithic (Leite et al. 2019). When designing services separately, each should aim towards *testable* and *deployable* design. Where testable aims to simple modules that may be tested locally by developer rather than being dependent on large ecosystems to test with, and deployable aims to have modules of the software to be independently and automatically deployed affecting as few other services of the software as possible (Humble). When done correctly, this is enabling a separate build, delivery, and deployment process for individual services rather than having to deploy the whole software monolith every time anything changes (Balalaie, Heydarnoori, and Jamshidi 2016).

Then the architectural definition has been followed successfully implemented, IEEE standard lists that the following outcomes should be reached and demonstrable. IEEE defines

that the architecture should be defined, to support the stakeholder's goals, and it should be described precisely and assessed throughout the solution. The architecture should take into consideration the security, and privacy by design, for both the organization and the applications users, follow and translate the legal requirements into technical requirements, and the architecture and its documentations should be available in accessible and consumable and form (IEEE 2021).

3.4 Testing in DevOps methodologies

Testing is defined as "... the process of executing a program with the intent of finding errors" by Myers in the Art of Software Testing (Myers et al. 2004). The reason software is tested is to validate, verify, improve quality and to estimate reliability of the software (Pan 1999). Without going much deeper in testing in general, may be added to these simplified definitions that, successful testing is meat to find errors that may be fixed, not to just run successfully (Myers et al. 2004).

To emphasize the importance of testing in continuous methods, the development pipeline is considered to be built so that each stage acts as a quality gate. When the development process advances to the next step of the process, the tests should ensure that the stage is ready and working as expected and not only for this stage but in all other stages of the pipeline as well, and it may go through the process to deployment or delivery (Chen 2015).

More related to DevOps and continuous methods, in mapping literature on testing in microservice architecture, By M. Wasseem et al., most commonly discussed testing approaches and tools were :**Unit testing** in which the smallest units of program are tested to function as wanted by themselves, and **Integration testing**, where the separate services and part of the program are tested together ensuring their wanted behaviour, both found effective methods in complex systems, but not sufficient to apply fully comprehensive testing by themselves. Beside these more than once were mentioned the following approaches: **Black Box Testing**, where the programs' functionality is examined excluding its inner structure, **Component Testing**, where components of the program are tested separately, **Contract Testing**, where programs APIs are tested to have implemented their requirements, **Consumer driven test-**

ing, where provider components are tested to offer services as intended, **System testing**, where the complete system is tested unified, also called **End-to-End Testing**, **Mocked Data Testing**, **Performance Testing** for example **Load Testing**, where the systems' ability to perform under load is tested, **Regression Testing**, where the program is tested after any alternation, during the whole lifetime of the application, and **White Box Testing**, where the structure of the application, the code, and documentation is examined, often statically(Waseem et al. 2020).

As mentioned multiple times so far in almost every step of the development process included in DevOps, the tests are crucial and should be implemented continuously, widely and thoroughly. While one notable and defining factor in testing DevOps and while using CI/CD methods is, that tests must be widely automated (Waseem et al. 2020).

This all considered, it may be said that testing in DevOps follows all the same principles as in software development in general, but automation in the process must be emphasized if following continuous development methods and DevOps. There are some major considerations in testing quantum programs, and more of that in chapter 4 Applying DevOps in quantum development.

3.5 Containers and container orchestration

Containers are an important building block in current systems, especially when based on microservice architecture and running in cloud infrastructure. Container are used to deploy programs on different platforms. In the deployment, containers are fast and offer high scalability. Some advantages of containers are not directly related to development process but more in the management of running software, such as fault tolerance and update ability (Kang, Le, and Tao 2016).

3.5.1 Containers

There are different approaches to virtualization, it may be done with virtual machines, which may be run either directly on hardware or in some solutions as an application on a host OS, or as we more closely here examine, with containers. Containers are running on top of the host

OS, using the host's services, but runs in isolation from the host OS or any other containers (Merkel et al. 2014). In the DevOps context, use of containers is used to deploy the software to the executing platform, may it be a cloud server platform, a local server or HPC cluster. Containers offer in many ways a one-stop-solution to deploy as instructed by DevOps. The container deployment is often automated using a CD pipeline with containers and container registries.

Images. A container image is a light standalone package of software. An image includes source code, runtime, system tools, library dependencies and for example environmental settings. An image becomes a container when executed (Pahl 2015). The implementation of container technology is based in kernel level Linux control group and namespace feature, that creates containers so that it has its own isolated *filesystem, PID, network, user, IPC, and hostname namespaces* as closely explained by Felter et al. (Felter et al. 2015). This provides the security for the host and the container by separating the access to database and restricts container's root user privileges outside container and vice versa (Merkel et al. 2014). Control group feature enables control over resources available for the container to consume (Khan 2017). Advantages offered by containers are to provide a lightweight, as light as one single process, and portable services, and a platform to develop, test and deploy software with different hardware and large number of servers (Pahl 2015; Felter et al. 2015).

Docker is one platform to develop, run and ship applications by the use of containers.(Docker Inc. 2024) Soon since it's launch in 2013 Docker rapidly gained popularity. At the time, Docker offered a more comprehensive user experience comparing to other container tools at that time, and its own image format leading it to quickly become the standard tool for image and container management by 2014 (Felter et al. 2015; Merkel et al. 2014). Docker demands root privileges, which limits use in some occasions.

Singularity created at Lawrence Berkeley National Laboratory is a very popular container engine in HPC clusters. Singularity is open source, and it is compatible with Docker Images (Sylabs Inc & Project Contributors 2024). For the future use of our solution, it is important to take in consideration and aim to have compatibility with HPC clusters. That will increase the

scope of usability, by enabling execution for circuits with higher qubit count and more depth.

Podman is another open source container platform, it is compatible with docker images and has widely similar usage and abilities. One major difference to Podman's advantage is that it may be run also by a non-privileged user, unlike Docker (Podman 2024).

Container registries are offered by various providers from cloud computing, version management and container platforms. These container image registries contain large scale of different base images, anything from ready to run software, development environments to databases, web servers and beyond (Merkel et al. 2014). In addition to the docker engine, capable of running the container images, Docker offers Docker Hub, a docker registry for storing and sharing images. For this Thesis' solution, we are using Docker Hub to store and share images.

3.5.2 Container orchestration

In microservice architecture based applications, an application may be composed of a multitude of containerized services. This leads to need for a container orchestration tools and automation (Khan 2017). Orchestration tools allow a cloud based application developer to define how the services should be run, distributed, monitored and configured to run as a multi-container application in could (Casalicchio and Iannucci 2020). According to Asif Khan (Khan 2017) container orchestration platform should be offering the enabling following features:

1. cluster state management and scheduling
2. providing high availability and fault tolerance
3. ensuring security
4. simplifying networking
5. enabling service discovery
6. making continuous deployment possible
7. providing monitoring and governance

Orchestration tool providers have some differences in the exact implementations and in the level of control offered over the containers. Different implementations and complete lack of given features may be found, for example in *resource limit control, scheduling, load balancing, health monitoring, fault tolerance tools, auto-scaling* (Casalicchio and Iannucci 2020).

