Integrating Quantum Software Tools with(in) MLIR

Patrick Hopf[†‡], Erick Ochoa Lopez[§*] Yannick Stade[†], Damian Rovara[†], Nils Quetschlich[†], Ioan Albert Florea[†], Josh Izaac[¶], Robert Wille[†‡‖], and Lukas Burgholzer[†‡]

[†]Technical University of Munich, Munich, Bavaria, Germany
[‡]Munich Quantum Software Company (MQSC), Garching near Munich, Bavaria, Germany
[‖]Software Competence Center Hagenberg, Hagenberg, Upper Austria, Austria
[¶]Xanadu Quantum Technologies Inc., Toronto, Ontario, Canada
[§]AMD, Markham, Ontario, Canada

{patrick.hopf, yannick.stade, damian.rovara, nils.quetschlich, ioan.florea, robert.wille, lukas.burgholzer}@tum.de
eochoalo@amd.com, josh@xanadu.ai

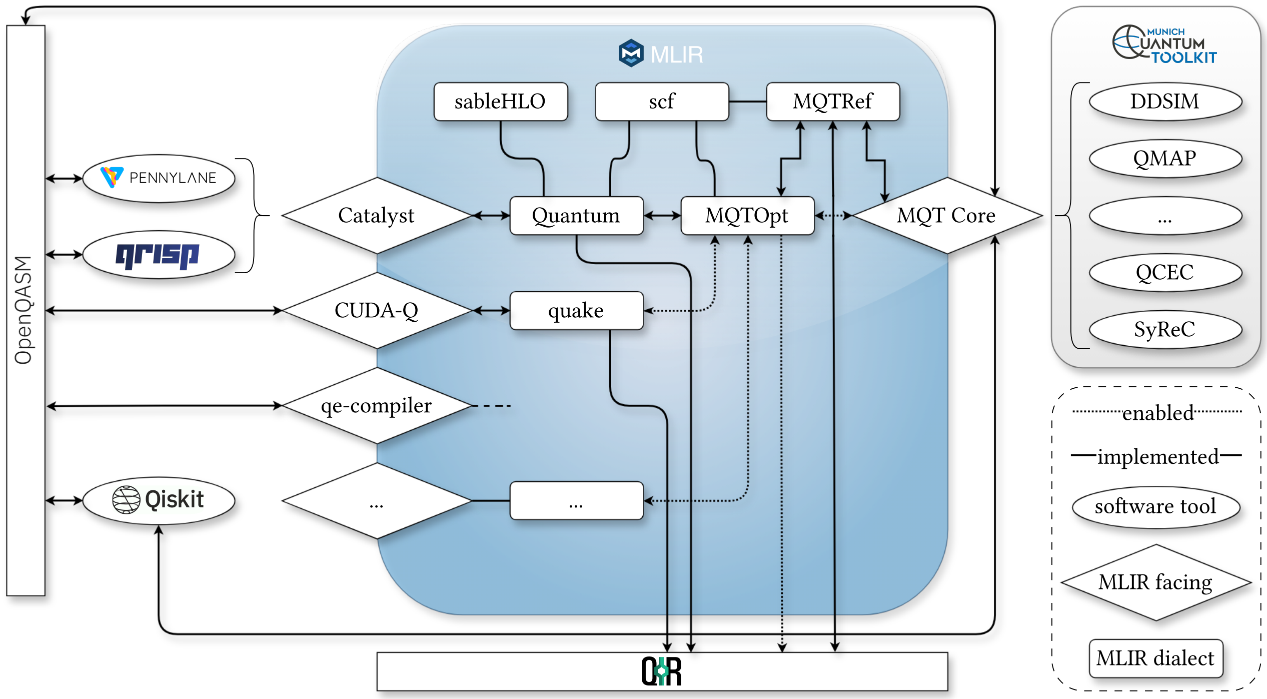


Figure 1: Quantum computing software tools within and outside of the MLIR ecosystem.

Abstract

Compilers transform code into action: They convert high-level programs into executable hardware instructions—a crucial step in enabling reliable and scalable quantum computation. However, quantum compilation is still in its infancy, and many existing solutions are ad hoc, often developed independently and from scratch. The resulting lack of interoperability leads to significant missed potential, as quantum software tools remain isolated and cannot be seamlessly integrated into cohesive toolchains.

The *Multi-Level Intermediate Representation (MLIR)* has addressed analogous challenges in the classical domain. It was developed within the LLVM project, which has long powered robust software stacks and enabled compilation across diverse software and hardware components, with particular importance in high-performance computing environments. However, MLIR's steep learning curve poses a significant barrier to entry, particularly in quantum computing, where much of the software stack is still predominantly built by experimentalists out of necessity rather than by experienced software engineers.

[*]Work done while at Xanadu Quantum Technologies Inc., Toronto, Canada.

SCA/HPCAsia 2026, Osaka, Japan
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-2067-3/2026/01
https://doi.org/10.1145/3773656.3773658

This paper provides a practical and hands-on guide for quantum (software) engineers to overcome this steep learning curve. Through a concrete case study linking Xanadu's PennyLane framework with the Munich Quantum Toolkit (MQT), we outline actionable integration steps, highlight best practices, and share hard-earned insights from real-world development. This work aims to support quantum tool developers in navigating MLIR's complexities and to foster its adoption as a unifying bridge across a rapidly growing ecosystem of quantum software tools, ultimately guiding the development of more modular, interoperable, and integrated quantum software stacks.

CCS Concepts

• **Hardware** → *Quantum computation*; • **Computer systems organization** → **Quantum computing**; • **Software and its engineering** → **Compilers**; **Retargetable compilers**; Source code generation.

Keywords

quantum software development, quantum compilation, intermediate representation, software stack, MLIR.

ACM Reference Format:
Patrick Hopf[†][‡], Erick Ochoa Lopez[§][*] Yannick Stade[†], Damian Rovara[†], Nils Quetschlich[†],, Ioan Albert Florea[†], Josh Izaac[¶], Robert Wille[†][‡][‖], and Lukas Burgholzer[†][‡] . 2026. Integrating Quantum Software Tools with(in) MLIR. In *SCA/HPCAsia 2026: Supercomputing Asia and International Conference on High Performance Computing in Asia Pacific Region (SCA/HPCAsia 2026), January 26–29, 2026, Osaka, Japan.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3773656.3773658

1 Introduction

With the rapid growth of quantum software tools and the increasing capabilities of quantum hardware, *quantum compilers* become increasingly important. Those quantum compilers serve as translators between high-level programming languages and low-level hardware instructions, thereby playing a critical role in connecting two rapidly evolving domains. On one side, industrial and academic teams are developing specialized quantum software tools optimized for diverse needs; on the other, hardware providers offer a wide range of quantum computing platforms—including superconducting, trapped-ion, neutral-atom, and photonic systems—each with their own native gate set and hardware-specific capabilities and constraints. To avoid reinventing the wheel each time a new software tool or hardware platform emerges, there is a clear need for an extensible and robust compilation infrastructure that can mediate between the zoo of high-level quantum programming languages and heterogeneous quantum backends—as is taken for granted in classical and HPC systems.

However, compared to the mature state of classical compilation frameworks, tools dedicated to quantum compilation are only emerging slowly. Many rely on ad-hoc solutions tailored to narrow use cases, are developed in isolation, and frequently reinvent core components from scratch. The resulting lack of interoperability leads to a significant loss of potential, as tools—and their users—could mutually benefit if they were designed to integrate seamlessly.

Fortunately, there is hope: For classical compilation flows, the *Multi-Level Intermediate Representation (MLIR)* [36] framework has already proven to be an effective solution to many of the challenges currently faced by quantum compilation. Adopting this well-established infrastructure for quantum compilers is a natural choice for several reasons:

- As part of the LLVM [34] project, MLIR has been widely adopted—not only by major software frameworks such as JAX [8], TensorFlow [1], and PyTorch [43], enabling deep integration with classical workflows, but also by hardware vendors such as Intel and NVIDIA, establishing crucial links to classical and HPC hardware—a key requirement for hybrid quantum-classical programs.
- Moreover, MLIR allows for the seamless composition of multiple compilation passes within a shared infrastructure, promoting modularity, reuse, and interoperability.
- Ultimately, it is more pragmatic for quantum software developers to extend a robust and ubiquitous classical compiler infrastructure with quantum-specific capabilities, rather than building entirely separate tooling from scratch and incrementally adding classical features such as hardware interfacing, linear algebra, or control flow.

However, MLIR is an overwhelmingly complex project, and its steep learning curve significantly hinders adoption—especially in a community that primarily consists of lateral entrants from various disciplines other than computer science or, specifically, compiler design. Hence, it requires pioneering interdisciplinary projects that connect both worlds: quantum computing on the one side and compiler design on the other side.

This paper aims to support developers in navigating the complexities of MLIR and foster its adoption as a unifying bridge across the fragmented quantum software ecosystem. To this end, we present a practical guide tailored to quantum software engineers, helping them overcome the steep learning curve of working with MLIR in the context of quantum compilation. We demonstrate how MLIR serves as a unified backbone for quantum program representation and transformation workflows, leveraging its lightweight and modular plugin infrastructure. A hands-on approach is emphasized throughout, highlighting best practices drawn from a year-long integration effort connecting two major quantum software frameworks: Xanadu's PennyLane [7] and the Munich Quantum Toolkit (MQT) [60].

The remainder of this paper is structured as follows: Section 2 reviews relevant concepts in both quantum and classical compilation and provides an overview of related work. Section 3 then outlines the problem of limited interoperability among quantum software tools and motivates MLIR as a potential solution, whose adoption is currently limited by the steep learning curve associated with the framework. To overcome the high entry barrier, Section 4 presents best practices and a detailed case study demonstrating the integration of two representative quantum software frameworks—PennyLane and MQT—within the MLIR ecosystem. In Section 6, this is followed by a discussion of key insights and future prospects arising from such an integration. Finally, Section 7 summarizes the contributions of this work.

