The Munich Quantum Software Stack

Connecting End Users, Integrating Diverse Quantum Technologies, Accelerating HPC

Lukas Burgholzer[†‡*], Jorge Echavarria[§*], Patrick Hopf[†‡], Yannick Stade[†], Damian Rovara[†],
Ludwig Schmid[†], Ercüment Kaya[†§], Burak Mete[†§], Muhammad Nufail Farooqi[§], Minh Chung[§],
Marco De Pascale[§], Laura Schulz[¶], Martin Schulz[†§], Robert Wille[†‡‖]

[†]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
[§]Leibniz Supercomputing Centre, Garching, Bavaria, Germany
[¶]Argonne National Laboratory, Lemont, Illinois, USA

{lukas.burgholzer, patrick.hopf, yannick.stade, damian.rovara, ludwig.schmid,
ercument.kaya, burak.mete, martin.schulz, robert.wille}@tum.de
{jorge.echavarria, muhammad.farooqi, minh.chung, marco.pascale}@lrz.de
lschulz@anl.gov

Abstract

Quantum computing is advancing rapidly, with notable progress in both hardware and algorithm development. However, to make quantum computing truly accessible and usable across disciplines, a comprehensive, efficient and unified software stack is essential. Such a software stack must be flexible enough to support diverse hardware platforms and evolving algorithms, while enabling programming models for a wide range of users from quantum specialists to users without deep quantum expertise. Additionally, a software stack must support dynamic resource management for efficient system utilization as well as seamless integration with classical High-Performance Computing (HPC) environments. The latter is key, as quantum systems are increasingly envisioned as accelerators within hybrid workflows, supporting various levels of integration from loosely to tightly coupled. Despite numerous discussions, conceptual proposals, and initial efforts in recent years, few tangible implementations of such full-featured software stacks currently exist.

This paper introduces the Munich Quantum Software Stack (MQSS)—a modular, open-source, and community-driven ecosystem designed to support hybrid quantum-classical workflows and applications. We present the multi-layered architecture of MQSS, which enables a seamless execution of high-level user applications on heterogeneous quantum hardware back-ends, while also

supporting the coupling with the corresponding classical workloads. Key components include front-end adapters for popular frameworks as well as new programming approaches, an HPC-integrated scheduler, a powerful MLIR-based compiler infrastructure, and a standardized hardware abstraction layer known as the Quantum Device Management Interface (QDMI).

While still under active development, the MQSS already now offers sophisticated concepts and open-source implementations that lay the foundation for a robust quantum computing software stack, with a forward-looking design that also considers the future requirements of fault-tolerant quantum computing, such as support for various qubit encodings and mid-circuit measurements. More information on the MQSS is available at

- https://github.com/Munich-Quantum-Software-Stack
- https://www.munich-quantum-valley.de/research/research-areas/mqss

CCS Concepts

• **Theory of computation** → **Quantum computation**; • **Computing methodologies** → *Distributed computing methodologies*; • **Computer systems organization** → *Heterogeneous (hybrid) systems*; • **Software and its engineering** → *Software infrastructures*.

Keywords

quantum computing, software stack, HPC, integration, accelerator, heterogeneous architectures, hybrid workflows

---

*Both authors contributed equally to this work.

1 Introduction

Quantum Computing (QC) promises to revolutionize computation by tackling problems intractable for even the most powerful classical supercomputers. Particularly in the realm of High-Performance Computing (HPC), quantum computers are envisioned as specialized accelerators, similar to Graphics Processing Units (GPUs) or Field Programmable Gate Arrays (FPGAs), that can efficiently solve specific computationally hard sub-problems in science and industry. However, realizing this potential requires more than just building quantum hardware. It requires a concerted and collaborative effort across a wide spectrum of disciplines. No single research group or company can overcome the multifaceted challenges alone. Instead, a successful quantum ecosystem relies on the synergy of: 1) experimental physicists and electrical engineers who provide the hardware; 2) mathematicians and theoreticians who lay the theoretical foundations for quantum algorithms and error correction; 3) computer scientists who design and implement the needed hardware and software environments; 4) software engineers who maintain and support the required software ecosystem suitable for production usage; and 5) domain experts from industry and science who identify and formulate compelling use cases.

Only in such a synergetic environment, can we create the needed sophisticated software stack, as the critical link that connects the diverse domains and stakeholders. Its role is to translate high-level problem descriptions from domain experts into low-level, hardware-specific instructions for the quantum computer. Furthermore, it must seamlessly integrate Quantum Processing Units (QPUs) into existing classical HPC infrastructures (a concept referred to as HPCQC integration), thereby enabling the hybrid quantum-classical workflows that are expected to dominate the field for the foreseeable future.

The necessity of such a software stack is underscored by the emergence of large-scale quantum initiatives worldwide. While none of these initiatives are primarily focused on software, they all recognize the importance of a robust software ecosystem. The Munich Quantum Valley (MQV), for example, is such a large-scale initiative that brings together over 400 researchers and practitioners from academia, industry, and government to create a comprehensive quantum ecosystem, where the software stack is a core pillar alongside hardware development.