Kubernetes is a cluster management framework to manage, containerized workloads and services. Kubernetes cluster is build from control plane and nodes. Nodes are the units that are running the containerized application, and each cluster has at least one worker node (Kubernetes 2024). Which then hosts the Pods, The Pod encapsulates at least one container each. a Pod has its own assigned resources, like CPU cores or memory limits, IP addresses, a set of options for its containers to run with (Rodriguez and Buyya 2019). The other important basic building block of Kubernetes cluster is kubelet which is an agent running on each node. The kubelet takes given PodSpecs and manages its node to run as instructed and stays healthy (Kubernetes 2024).

Kubernetes supports several container runtime engines natively, and is compatible with practically with all current commonly used container platforms based on containerd or CRI (Container runtime interface) (Rodriguez and Buyya 2019).

Category	Examples	Actors	Goals	Concepts
Knowledge sharing	Rocket Chat GitLab wiki Redmine Trello	Everyone	Human collaboration	Culture of collaboration Sharing knowledge Breaking down silos Collaborate across departments
Source code management	Git SVN CVS ClearCase	Dev/Ops	Human collaboration Continuous delivery	Versioning Culture of collaboration Sharing knowledge Breaking down silos Collaborate across departments
Build process	Maven Gradle Rake JUnit Sonar	Dev	Continuous delivery	Release engineering Continuous delivery Automation Testing automation, Correctness Static analysis
Deployment Integration	Chef Puppet Docker Heroku Open Stack AWS Cloud Formation Rancher Flyaway	Dev / Ops	Continuous Delivery Reliability	Frequent and reliable release process Release engineering Configuration management Continuous delivery Infrastructure as code Virtualization, Containerization Cloud services, Automation
Monitoring and Logging	Nagios Zabbix Prometheus Logstash Graylog	Dev / Ops	Reliability	You built it, you run it After-hours support for Devs Continuous runtime monitoring Performance, Availability, Scalability Resilience, Reliability, Automation Metrics, Alerting, Experiments Log management, Security

Table 1. Example listing of DevOps tools and their usage by Leite et al. in A Survey of DevOps Concepts and Challenges (Leite et al. 2019)

4 Applying DevOps in quantum development

4.1 DevOps for quantum

The aim for the methodologies suggested in the literature, is to fill the gaps between quantum computer and the classical software development for real-world applications leveraging quantum computing (Gheorghe-Pop et al. 2020). In general all the applicable methods of classical DevOps should be implemented in quantum DevOps, and among them developer should be pay attention to the special characteristic of quantum during the workflow.

Code execution on quantum hardware is prone to errors, and the computers should be monitored closely. Gheorghe-Pop et al. suggest the following summarized list of actions to be made when applying DevOps in quantum computing.

- At regular intervals, various available QC instances are being checked for the calculation of basic gates.
- This provides an estimation of whether a QC instance is currently likely to be able to perform a large critical calculation correctly.
- Based on these checks, the most promising QC instance for a calculation is then selected (also among different cloud quantum providers).
- This process is applied in the **development, testing, and operations** and merged into a kind of Quantum DevOps

(Gheorghe-Pop et al. 2020)

In the following sections I will go through the suggested steps and actions divided into two phases, Development and Operations.

4.2 Development

Plan. The programmer or algorithm designer analyses the problem and it's requirements for the system. Planning also includes design of the models, architecture and the algorithms to use (Gheorghe-Pop et al. 2020). These two sub-steps are affecting each other, as much

of the requirements are dependent on the design and design needs a system to support it. One characteristic feature in quantum programming with current systems is the quantum - classical splitting. Chosen algorithm and the problem to be solved will be affecting the split, and in hybrid solutions like Variational Quantum Eigensolvers the quantum algorithms might have classical algorithms as part of them and in other solutions like Quantum phase estimation the algorithm might be fully quantum (Weder et al. 2020).

Code. Coding and code management in quantum is much similar to the classical computing but has some of its own characteristics. When the process moves from planning to algorithm design and implementation, the code should be designed as hardware independent, in a quantum programming language chosen suitable (Weder et al. 2020). While the initial coding should be hardware-agnostic, the choice of Quantum SDK may impact compatibility with quantum hardware. Meaning, that hardware platform most likely to be targeted, should be noted in the choice of SDK early on, if possible as it might have noticeable impact on available libraries and other dependencies.

When ensured the algorithms wanted behaviour in simulator, and moving on towards execution on quantum computers, the hardware software coupling comes more and more important factor (Yue et al. 2023; Weder et al. 2020). When moving to QC execution of algorithms, the code needs to be hardware optimized. The hardware might be very different depending on the vendor, or even the by the device, varying from qubit implementation, to circuit design and the coherence of single qubits and gates (Yue et al. 2023). The source code management should follow similar principalities as in classical DevOps, including version control and source code management. Common tools like Git is compatible to quantum code management.

Build. When the code is optimized to the target hardware, and it is executable, it may be compiled and executed (Gheorghe-Pop et al. 2020). In most of the current ecosystems the hardware provider is controlling the access tighter than in classical computing, but if working with a full stack ecosystem by the provider, this is somewhat similar experience as in classical. In the build phase of a quantum software of course has to be noted that the guides like *build binaries once*, and *deploy the same way to every environment* are not possible and has to be adjusted to suit each situation and to note that in current systems quantum code is

often transpiled for each execution for each device.

Testing. When talking testing for quantum, one of the most notable things to consider quantum computers is that by the nature a quantum system may not be debugged such way as we have used to in traditional computer. This is caused by the fact that the qubits state collapses when measured, and the execution may not be continued as the superposition is gone. The no-cloning theorem in quantum mechanics also proves it impossible to copy the state of a qubit when not knowing it's state. Meaning that debugging, stopping and step-by-step execution has to be done all only in classical simulators, which emphasizes the need to use simulators rather than QC in coding and earlier stages of development process. Other white-box testing activities listings, reviews, and inspections may and should be done similarly as in classical. All these test methods are important, but do not remove the need to execute code in testing. Other major factor to be considered in all quantum testing is the probabilistic nature of quantum computing, which is affecting all testing done to quantum programs (Miranskyy and Zhang 2019; García de la Barrera et al. 2023). Suggested flow of testing the quantum code by Gheorghe-Pop et al., is to apply testing in three phases, with **simulator without noise**, **simulator with noise model** and to test with **QC hardware**. To apply frequent testing as suggested by DevOps, the need for high performance simulator and access to QC platforms emphasizes(Gheorghe-Pop et al. 2020).

4.3 Operations

Deployment While the process during development, testing, and build follows previous steps explained here, should the deployment be possible to be performed similarly as in classical, but again with noticeable difference in the non standardized access points available for the current QC by different vendors. When the target quantum hardware is chosen and the application is ready to be deployed, it will be transpiled for the target in question, possibly reconfigured for the target platform and then finally deployed to the target for execution (Gheorghe-Pop et al. 2020). For the deployment developer should use suitable deployment tool, as in classical, using dedicated CD tools like GitHub actions, or customized container tools, docker and kubernetes (Weder, Barzen, and Leymann 2021; Romero-Álvarez et al. 2023).

The current QC cloud services offer their own deployment services for their accessible Quantum platforms, where the implementation might be distinctive for each vendor. With the current systems like Qiskit serverless, the quantum code execution is implemented as a single job each time the code is executed, rather than deploying the application to a server. [1]

4.4 Challenges in applying DevOps to quantum computing

As the most common, briefly introduced earlier in section 3.2.2 quantum SDK's, are built from early-on to be fully compatible with classical software. Many of them are implemented simply as libraries to a classical programming language, applying any development method and to integrate them into classical software is not a problem from that point of view. In current landscape, a big challenge to apply DevOps to quantum and quantum-hybrid computing is the integration of the hardware. Most of the hardware has their own access interface, often only implemented and integrated into only a certain cloud platform, tying the development process to that environment.