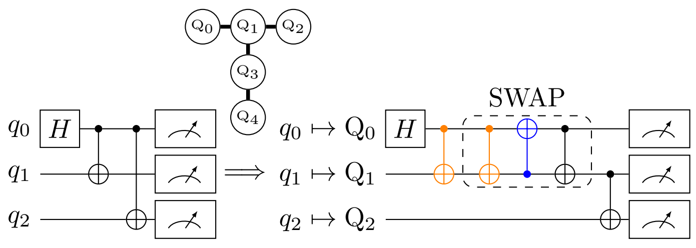


Figure 2: Compilation of a quantum program—preparing the GHZ state—for a T-shaped five-qubit architecture which supports the native gate set {H, T, CNOT}.

2 Background

To keep this paper self-contained, this section briefly reviews the fundamentals of quantum and classical compilation and concludes with an overview of related work on intermediate representations.

2.1 Quantum Compilation

To execute a quantum program on a chosen quantum device, the program must first be converted into an executable. This process is conducted by *compilers* that transform the quantum program to a sequence of hardware-specific instructions (in rough analogy to a classical instruction set architecture, or ISA). Those instructions must adhere to the constraints induced by the chosen quantum computer.

To this end, the *compilation* is typically divided into compilation *passes* that operate on a quantum circuit, which consists of qubit wires and gate operations. These passes can be executed sequentially or invoked multiple times, resulting in complex compilation *flows*. All passes can be broadly classified into the following three categories:

- *Placement and Routing*: These passes assign each qubit present in the quantum program to a physical qubit on the target hardware. They ensure that all gates are executable with respect to the device's topology. To resolve topological constraints, additional operations—such as SWAP gates on superconducting systems or move/shuttling operations on neutral atom and trapped-ion platforms—may be introduced, as demonstrated in [5, 33, 51, 52, 61].
- *Synthesis*: These passes decompose quantum operations that are not natively supported by the target device into sequences of gates from its native gate set, using strategies such as those proposed in [3, 18, 31, 45, 66].
- *Optimization*: These passes aim to improve the efficiency of a quantum program and thereby mitigate some of the inherent limitations of current noisy quantum computers, employing techniques such as those proposed in [24–26, 41, 42].

Example 1. *Assume the quantum program preparing a GHZ state with three qubits shown on the left-hand side of Fig. 2 shall be mapped onto a T-shaped five-qubit architecture (as depicted at the top) and synthesized to its native gate set* {H, T, CNOT}*. By choosing a one-to-one mapping between the circuit and the device qubits and inserting a single SWAP gate—synthesized as three CNOT gates—a fully executable circuit that matches the device's topology can be derived. Subsequent optimization passes may identify that the two orange* CNOT *gates cancel, and, assuming initial qubit states of* $|0\rangle$*, the blue* CNOT *can likewise be removed—leaving only the two trailing* CNOT *operations.*

To construct such compilation flows, all constituent compilation passes usually consume and produce the quantum program in a common format—the so-called *intermediate representation* (IR). To this end, many quantum software tools have introduced their own IRs at both high and low levels of abstraction; examples include Qiskit's `DAGCircuit` [29], TKET's `Circuit` [55], TKET2's `HUGR` [32], as well as MQT's `QuantumComputation` [12] and PennyLane's `QuantumTape` [7]. While this approach works well *within* an individual tool, it complicates the integration of different tools in a unified compilation flow as demonstrated in Example 2. Furthermore, no shared framework or standardized infrastructure is currently established to facilitate efficient IR translation across different quantum software tools for the purpose of compilation.

Example 2. *The MQT Predictor [49] enables the composition of compiler passes from both Qiskit and TKET to construct optimized compilation flows that surpass the capabilities of the individual tools. While the reinforcement learning-based interleaving of various passes yields highly optimized quantum programs, the underlying compilation routine—and consequently the model training process—must continually translate between the distinct circuit representations employed by Qiskit and TKET, posing a significant runtime overhead.*

2.2 Classical Compilation

In contrast to the still-maturing quantum software ecosystem, classical compilation has benefited from decades of development. A key success factor has been the use of shared frameworks, most notably LLVM [34], a widely adopted compiler infrastructure offering reusable libraries and tools. LLVM enables the translation of high-level languages (such as C/C++, Rust, or Java) into low-level, hardware-specific instructions (e.g., assembly) using a unified IR known as LLVM IR.

Beyond language translation, LLVM has become a cornerstone of modern HPC software stacks. It provides the foundation for widely used production compilers such as Clang, supports vectorization and parallelization techniques critical to HPC workloads, and serves as a backend for domain-specific languages and accelerator toolchains. By offering reusable optimization passes, target-specific backends, and a well-defined IR, LLVM enables a high degree of portability and performance tuning across diverse HPC architectures, ranging from multi-core CPUs to GPUs and other specialized accelerators, among which quantum processors are beginning to emerge [16].

While LLVM IR facilitates modular compiler design, relying on a single, static IR can make it difficult to express high-level abstractions or low-level hardware details. This gap led to the development of the *Multi-Level Intermediate Representation (MLIR)* [35], a more flexible infrastructure that supports distinct levels of abstraction and is designed for domain-specific compiler engineering.

MLIR allows developers to define custom IRs via dialects and compose tailored passes for their specific domains. It has been widely adopted by classical software tools such as JAX [8], TensorFlow [1], and PyTorch [43]. The framework provides native support for packaging and distributing custom dialects and compiler passes as plugins. By chaining together multiple passes, developers can construct flexible and modular compilation pipelines.

Internally, MLIR follows Static Single Assignment (SSA) semantics: Meaning each *value* is defined exactly once. All values are explicitly linked to the *operation* that defines them, as well as to all operations that use them. This so-called *definition-use* chain enables efficient program traversal in compilation passes. MLIR's graph-like program representation—comprising operation nodes and value edges—aligns naturally with quantum circuits, which are often modeled as directed acyclic graphs.

Taken together, these features make MLIR a compelling foundation for representing and transforming quantum programs via custom dialects and pass logic, ultimately enabling the deployment of stand-alone plugins.

2.3 Related Work

Given the plethora of quantum software tools available today, it is unsurprising that an equally diverse set of quantum program representations has emerged, each tailored to specific needs and use cases. A variety of data structures have been proposed to describe the functionality of quantum circuits, most notably tensor networks [47] and decision diagrams [62, 63], which aim to represent the exponentially large state space acted upon by quantum operations and are therefore useful for classical quantum circuit simulation [20, 22, 67] or verification [50]. Other representations are tailored to specific compilation tasks, such as the ZX-calculus [30] and its derivatives [2, 40], the stabilizer formalism for Clifford circuits [19], and graph-based models [6, 58, 59]. However, these representations are not designed to serve as general-purpose exchange formats for quantum programs, as they typically prioritize compactness or tractability of specific features over broad expressiveness and interoperability.

Early efforts toward a universal and extensible IR that unifies quantum and classical computation have been introduced with IBM's OpenQASM 3 [14]. It has become widely adopted within the quantum computing community, building on the success of its predecessor, OpenQASM 2 [15]. However, it necessitates reimplementing many concepts that have long been established and refined in classical compiler infrastructures [56].

To address this limitation, the Quantum Intermediate Representation (QIR) [48]—originally proposed by Microsoft—has begun to emerge slowly [56]. QIR is built on top of the classically established LLVM framework and extends it with a universal interface between quantum programming languages and various quantum hardware backends. To this end, it defines a set of protocols for expressing quantum programs in a hardware- and language-agnostic format embedded within LLVM IR and has been adopted in compilation tools such as [21, 64]. As it is based on LLVM, QIR faces the same challenges encountered in classical compilation.

Consequently, it is not surprising that MLIR—offering customizable abstraction levels and an extensible IR framework—has recently gained popularity within the quantum computing community. Q-MLIR [39] explores the integration of quantum constructs into the MLIR ecosystem in a manner that remains compatible with QIR interface standards. Similarly, QIRO [28] extends MLIR to support quantum-classical co-optimization. Industry frameworks such as NVIDIA's CUDA-Q [57] and Xanadu's Catalyst [27] also build on MLIR, providing extensible compiler backends and plugin access for third-party tool developers.

A comprehensive overview and comparative study of initial quantum IRs (such as, e.g., QSSA [44], XACC[38], or QBIR [37]) as well as more recent MLIR-based approaches (such as Q-MLIR [39] and QIRO [28]) can be found in [13].

Ultimately, despite initial efforts toward MLIR-based IRs for quantum compilation, a unified integration path for the diverse set of quantum computing tools remains absent. The compiler and quantum communities remain largely disconnected—mainly due to the substantial expertise required in both domains.