This paper introduces the Munich Quantum Software Stack (MQSS), an open-source, community-driven effort to create such a software ecosystem. The MQSS is designed to be a modular, efficient, and extensible platform that bridges the gap between high-level quantum applications and the intricacies of diverse quantum hardware back-ends. It provides a comprehensive set of tools for compilation, optimization, and execution of quantum circuits, while also facilitating the deep integration with classical HPC systems for hybrid workflows and applications, which are required for practical quantum advantage. The MQSS's capabilities were first demonstrated when it successfully achieved HPCQC tightly-coupled integration, a first between on-premise HPC and QC systems. To achieve this, SuperMUC-NG—the flagship HPC system at the Leibniz Supercomputing Centre (LRZ), one of Germany's three national HPC centers—was coupled with Q-Exa, Germany's first quantum demonstrator: a 20-qubit IQM Radiance superconducting device.

This integration enabled the execution of a VQE application, resulting in one of the world's first HPC-quantum computing (HPCQC) demonstrations [1].

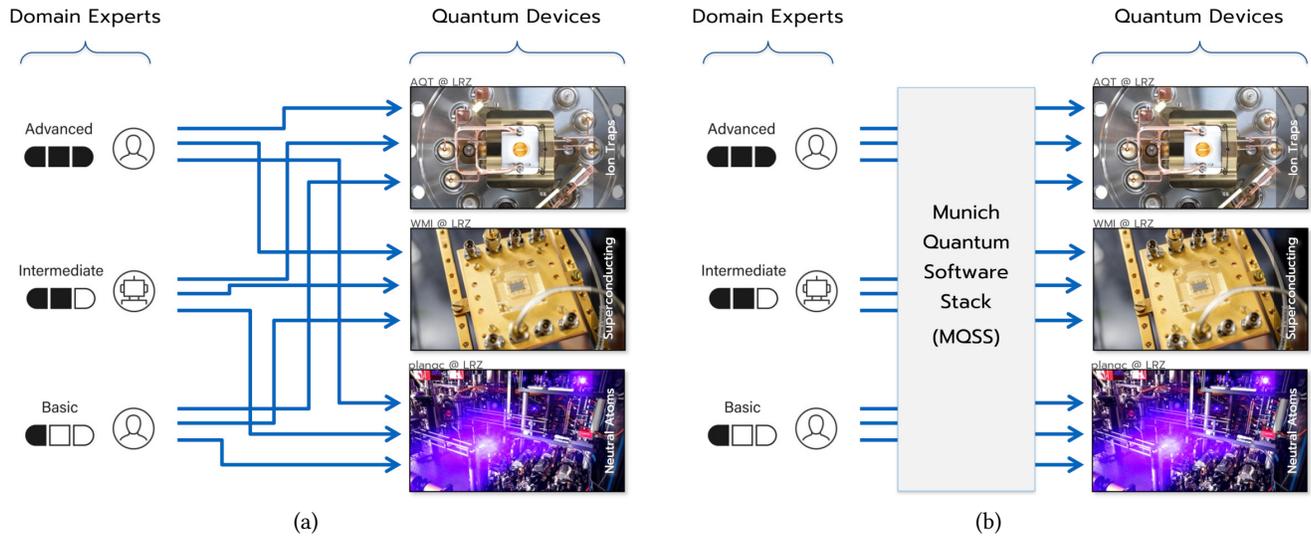## 1.1 The Need and Challenges for a Quantum Software Stack

In classical computing, we typically expect and rely on a mature and reliable software stack as an essential component of the ecosystem, providing layers of abstraction that allow developers to productively use complex hardware[1]. The same necessity holds for quantum computing, and even more so in the context of hybrid HPCQC, where quantum processors are envisioned to function as accelerators, analogous to GPUs or FPGAs. A combined software stack, with shared components across both ecosystems, is the critical infrastructure that enables this vision of hybrid quantum-classical computing [7, 25, 28, 29]. Moreover, users, both humans and automated tools, differ in expertise, requirements, and capabilities. As shown in Figure 1.a, currently, many existing stacks address these user types separately for each quantum technology. Figure 1.b, on the other hand, shows the MQSS's unified solution that supports all user domain–technology permutations.

A stack of this nature must be modular, flexible as well as efficient. It needs to support a diverse and evolving landscape of quantum hardware, from superconducting circuits to trapped ions, from neutral atoms to photonic systems, and more [2, 8, 19, 20]. Each of these modalities has its own unique characteristics, constraints, and performance metrics, which must be abstracted away to provide consistent access. The stack must also be capable of accommodating a wide range of current and future quantum algorithms, as well as hybrid applications that combine the usage of classical and quantum resources. To make quantum resources accessible to users who are not quantum physicists, the stack must offer high-level abstraction layers, covering the complex workflow for given problem descriptions to their execution on a back-end. Further, quantum computing adds new requirements, such as dynamic Just-In-Time (JIT) compilation, as the software stack must adapt to real-time changes in the hardware's state as well as performance characteristics to optimize execution.