There is a clear business logic to support this, and it is practically repeating how the current classical cloud infrastructure is built to work. For developer's point of view the situation is different as classically simulating quantum is computationally so demanding. Powerful classical processors, both CPUs and GPUs, are widely available in HPC clusters and data centres, but it is not always a trivial task to include them in the development process. As DevOps suggests that the application developed should be tested frequently, that means that the quantum parts should be run as frequently, which may turn out to be costly. If run either locally, which is time-consuming, or in cloud platform, where frequent executions might turn out to be expensive, and caused by the batch computation model end up queuing in the targeted platform.

The other major consideration raised by Gheorghe-Pop et al. and Weder et al. is the quantum hardware selection (Gheorghe-Pop et al. 2020; Weder et al. 2020). Thou, related to what is said earlier about the accessing hardware, this is another problem. To provide the user the best possible way of knowing where to execute the quantum code, they would need to

1. https://docs.quantum.ibm.com/run/quantum-serverless

have detailed information on things like, devices calibration, qubit topology, and capable operations of the exact device (Stirbu et al. 2023). While this may remain to be a challenge in the near future, and it's solving is highly dependent on hardware providers, creating a system with possible access to different machines may help the developer to make a choice of execution platform most suitable for their needs.

5 Solution

5.1 Content of the chapter

To answer the object and research question of this thesis, in this chapter is presented a model for a containerized software execution platform for quantum programming. The solution is built from container components, of which an implementation is presented here, and the solutions' architecture is explained, from both static, and dynamic point of view. Followed by the demonstration scenarios Notebook and GPU, and Quantum execution.

5.2 Containerised quantum application

In quantum computing as the execution target is often not the developer's local machine, the containers offer support in the development process before the deployment phase, which they are most commonly used in classical development. Testing, each execution of code during development, debugging, practically every execution of the quantum code, beyond certain complexity if possible is justified to run in either higher powered simulator or suitable QC. Containers enable this to be done in a controlled and easy manner, saving developers time, which is a very valuable resource.

Image. The container image is consisting of separate layers. Operating system, we have chosen Ubuntu latest LTS, currently `Ubuntu 22.04` version and for the GPU enabled container images we use a base image by Nvidia for Cuda, which comes on `Ubuntu 22.04`.

Architectural structure of Cuda container presented in figure 8 for Cuda to work, we also include the `LD_LIBRARY_PATH` in the GPU base image.

Following the OS, the images have the compatible `python` and `pip` version installation and update commands. With pip installed, the executed container installs requirements, which are python libraries related to the project under work. In this example in listing 5.1 packages installed are nvidia-cuda packages and chosen quantum framework, and it's backend, `pennylane, pennylane-lightning, pennylane-lightning[gpu]` and

`matplotlib`. Finally, we pack the executable quantum code `simple_qml.py` into the container image.

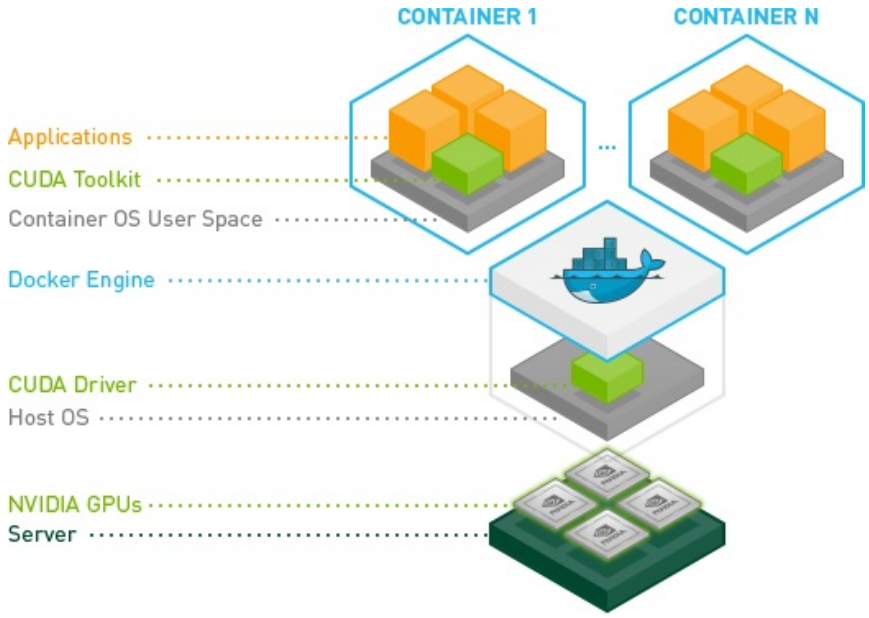


Figure 8. Nvidia CUDA architecture in containerized application (NVIDIA Corporation 2024).

Quantum Node. The quantum capable node is a specialized node in the cluster for the quantum code from the container to be executed in. It is separated by adding a label, in our solution we have chosen `accelerator:qpu`, which is later used to target the quantum workloads for that node. The purpose of this quantum node is to be the execution platform for the finalized quantum algorithm, or the production level quantum or quantum hybrid software. In our demonstration, we have implied a virtual QPU with ssh connection to LUMI cluster's HELMI QPU.

Simulator Node. When aiming the execution on high performance simulator, for example a GPU or even a HPC cluster with several GPUs or CPUs, may this be done similarly from the kubernetes cluster, by assigning the suitable node to the wanted platform, and choosing the node in th job description. This may be done for simply using node selector `accelerator:gpu`. GPU simulator nodes would be suitable to use during development, in testing the algorithm, in debugging, or to run algorithms within the reach of simulating.

Job. Kubernetes has different options for workload resources, of which we found Job to be

Listing 5.1. example of a dockerfile aimed for PennyLane using GPU powered lightning backend

```
FROM nvidia/cuda:11.0-base-ubuntu22.04

RUN apt-get update && apt-get install -y python3-pip
RUN apt-get -y install cuquantum-cuda-11

RUN pip install --no-cache-dir \
   pennylane \
   pennylane-lightning \
   'pennylane-lightning[gpu]'\
   matplotlib

COPY entrypoint.sh .
COPY install.sh .

RUN chmod +x ./install.sh && ./install.sh

ADD simple_qml.py .

ENV LD_LIBRARY_PATH='/usr/local/lib/python3.10/..'


CMD ["./entrypoint.sh", "--gpus=all"]
```

Listing 5.2. Pod specification created from the template described in the Job

```
apiVersion: v1
kind: Pod
metadata:
  name: quantum-pod
spec:
  nodeSelector:
    accelerator: qpu
  containers:
    - name: quantum-task
      image: "registry.example.com/user/program:v1.2.3"
      resources:
        requests:
          vendor.example.com/qpu: 1
        limits:
          vendor.example.com/qpu: 1
```

most suitable for our purpose. When a Job is created it runs the pod or pods as instructed in a Job spec file, in this solution executing the container with our quantum code as a batch type of execution. The cluster does the tracking and logging for the pod. In the JobSpec file it is determined which container is executed, and on what kind of hardware it was designed to run on e.g. `qpu` or `gpu`, and how many times the pod is wanted to be executed. An example in the listing 5.3. The JobSpec example in the listing also shows that the container and volume are listed separately. This is done to have the executable code, mounted as volume, as it is often changing more frequently than the dependencies of the container. This speeds up the code execution, as installation of the base image and dependencies for each code change would slow the execution noticeably each time.