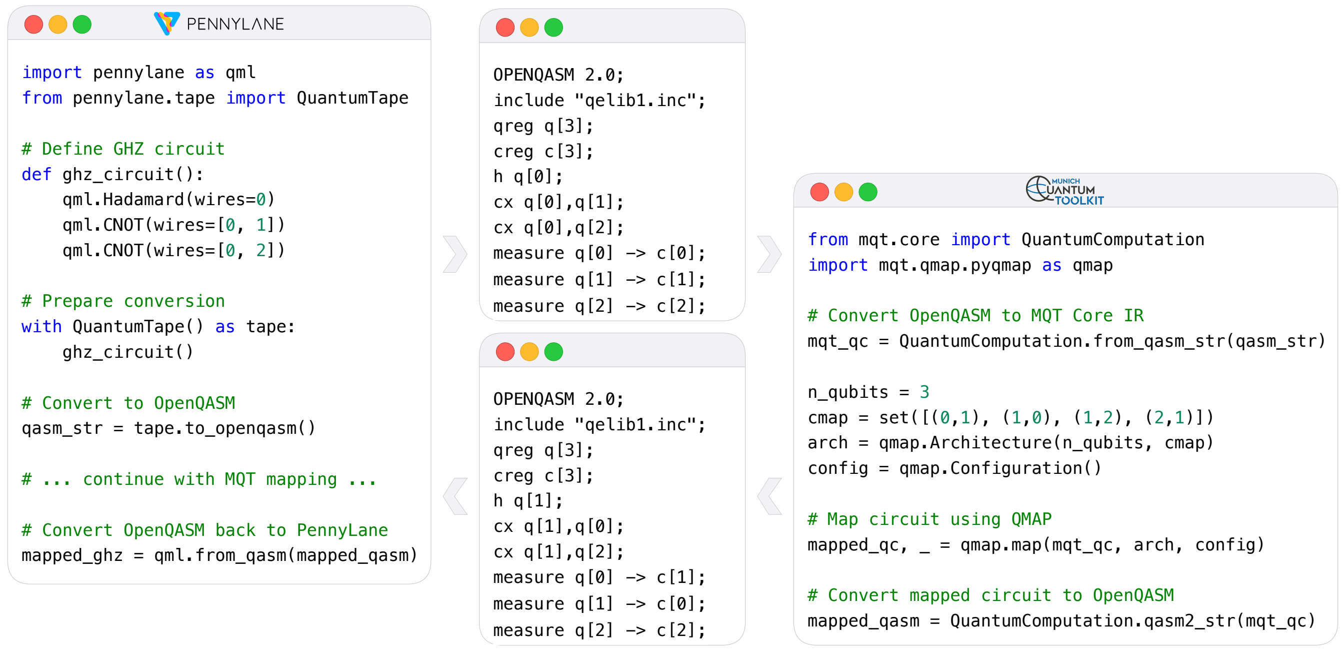
To unlock the full potential at the intersection of these disciplines, it is essential to make MLIR more accessible and usable for the quantum computing community.

3 Motivation

Although MLIR provides a powerful infrastructure for building sophisticated quantum compilers capable of connecting the diverse ecosystem of quantum computing tools, its adoption within the community remains limited. This section examines the key obstacles hindering the broader adoption of MLIR within the quantum community and outlines how this work aims to address them.

3.1 Considered Problem

In classical and HPC computing, compilation has become increasingly standardized and unified, with LLVM IR playing a central role. The introduction of MLIR and its support for custom dialects has relaxed the constraints of using a single IR in a controlled and coherent way. In contrast, the variety of IRs used in quantum computing lacks a unified and shared infrastructure suitable for compilation. A major reason for that is that quantum computing—and more specifically, quantum software engineering—is still a young area of research without established standards. The quantum devices themselves have often been the primary focus area of many researchers and hardware vendors, while the software was treated as an afterthought. This has resulted in a highly heterogeneous quantum computing software landscape that is characterized by a wide variety of quantum software tools. These not only differ in their underlying data structures, compilation strategies, and development philosophies, but they also suffer from limited interoperability due to the absence of a shared IR—effectively prohibiting the combination of different tools without significant effort. This is exacerbated by the rapid pace of the still-young quantum (software) ecosystem



```python
import pennylane as qml
from pennylane.tape import QuantumTape

# Define GHZ circuit
def ghz_circuit():
    qml.Hadamard(wires=0)
    qml.CNOT(wires=[0, 1])
    qml.CNOT(wires=[0, 2])

# Prepare conversion
with QuantumTape() as tape:
    ghz_circuit()

# Convert to OpenQASM
qasm_str = tape.to_openqasm()

# ... continue with MQT mapping ...

# Convert OpenQASM back to PennyLane
mapped_ghz = qml.from_qasm(mapped_qasm)
```

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[3];
creg c[3];
h q[0];
cx q[0],q[1];
cx q[0],q[2];
measure q[0] -> c[0];
measure q[1] -> c[1];
measure q[2] -> c[2];
```

```python
from mqt.core import QuantumComputation
import mqt.qmap.pyqmap as qmap

# Convert OpenQASM to MQT Core IR
mqt_qc = QuantumComputation.from_qasm_str(qasm_str)

n_qubits = 3
cmap = set([(0,1), (1,0), (1,2), (2,1)])
arch = qmap.Architecture(n_qubits, cmap)
config = qmap.Configuration()

# Map circuit using QMAP
mapped_qc, _ = qmap.map(mqt_qc, arch, config)

# Convert mapped circuit to OpenQASM
mapped_qasm = QuantumComputation.qasm2_str(mqt_qc)
```

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[3];
creg c[3];
h q[1];
cx q[1],q[0];
cx q[1],q[2];
measure q[0] -> c[1];
measure q[1] -> c[0];
measure q[2] -> c[2];
```

Figure 3: Mapping a quantum program defined in PennyLane [7] with MQT QMAP [61] without MLIR integration.

and the maintenance and continuous development demands that result from the high volatility.

While there exist "workarounds" made possible by early standardization efforts (such as OpenQASM), these formats are better suited as human-readable, static program descriptions rather than as dynamic, in-memory representations suitable for compiler infrastructure. Consequently, they are not typically used as the internal representation in modern quantum software stacks, which leads to significant parsing overhead and the frequent loss of structural and semantic information during translation.

Example 3. *Assume that the quantum program described in Example 1, which prepares a GHZ state and is shown on the left-hand side of Fig. 2, is defined using Xanadu's PennyLane [7] and should be mapped with MQT's QMAP [61]. Since the tools have no shared IR, there is no trivial solution to implement the joint compilation flow. Therefore, a workaround must be taken as visualized in Fig. 3.*

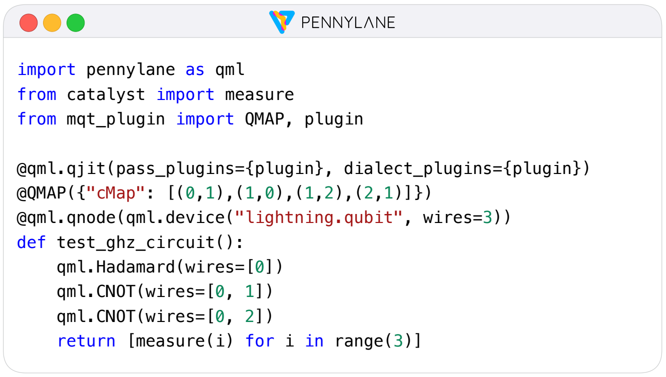
After defining the program as a PennyLane `QuantumTape`*, it must first be exported as an OpenQASM string. This string can then be converted into MQT Core's IR,* `QuantumComputation`*, and subsequently mapped using QMAP. Once the mapping is complete, the compiled circuit must be converted back through all intermediate steps to make it usable in PennyLane again. Ultimately, this process requires an installation of Qiskit, which is used by* `pennylane-qiskit` *for the import of OpenQASM—introducing yet another dependency (with its own IR and conversion routines).*

Furthermore, the qubit assignment resulting from the mapping process is not explicitly preserved in the OpenQASM IR. Thus, there is no way to infer which device qubit a particular qubit from the original circuit has been mapped to, which might be crucial for reasoning about the correctness of the compiled circuit [11].

Instead of reinventing the wheel by developing yet another IR for each new quantum software tool and relying on cumbersome workflows to communicate between them, decades of experience in classical compiler and software infrastructure can be leveraged. The MLIR framework has demonstrated significant benefits in the classical domain and offers a robust foundation for extensible and reusable quantum compiler design. However, understanding and effectively using such a complex and comprehensive framework—developed in C++—is far from trivial and comes with two main difficulties.

The first is technical: Quantum software engineers, often coming from a physics or engineering background, are typically more familiar with Python, as reflected in the predominantly Python-based quantum computing software landscape. This results in a natural barrier to entry, as learning MLIR often requires becoming proficient in C++ as well. MLIR's heavy use of template programming patterns only further complicates the situation. Even just setting up the large LLVM project—of which MLIR is a subcomponent—can be a challenge in its own right. These projects involve millions of lines of code, come with many dependencies, and rely on CMake for configuration and building—another powerful but intricate tool that even computer scientists can find challenging to master.

The second is conceptual: even when technical barriers are lowered, MLIR introduces compiler concepts—such as SSA semantics, IR transformations, and pass pipelines—that may be unfamiliar to physicists more accustomed to unitary matrix representations, pattern matching, and physical design aspects. While projects like xDSL [17] aim to reduce this friction by exposing MLIR-like abstractions through Python, the learning curve remains steep for those without a background in compiler construction.



```python
import pennylane as qml
from catalyst import measure
from mqt_plugin import QMAP, plugin

@qml.qjit(pass_plugins={plugin}, dialect_plugins={plugin})
@QMAP({"cMap": [(0,1),(1,0),(1,2),(2,1)]})
@qml.qnode(qml.device("lightning.qubit", wires=3))
def test_ghz_circuit():
    qml.Hadamard(wires=[0])
    qml.CNOT(wires=[0, 1])
    qml.CNOT(wires=[0, 2])
    return [measure(i) for i in range(3)]
```

Figure 4: Mapping a quantum program defined in PennyLane [7] with MQT QMAP [61] using the MLIR plugin system.

We believe that these factors significantly contribute to the fact that MLIR has not yet seen widespread adoption within the quantum computing community. Its largely untapped potential to interconnect the fragmented landscape of quantum software tools underscores the need for a clear and accessible guide to help quantum engineers integrate this well-established classical framework into their tools.

3.2 Contribution

This paper aims to provide such a practical guide for quantum software developers aiming to harness the potential of MLIR as a compilation framework. It demonstrates how MLIR can be effectively leveraged to bridge heterogeneous quantum software tools through a shared compilation backbone.

To this end, we present a case study showcasing the integration of two major quantum software frameworks: Xanadu's PennyLane and the Munich Quantum Toolkit (MQT). By providing the foundational components required for a successful integration within the MLIR ecosystem, this demonstration focuses on only the most essential concepts while also pointing to valuable resources for more in-depth exploration—helping readers get started without being overwhelmed by the full intricacy of MLIR. We demonstrate a lightweight and modular approach by leveraging MLIR's dedicated plugin infrastructure, thereby minimizing complexity and external dependencies—common barriers to widespread adoption. A hands-on approach is emphasized throughout, with best practices highlighted and key insights shared from our integration efforts. To further lower the entry barrier and facilitate broader adoption across the community, the full code is made publicly available as part of the MQT repository [60].