The architectural principles of such a stack must be flexible enough to support a wide range of deployment models from on-premise setups and cloud access to QCs. Only this will allow for a broad adoption and a reuse of compiler components, programming models and back-end solutions. In this paper, though, we focus on a deep integration with on-premise HPC systems. For one, this represents one of the extreme cases stressing the software stack the most, and, second, it enables tightly-coupled hybrid computations, which are expected to have the most immediate impact for real-world use cases. This integration is key to unlocking the full potential of quantum accelerators, moving beyond simple, loosely-coupled cloud access.

Finally, the software stack must be designed with the future in mind, providing a clear path from today's Noisy Intermediate-Scale Quantum (NISQ [21]) devices to tomorrow's fault-tolerant

---

[1]Typically in the form of an operating system environment, such as Linux or Windows. Another example is the E4S stack stemming from the US Exascale Computing Program (ECP), see https://e4s.io.

Figure 1: a) Traditional view: many users, many systems, many stacks. b) Vision going forward: decoupling front-end and back-end in a single stack via shared IRs.

machines [6, 24, 30]. This will require flexibility on representing encodings, supporting mid-circuit measurements, enabling backends with hardware support, as well as supporting bi-directional state transformation up and down the stack.

The development of a comprehensive quantum software stack is fraught with challenges that stem from the unique nature of quantum computing and its ecosystem. As a relatively new field, quantum computing lacks the established standards and deep-rooted abstractions that we naturally rely on in classical computing. The underlying hardware is incredibly diverse, with various physical realizations of qubits—such as superconducting circuits, trapped ions, neutral atoms, or photons—each presenting distinct properties, connectivity, and error characteristics [4, 8, 26, 33]. Even systems based on the same underlying qubit technology can vary wildly in their architecture and performance. This heterogeneity makes it difficult to create a unified software layer that can effectively target all platforms.

Furthermore, the quantum computing community is a melting pot of different expertise. Software development has often been driven by physicists focused on the underlying quantum systems, rather than by computer scientists with expertise in building scalable and maintainable software. On the user side, domain experts from fields like chemistry or finance want to solve their specific problems, but are often neither experts in quantum computing nor in HPC. Conversely, HPC experts may lack a deep understanding of quantum mechanics, and quantum computing experts may not be familiar with the intricacies of large-scale classical resource management. These conditions create significant communication gaps between stakeholders, which need to be bridged by a concerted effort to establish a common language and shared understanding. This is essential for creating tools that are both powerful and usable.

1.2 Our Approach: The Munich Quantum Software Stack

The MQSS is our answer to these challenges. Developed by computer scientists, but in close connection with all other stakeholders and their expertise, it provides both a conceptual blueprint, which covers the fundamental design of hybrid quantum/classic software stacks, and a tangible, open-source software stack with a concrete implementation that can be deployed to and used by a wide range of users, systems, and deployment environments.

Our approach is centered on several key principles that directly address the challenges outlined above:

- To tackle hardware **heterogeneity**, the MQSS features a **modular architecture** designed for extensibility and flexibility, allowing for the seamless integration of new components and support for a growing number of quantum platforms.
- To bridge the gap with classical computing, it is built from the ground up for **deep integration with classical HPC**, enabling the tightly-coupled hybrid workflows that are essential for quantum acceleration.
- To foster a collaborative ecosystem and overcome the lack of standards, we develop and promote **open interfaces and specifications**, such as the Quantum Device Management Interface (QDMI [38]).

This commitment to openness and interoperability stands in contrast to other, more monolithic, single-vendor stacks and ensures that the MQSS can serve as a versatile and future-proof platform for the entire quantum community, empowering users and developers alike. These principles are not just abstract concepts, but concrete solutions that guide the development of the MQSS, ensuring it is a robust, usable, and impactful tool for advancing quantum computing.

The MQSS is actively being developed and deployed, supported with Continuous Integration (CI) and Continuous Deployment (CD) techniques, and its components are successively being released as open source and integrated with external components. This makes the MQSS a community-driven effort that has the ability to bring together a wide range of users, vendors, centers, and compiler developers, which all can base their work on the open platform and then contribute components back to the overall ecosystem.

2 Architectural Overview

This section provides a high-level overview of the MQSS. We begin by outlining the intended use cases and target user groups of the stack to illustrate the problems it aims to solve. We then describe the overall architecture of the MQSS, detailing its main components and how they interact to form a cohesive and powerful software ecosystem for HPCQC integration. This overview sets the stage for a more detailed look at the individual components in the following section.

