Quantum DevOps: Automating and Integrating Processes in the Quantum Software Lifecycle

Gabriele Morello and Gard Aasness

"We certify that generative AI, incl. ChatGPT, has not been used to write this essay. Using generative AI without permission is considered academic misconduct."

1 Introduction

Recent developments in quantum computing hardware are uncovering a lot of new applications. While the research is moving fast for quantum hardware and information, quantum software engineering is still at a very early stage, and the gap between these areas is growing larger. In addition, every quantum provider is moving in different directions making it hard to have a clear unified software lifecycle. This essay wants to give the reader an introduction to quantum computing, and then provide an overview of the state of the art of software lifecycle for Quantum Computing and present what one of the main actor in the industry are offering.

2 Quantum Computing

To give context to our essay as it is aimed at Master's students of Computer Science we briefly overview Quantum Computing, this summary will not be an exhaustive explanation but it should provide enough information to understand the rest of the essay.

The fundamental elements of quantum computation are the qubits (from Quantum Bits), which are the basic unit of information as the bits are in classical systems, in the classical world the information in a bit can be 0 or 1. In the quantum world we can have multiple states at once, this phenomenon is known as superposition, and we associate a probability to the possible states.

Another important property of qubits is entanglement, it is a quantum mechanical phenomenon in which two or more particles (Qubits) become correlated in a way that their properties become dependent on each other. This means that if one of the particles is observed or measured, it will affect the state of the other particles, this

has important implications for quantum computing, as entangled qubits can be used to create quantum circuits that perform operations in parallel.

The building blocks of computations are the Quantum Gates, which are like logical gates but they operate on qubits instead of bits. They can manipulate the state of qubits by changing the probabilities of measuring them in different states. Some examples of quantum gates are the Hadamard gate which puts a qubit into a superposition state, and the phase gate, which introduces a relative phase shift between the 0 and 1 states of a qubit.

With quantum gates, we can create quantum circuits, where each gate performs a specific operation on one ore qubit changing its state and in the end, each qubit is measured. Some notable examples of quantum circuits to solve real-world problems are for example Shor's algorithm circuit[1] that is used to factorize large numbers and its improvement compared to a classical factorization is exponential. This result is significant in cryptography as it deems unsafe cryptosystems based on integer factorization.

It is also possible to implement the Variational Quantum Eigensolver (VQE) algorithm [2] which is a classical-quantum circuit to calculate the ground state energy of a molecule. It is particularly useful for molecules that are too complex to be solved exactly with classical computers, therefore it provides a powerful tool for quantum chemistry research and drug discovery.

To develop programs there are several solutions, Microsoft proposed its own language called Q#[3] based on C#. Another option is OpenQASM, which is comparable to Verilog as a level of abstraction. IBM instead developed a framework called Qiskit[4], where developers can write quantum circuits in Python. A rather large area of research is focusing on compilers that allow developers and programmers to write, build, and execute software for quantum computers.

It is important to point out that since quantum computers are still in an early stage and they can't exist outside a research environment, therefore users can exploit quantum computing platforms as a service (QCaaS). QCaaS reduce the requirements for owning or maintaining quantum computers, and it enables to wrap of data and computations inside loosely coupled, fine-grained modules of source code. [5]

3 Quantum DevOps

Seeing how quantum computing is in the infancy stages, quantum software engineering is an emerging research area, meaning that experts are still exploring the optimal practices to develop and maintain quantum applications in regard to concepts and principles.[6] As quantum computing deviates from classical software engineering and it is more complex, another obstacle is the lack of quantum software specialists.[5] A combination of these two problems emphasizes the importance of systematically applying software engineering principles as a part of all phases of the development process, to develop quality quantum applications which are reusable. [7]

3.1 Quantum Software Development Lifecycle

Quantum software is often just a part of a more advanced system which also includes classical software components. This leads to a hybrid system, which is more complex to develop, deploy and maintain, as both quantum- and classical software have to be considered. [8]