The following section demonstrates how to achieve the seamless interoperability enabled by MLIR's extensible plugin architecture.

Example 4. *The workflow described in Example 3 is obviously not optimal. However, using the MLIR plugin system, the complexity of such a workflow is significantly reduced as shown in Fig. 4. Behind the scenes, the program is efficiently converted within MLIR using dedicated dialects and passes. Providing the custom plugin and pipeline in PennyLane is enough to trigger the compilation—avoiding any OpenQASM parsing, dumping, and loading efforts, as well as fully abandoning the Qiskit dependency.*

4 Integration

This section presents a practical integration effort that enables interoperability between two major quantum software frameworks using MLIR. The first two parts introduce the individual frameworks, while the third outlines the key components necessary to facilitate their integration within the MLIR ecosystem. Together, these present a practical blueprint for quantum software engineers seeking to adopt MLIR in their own tooling.

4.1 Xanadu's PennyLane

PennyLane [7] is a cross-platform Python library for quantum programming, with a key differentiator compared to other quantum software frameworks being built-in support for end-to-end autodifferentiation through quantum and classical instructions. Recently, PennyLane has added support for quantum just-in-time (QJIT) compilation via the Catalyst compiler [27]. Inspired by similar approaches in the Python ecosystem (including frameworks such as Numba and JAX), quantum just-in-time compilation allows dynamic, hybrid quantum-classical programs to be written in Python, but not executed by the Python interpreter. Instead, when executed from Python, the program is captured (including all classical processing, quantum instructions, and control flow), and compiled to an optimized machine binary via MLIR and LLVM. On subsequent calls to the Python program, this pre-compiled binary is instead executed on the specified simulator or hardware device, without the need to recompile or re-capture the program.

Internally, the Catalyst compiler consists of three main components:

(1) **A Python frontend**. When the user executes a function to be QJIT-compiled, it is executed with *tracers* representing abstract function parameters, allowing the function's behaviour to be captured. This frontend extends JAX's [8] tracing infrastructure to capture Python functions that contain classical instructions, PennyLane quantum operations, and native Python control flow. The frontend then lowers the program representation to MLIR, using a bespoke quantum dialect for the quantum operations, and the StableHLO dialect by OpenXLA for the classical instructions.

The bespoke MLIR Quantum Dialect provided by Catalyst allows users to denote their own custom gates, which would be lowered to a runtime function call passing the name of the gate as a parameter. The quantum operations are expressed in SSA form, where a quantum operation takes input qubits and returns output qubits.

(2) **An MLIR compiler**. During compilation, Catalyst optimizes the program representation to LLVM and compiles the program to a binary. During this process, quantum optimizations are applied directly to the structured program (that is, without removing or unrolling any classical control flow).

(3) **A runtime**. During execution of the compiled binary, Catalyst will execute quantum instructions on user-specified quantum hardware or simulator devices, such as PennyLane's Lightning simulator suite [4].

4.2 The Munich Quantum Toolkit (MQT)

The *Munich Quantum Toolkit (MQT)* [60] is a collection of open-source software tools for quantum computing developed by the Chair for Design Automation at the Technical University of Munich as well as the Munich Quantum Software Company (MQSC). Among others, it is part of the Munich Quantum Software Stack (MQSS) [9] ecosystem, which is being developed as part of the Munich Quantum Valley (MQV) initiative. Its overarching objective is to provide solutions for design tasks across the entire quantum software stack. This entails high-level support for end users in realizing their applications, efficient methods for the classical simulation, compilation, and verification of quantum circuits, tools for quantum error correction, support for physical design, and more. These methods are supported by corresponding data structures (such as decision diagrams or the ZX-calculus) and core methods (such as SAT encodings/solvers).

Given the need throughout the entire toolkit to effectively represent and manipulate quantum circuits and the fact that the origins of the project date back to a time when OpenQASM 2 had just been proposed and none of the previously discussed tooling had been around, the MQT features its own C++-based `QuantumComputation` IR, which is part of the MQT Core [12] library and referred to as *MQT Core IR* in the following.

The IR is used in various top-level libraries throughout the MQT. One of these is MQT QMAP [61]. Its name originates from the fact that it was mainly a tool for mapping quantum circuits to superconducting architectures with a limited connectivity (i.e., solving the placement and route problem). Over the years, it has grown to a collection of compilation tools that allows mapping quantum circuits to various qubit technologies, now also including neutral atom quantum computers [51].

As of this work, MQT Core additionally provides dedicated MLIR dialects that are available starting with version 3.3.3.

https://github.com/munich-quantum-toolkit/core/tree/v3.3.3
https://mqt.readthedocs.io/projects/core/en/v3.3.3

Such dialects are crucial for the plugin, with one particular dialect, namely `MQTOpt`, discussed in the following.

4.3 Implementation

Integrating PennyLane and MQT into MLIR requires several foundational components. To facilitate a clear understanding of these building blocks, we present a dedicated MLIR *dialect*, associated *transformations*, and the embedding *plugin* infrastructure. We encourage readers to explore the accompanying open-source implementation available in the MQT GitHub repository:

https://github.com/munich-quantum-toolkit/core-plugins-catalyst
https://mqt.readthedocs.io/projects/core-plugins-catalyst

Dialect: The `Quantum` MLIR dialect provided by Xanadu's Catalyst enables direct access to the structure and semantics of quantum programs. For example, parts of the MLIR representation of the GHZ circuit implemented in Fig. 4 are shown in Fig. 5a using Catalyst's SSA format.

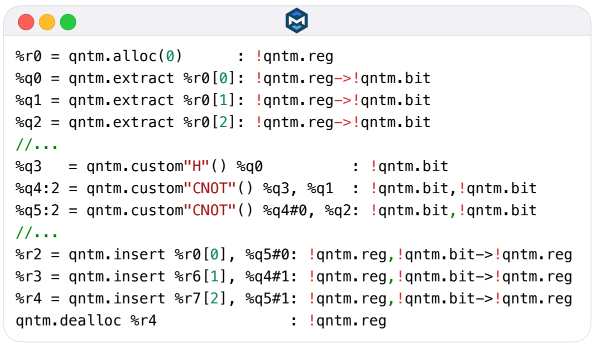
However, as outlined in Section 2, many quantum software tools do not support MLIR out of the box. Since the MQT similarly lacks native MLIR support, a translation between the MQT Core IR and MLIR is necessary.

To ease this transition and minimize the disruption caused by such a change, a dedicated quantum dialect (called `MQTOpt`) has been created that very closely matches the semantics of the existing MQT Core IR. In addition to SSA semantics, it enforces a single-use constraint, where each value is consumed only once. Such a linear typing property naturally enforces the no-cloning theorem [65], as each qubit is guaranteed to be defined and used exactly once. This further simplifies the tracking of register state changes during qubit extraction or insertion, thereby easing subsequent translations to and from program representations that explicitly model physical qubit assignments on hardware. For instance, extracting a qubit from a quantum register yields both a qubit and an updated register, and is represented in the custom `MQTOpt` dialect as illustrated in Fig. 5b.

Moreover, Fig. 5b highlights dialect-specific operand and result *types* (e.g., the `QubitRegister`) and *operations* (e.g., the `x`-gate), along with an associated modifier (here, `ctrl`). Another key difference from the `Quantum` dialect (cf. Fig. 5a) is that, while Catalyst represents gates as generic operations distinguished by a `gate_name` attribute, the `MQTOpt` dialect defines each gate as a dedicated, dialect-specific operation. Note that the code shown is only a textual representation of the program (automatically produced by MLIR) and does not necessarily align with the names of classes, types, and attributes in the C++ implementation.

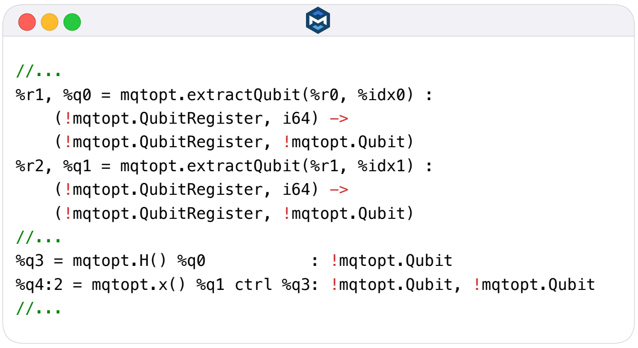
The implementation of a dialect can be further streamlined by defining *traits*, which encapsulate common properties shared across different types or operations. In the context of the `MQTOpt` dialect, structural invariants—such as the expected number of parameters for a given operation (e.g., a rotation gate)—are enforced by implementing a `verifyTrait()` function within the corresponding trait definition.