Label is a selector used in JobSpec to set a certain type of instruction for the job. Label is not used as a name to select a unique or specific node as a target, but to specify that the selected node needs to have a certain ability, like an accelerator. In our solution, the wanted target for a job to be executed would be accelerator GPU for simulator backend or QPU for quantum hardware.

5.3 Architecture

The artefact is designed and developed using current open source tools, kubernetes and docker. Both are commonly used in classical development to implement containers and to execute them in cloud platforms. The containers images are stored in container registry, from where they are pulled and pushed to. In our demos, we have used Docker hub. In testing and development quantum programs we have used Qiskit, PennyLane and Cirq toolkits, all of which are commonly used in quantum hybrid software development and in quantum algorithm development.

The cluster, presented in figure 9, has a control plane, with a scheduler and API server, which the developer or deployment tool is communicating with. This control plane in kubernetes is called a master node. From the master node, the jobs are assigned to the matching nodes, to a worker node. The worker node has a kubelet communicating with the master node's API server. When a job is assigned to a worker node, it retrieves the container directly from the

Listing 5.3. example of a JobSpec to send a job to be executerd in GPU accelerator

```
apiVersion: batch/v1
kind: Job
metadata:
  name: "gpu-quantum-job"
spec:
  template:
    metadata:
      name: "gpu-quantum-pod"
    spec:
      containers:
        - name: "gpu-quantum-task"
          image: registry.example.com/user/job-dependencies:v1
          command: ["python", "/app/main.py"]
          resources:
            requests:
              nvidia.com/gpu: '1' # requires GPU usage
          volumeMounts:
          - name: config-volume
            mountPath: /app
      volumes:
        - name: config-volume
          configMap:
            name: task-files #{"main.py": "code"}
      restartPolicy: Never
```

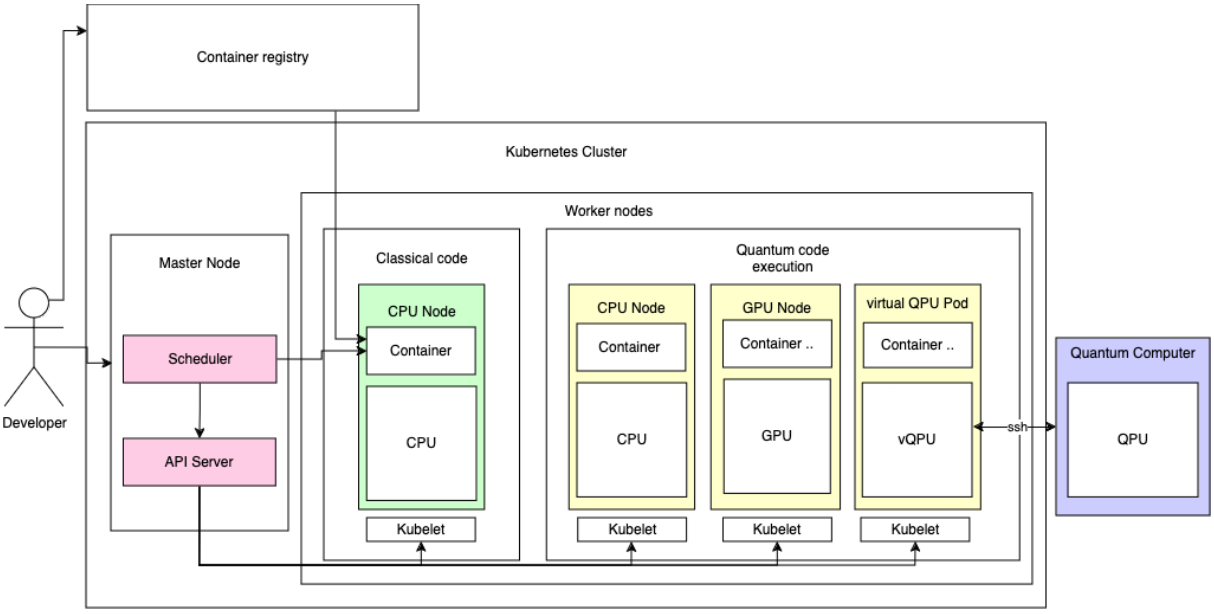


Figure 9. Example structure of quantum cluster with CPU, GPU and QPU nodes

container registry, and executes the code according to the job specification.

Execution. When the CD/CI Pipeline, User or other application using the solution executes the program with quantum code and the target is set to GPU or QPU triggers it the following process, also presented as a flow sequence diagram in figure 10. The execution creates job and with the given specs, as explained earlier, in the Master node the scheduler schedules job, and it is assigned to a suitable node. Then the worker node selected pulls the container from the registry, and executes it as instructed. With the first assigned run the dependencies are installed, later only the quantum code, from the mounted volume is executed. Finally, after execution or failure the logs are pulled from the worker node and passed back as intended to the instance that launched the process.

5.4 Demonstration scenarios

5.4.1 Notebook and GPU

To demonstrate the use of containerized quantum applications and execution's advantage for a developer, we have created a notebook backend leveraging containerization and a kubernetes cluster. The backend was implemented as a custom IPython kernel, responsible for

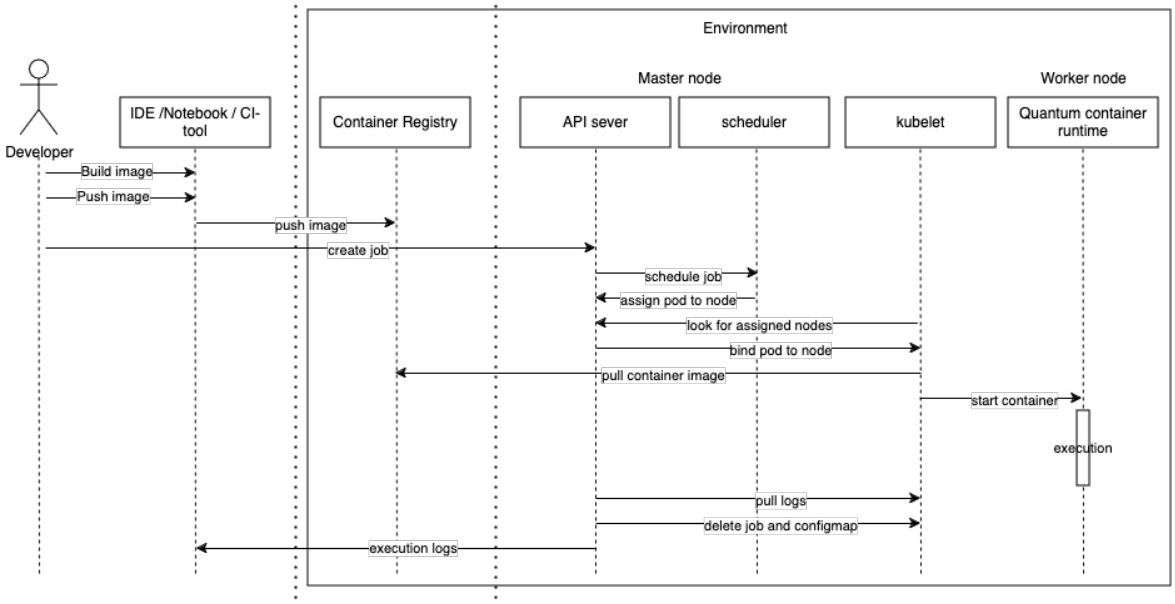
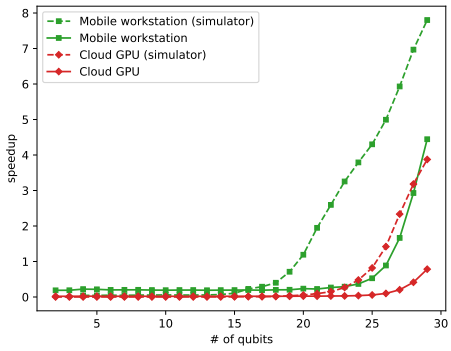