2.1 Use Cases and Target Groups

The MQSS is designed to serve a diverse set of users and enable a wide range of applications and use cases, bridging the gap between the abstract potential of quantum computing and its practical implementation within an HPC context. Our primary goal is to make quantum resources accessible and useful to several key target groups:

- **Domain Experts** from fields such as material science, chemistry, operational research, or finance who possess deep knowledge of their problem domain, but are not necessarily experts in quantum computing. For them, the MQSS provides high-level abstractions that allow them to formulate their problems without needing to manage the complexity of the underlying quantum hardware.
- **HPC Experts and Users** who are accustomed to heterogeneous computing environments and wish to leverage quantum computers as another type of specialized accelerator, much like a GPU or an FPGA. The MQSS facilitates this by integrating QPUs as offload devices with classical HPC resources, enabling various levels of hybrid workflows from loosely to tightly-coupled [7].
- **Quantum Algorithm and Software Developers** who require a flexible, modular platform to design, test, and benchmark new algorithms, compilation strategies, and software tools. The open-source and extensible nature of the MQSS provides an ideal environment for this kind of research and development.
- **Hardware Providers and Physicists** who develop and operate quantum computing hardware. For this group, the MQSS offers (quasi-)standardized interfaces, like QDMI, for integrating new devices into the ecosystem, allowing them to expose their systems to a broader user base and receive valuable feedback. At the same, it also enables low-level access through the same infrastructure to facilitate low-level experiments and manual access.
- **System Administrators and User Managers** who are responsible for operating QC or HPCQC systems. For this

group, the MQSS offers introspection interfaces to understand system health and status, as well as interfaces to integrate issues, like authentication and quota management, with the matching HPC components.

The MQSS supports this wide range of users and use cases, merging the access into a single unified infrastructure. Further, it does so by not "just" supporting the execution of quantum programs, but by also offering a native integration with existing classical systems, in particular High Performance Computing (HPC) systems. This is critical for the success of any quantum ecosystem, because QC systems alone cannot support themselves, as they need classical systems for staging and control, and because they cannot support arbitrary workloads, as quantum algorithms only target very specific computational problems.

As a consequence, QC system will for the foreseeable future operate as quantum accelerators, where a QPU tackles a computationally hard subtask or kernel of a larger problem that, in its whole, still continues to be orchestrated by a classical HPC system. The result is a powerful hybrid workflow exploiting and combining the respective strengths from both worlds.

Executing such a hybrid workflow, though, involves a complex interplay of data flow, resource management, and scheduling. For example, a user's quantum program must be compiled for a specific device, the device must be available, and the classical and quantum computations must be synchronized., while at the same time minimizing idle times on both the QC and HPC system.

A concrete instance of the MQSS is deployed at the Leibniz Supercomputing Centre (LRZ), which currently hosts many of the quantum systems available via the MQV. This deployment, which marks the first HPCQC integration in Germany, as well as one of the first co-located on-premise integration world-wide, demonstrates our stack's versatility in a production environment.

The MQSS offers multiple access paths—including a web portal (the "MQSS Dashboard"), command-line access, and tightly integrated hybrid access from within the HPC environments.
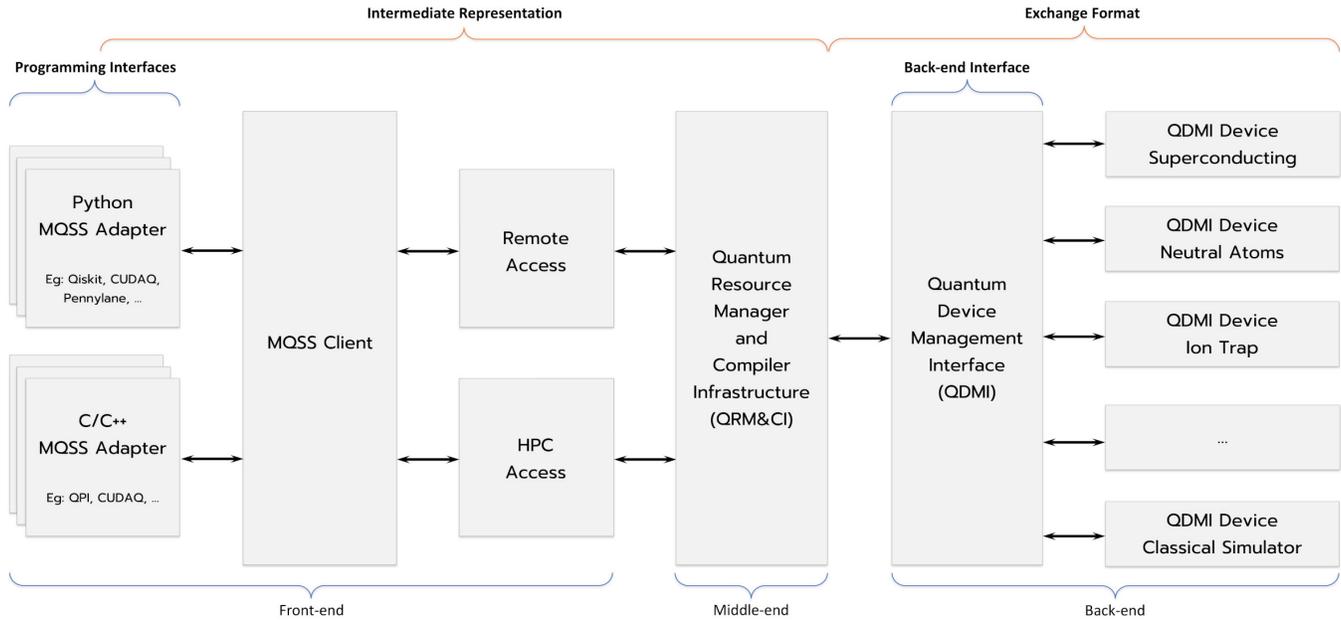
2.2 The MQSS Core Architecture

To realize the use cases described above, the MQSS is built as a modular, multi-layered architecture, implemented as a series of connected components. Figure 2 sketches the basic architecture, forming the bridge from users (left) to systems (right).