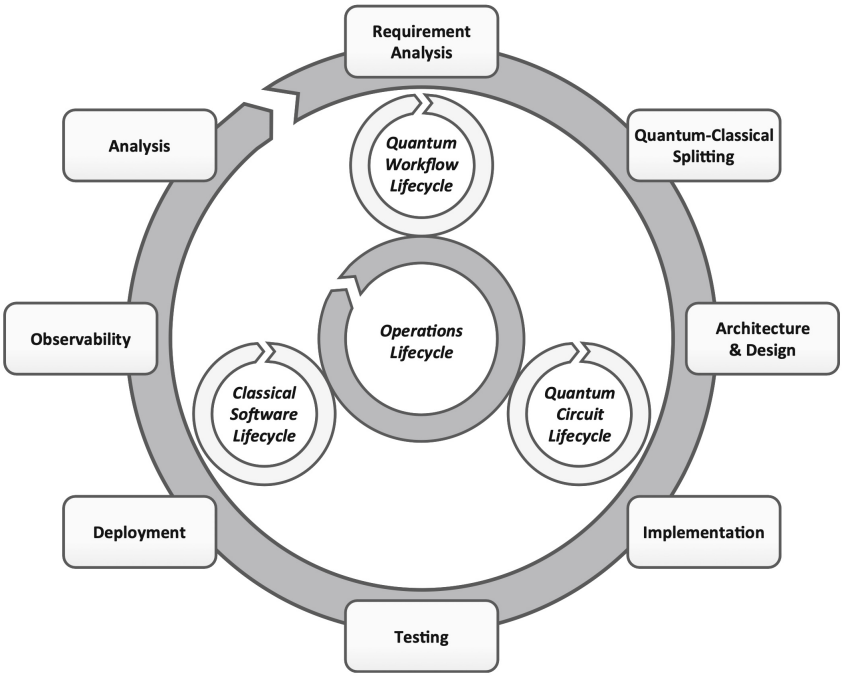

Fig. 1 The quantum software development lifecycle

As a result of being a hybrid system, the software lifecycle of a quantum application is split into smaller lifecycles, shown in figure 1. The lifecycle consists of the quantum workflow lifecycle, quantum circuit lifecycle, classical software lifecycle, and operations lifecycle, which all together combine the maintenance and development of the software with the human management of the software. In accordance with the use of DevOps methodology for applications using classical software, this also allows for continuous integration and continuous delivery for a quantum application. As previously mentioned, considering the quantum computing field is constantly developing, with new hardware being built and new software coming to light, it is important to be easily able to adapt existing quantum applications to these new releases. The DevOps methodology provides the foundation to achieve such adaptability.

The report will now briefly present the different lifecycles and steps that only apply to a quantum application. If a step is present in multiple lifecycles, it will only be described once.

The enclosing lifecycle, shown in figure 1, includes some steps which are absent in a classical software lifecycle. Quantum-classical splitting refers to the process of deciding which parts of a quantum application that should be executed on quantum hardware and which should be executed on classical hardware, based on the output discovered in the requirement analysis step.[6]