Transformation: Next, to enable seamless translation between the two dialects, MLIR transformation patterns offer an effective and convenient mechanism for defining translations at different levels of abstraction. In general, three types of MLIR transformations can be distinguished: Importing/exporting to/from non-MLIR representations, conversions between different MLIR dialects, and transformations within a single dialect. Specifically, conversions



```
%r0 = qntm.alloc(0)      : !qntm.reg
%q0 = qntm.extract %r0[0]: !qntm.reg->!qntm.bit
%q1 = qntm.extract %r0[1]: !qntm.reg->!qntm.bit
%q2 = qntm.extract %r0[2]: !qntm.reg->!qntm.bit
//...
%q3   = qntm.custom"H"() %q0          : !qntm.bit
%q4:2 = qntm.custom"CNOT"() %q3, %q1  : !qntm.bit,!qntm.bit
%q5:2 = qntm.custom"CNOT"() %q4#0, %q2: !qntm.bit,!qntm.bit
//...
%r2 = qntm.insert %r0[0], %q5#0: !qntm.reg,!qntm.bit->!qntm.reg
%r3 = qntm.insert %r6[1], %q4#1: !qntm.reg,!qntm.bit->!qntm.reg
%r4 = qntm.insert %r7[2], %q5#1: !qntm.reg,!qntm.bit->!qntm.reg
qntm.dealloc %r4               : !qntm.reg
```

(a) Catalyst `Quantum` MLIR dialect



```
//...
%r1, %q0 = mqtopt.extractQubit(%r0, %idx0) :
    (!mqtopt.QubitRegister, i64) ->
    (!mqtopt.QubitRegister, !mqtopt.Qubit)
%r2, %q1 = mqtopt.extractQubit(%r1, %idx1) :
    (!mqtopt.QubitRegister, i64) ->
    (!mqtopt.QubitRegister, !mqtopt.Qubit)
//...
%q3 = mqtopt.H() %q0           : !mqtopt.Qubit
%q4:2 = mqtopt.x() %q1 ctrl %q3: !mqtopt.Qubit, !mqtopt.Qubit
//...
```

(b) `MQTOpt` MLIR dialect

Figure 5: Two MLIR dialects representing (parts of) the same quantum program.

from higher-level dialects or IRs to lower-level dialects are referred to as lowerings.

In the context of the integration, a (partial) conversion was implemented by providing a `TypeConverter` along with several `ConversionPatterns` to translate between the source and target quantum dialects (leaving non-quantum logic untouched). In contrast to the dialect definitions, which largely consist of boilerplate code, such patterns typically need to be implemented manually, as they often involve non-trivial rewrite logic—for example, enforcing the single-use constraint in the `MQTOpt` dialect.

At this point, it is important to note that having implemented a dialect and its associated conversions has already established all the necessary components to support compilation of quantum programs within the MLIR framework. With these components in place, future development can—but does not have to—build upon MLIR's pass infrastructure to enable advanced compiler analyses and transformations.

Example 5. *To implement a qubit mapping routine, one may define analysis passes to extract qubit interaction patterns, followed by transformation passes that minimize the number of required* `SWAP` *operations. MLIR's pass infrastructure supports runtime configurability through* `PassOptions`*, which enable, for example, the specification of a coupling map via key-value pairs, as illustrated in Fig. 4.*

Plugin: Ultimately, our goal is to make the MLIR passes defined above available without requiring users to compile the entire library—or, more broadly, the entire MLIR project. Fortunately, MLIR's pass manager infrastructure supports the registration of passes and the dynamic loading of plugin-based dialects and passes. To enable this dynamic approach, one must compile an MLIR `DialectPlugin` and `PassPlugin` into a shared library. The compiled files can then, for example, be distributed as pre-compiled binary wheels similar to traditional compiled extensions.

Similarly, tool developers do not have to compile the entire Catalyst project in order to implement a dedicated plugin. In the context of its MQT integration, the implementation only requires the `Quantum` dialect header files instead of having to build the entire library.

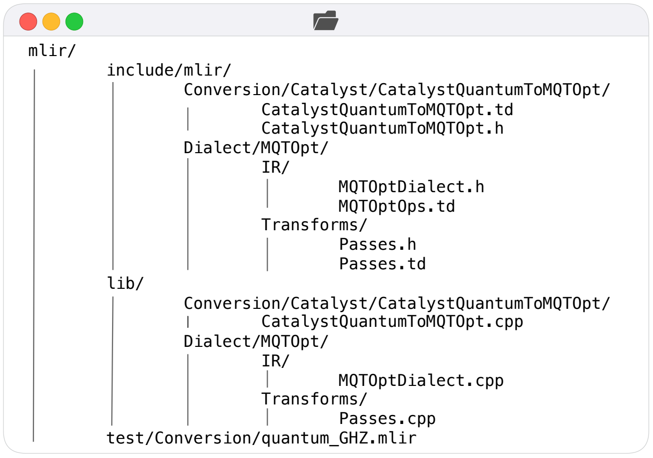
Finally, by leveraging the shared library plugin, the pass pipeline—specifying all relevant passes required for the full round trip—can be executed. The definition of a custom dialect, the transformations to and from this dialect, and the accompanying plugin infrastructure provide all the essential building blocks for integrating quantum software tools within the MLIR ecosystem.

Example 6. *Consider, once more, the task of mapping a quantum program defined in PennyLane using QMAP—i.e., performing a full round trip between Catalyst and the MQT. An example invocation of the complete round-trip pipeline is illustrated in Fig. 6b. Starting from the GHZ circuit defined in the* `Quantum` *dialect (shown in Fig. 5a and provided in* `quantumGHZ.mlir`*), the conversion to* `MQTOpt` *and back is handled by the* `catalystquantum-to-mqtopt` *and* `mqtopt-to-catalystquantum` *passes, respectively. Mapping via QMAP, along with export to and re-import from the MQT Core IR, is performed using the* `mqt-qmap` *pass. Note that the coupling map is specified using the* `cmap` *pass option. The entire round trip can be executed using the* `mlir-opt` *command-line utility, which supports loading dialects and pass plugins via shared library files.*

5 Best Practices

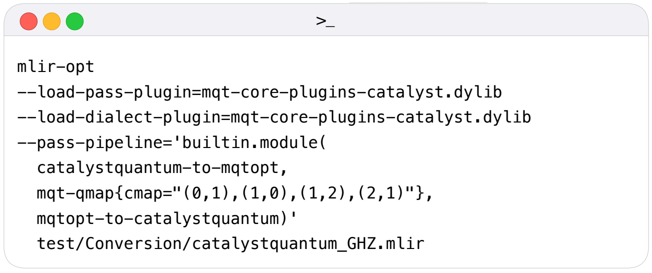
After gaining familiarity with the fundamental building blocks of a representative MLIR integration, the initially high entry barrier has already been lowered. To further ease the adoption process, we present a set of best practices that have proven essential for the successful realization of the integration and can greatly reduce the development effort of similar projects. Quantum software developers seeking to adopt MLIR in their own tooling are encouraged to take these into account.

First, do not reinvent the wheel. MLIR is a rapidly growing ecosystem, supported by contributions from both industry and academia, and includes numerous well-maintained open-source projects. To avoid duplicating existing solutions, it is advisable to engage with the broader MLIR community—particularly through platforms like Discord or GitHub—and to study existing implementations. Numerous MLIR dialects, such as the arithmetic dialect `arith` and the structured control flow dialect `scf`, provide sophisticated, reusable components that can be directly adopted or extended in new projects. In the spirit of encouraging reuse, we have made the implementation of the MLIR dialect and plugin described in this paper available as open source in the MQT GitHub repository.



```
mlir/
│       include/mlir/
│       │       Conversion/Catalyst/CatalystQuantumToMQTOpt/
│       │       │       CatalystQuantumToMQTOpt.td
│       │       │       CatalystQuantumToMQTOpt.h
│       │       Dialect/MQTOpt/
│       │       │       IR/
│       │       │       │       MQTOptDialect.h
│       │       │       │       MQTOptOps.td
│       │       │       Transforms/
│       │       │       │       Passes.h
│       │       │       │       Passes.td
│       lib/
│       │       Conversion/Catalyst/CatalystQuantumToMQTOpt/
│       │       │       CatalystQuantumToMQTOpt.cpp
│       │       Dialect/MQTOpt/
│       │       │       IR/
│       │       │       │       MQTOptDialect.cpp
│       │       │       Transforms/
│       │       │       │       Passes.cpp
│       test/Conversion/quantum_GHZ.mlir
```

(a) MQT's MLIR project structure.



```
mlir-opt
--load-pass-plugin=mqt-core-plugins-catalyst.dylib
--load-dialect-plugin=mqt-core-plugins-catalyst.dylib
--pass-pipeline='builtin.module(
  catalystquantum-to-mqtopt,
  mqt-qmap{cmap="(0,1),(1,0),(1,2),(2,1)"},
  mqtopt-to-catalystquantum)'
  test/Conversion/catalystquantum_GHZ.mlir
```

(b) Executing a mapping round trip with an MLIR plugin.

Figure 6: Overview of a typical file structure of an MLIR-based project and its `mlir-opt` usage.

The project structure is illustrated in Fig. 6a, and it follows the directory layout commonly found in MLIR-based projects. Gaining familiarity with existing projects helps prevent redundant development efforts and provides valuable insights into dialect design and implementation.

Secondly, prioritize modular and lightweight design. Without careful design choices, quantum software engineers—who are often not deeply familiar with the intricacies of LLVM or MLIR—can easily be overwhelmed by their size and complexity (the very reason for writing this article). Modularity—a foundational principle in modern compiler development—helps mitigate their overhead by promoting maintainability, testability, and extensibility in increasingly complex software ecosystems. In practice, this means isolating dependencies and avoiding unnecessary compilation of massive codebases like the entire LLVM project. For example, as full-stack frameworks such as PennyLane continue to grow, compiling or linking against large monolithic toolchains becomes increasingly impractical. Fortunately, MLIR's pass and plugin infrastructure allows developers to cleanly encapsulate custom functionality without requiring deep knowledge or changes to the core system. In the integration effort presented above, we therefore provided the `MQTOpt` dialect and its associated passes as a precompiled MLIR plugin. Embracing modular, lightweight, and plugin-based architectures improves maintainability and lowers the barrier to entry for new contributors. While some situations (like, e.g., debugging and testing) may still require building MLIR (or other large dependencies) from source, a well-isolated design ensures this remains an exception rather than the norm.

Finally, use dedicated tooling. LLVM and MLIR provide a wide range of utilities that can greatly simplify the development of quantum software tools. Among these, two tools are particularly helpful and are used throughout the open-source repository accompanying this paper. Rather than providing a full tutorial, our goal here is to highlight their purpose and role in the development process.