At its heart, the MQSS consists of a common and agnostic infrastructure, built on top of established open source principles, which can then be instantiated in many scenarios and customized via plugins operating on standardized interfaces. This design philosophy emphasizes flexibility and extensibility, allowing components to be replaced or upgraded and enabling many-to-many connections between the layers.

The key architectural layers, which are explicitly designed to support the different user groups, include:

- **Front-end Layer:** This is the highest level of abstraction, providing user-facing interfaces and decoupling them from the underlying runtime system. It includes support for popular quantum programming frameworks, like Qiskit and

Figure 2: Overview of the MQSS and its components, connecting end-users and their high-level problem descriptions (left) to a diverse set of quantum hardware back-ends, as well as simulators (right).

PennyLane, domain-specific tools that allow experts to formulate problems in a familiar language, and new programmatic interfaces, like the Quantum Programming Interface (QPI; [14]) for hybrid environments.
- **Middle-end Layer:** This layer is responsible for embedding the quantum workflows into the classical HPC environment. It handles the scheduling of hybrid jobs, manages the allocation of quantum resources, and orchestrates the communication between them. It also contains the powerful and flexible compiler infrastructure that translates high-level quantum circuit descriptions into executable instructions optimized for a specific hardware back-end.
- **Back-end Layer:** This layer provides a unified interface to the underlying quantum hardware. Through a dedicated interface, called QDMI, it abstracts access to the specific details of each back-end, allowing the higher levels of the stack to interact with different QPUs in a standardized way. It also exposes crucial information about the hardware, such as calibration data and Figures of Merit and Constraints (referred to as FoMaCs in the remainder of the text), which are essential for hardware-aware compilation and optimization [10].

This layered architecture ensures that the MQSS can evolve with the rapidly changing quantum landscape, providing a stable, yet adaptable platform for HPCQC integration.
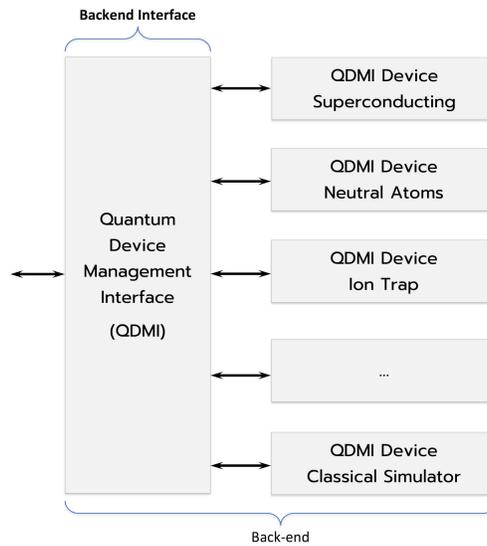
3 Individual Components of the Software Stack

This section details the key components of the MQSS, looking at the stack bottom up, adding layers of abstraction on top of the hardware systems up to the point where the MQSS directly supports domain users and their applications.

At the bottom of the stack, the MQSS features a unified interface for running individual quantum kernels (using gate or pulse notation [5]), delivering results, and querying both static and dynamic information of QC systems and their physical environment. The latter information is then aggregated into higher-level data in the form of Figures of Merit and Constraints (FoMaCs), which summarize the status, the health, and the system configuration. This data is then fed to other layers of the stack to influence and optimize compilation and scheduling decisions, as well as to inform users and administrators.

Building on this, we introduce the Quantum Resource Manager & Compiler Infrastructure (QRM&CI) that handles the entire compilation, resource management and scheduling.

Finally, we present the front-end adapters, which provide various levels of abstraction, from familiar—typically low-level—SDKs, like Qiskit, to hybrid HPCQC languages, offload approaches and high-level Domain Specific Languages (DSLs) and libraries. These enable users to leverage the full power of the underlying stack, based on their application requirements and prior knowledge.

Figure 3: Structure of the MQSS back-end, featuring the implementation of the QDMI interface as well as a series of system device plugins.

The following sections summarize the main concepts in these layers. Links to more details and actual (open-source) implementations are provided where available for a deeper understanding.

3.1 Quantum Device Management Interface

As depicted in Figure 3, the lowest layer of the software stack interfaces directly with the hardware devices and their control systems. The level of abstraction needed at that level can depend on the targeted device, its control structure and internal logic, its type of access (e.g., gate or pulse-based), and its ability to execute code itself. This can reach from directly accessing the control logic to managing autonomous devices, e.g., in the form of control processors with conditional logic, with their own complex firmware stack.