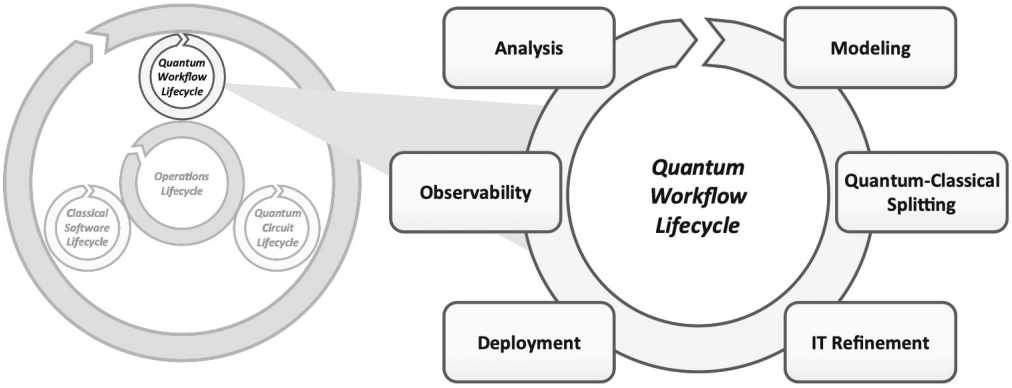

Fig. 2 The quantum workflow lifecycle

The quantum workflow lifecycle shows how the quantum and classical software lifecycles are orchestrated. The modeling step uses the output of the architecture and design phase of the enclosing lifecycle as input and defines all activities necessary to implement the quantum application, including the data flow between these activities. This gives an abstract flow of the system.

The IT refinement step aims to turn the abstract workflow into an executable workflow. To achieve this, the first step should be to look for existing solutions and workflows that can be reused as a part of the solution. If there are no reusable implementations, the next step in this phase is to implement it, which leads to entering the quantum circuit lifecycle shown in figure 3.[6]

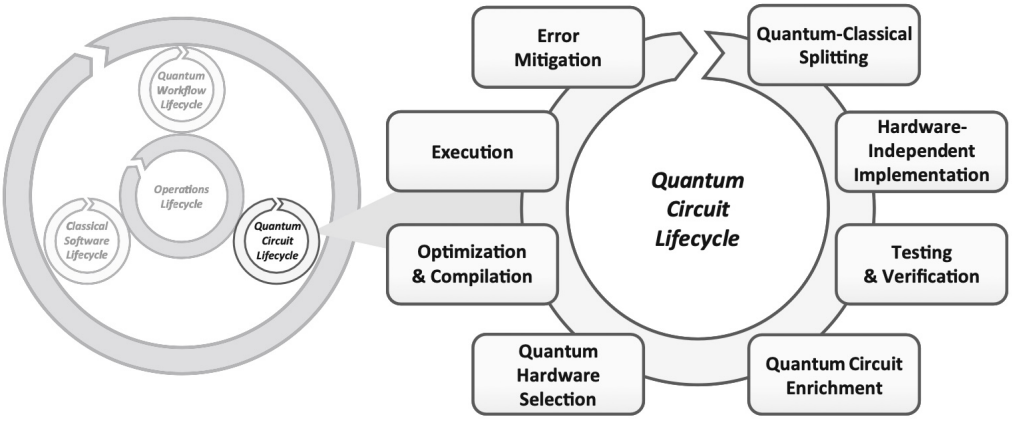

Fig. 3 The quantum circuit lifecycle

The quantum circuit lifecycle shows how to implement a quantum circuit. However, most of the steps include hardware concepts that go out of scope for this report, so we will not discuss it further.

3.2 DevOps Practices for Quantum Computing

A hybrid quantum application is more complex both in terms of the software and the process as a whole. It affects most aspects of DevOps, including the people, process, and product, so the practices have to adapt accordingly. In terms of the people, the different roles of a team require knowledge and skills that would not be necessary in a classical development team, for example, a designer. To optimize a quantum application, most roles need quantum knowledge to understand the limits and boundaries. Regarding the process, the activities and progress between the quantum and classical components should be aligned, to ensure easier integration between the parts, as they are developed individually.

To begin with, the infrastructure as code (IaC) has to be adapted, as the quantum components have to have their own environment, which is separated from the environment of the classical components. In the same way, a classical environment contains all necessary resources to execute the classical software jobs, the quantum environment has to contain all necessary resources to execute quantum jobs. These environments can be automatically provisioned using various web services.