First, the implementation of key MLIR components can be significantly streamlined using the so-called `TableGen` tool and its associated definition files. These files enable the automatic generation of much of the boilerplate code required by MLIR, including header declarations and getter/setter methods—supporting implementations that range from highly general to fully customized (i.e., manually defined). As illustrated in Fig. 6a, the operations in the `MQTOpt` dialect are defined in the corresponding `MQTOptOps.td` file (following naming conventions in the MLIR ecosystem). While this definition file contains fewer than 700 lines of code, the corresponding `MQTOptOps.cpp.inc` file generated by TableGen comprises nearly 20,000 lines of complicated template code—demonstrating the considerable implementation effort that TableGen abstracts away.

Second, `mlir-opt` (already briefly touched on in Example 6) provides a centralized driver that connects and orchestrates the execution of MLIR passes. Its command-line entry point is particularly useful during development and debugging, allowing fine-grained inspection of the IR at different compilation stages. Combined with the LLVM tools `lit` and `FileCheck`, it can also facilitate the systematic testing of MLIR passes and transformations. In the context of the PennyLane–MQT MLIR integration, dedicated test files (e.g., `quantumGHZ.mlir`) are annotated with `lit` commands and `FileCheck` directives that run `mlir-opt` and verify the correctness of its output after specific transformations have been applied. In summary, dedicated tools like `TableGen` and `mlir-opt` not only reduce manual effort and potential implementation errors but also promote consistency across the codebase and enable the application of automated testing workflows.

Adhering to these best practices promotes an effective and sustainable MLIR integration, whose immediate and long-term advantages are discussed next.

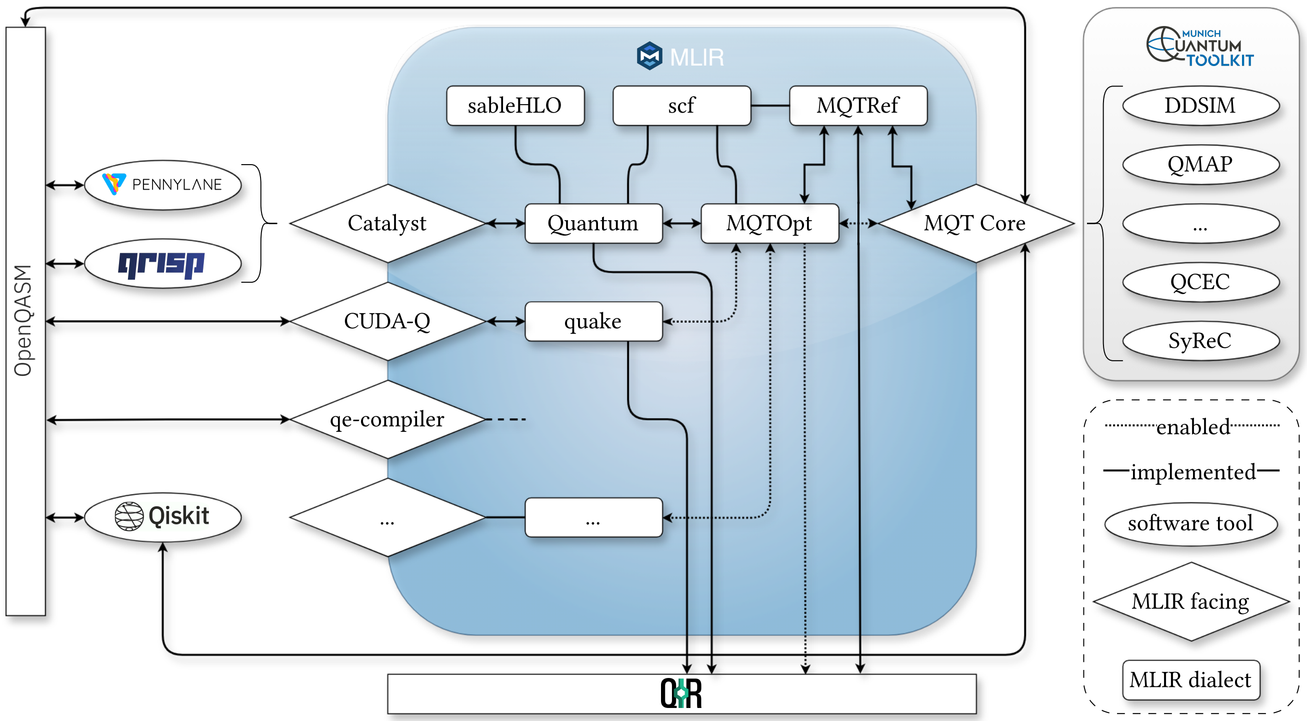


Figure 7: Map of quantum computing software tools within and outside of the MLIR ecosystem.

6 Discussion

The sections above have outlined how the integration of quantum software tools within the MLIR framework can be efficiently achieved, thereby significantly lowering the entry barrier for quantum software developers. Next, key insights gained throughout the integration, as well as directions for future work, are discussed. While the following highlights are drawn from the MQT integration, the insights are broadly applicable to any quantum software tool aiming to adopt MLIR.

6.1 Insights

With the key components and best practices for a successful MLIR integration established, we now take a step back to evaluate the *immediate* implications of this effort. To this end, consider Fig. 7, which illustrates the newly achieved interaction between Catalyst and MQT Core, facilitated entirely within the MLIR framework.

The integrated tool is now part of a largely community-developed ecosystem, joining a growing family of MLIR adopters. As a result, it benefits from ongoing support and advancements across the MLIR ecosystem. In particular, performance improvements and updates by experts from the MLIR community directly enhance the performance of tools that utilize the MLIR infrastructure.

Another advantage is the improved usability for users on both sides—those working with PennyLane as well as those using the MQT. Quantum algorithm experts, for example, can now focus on implementing their algorithms using the straightforward Catalyst decorators for mapping with QMAP, as illustrated in Fig. 4, rather than navigating the cumbersome workflow depicted in Fig. 3.

Leveraging the new integration, the interaction between Catalyst and MQT Core enables interoperability between third-party quantum computing tools that were previously only connected through text-based exchange formats such as OpenQASM. On one side, tools that build upon the Catalyst compiler, such as Qrisp [53], can now directly interface with the MQT compilation tools. While this has not yet been tested at the time of writing, it is expected to require minimal effort. On the other end, any tool built on top of the MQT Core library—such as DDSIM [10] or QCEC [46]—can now consume MLIR input. This connection directly facilitates interoperability across these software tools on the compiled level, significantly expanding the landscape of composable quantum compilation workflows.

6.2 Outlook

Taking an even broader perspective on Fig. 7, additional *long-term* advantages of an MLIR-based integration become evident. Specifically, the integration enables easy extensibility to and seamless interoperability with a broad range of quantum software tools, and paves the way for a more general integration into established software stacks. For instance, integration with other MLIR-based frameworks—such as NVIDIA's CUDA-Q—can now be achieved with minimal additional effort. In such a scenario, little more than conversion passes between the `MQTOpt` dialect and CUDA-Q's `quake` dialect would need to be implemented—an extension made low-effort by the already existing and modular code base.

Furthermore, the integration of classical dialects, such as `scf`, can now be conducted with ease. Catalyst's tracing capabilities, for example, already benefit significantly from structured control flow constructs like loops, as provided by `scf`. Extending the `MQTOpt` dialect to support structured control flow will enable more compact program representations and improve the applicability of compilation passes. For instance, quantum circuits composed of repeated patterns can be expressed with constant program size, an advantage that becomes especially relevant in nontrivial instances of algorithms such as Grover's. Moreover, in these cases, it may suffice to map the fixed qubit interactions within a single loop iteration, rather than unrolling the entire circuit and redundantly mapping identical patches—common, e.g., in quantum machine learning circuits—multiple times.

More broadly, this observation holds for many quantum software tools considering future MLIR integration. Tasks such as circuit analysis and subsequent transformations—often computationally intensive and typically implemented in Python—stand to benefit substantially from the efficient, structured program representation that MLIR provides. By shifting such functionality to the MLIR layer, significant performance gains and improved scalability can be achieved.

Another observation evident from Fig. 7 is that Qiskit has thus far remained relatively isolated from the MLIR ecosystem. The associated `qe-compiler` [23], which converts OpenQASM into several dedicated MLIR dialects, has not seen widespread adoption and remains largely unmaintained. At the same time, MQT Core already supports direct integration with Qiskit at the Python level. With Qiskit's ongoing transition to a Rust-based backend, directly connecting the MLIR and Qiskit ecosystems presents an interesting challenge. The recent development of a C API for Qiskit's compiled components may open up a viable path forward, making MQT Core a promising candidate for enabling integration between the two ecosystems.

Finally, the demonstrated MLIR plugin paves the road to a unified interaction with QIR as an exchange format. Because QIR is built on top of LLVM IR, it is already being utilized for deploying quantum accelerators in HPC software stacks [54]. For the same reason, QIR is familiar to MLIR and can easily be integrated with an MLIR-based compilation flow. As a result, MLIR-integrated tools, such as MQT, can be applied directly to QIR files. Conversely, any tool that already supports QIR can be readily incorporated into the flow, contributing to a modular and extensible quantum software stack.

In summary, the immediate and long-term benefits demonstrate that lowering the entry barrier for quantum software engineers seeking to adopt the MLIR framework is a worthwhile endeavor. Their integration efforts will significantly enhance interoperability and lay the foundation for a more interconnected quantum computing ecosystem, ultimately paving the way for its adoption into classical and HPC software stacks.

7 Conclusions

This work demonstrated how to leverage MLIR's dialect and pass plugin system to develop a modular and lightweight compilation backbone—building upon decades of established classical compiler expertise. To illustrate this, we presented a practical example of integrating two major quantum computing frameworks, replacing previously cumbersome and ad hoc parsing mechanisms with a streamlined MLIR-based solution. Recognizing the steep learning curve posed by MLIR for quantum software developers, we provided best practices and insights derived from this integration effort. The resulting implementation is provided as open-source software, intended as both a foundation and inspiration for similar integration efforts within the broader quantum computing community. Ultimately, this work provides essential guidance for quantum software engineers who are unfamiliar with MLIR and wish to integrate their tools with(in) the framework and thereby improve interoperability with other software tools, as well as with classical and HPC infrastructures. In doing so, it significantly lowers the entry barrier traditionally associated with adopting MLIR for quantum compilation and paves the way for its integration into established HPC software stacks.