Figure 10. Sequence diagram presenting the data and command flow of the solution

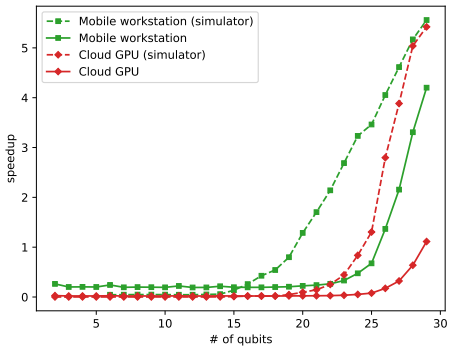
packaging the code from the notebook, to the selected platform to run on, in our demo a GPU laptop and in a data-center GPU in the cloud.

Using the solution as a part of application is one important way of showing that the versatility of containers may be transferred to a certain point into quantum computing in the current landscape. These types of notebooks are very commonly used by people working in science, and similar applications are offered by IBM Qiskit, Google Ai and most other big and small QC providers in their closed ecosystems. Our notebook backend differs from other similar solutions as it is completely open source and easy to install into Jupyter notebook locally, and may be adjusted to be used with any accessible QPU or GPU target.

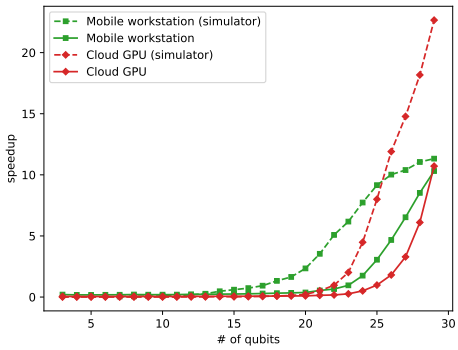
This notebook experiment shows that the solution is useful and did enable great advantage in terms of execution time and addition to the range of complexity of the quantum code within the reach of a developer. The advantage GPUs offer over CPU, when moving to more complex circuits, is significant, and remains to provide speed-ups even with the overhead created by the additional layer and network delay created in the cluster. The system provided speed-ups in quantum code from 4 to 10 times shorter execution times, depending on the benchmark in question, viewable and explained in the figure 11.



(a) QFT



(b) QV



(c) QAOA

Figure 11. Results achieved in the notebook execution times, with three benchmarks QTF, QV and QAOA. The dotted lines are presenting the advantage provided by the GPU's, when executed locally also showing the maximum advantage reachable with the chosen hardware. The solid lines are presenting the speed-ups with the overhead created by the kubernetes cluster and with the use of container apps.

5.4.2 Quantum execution

In other demonstration we did and experimental run with Quantum computer HELMI, operated by CSC[1]. In this experiment the Quantum node in the Kubernetes cluster was set up as a virtual QPU, which was connection to HELMI control unit, with SSH connection. HELMI is a small 5 qubit Quantum computer and is not highly integrated into any larger service, and does not offer other public access points.

In this proof experiment, the goal was to deliver and execute the code using QPU and the Kubernetes cluster. In the experiment, we assigned a quantum job to this virtual QPU node. The container used included quantum code equivalent to classical "Hello World", and logged the results in the job's log. Both goals set were succeeded as aimed. The QC used in this experiment is small and older generation with only few qubits and this experiment was just a proof of concept, for the future this leaves room for more research, different algorithm executions and experimenting with different more recent and higher end QC and implement the QPU more tightly in the kubernetes cluster.

1. https://docs.csc.fi/computing/quantum-computing/helmi/helmi-specs/

6 Discussion and conclusions

6.1 Evaluating the artefact

I am evaluating the solution presented here with a framework introduced by Hevner et al. for Information system design using Analytical evaluation method with four sub steps :

- Static Analysis: Examine structure of artifact for static qualities
- Architecture Analysis: Study fit of artifact into technical IS architecture
- Optimization: Demonstrate inherent optimal properties of artifact or provide optimality bounds on artifact behaviour
- Dynamic Analysis: Study artifact in use for dynamic qualities (e.g., performance)

(Hevner et al. 2004)

6.1.1 Static analysis

To analyse the complexity of the solution, need to consider the alternatives for the solution and the purpose it is aimed for. One main purpose of development of software containers has been the control over the execution environment, and to handle dependencies when transferring the program to the targeted platform. When using GPU or QPU as an accelerator the dependencies' management gets more complex creating a need for suitable toolkit. Containerising the quantum application will provide advantage over non containerised application in dependency management.

When moving forward to more complex software, several services including quantum, the Kubernetes, or other container platform will provide necessary tools like scheduler, volumes, workload distributors for the developer. Kubernetes may be used to provide similar services to a quantum application as to a classical, helping the implementation of DevOps encouraged architectures like Micro-service architecture. Containers and container management platforms suit to be used in quantum and in quantum-hybrid applications when applying DevOps methods.

6.1.2 Architecture analysis

The architectural choices of the solution are justified by two important points of view, 1. the direction classical software development methods have moved in past years, and 2. the development which has been happening frequently with GPU accelerators and their integration to cloud based systems.

Microservice architecture is suggested to be implemented in DevOps, and containers are often mentioned as go-to deployment tool when applying DevOps. Both DevOps method and containers have been gaining popularity for the past ten years, and are very widely adapted in the community in applications of all sizes and shapes. The architecture of software containers is suitable and applicable and may be used in DevOps for to quantum and in quantum-hybrid applications.

Quantum computing may provide acceleration in many current applications in the future. This will mean that the quantum algorithms will need to be integrated to previously built applications. In our solution we suggest use of Kubernetes, which is already found in many cloud native applications. The possibility to integrate into wide variety of existing application and to follow their chosen architecture supports the choice of containers for quantum applications presented in this thesis.

6.1.3 Optimization

Building the containerised solution based on current technologies, adapted to quantum computing, has great advantages for developers as the know-how for these solutions exist. This provides huge advantage to the developer when comparing to learning completely novel technology. Quantum simulators needing high computation power, and being suitable workloads for GPU's, have many things in common with currently trending machine learning and artificial intelligence. In classical development use of containers is considered to be optimal solution for deployment, and as proven they may be transferred to quantum DevOps we believe they will be suitable and often optimal solution for at least deploying the quantum code in development process.

6.1.4 Dynamic analysis

For thorough dynamic analysis, we have little to no measured data, which would require several participants to experiment with development with and without the solution. The numerical data we have from the Notebook solution shows that the execution times are noticeably shorter with GPU than CPU, and while the difference is provided by the hardware, not our solution, the experiment shows that the solution improves the execution times even with the overhead in the cluster. Speed-ups in different situations are discussed in 5.4 and figure 11. Without wider experiments, it is impossible to estimate how much the solution could improve the development process and save time for developers.