The Quantum Device Management Interface (QDMI) is designed to bridge these differences and offers a standardized connection to QC devices covering multiple access options and supporting different types of firmware stacks. It is split into three main components (see Figure 4): 1) a session management enables the software stack to work with multiple users on multiple devices in a safe and reliable manner, 2) a job submission and management interface to send jobs (in one of many forms) to a device, and 3) a query interface to extract device properties and constraints, e.g., live fidelity values or concrete gate sets supported by the device.

Its primary role is to abstract the hardware-specific details of device control and data acquisition, enabling the submission and management of quantum kernels in a uniform manner. Kernels are represented in an exchange format, which is produced directly by the compiler, as explained below, and then handed on to the respective back-end for final transformation on the vendor side (if needed) and execution. For the latter, QDMI defines a simple queue management (incl. submission, asynchronous progress, and job status queries) associated with clearly defined and separated

sessions, enabling the execution of multiple kernels from multiple users on multiple devices.

Additionally, QDMI allows software tools to retrieve device properties. This includes static information, like the number of qubits or available gate sets, as well as dynamic information, like gate fidelities, calibration data, or data from environmental sensors. This information can then be analyzed and used by compilers, the underlying runtime or the user to understand system properties and adjust their usage accordingly.

On the implementation side, QDMI can be split into two main parts: the interface specification describing the interface towards both the user and the hardware vendor, and a core implementation that implements the session, job and query management. This implementation can then be customized with plugins, each representing the needed control software for individual or families of quantum devices. This structure allows the software stack to access the devices via a firmly defined stack-facing interface, while vendors can add support for QDMI by implementing a thin plugin based on the hardware-facing interface. This approach not only supports easy expandability, but also supports portability across software stacks and quantum technologies.

Looking forward, this modular and portable approach opens QDMI for further extensions. In particular, it easily supports different levels of abstraction from gate-based to pulse-based circuits, it enables both directly controlling execution to code shipping to "close to metal" inside the quantum device, all with the same interface. Further, it will be possible to integrate virtual devices, external software stacks, as well as ISV software, or remote execution on devices, without changes to the interface.

For more information on QDMI, including its specification and implementation, please refer to:
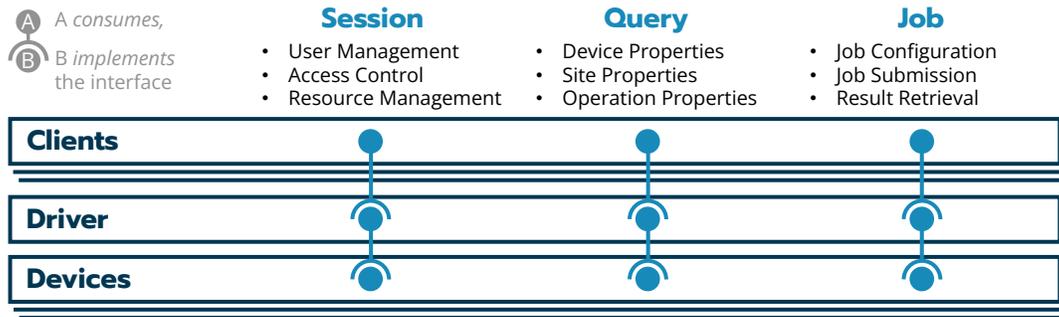
- https://github.com/Munich-Quantum-Software-Stack/QDMI
- https://munich-quantum-software-stack.github.io/QDMI

Figure 4: The three main components of the QDMI: Session Management, Job Submission, and Query Interface.

3.2 Quantum Resource Manager & Compiler Infrastructure

The Quantum Resource Manager & Compiler Infrastructure (QRM&CI), shown in Figure 5, is the central runtime component responsible for orchestrating the execution of quantum kernels. It combines a dynamic compiler infrastructure with scheduling and resource management, acting as the bridge between user input in the form of quantum kernels on one side and execution on the quantum systems on the other. It receives kernels in some, likely high-level, Intermediate Representation (IR), transforms and lowers them to execution-ready kernels in a, likely low-level and system-specific IR, allocates the needed quantum resources, and then uses QDMI to execute the kernel. Once complete, it reads the result, again via QDMI, transforms the results into the form requested by the user, and returns it via the initial programming abstraction.

The QRM&CI is composed of several key components, including support libraries for Figures of Merit and Constraints (FoMaCs), a compiler infrastructure, and a scheduler, which are detailed in the following subsections.