Second, the practices for continuous integration (CI) has to be tweaked. The quantum software has to be tested as it has got its own environment. In addition, there is some classical software which is managing the quantum components, with tasks such as submitting and monitoring quantum jobs, which also need to be tested. To achieve automated testing of quantum components, there are certain activities which need to be performed, including unit tests executed in a quantum simulator, calculation of an estimated amount of resources necessary to run the component on an actual quantum computer in the future, and tests actually executed on real quantum computers. The

testing activities have some requirements to ensure good testing quality. Firstly, the tests which deal with probabilistic behavior need to be executed on real quantum hardware, and it needs to be run several times to ensure that a test stays successful upon subsequent runs. For most other test cases, a quantum simulator containing quantum bits (QuBits) should be sufficient. Secondly, validation of the results of a test can be challenging in the context of quantum computing, due to the characteristics of quantum systems. Therefore, using quantum-specific tools and tricks might be necessary to achieve validation based on whether the solution is valid according to the constraints.

Continuous delivery involves having changes in code and application automatically prepared for release to production and includes activities such as building, configuring, and deployment. There are some adjustments that relate to these activities to have in mind when dealing with a quantum application. To begin with, as mentioned in the previous paragraph, there are some classical jobs that submit quantum jobs on demand. In practice, this means that the quantum code does not persist on the quantum components, and there is therefore no need to re-provision these components for every code change, but rather on scenarios such as a clean deployment. However, the classical components continue to be re-provisioned for every code change. As for the configuration, the classical components which are responsible for submitting and monitoring the quantum jobs, need to be granted access to the quantum workspace.[8]

3.2.1 Overview of Quantum Computing DevOps Cycle

The DevOps paradigm usually splits a cycle, which is an iterative process, into two loops: the inner and outer loop. The outer loop typically deals with more managing-related purposes, such as maintaining a backlog and monitoring the application, whereas the inner loop focuses more on the development process, with tasks such as coding and testing. In a classical software application, there is just one inner loop for the classic components. However, in a quantum application, there is an additional inner loop for the quantum components, as shown in figure 4. This is because they have separate environments.

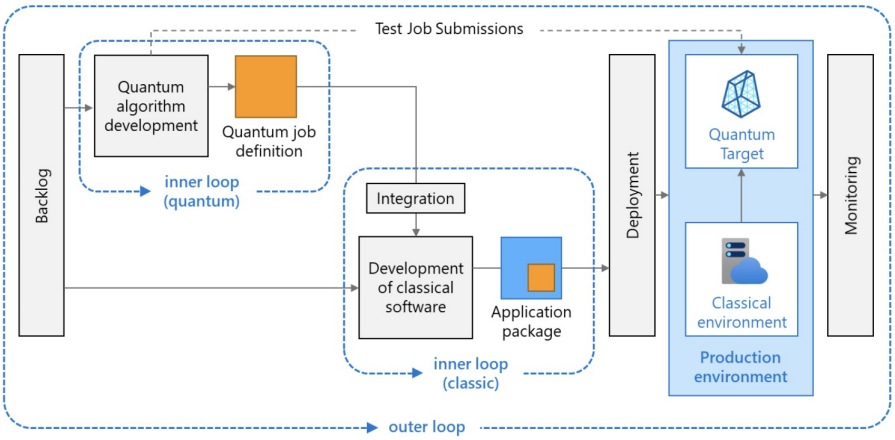

Fig. 4 Full DevOps cycle for a quantum computing project, containing both an inner and outer loop

The inner loops of the two environments are relatively similar, except for switching out writing classical code with quantum code and performing some additional steps which were mentioned earlier, namely testing the quantum code in simulation and actual quantum hardware, and estimating the required resources needed to run the job. Figure 4 shows the DevOps cycle for a hybrid application, and therefore also contains an additional step for integrating the two environments.[8]

3.3 Implementation

To provide a practical example of DevOps in Quantum Computing and to present the state of the art, we now show how we can implement it using Azure's Quantum Development Kit and GitHub Actions. As presented above we are using a hybrid system, in particular in this case we use a loosely coupled architecture. Microsoft proposes two possible hybrid approaches, the differences between them are in the orchestration activities: preparation of input data, submission of quantum computing jobs to target quantum environments, monitoring of job execution, and post-processing of job results. In the tightly coupled approach, the logic for the orchestration of quantum resources is integrated into the classical component, while in the loosely coupled the logic for the orchestration of quantum resources is exposed as an API that can be called by various classical software components. The tightly coupled is preferred is integrated and shares the same lifecycle with the classical code, or when the problem that we want to solve is hybrid by nature such as in the Variational Quantum Eigensolvers (VQE) presented in the introduction to quantum computing. The loosely coupled approach is preferred when a team of quantum specialists provides functionalities to other teams and the quantum job represents a generic solution that can be reused by multiple classical applications.[9]

We will focus solely on the loosely coupled model, and figure 4 represents a slightly simplified example that covers the aspects that we are going to discuss.