Acknowledgments

Authors affiliated with the Technical University of Munich acknowledge funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (DA QC, grant agreement No. 101001318 and MILLENION, grant agreement No. 101114305), the Munich Quantum Valley, which is supported by the Bavarian state government with funds from the Hightech Agenda Bayern Plus. This work has been supported by the BMK, BMDW, the State of Upper Austria in the frame of the COMET program, the BMFTR within the FullStaQD initiative and on the basis of a decision by the German Bundestag through the project SYNQ, as well as the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation – 563436708).

Xanadu authors thank the Catalyst and PennyLane teams at Xanadu for useful discussions and reviews of the MLIR plugin features of Catalyst that made this implementation possible, including but not limited to David Ittah, Mehrdad Malek, Paul Haochen Wang, Joey Carter, and Raul Torres. All authors thank the MLIR and LLVM contributors for all their hard work.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, et al. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software.

[2] Matthew Amy, Dmitri Maslov, and Michele Mosca. 2014. Polynomial-Time T-Depth Optimization of Clifford+T Circuits Via Matroid Partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33, 10 (2014), 1476–1489. https://doi.org/10.1109/TCAD.2014.2341953

[3] Juan Miguel Arrazola, Thomas R Bromley, Josh Izaac, Casey R Myers, Kamil Brádler, and Nathan Killoran. 2019. Machine learning method for state preparation and gate synthesis on photonic quantum computers. *Quantum Science and Technology* 4, 2 (2019), 024004. https://doi.org/10.1088/2058-9565/aaf59e

[4] Ali Asadi, 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] https://arxiv.org/abs/2403.02512

[5] Bao Bach, Ilya Safro, and Ed Younis. 2025. Efficient Compilation for Shuttling Trapped-Ion Machines via the Position Graph Architectural Abstraction. arXiv:2501.12470 [quant-ph] https://arxiv.org/abs/2501.12470 arXiv:2501.12470.

[6] Medina Bandic, Carmen G Almudever, and Sebastian Feld. 2023. Interaction graph-based characterization of quantum benchmarks for improving quantum circuit mapping techniques. *Quantum Machine Intelligence* 5, 2 (2023), 40.

[7] Ville Bergholm, Josh Izaac, Maria Schuld, Christian Gogolin, Shahnawaz Ahmed, and Vishnu Ajith others. 2022. PennyLane: Automatic differentiation of hybrid quantum-classical computations. arXiv:1811.04968 [quant-ph] https://arxiv.org/abs/1811.04968 arXiv:1811.04968.

[8] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, et al. 2018. JAX: composable transformations of Python+NumPy programs. http://github.com/jax-ml/jax Software.

[9] L. Burgholzer, J. Echavarria, P. Hopf, Y. Stade, D. Rovara, L. Schmid, E. Kaya, B. Mete, M. N. Farooqi, M. Chung, M. De Pascale, L. Schulz, M. Schulz, and R. Wille. 2026. The Munich Quantum Software Stack: Connecting End Users, Integrating Diverse Quantum Technologies, Accelerating HPC. In *SCA/HPCAsia: Supercomputing Asia and International Conference on High Performance Computing in Asia Pacific Region.* https://doi.org/10.1145/3773656.3773669

[10] Lukas Burgholzer, Alexander Ploier, and Robert Wille. 2022. Exploiting arbitrary paths for the simulation of quantum circuits with decision diagrams. In *Proceedings of the 2022 Conference & Exhibition on Design, Automation & Test in Europe* (Antwerp, Belgium) *(DATE '22)*. European Design and Automation Association, 64–67.

[11] Lukas Burgholzer, Rudy Raymond, and Robert Wille. 2020. Verifying results of the IBM Qiskit quantum circuit compilation flow. In *Int'l Conf. on Quantum Computing and Engineering*. https://doi.org/10.1109/QCE49297.2020.00051

[12] Lukas Burgholzer, Yannick Stade, Tom Peham, and Robert Wille. 2025. MQT Core: The Backbone of the Munich Quantum Toolkit (MQT). *Journal of Open Source Software* 10, 108 (2025), 7478. https://doi.org/10.21105/joss.07478

[13] F. Javier Cardama, Jorge Vázquez-Pérez, César Piñeiro, Juan C. Pichel, Tomás F. Pena, and Andrés Gómez. 2025. Review of intermediate representations for quantum computing. *The Journal of Supercomputing* 81, 2 (2025), 418. https://doi.org/10.1007/s11227-024-06892-2

[14] Andrew Cross, Ali Javadi-Abhari, Thomas Alexander, Niel De Beaudrap, Lev S. Bishop, Steven Heidel, et al. 2022. OpenQASM3: A Broader and Deeper Quantum Assembly Language. *ACM Transactions on Quantum Computing* 3, 3, Article 12 (2022), 50 pages. https://doi.org/10.1145/3505636

[15] Andrew W Cross, Lev S Bishop, John A Smolin, and Jay M Gambetta. 2017. Open quantum assembly language. *arXiv preprint arXiv:1707.03429* (2017).

[16] Amr Elsharkawy, Xiao-Ting Michelle To, Philipp Seitz, Yanbin Chen, Yannick Stade, Manuel Geiger, Qunsheng Huang, Xiaorang Guo, Muhammad Arslan Ansari, Christian B. Mendl, Dieter Kranzlmüller, and Martin Schulz. 2025. Integration of Quantum Accelerators with High Performance Computing—A Review of Quantum Programming Tools. *ACM Transactions on Quantum Computing* 6, 3, Article 24 (July 2025), 46 pages. https://doi.org/10.1145/3743149

[17] Mathieu Fehr, Michel Weber, Christian Ulmann, Alexandre Lopoukhine, Martin Paul Lücke, Théo Degioanni, et al. 2025. xDSL: Sidekick Compilation for SSA-Based Compilers. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization* (Las Vegas, NV, USA) *(CGO '25)*. Association for Computing Machinery, 179–192. https://doi.org/10.1145/3696443.3708945

[18] Brett Giles and Peter Selinger. 2013. Exact synthesis of multiqubit Clifford+T circuits. *Physical Review A* 87, 3 (2013), 032332. https://doi.org/10.1103/PhysRevA.87.032332

[19] Daniel Gottesman. 1997. *Stabilizer codes and quantum error correction.* California Institute of Technology.

[20] Thomas Grurl, Jürgen Fuß, and Robert Wille. 2025. *Quantum Circuit Simulation With Decision Diagrams*. Springer Nature Switzerland, 21–35. https://doi.org/10.1007/978-3-031-71036-0_3

[21] Harshit Gupta, Rohan Jain, Samuel Kushnir, Priyansh Parakh, and Ryan James Hill. 2025. qBraid-QIR: Python package for QIR conversions, integrations, and utilities. https://github.com/qBraid/qbraid-qir Software.

[22] Johannes Hauschild and Frank Pollmann. 2018. Efficient numerical simulations with Tensor Networks: Tensor Network Python (TeNPy). *SciPost Physics Lecture Notes* (2018). https://doi.org/10.21468/scipostphyslectnotes.5

[23] Michael B. Healy, Reza Jokar, Soolu Thomas, Vincent R. Pascuzzi, Kit Barton, Thomas A. Alexander, Roy Elkabetz, Brian C. Donovan, Hiroshi Horii, and Marius Hillenbrand. 2024. Design and architecture of the IBM Quantum Engine Compiler. *arXiv preprint arXiv:2408.06469* (2024). arXiv:2408.06469 [quant-ph] https://arxiv.org/abs/2408.06469

[24] Patrick Hopf, Nils Quetschlich, Laura Schulz, and Robert Wille. 2025. Improving Figures of Merit for Quantum Circuit Compilation. In *2025 Design, Automation and Test in Europe Conference (DATE)*. 1–7. https://doi.org/10.23919/DATE64628.2025.10992761

[25] Thomas Häner, Torsten Hoefler, and Matthias Troyer. 2020. Assertion-based optimization of Quantum programs. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–20. https://doi.org/10.1145/3428201

[26] Toshinari Itoko, Rudy Raymond, Takashi Imamichi, and Atsushi Matsuo. 2020. Optimization of quantum circuit mapping using gate transformation and commutation. *Integr. VLSI J.* 70, C (2020), 43–50. https://doi.org/10.1016/j.vlsi.2019.10.004

[27] David Ittah, Ali Asadi, Erick Ochoa Lopez, Sergei Mironov, Samuel Banning, Romain Moyard, Mai Jacob Peng, and Josh Izaac. 2024. Catalyst: a Python JIT compiler for auto-differentiable hybrid quantum programs. *Journal of Open Source Software* 9, 99 (2024), 6720. https://doi.org/10.21105/joss.06720

[28] David Ittah, Thomas Häner, Vadym Kliuchnikov, and Torsten Hoefler. 2022. QIRO: A Static Single Assignment-based Quantum Program Representation for Optimization. *ACM Transactions on Quantum Computing* 3, 3 (2022), 1–32. https://doi.org/10.1145/3491247

[29] Ali Javadi-Abhari, Matthew Treinish, Kevin Krsulich, Christopher J. Wood, Jake Lishman, Julien Gacon, et al. 2024. Quantum computing with Qiskit. arXiv:2405.08810 [quant-ph] https://arxiv.org/abs/2405.08810 arXiv:2405.08810.