3.2.1 Figures of Merit and Constraints Support Libraries.
While QDMI provides the interface to query raw hardware and system properties, this information is low-level and hard to use directly. It must first be analyzed and refined, a task that every consumer of the data needs and hence should not be replicated. This task is, therefore, relegated to a set of libraries, which use the QDMI-provided raw data and create concrete Figures of Merit and Constraints (FoMaCs). This is particularly helpful for complex, dynamic, and time-sensitive information, such as gate fidelities, coherence times, and readout error rates, which are aggregated from the low-level data provided by the hardware.
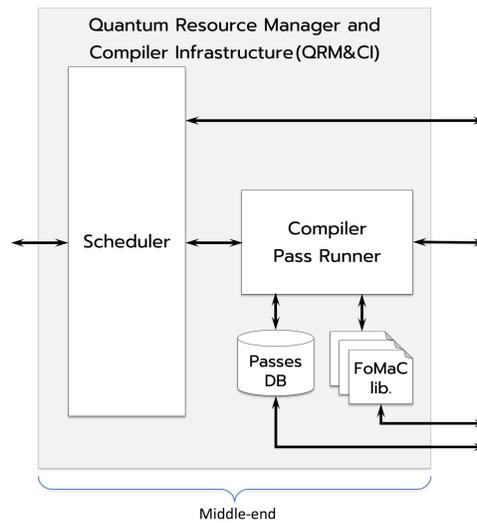
Such live, aggregated telemetry is critical for the hardware-aware compilation and optimization and, hence, is made available by a set of FoMaC libraries to the rest of the MQSS software stack. For example, the compiler can use up-to-date fidelity data to choose the best physical qubits for a computation or to adjust its optimization strategy to avoid particularly noisy gates. The FoMaC libraries also play a role in monitoring the broader operational environment. They can aggregate data from environmental sensors—provided through QDMI—that track factors like temperature and magnetic

fields, which can influence the performance of sensitive quantum hardware. By providing this comprehensive and dynamic view of the hardware's status, FoMaC libraries enable the software stack to make intelligent, data-driven decisions that maximize the performance and reliability of the targeted quantum systems.

A simple example implementation of a FoMaC library is provided as part of the QDMI repository. A more complex example is the sys-sage library, which can be used to capture, analyze and track topology and reliability data from quantum systems [18]. sys-sage was originally developed for HPC systems to track classic node topologies and performance properties, and was extended to include quantum data, offering a unified view of HPC and quantum system information. This demonstrates how the MQSS supports the integration of the two technologies.

3.2.2 Compiler Infrastructure.
One of the key elements of the QRM&CI is its compilation infrastructure (in the following referred to as compiler), which translates input kernels to low-level, hardware-specific sequences of instructions, which can be consumed by QDMI and its back-end plugins. For this, the compiler uses a family of Intermediate Representations (IR) to describe the quantum circuits and applies a series of compilation passes to these kernels to transform them step by step and gradually lower them from a likely high-level and generic input circuit description to likely low-level and execution-ready code. In this process, each pass can apply optimizations for the circuits, be guided by the FoMaC input, verify properties of circuits for debugging, select appropriate devices based on circuit properties, or transpile circuits to the available native gate sets, as queried via QDMI.

To manage this complexity, to support a compatible family of IRs, to maintain flexibility, and to reuse existing work, the MQSS leverages the Multi-Level Intermediate Representation (MLIR) framework [15], developed for and widely used in classical systems. MLIR allows for the creation of a modular and extensible compiler pipeline where different levels of abstraction are represented by distinct IR "dialects". The framework further supports extensibility via custom dialects and transformation passes through its plugin system [9]. This allows the MQSS to be both back-end-agnostic at its higher levels and highly specific and optimized at its lower levels, all the way to low-level representations, like QIR [22, 32], while at the

Figure 5: The structure of the MQSS middle-end, featuring the QRM&CI and its internal components implementing the compiler.

same time supporting low-level intrinsics and annotations where helpful in high-level abstractions. It also allows the combination of dialects of IRs for classical and quantum kernels, enabling a truly hybrid compilation and cross-paradigm optimization. Further, it supports different levels of abstractions on the back-end side, easing the integration of new back-ends.

The compiler is designed as part of the runtime system to be highly dynamic, using the live data provided by the FoMaC support libraries (see subsubsection 3.2.1) to inform its optimization choices. The actual passes are picked from a library of passes based on the matching IRs, the targeted systems, as well as user directives, similar to compiler optimization techniques from classical computing. To further support pass selection and optimization, the MQSS can also employ AI-based pass selection, which can be integrated via a separate pass selection interface. Tools, like the MQT Predictor tool [23], can use this to select a dedicated compilation pass sequence for a given circuit based on its properties and the current state of the available hardware.

The modular structure of the MQSS compiler and the distinction between a generic infrastructure with a customizable pass selection mechanism, coupled with an extensive and expandable pass library (with MQSS and external contributions) offers the ability to grow and extend the MQSS capabilities, as well as to reuse existing techniques and software. Specifically, the MQSS leverages existing passes from frameworks such as Qiskit, but also incorporates individually developed passes for superconducting systems [16, 37, 39, 40], neutral atoms [33, 34], and trapped ions [27, 35], such as those available from the Munich Quantum Toolkit (MQT) [3, 36].