6.2 Threats

6.2.1 Internal threats

Internal threat to the validity of this work may be considered to be mostly in the following points.

We have created and tested containerized apps on PennyLane, and Qiskit, which are popular toolkits but do not present the whole field of quantum software development. For example, Cirq may be an equally good and popular choice of framework for machine learning applications. Similarly, GPU all simulations in our experiments have been executed on Nvidia hardware on Cuda platform, which is currently offering the widest support for chosen quantum toolkits, but is not the only GPU providers. The similar pattern goes on also with container engine and container management platform, in which the experiments could be repeated in other platforms to prove the results to be reliable.

Quantum hardware experiment was experimented on one computer. Partial reasoning for this and discussion of closed environments has been explained in this work, but to transfer the suggested containerised model to practice more experiments with different hardware should be done.

Also, due to limited time and resources on this thesis process a larger scale full stack application, leveraging the introduced solution and applying DevOps has not yet been developed.

This would likely be the best way to provide feedback on the solution and develop it further. To evaluate some aspects of the solution is impossible without having done a full process leveraging it's in theory claimed advantages.

6.2.2 External threats

The solution is built to rely on the possibility to access quantum hardware from outside the vendor's own cloud infrastructure. That makes it dependent on finding accessible machines in future. Current small scale machines like CSC HELMI, with 5 qubits will not be the main goal for the future quantum programs, and there is a risk that most of larger scale machines will remain outside the access of applications like ours.

Current vendors are trying to build as good and versatile platforms to attract users. That has already provided great user share to Qiskit SDK, and may continue to do so, as IBM is actively bringing more features into the platform and continues to integrate their new hardware into the system. This may lead to a more centred market where smaller actors may disappear or be leaving behind in the race.

7 Conclusions and future directions

7.1 Conclusion

In this work I have presented a model to implement modern development methods, especially DevOps into quantum software development. As a novelty, I have created a containerised solution to be used in development. From the demonstrations presented and the theory supporting them, this work shows that containers may be used in quantum software development. The classically efficient container tools like Kubernetes will provide same efficiency to quantum software development.

In Quantum software development, the execution platform plays a big role, and containerising the application for execution during development may provide advantage in time savings and by enabling easier runs. Quantum computers have characteristics why simulations will stay relevant as part of the development even as the QC hardware improves, containerised solution provides ease to deploying into efficient simulator targets. In this way, containers may bring great advantage to the development process.

7.2 Future directions

Quantum computing is still young and has many directions it may develop into, yet may be seen that the interest and investments by large classical IT companies has raised lately. Many of the cloud giants have in most parts suitable infrastructure to add QC into. This is happening by both, building own hardware and adding smaller vendor's machines into their cloud platforms.

For other actors in QC, it will be crucial to offer good connectivity and access points to be able to survive when competition gets harder. To secure this for European actors, EuroHCP has made a decision to work towards Universal Quantum Access. Universal Quantum Access would enable standardized access points and interfaces to communicate with quantum computers (EuroHPC 2024). This kind of advancement will help smaller hardware providers to stay relevant, without having to create complete ecosystems around their hardware, and

on the other hand it will help software developers to include quantum computers in applications. When a standardised API is created and generalising, it's or their implementation to Kubernetes should be researched.

Even if the accessibility for the quantum computers improve, for a while the hardware remains to be noisy and incoherent, which sustains the need for more and more efficient simulators. With the solution introduced in this thesis, there is still room to expand from single GPU into multi-node GPU's. This may increase the scope of the execution's again by a some qubits, enable added complexity to circuits, and maybe most importantly cut the execution times by an order of a magnitude. Kubeflow MPI operatorhttps://github.com/kubeflow/mpi-operator allows more HPC-like infrastructure to be implemented in Kubernetes. This should allow distributed computation and performing multi-node execution of quantum code using the Kubernetes cluster, as presented in this work.

Acknowledgements

This work has been supported by the Academy of Finland (project DEQSE 349945) and Business Finland (project TORQS 8582/31/2022).

Bibliography

Ajallooiean, Hossein, Thomas Alexander, Lev Bishop, Yudong Cao, Andrew Cross, Niel de Beaudrap, Jay Gambetta, et al. 2024. *OpenQASM documentation.* https://openqasm.com/intro.html.

Amazon Web Services Inc. 2023. *Amazon Braket Getting Started.* https://aws.amazon.com/braket/getting-started/.

Arute, Frank, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, et al. 2019. "Quantum supremacy using a programmable superconducting processor". *Nature* 574 (7779): 505–510.

Asadi, Ali, Amintor Dusko, Chae-Yeun Park, Vincent Michaud-Rioux, Isidor Schoch, Shuli Shu, Trevor Vincent, and Lee James O'Riordan. 2024. *Hybrid quantum programming with PennyLane Lightning on HPC platforms.* arXiv: 2403.02512 [quant-ph].

Balalaie, Armin, Abbas Heydarnoori, and Pooyan Jamshidi. 2016. "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture". *IEEE Software* 33 (3): 42–52. https://doi.org/10.1109/MS.2016.64.

Bergholm, Ville, Josh Izaac, Maria Schuld, Christian Gogolin, Shahnawaz Ahmed, Vishnu Ajith, M. Sohaib Alam, et al. 2022. *PennyLane: Automatic differentiation of hybrid quantum-classical computations.* arXiv: 1811.04968 [quant-ph].

Byrd, Gregory T., and Yongshan Ding. 2023. "Quantum Computing: Progress and Innovation". *Computer* 56 (1): 20–29. https://doi.org/10.1109/MC.2022.3217021.

Casalicchio, Emiliano, and Stefano Iannucci. 2020. "The state-of-the-art in container technologies: Application, orchestration and security". E5668 cpe.5668, *Concurrency and Computation: Practice and Experience* 32 (17): e5668. https://doi.org/https://doi.org/10.1002/cpe.5668. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.5668. https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5668.

Chen, Lianping. 2015. "Towards architecting for continuous delivery". In *2015 12th Working IEEE/IFIP Conference on Software Architecture,* 131–134. IEEE.

DiVincenzo, David P. 1995. “Quantum computation”. *Science* 270 (5234): 255–261.

———. September 2000. “The Physical Implementation of Quantum Computation”. *Fortschritte der Physik* 48, number 9–11 (): 771–783. ISSN: 1521-3978. https://doi.org/10.1002/1521-3978(200009)48:9/11<771::aid-prop771>3.0.co;2-e. http://dx.doi.org/10.1002/1521-3978(200009)48:9/11%3C771::AID-PROP771%3E3.0.CO;2-E.

Docker Inc. 2024. *Docker Overview.* https://docs.docker.com/get-started/overview/.

Ebert, Christof, Gorka Gallardo, Josune Hernantes, and Nicolas Serrano. 2016. “DevOps”. *Ieee Software* 33 (3): 94–100.

EuroHPC. 2024. *EuroHPC JOINT UNDERTAKING DECISION OF THE GOVERNING BOARD OF THE EuroHPC JOINT UNDERTAKING No 11/2024 Amending the Joint Undertaking’s Work Programme and Budget for the year 2024 (Amendment no 1).* https://eurohpc-ju.europa.eu/document/download/a7d83d79-b70e-4700-a401-8de4c65f9d94_en?filename=Decision%2011.2024.-%20WP%202024%20amendment_V2.pdf.