[30] Aleks Kissinger and John van de Wetering. 2020. PyZX: Large Scale Automated Diagrammatic Reasoning. *Electronic Proceedings in Theoretical Computer Science* 318 (2020), 229–241. https://doi.org/10.4204/eptcs.318.14

[31] A. Yu Kitaev. 1997. Quantum computations: algorithms and error correction. *Russian Mathematical Surveys* 52, 6 (1997), 1191–1249. https://doi.org/10.1070/RM1997v052n06ABEH002155

[32] Mark Koch, Agustín Borgna, Seyon Sivarajah, Alan Lawrence, Alec Edgington, Douglas Wilson, et al. [n. d.]. HUGR: A Quantum-Classical Intermediate Representation. Software.

[33] Fabian Kreppel, Christian Melzer, Diego Olvera Millán, Janis Wagner, Janine Hilder, Ulrich Poschinger, et al. 2023. Quantum Circuit Compiler for a Shuttling-Based Trapped-Ion Quantum Computer. *Quantum* 7 (2023), 1176. https://doi.org/10.22331/q-2023-11-08-1176

[34] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, 75.

[35] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, et al. 2020. MLIR: A Compiler Infrastructure for the End of Moore's Law. arXiv:2002.11054 [cs.PL] https://arxiv.org/abs/2002.11054 arXiv:2002.11054.

[36] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, et al. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[37] Xiu-Zhe Luo, Jin-Guo Liu, Pan Zhang, and Lei Wang. 2020. Yao.jl: Extensible, Efficient Framework for Quantum Algorithm Design. *Quantum* 4 (2020), 341. https://doi.org/10.22331/q-2020-10-11-341

[38] Alexander J. McCaskey, Dmitry I. Lyakh, Eugene F. Dumitrescu, Sarah S. Powers, and Travis S. Humble. 2020. XACC: A system-level software infrastructure for heterogeneous quantum-classical computing. *Quantum Science and Technology* 5, 2 (2020). https://doi.org/10.1088/2058-9565/ab6bf6

[39] Alexander McCaskey and Thien Nguyen. 2021. A MLIR Dialect for Quantum Assembly Languages. In *2021 IEEE International Conference on Quantum Computing and Engineering (QCE)*. IEEE Computer Society, 255–264. https://doi.org/10.1109/QCE52317.2021.00043

[40] Arianne Meijer van de Griend. 2025. A comparison of quantum compilers using a DAG-based or phase polynomial-based intermediate representation. *J. Syst. Softw.* 221, C (2025), 14 pages. https://doi.org/10.1016/j.jss.2024.112224

[41] Siyuan Niu, Akel Hashim, Costin Iancu, Wibe Albert De Jong, and Ed Younis. 2024. Effective Quantum Resource Optimization via Circuit Resizing in BQSKit. In *Proceedings of the 61st ACM/IEEE Design Automation Conference* (San Francisco, CA, USA) *(DAC '24)*. Association for Computing Machinery, New York, NY, USA, Article 321, 6 pages. https://doi.org/10.1145/3649329.3656534

[42] Siyuan Niu, Akel Hashim, Costin Iancu, Wibe Albert De Jong, and Ed Younis. 2024. Effective Quantum Resource Optimization via Circuit Resizing in BQSKit. In *Proceedings of the 61st ACM/IEEE Design Automation Conference* (San Francisco, CA, USA) *(DAC '24)*. Association for Computing Machinery, Article 321, 6 pages. https://doi.org/10.1145/3649329.3656534

[43] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, et al. 2019. PyTorch: an imperative style, high-performance deep learning library. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Curran Associates Inc., Article 721, 12 pages.

[44] Anurudh Peduri, Siddharth Bhat, and Tobias Grosser. 2022. QSSA: an SSA-based IR for Quantum computing. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction (CC '22)*. ACM, 2–14. https://doi.org/10.1145/3497776.3517772

[45] Tom Peham, Nina Brandl, Richard Kueng, Robert Wille, and Lukas Burgholzer. 2023. Depth-optimal synthesis of Clifford circuits with SAT solvers. In *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*. https://doi.org/10.1109/QCE57702.2023.00095 arXiv:2305.01674

[46] Tom Peham, Lukas Burgholzer, and Robert Wille. 2023. Equivalence Checking of Parameterized Quantum Circuits: Verifying the Compilation of Variational Quantum Algorithms. In *Proceedings of the 28th Asia and South Pacific Design Automation Conference (ASPDAC '23)*. ACM, 702–708. https://doi.org/10.1145/3566097.3567932

[47] B Pirvu, V Murg, J I Cirac, and F Verstraete. 2010. Matrix product operator representations. *New Journal of Physics* 12, 2 (2010), 025012. https://doi.org/10.1088/1367-2630/12/2/025012

[48] QIR Alliance: https://qir-alliance.org [n. d.]. *QIR Specification*. QIR Alliance: https://qir-alliance.org. https://github.com/qir-alliance/qir-spec

[49] N. Quetschlich, L. Burgholzer, and R. Wille. 2025. MQT Predictor: Automatic Device Selection with Device-Specific Circuit Compilation for Quantum Computing. *ACM Transactions on Quantum Computing (TQC)* (2025). https://doi.org/10.1145/3673241 arXiv:2310.06889

[50] Aaron Sander, Lukas Burgholzer, and Robert Wille. 2024. Equivalence Checking of Quantum Circuits via Intermediary Matrix Product Operator. *arXiv preprint arXiv:2410.10946* (2024).

[51] Ludwig Schmid, Sunghye Park, and Robert Wille. 2024. Hybrid circuit mapping: Leveraging the full spectrum of computational capabilities of neutral atom quantum computers. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*. 1–6.

[52] Daniel Schoenberger, Stefan Hillmich, Matthias Brandl, and Robert Wille. 2024. Shuttling for Scalable Trapped-Ion Quantum Computers. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2024), 1–1. https://doi.org/10.1109/TCAD.2024.3513262

[53] Raphael Seidel, Sebastian Bock, René Zander, Matic Petrič, Niklas Steinmann, Nikolay Tcholtchev, and Manfred Hauswirth. 2024. Qrisp: A Framework for Compilable High-Level Programming of Gate-Based Quantum Computers. arXiv:2406.14792 [quant-ph] https://arxiv.org/abs/2406.14792

[54] Amir Shehata, Peter Groszkowski, Thomas Naughton, Muralikrishnan Gopalakrishnan Meena, Elaine Wong, Daniel Claudino, Rafael Ferreira da Silva, and Thomas Beck. 2026. Bridging paradigms: Designing for HPC-Quantum convergence. *Future Generation Computer Systems* 174 (Jan. 2026), 107980. https://doi.org/10.1016/j.future.2025.107980

[55] Seyon Sivirajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. 2020. TKET: A Retargetable Compiler for NISQ devices. *Quantum Science and Technology* 6, 1 (2020), 014003. https://doi.org/10.1088/2058-9565/ab8e92

[56] Yannick Stade, Lukas Burgholzer, and Robert Wille. 2024. Towards Supporting QIR: Thoughts on Adopting the Quantum Intermediate Representation. arXiv:2411.18682 [quant-ph] https://arxiv.org/abs/2411.18682

[57] The CUDA-Q development team. [n. d.]. CUDA-Q. https://github.com/NVIDIA/cuda-quantum

[58] Davide Venturelli, Minh Do, Bryan O'Gorman, Jeremy Frank, Eleanor Rieffel, Kyle EC Booth, Thanh Nguyen, Parvathi Narayan, and Sasha Nanda. 2019. Quantum circuit compilation: An emerging application for automated reasoning. In *Scheduling and Planning Applications Workshop*.

[59] Hanrui Wang, Pengyu Liu, Jinglei Cheng, Zhiding Liang, Jiaqi Gu, Zirui Li, Yongshan Ding, Weiwen Jiang, Yiyu Shi, Xuehai Qian, et al. 2022. Quest: Graph transformer for quantum circuit reliability estimation. *arXiv preprint arXiv:2210.16724* (2022).

[60] Robert Wille, Lucas Berent, Tobias Forster, Jagatheesan Kunasaikaran, Kevin Mato, Tom Peham, et al. 2024. The MQT Handbook: A Summary of Design Automation Tools and Software for Quantum Computing. In *IEEE International Conference on Quantum Software (QSW)*. https://doi.org/10.1109/QSW62656.2024.00013 arXiv:2405.17543

[61] Robert Wille and Lukas Burgholzer. 2023. MQT QMAP: Efficient Quantum Circuit Mapping. In *Proceedings of the 2023 International Symposium on Physical Design (ISPD '23)*. ACM, 198–204. https://doi.org/10.1145/3569052.3578928

[62] Robert Wille, Stefan Hillmich, and Lukas Burgholzer. 2022. Tools for Quantum Computing Based on Decision Diagrams. *ACM Transactions on Quantum Computing* 3, 3, Article 13 (2022), 17 pages. https://doi.org/10.1145/3491246

[63] Robert Wille, Stefan Hillmich, and Lukas Burgholzer. 2023. *Decision Diagrams for Quantum Computing*. Springer International Publishing, 1–23. https://doi.org/10.1007/978-3-031-15699-1_1

[64] Elaine Wong, Vicente Leyton Ortega, Daniel Claudino, Seth Johnson, Sharmin Afrose, Meenambika Gowrishankar, et al. 2024. A Cross-Platform Execution Engine for the Quantum Intermediate Representation. *arXiv preprint arXiv:2404.14299* (2024). arXiv:2404.14299 [quant-ph] https://arxiv.org/abs/2404.14299

[65] W. K. Wootters and W. H. Zurek. 1982. A single quantum cannot be cloned. *Nature* 299, 5886 (1982), 802–803. https://doi.org/10.1038/299802a0

[66] Ed Younis, Koushik Sen, Katherine Yelick, and Costin Iancu. 2021. Qfast: Conflating search and numerical optimization for scalable quantum circuit synthesis. In *2021 IEEE International Conference on Quantum Computing and Engineering (QCE)*.

[67] Alwin Zulehner, Stefan Hillmich, and Robert Wille. 2019. How to Efficiently Handle Complex Values? Implementing Decision Diagrams for Quantum Computing. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–7. https://doi.org/10.1109/iccad45719.2019.8942057