3.3.1 Inner Quantum and classical loops

Our quantum job for this example is a random number generator, which is an appropriate task for a quantum computer since quantum properties guarantee true randomness, this is an example of a random number generator in Q#.

```
namespace QApp.Qsharp {
  open Microsoft.Quantum.Intrinsic;
  open Microsoft.Quantum.Canon;
  open Microsoft.Quantum.Measurement;

  @EntryPoint()
  operation SampleRandomNumberInRange(nQubits : Int) : Result[] {
    use qubits = Qubit[nQubits];
    ApplyToEach(H, qubits);
    return MultiM(qubits);
  }
}
```

Now that we have a quantum job, we can use Azure's QDK to write a test that runs on a simulator:

```
@Test("QuantumSimulator")
operation BeOfSpecifiedLength() : Unit {
    let specifiedLength = 8;
    let value = SampleRandomNumberInRange(specifiedLength);
    let length = Length(value);

    Fact(length == specifiedLength, "Expected a result of specified length.");
}
```

Before running on the real hardware we have to estimate the resources that we need to run out job, the QDK provides us with resources, we can automate resource estimation as well as testing via the xUnit framework.

```
[Fact]
public void UseMinimumQubitCount()
{
  int specifiedLength = 3;
  ResourcesEstimator estimator = new ResourcesEstimator();
  SampleRandomNumberInRange.Run(estimator, specifiedLength).Wait();
  var data = estimator.Data;
  int qubitCount = Convert.ToInt32(data.Rows.Find("QubitCount")["Max"]);
  Assert.True(qubitCount <= specifiedLength);
}
```

The inner classic loop includes all the activities for building, debugging, and testing classical components. The integration between the two inner loops is made through API since we are following a loosely coupled approach.

3.3.2 Outer loop

In the outer loop, CI/CD pipelines do building and unit testing automatically by accessing the source code repository. The pipelines then perform unit testing and deploy the application with the required Azure services. Azure offers two possibilities to describe infrastructure as code: Azure Resource Manager (ARM) templates and Azure Bicep, with them we can create instances of Azure Quantum workspace, Azure Storage account, App Service plan, Function App, and Application Insights

We can implement Continuous Integration with GitHub Actions to ensure automatic testing, here's a fragment of a GitHub Action to build and test the quantum component at each commit of new code.

```
    # Checkout code
  - name: 'Checkout code'
    uses: actions/checkout@main

    # Build the Quantum Component
  - name: 'Build the Quantum Component'
    shell: pwsh
    run: |
      pushd '${{ env.AZURE_QUANTUM_PACKAGE_PATH }}'
      dotnet restore
      dotnet build --configuration Release --output ./output
      popd

    # Test the Quantum Component
  - name: 'Unit Test the Quantum Component'
    shell: pwsh
    run: |
      pushd '${{ env.AZURE_QUANTUM_UNIT_TEST_PACKAGE_PATH }}'
      dotnet test --verbosity normal
      popd
```

We use GitHub Actions for Continuous Delivery as well, we have to implement two different workflows, one for infrastructure deployment, and one for application deployment. The steps of first one have to check out the code, login, deploy the infrastructure and configure roles for the quantum workspace and for the Azure Function App. After this, we can deploy the application and the steps are: build the application, Deploy the Azure Functions (containing the packages for the quantum logic), executing tests. [10]