Felter, Wes, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. 2015. “An updated performance comparison of virtual machines and linux containers”. In *2015 IEEE international symposium on performance analysis of systems and software (ISPASS),* 171–172. IEEE.

Feynman, Richard P, et al. 2018. “Simulating physics with computers”. *Int. j. Theor. phys* 21 (6/7).

Fitz, Timothy. 2009. *Continous Deployment.* http://timothyfitz.com/2009/02/08/continuous-deployment/.

Fowler, Martin, and Matthew Foemmel. 2006. *Continuous integration.*

García de la Barrera, Antonio, Ignacio García-Rodríguez de Guzmán, Macario Polo, and Mario Piattini. 2023. “Quantum software testing: State of the art”. *Journal of Software: Evolution and Process* 35 (4): e2419.

Gheorghe-Pop, Ilie-Daniel, Nikolay Tcholtchev, Tom Ritter, and Manfred Hauswirth. 2020. “Quantum DevOps: Towards Reliable and Applicable NISQ Quantum Computing”. In *2020 IEEE Globecom Workshops (GC Wkshps,* 1–6. https://doi.org/10.1109/GCWkshps50303.2020.9367411.

Gill, Sukhpal Singh, Adarsh Kumar, Harvinder Singh, Manmeet Singh, Kamalpreet Kaur, Muhammad Usman, and Rajkumar Buyya. 2022. “Quantum computing: A taxonomy, systematic review and future directions”. *Software: Practice and Experience* 52 (1): 66–114. https://doi.org/https://doi.org/10.1002/spe.3039. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.3039. https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.3039.

Giri, Pulak Ranjan, and Vladimir E Korepin. 2017. “A review on quantum search algorithms”. *Quantum Information Processing* 16:1–36.

Google Quantum AI. 2024. *Quantum AI Cirq.* https://quantumai.google/.

Hevner, Alan R., Salvatore T. March, Jinsoo Park, and Sudha Ram. 2004. “Design Science in Information Systems Research”. *MIS Quarterly* 28 (1): 75–105. https://doi.org/10.2307/25148625. https://doi.org/10.2307/25148625.

Huang, He-Liang, Dachao Wu, Daojin Fan, and Xiaobo Zhu. 2020. “Superconducting quantum computing: a review”. *Science China Information Sciences* 63:1–32.

Humble, Jez. 2010. *Continous Delivery vs Continuous Deployment.* https://continuousdelivery.com/2010/08/continuous-delivery-vs-continuous-deployment/.

———. *Architecture.* https://continuousdelivery.com/implementing/architecture/.

Humble, Jez, and David Farley. 2010. *Continuous delivery: reliable software releases through build, test, and deployment automation.* Pearson Education.

IBM. 2023. *The IBM Quantum Development Roadmap.* https://www.ibm.com/quantum/roadmap.

IEEE. 2021. “IEEE Standard for DevOps:Building Reliable and Secure Systems Including Application Build, Package, and Deployment”. *IEEE Std 2675-2021,* 1–91. https://doi.org/10.1109/IEEESTD.2021.9415476.

IMB Quantum. 2024a. *Compute resources.* https://quantum.ibm.com/services/resources?tab=systems.

———. 2024b. *Qiskit.* https://quantum.ibm.com/services/resources?tab=systems.

———. 2024c. *Qiskit ecosystem.* https://qiskit.github.io/ecosystem/.

Intel Corporation. 2023. *Intel Releases Quantum Software Development Kit Version 1.0 to Grow Developer Ecosystem.* https://www.intel.com/content/www/us/en/newsroom/news/intel-releases-quantum-sdk.html#gs.1in4lh.

Jabbari, Ramtin, Nauman bin Ali, Kai Petersen, and Binish Tanveer. 2016. "What is DevOps? A systematic mapping study on definitions and practices". In *Proceedings of the scientific workshop proceedings of XP2016,* 1–11.

Jones, Cody, Michael A. Fogarty, Andrea Morello, Mark F. Gyure, Andrew S. Dzurak, and Thaddeus D. Ladd. 2018. "Logical Qubit in a Linear Array of Semiconductor Quantum Dots". *Physical Review X* 8 (2). ISSN: 2160-3308. https://doi.org/10.1103/physrevx.8.021058. http://dx.doi.org/10.1103/PhysRevX.8.021058.

Kang, Hui, Michael Le, and Shu Tao. 2016. "Container and Microservice Driven Design for Cloud Infrastructure DevOps". In *2016 IEEE International Conference on Cloud Engineering (IC2E),* 202–211. https://doi.org/10.1109/IC2E.2016.26.

Khan, Asif. 2017. "Key Characteristics of a Container Orchestration Platform to Enable a Modern Application". *IEEE Cloud Computing* 4 (5): 42–48. https://doi.org/10.1109/MCC.2017.4250933.

Kim, G., J. Humble, P. Debois, and J. Willis. 2016. *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations.* ITpro collection. IT Revolution Press. ISBN: 9781942788072. https://books.google.fi/books?id=ui8hDgAAQBAJ.

Kjaergaard, Morten, Mollie E. Schwartz, Jochen Braumüller, Philip Krantz, Joel I.-J. Wang, Simon Gustavsson, and William D. Oliver. 2020. "Superconducting Qubits: Current State of Play". *Annual Review of Condensed Matter Physics* 11 (1): 369–395. https://doi.org/10.1146/annurev-conmatphys-031119-050605. eprint: https://doi.org/10.1146/annurev-conmatphys-031119-050605. https://doi.org/10.1146/annurev-conmatphys-031119-050605.

Kubernetes. 2024. *Kubernetes documentation.* https://kubernetes.io/docs/concepts/overview/.

Lau, Jonathan Wei Zhong, Kian Hwee Lim, Harshank Shrotriya, and Leong Chuan Kwek. 2022. “NISQ computing: where are we and where do we go?” *AAPPS bulletin* 32 (1): 27.

Leite, Leonardo, Carla Rocha, Fabio Kon, Dejan Milojicic, and Paulo Meirelles. 2019. “A survey of DevOps concepts and challenges”. *ACM Computing Surveys (CSUR)* 52 (6): 1–35.

McKay, David C, Thomas Alexander, Luciano Bello, Michael J Biercuk, Lev Bishop, Jiayin Chen, Jerry M Chow, Antonio D Córcoles, Daniel Egger, Stefan Filipp, et al. 2018. “Qiskit backend specifications for openqasm and openpulse experiments”. *arXiv preprint arXiv:1809.03452*.

McMahon, David. 2007. *Quantum computing explained.* John Wiley & Sons.

Merkel, Dirk, et al. 2014. “Docker: lightweight linux containers for consistent development and deployment”. *Linux j* 239 (2): 2.

Microsoft. 2024. *What is Azure Quantum.* https://learn.microsoft.com/en-us/azure/quantum/.

Miranskyy, Andriy, and Lei Zhang. 2019. “On testing quantum programs”. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER),* 57–60. IEEE.

Morino, Shinya, Andreas Hehn, and Leo Fang. 2024. *Accelerating Quantum Circuit Simulation with NVIDIA cuStateVec.* https://developer.nvidia.com/blog/accelerating-quantum-circuit-simulation-with-nvidia-custatevec.