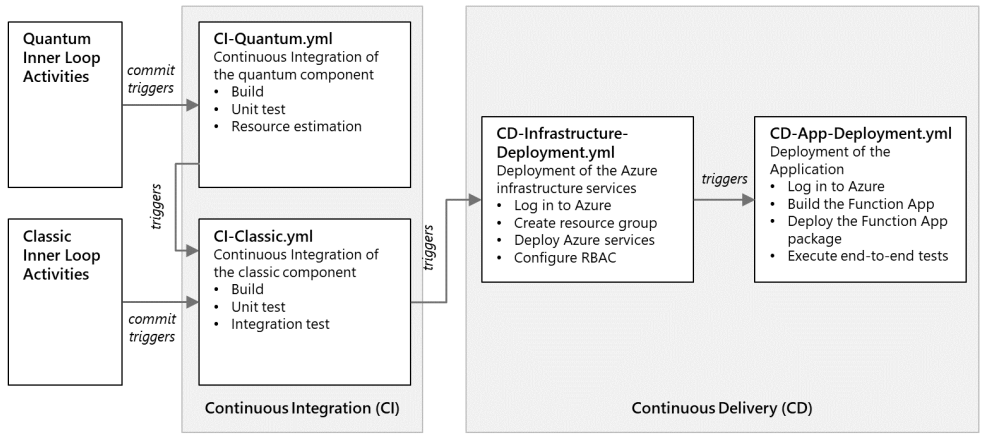

Fig. 5 CI/CD-pipelines for hybrid quantum apps

4 Conclusion

To conclude, a quantum component is typically part of a larger system containing classical components as well. These hybrid applications have a different lifecycle than classical software and therefore need special treatment in regard to DevOps practices to automate for continuous integration and delivery. As our knowledge and use of quantum engineering continue to evolve with time, the importance of quantum DevOps will increase correspondingly, and new and better practices will arise. For now, Azure's Quantum Development Kit provides a good opportunity to follow the quantum DevOps paradigm.

References

[1] Shor, P.W.: Algorithms for quantum computation: discrete logarithms and factoring. In: Proceedings 35th Annual Symposium on Foundations of Computer Science, pp. 124–134 (1994). https://doi.org/10.1109/SFCS.1994.365700

[2] Peruzzo, A., et al.: A variational eigenvalue solver on a photonic quantum processor. (2014) https://doi.org/10.1038/ncomms5213

[3] The Q Quantum Programming Language User Guide. https://learn.microsoft.com/en-us/azure/quantum/user-guide/?view=qsharp-preview

[4] Aleksandrowicz, G., Alexander, T., Barkoutsos, P.: Qiskit: An Open-source Framework for Quantum Computing. https://doi.org/10.5281/zenodo.2562111 . https://doi.org/10.5281/zenodo.2562111

[5] Ahmad, A., Waseem, M., Liang, P., Fehmideh, M., Khan, A.A., Reichelt, D.G., Mikkonen, T.: Engineering Software Systems for Quantum Computing as a Service: A Mapping Study (2023)

[6] Weder, B., Barzen, J., Leymann, F., Vietz, D.: In: Serrano, M.A., Pérez-Castillo, R., Piattini, M. (eds.) Quantum Software Development Lifecycle, pp. 61–83. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-05324-5_4 . https://doi.org/10.1007/978-3-031-05324-5_4

[7] Gheorghe-Pop, I.-D., Tcholtchev, N., Ritter, T., Hauswirth, M.: Quantum DevOps: Towards Reliable and Applicable NISQ Quantum Computing (2020). https://doi.org/10.1109/GCWkshps50303.2020.9367411

[8] Sirtl, H.: DevOps for Quantum Computing. https://devblogs.microsoft.com/qsharp/devops-for-quantum-computing/

[9] Sirtl, H.: Quantum Computing Integration with Classical Apps. https://learn.microsoft.com/en-us/azure/architecture/example-scenario/quantum/quantum-computing-integration-with-classical-apps

[10] Sirtl, H.: Implementing DevOps for Quantum Applications. https://devblogs.microsoft.com/qsharp/implementing-devops-for-quantum-applications/