Myers, Glenford J, Tom Badgett, Todd M Thomas, and Corey Sandler. 2004. *The art of software testing*. 2:6–18. Wiley Online Library.

Nielsen, Michael A, and Isaac L Chuang. 2010. *Quantum computation and quantum information.* Cambridge university press.

Nvidia. 2024. *Nvidia cuQuantum documentation.* https://docs.nvidia.com/cuda/cuquantum/.

NVIDIA Corporation. 2024. *NVIDIA Cloud Native Container Toolkit Documentation.* https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/. Last updated on January 24, 2024.

O'Riordan, Lee J, and PennyLane Team. 2022. *Lightning-fast simulations with PennyLane and the NVIDIA cuQuantum SDK.* https://pennylane.ai/blog/2022/07/lightning-fast-simulations-with-pennylane-and-the-nvidia-cuquantum-sdk/.

Pahl, Claus. 2015. "Containerization and the paas cloud". *IEEE Cloud Computing* 2 (3): 24–31.

Pan, Jiantao. 1999. "Software testing". *Dependable Embedded Systems* 5 (2006): 1.

Pednault, Edwin, John A Gunnels, Giacomo Nannicini, Lior Horesh, and Robert Wisnieff. 2019. "Leveraging secondary storage to simulate deep 54-qubit sycamore circuits". *arXiv preprint arXiv:1910.09534.*

Peffers, Ken, Tuure Tuunanen, Marcus A. Rothenberger, and ChatterjeeSamir. 2007. "A Design Science Research Methodology for Information Systems Research". *Journal of Management Information Systems* 24 (3): 45–77. https://doi.org/10.2753/MIS0742-1222240302. eprint: https://doi.org/10.2753/MIS0742-1222240302. https://doi.org/10.2753/MIS0742-1222240302.

Podman. 2024. *Podman Documentation.* https://docs.podman.io/en/latest/.

Preskill, John. 2012. *Introduction to Quantum Information (Part 1).* The lecture Introduction to Quantum Information at the Canadian Summer School on Quantum Information, held at the University of Waterloo. Available at https://www.youtube.com/watch?v=Q4xBlSi_fOs [Accessed: 2024-05-20].

———. August 2018. "Quantum Computing in the NISQ era and beyond". *Quantum* 2 (): 79. ISSN: 2521-327X. https://doi.org/10.22331/q-2018-08-06-79. https://doi.org/10.22331/q-2018-08-06-79.

———. 2023. "Quantum computing 40 years later". In *Feynman Lectures on Computation,* 193–244. CRC Press.

Qiskit contributors. 2023. *Qiskit: An Open-source Framework for Quantum Computing.* https://doi.org/10.5281/zenodo.2573505.

Red Hat. 2022. *What is CI/CD?* https://www.redhat.com/en/topics/devops/what-is-ci-cd.

Resch, Salonik, and Ulya R Karpuzcu. 2021. “Benchmarking quantum computers and the impact of quantum noise”. *ACM Computing Surveys (CSUR)* 54 (7): 1–35.

Rieffel, Eleanor, and Wolfgang Polak. 2000. “An introduction to quantum computing for non-physicists”. *ACM Computing Surveys (CSUR)* 32 (3): 300–335.

Riungu-Kalliosaari, Leah, Simo Mäkinen, Lucy Ellen Lwakatare, Juha Tiihonen, and Tomi Männistö. 2016. “DevOps adoption benefits and challenges in practice: A case study”. In *Product-Focused Software Process Improvement: 17th International Conference, PROFES 2016, Trondheim, Norway, November 22-24, 2016, Proceedings 17,* 590–597. Springer.

Rodriguez, Maria A., and Rajkumar Buyya. 2019. “Container-based cluster orchestration systems: A taxonomy and future directions”. *Software: Practice and Experience* 49 (5): 698–719. https://doi.org/https://doi.org/10.1002/spe.2660. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2660. https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2660.

Romero-Álvarez, Javier, Jaime Alvarado-Valiente, Enrique Moguel, Jose Garcia-Alonso, and Juan M Murillo. 2023. “Enabling continuous deployment techniques for quantum services”. *Software: Practice and Experience.*

Shahin, Mojtaba, Muhammad Ali Babar, and Liming Zhu. 2017. “Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices”. *IEEE Access* 5:3909–3943. https://doi.org/10.1109/ACCESS.2017.2685629.

Shahin, Mojtaba, Muhammad Ali Babar, Mansooreh Zahedi, and Liming Zhu. 2017. “Beyond continuous delivery: an empirical investigation of continuous deployment challenges”. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM),* 111–120. IEEE.

Soares, Eliezio, Gustavo Sizilio, Jadson Santos, Daniel Alencar da Costa, and Uirá Kulesza. 2022. “The effects of continuous integration on software development: a systematic literature review”. *Empirical Software Engineering* 27 (3): 78.

Stahl, Daniel, Torvald Martensson, and Jan Bosch. 2017. “Continuous practices and devops: beyond the buzz, what does it all mean?” In *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA),* 440–448. https://doi.org/10.1109/SEAA.2017.8114695.

Steane, Andrew. 1998. “Quantum computing”. *Reports on Progress in Physics* 61 (2): 117.

Stirbu, Vlad, Majid Haghparast, Muhammad Waseem, Niraj Dayama, and Tommi Mikkonen. 2023. “Full-stack quantum software in practice: ecosystem, stakeholders and challenges”. In *2023 IEEE International Conference on Quantum Computing and Engineering (QCE),* 2:177–180. IEEE.

Sylabs Inc & Project Contributors. 2024. *SingularityCE User Guide.* https://docs.sylabs.io/guides/latest/user-guide.

Ugwuishiwu, CH, UE Orji, CI Ugwu, and CN Asogwa. 2020. “An overview of quantum cryptography and shor’s algorithm”. *Int. J. Adv. Trends Comput. Sci. Eng* 9 (5).

Vedral, Vlatko, and Martin B Plenio. 1998. “Basics of quantum computation”. *Progress in quantum electronics* 22 (1): 1–39.

Waseem, Muhammad, Peng Liang, Gastón Márquez, and Amleto Di Salle. 2020. “Testing microservices architecture-based applications: A systematic mapping study”. In *2020 27th Asia-Pacific Software Engineering Conference (APSEC),* 119–128. IEEE.

Weder, Benjamin, Johanna Barzen, and Frank Leymann. 2021. “MODULO: Modeling, Transformation, and Deployment of Quantum Workflows”. In *2021 IEEE 25th International Enterprise Distributed Object Computing Workshop (EDOCW),* 341–344. https://doi.org/10.1109/EDOCW52865.2021.00067.

Weder, Benjamin, 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,* 2–9.

Willsch, Dennis, Madita Willsch, Fengping Jin, Kristel Michielsen, and Hans De Raedt. 2022. “GPU-accelerated simulations of quantum annealing and the quantum approximate optimization algorithm”. *Computer Physics Communications* 278:108411. ISSN: 0010-4655. https://doi.org/https://doi.org/10.1016/j.cpc.2022.108411. https://www.sciencedirect.com/science/article/pii/S0010465522001308.

Yue, Tao, Wolfgang Mauerer, Shaukat Ali, and Davide Taibi. 2023. “Challenges and Opportunities in Quantum Software Architecture”. *Software Architecture: Research Roadmaps from the Community,* 1–23.