For more information on the MQSS compiler infrastructure, including a collection of passes, please refer to:
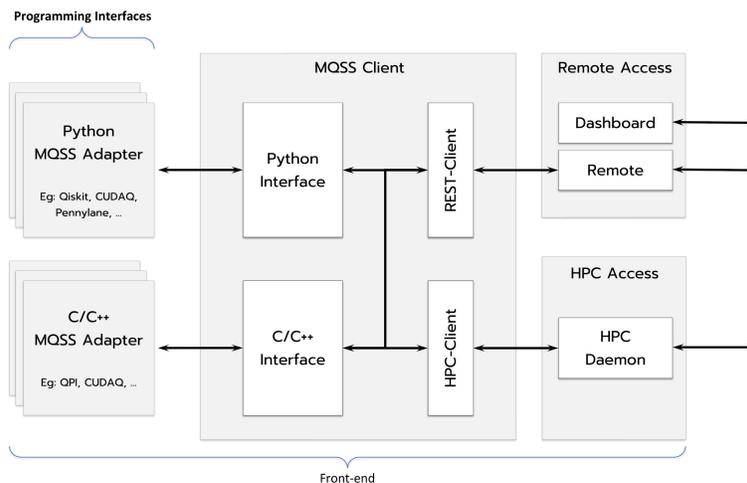
   https://github.com/Munich-Quantum-Software-Stack/
   MQSS-Passes-Suite

3.2.3 Scheduling and Resource Management. To effectively execute quantum circuits on quantum systems, which are in most cases a rare resource, a software stack must include the needed resource management. This will enable the user and/or the system to select an appropriate QC system and then schedule a kernel to that system in a way that both reduces latency and improves throughput. Both aspects are potentially highly dependent on kernel and system properties, as well as the current live state of the system.

Such a scheduler will naturally have to be bi-level. At the upper level, the QRM&CI selects a device for execution and then coordinates execution with simple execution and shot management inside the QDMI back-ends. While the latter is kept explicitly simple and typically only uses single priority queue for each device, the former scheduler is intended to be more sophisticated. Through explicit interfaces, plugins can be loaded to support adaptive device selection, e.g., using AI methods [23] as well as performance prediction to estimate workload and system occupancy.

3.2.4 HPC Integration. For the integration into classical systems, in particular into HPC environments with their own resource management and scheduler, the topic of resource management becomes even more critical. Here, the QRM&CI scheduler additionally has to coordinate with the HPC scheduler, such as Slurm [13], that manages the HPC component of the hybrid workload, resulting a three-level scheduling system. The goal of this scheduler is to minimize idle times for both the HPC and the QC system, while at the same time selecting the most suited quantum system for the supplied kernel.

In order to fulfill these requirements, the QRM&CI requires a sophisticated scheduling policy as well as tight coordination with the HPC scheduler, the quantum scheduler, as well as the system status via QDMI. Further, it must understand the characteristics of the input circuit as they would map to a specific technology with its respective native gate set, fidelity and execution speed. Based

Figure 6: Structure of the MQSS front-end, featuring the MQSS Client connecting end-user programming to the rest of the stack.

on that, it must be able to predict execution times to understand resource requirements and length of reservations, so kernel executions can be scheduled, possibly across jobs and users, in a way that the QC resource is available when needed, but used otherwise when not. This requires the ability to model and predict the performance of both the runtime stack and the actual quantum execution in the respective back-end. For this, the MQSS offers the needed interfaces to add and to customize such models, and with that to optimize overall system utilization.

3.3 Front-end Adapters and MQSS Client

Facing the user and/or the application, the MQSS Adapters, as shown on the left side of Figure 6, provide the primary entry point for defining and executing quantum kernels, either via direct coding or via automated tools. These adapters, therefore, allow users to leverage the entire stack using a wide range of APIs, abstraction libraries, DSLs or programming models. From a technical point of view, they decouple the programming model from the underlying compilation and runtime stack, a concept well-known from classical computation, but so far not widely used for quantum programming.

Recognizing that users as well as existing tools have diverse backgrounds and prefer different programming paradigms, the MQSS is designed to be compatible with a wide range of popular quantum software development kits and interfaces, each connected with its own, lightweight front-end adapter, but connecting to the same stack (see Figure 1). This compatibility ensures that developers can continue to work in the environments they are most comfortable with, without needing to learn a new proprietary language to access the features of the MQSS. The modular design also allows for the extension to new and emerging frameworks.

We currently provide robust support for widely-used frameworks such as *Qiskit*, *PennyLane*, and *CUDA-Q*. This support allows the large communities built around these tools to seamlessly target

MQSS-enabled back-ends, benefiting from the underlying compilation, optimization, and execution management capabilities. Built on top of a uniform MQSS Client interface (center of Figure 7), these adapters provide the final layer of abstraction, shielding the user from the complexities of the underlying hardware and software infrastructure. The MQSS Client automatically detects whether a submission originates from within the HPC environment or from a remote location and routes it accordingly. This also provides a hook for external tools, such as those at other data centers, to submit jobs to any given instance of the MQSS.
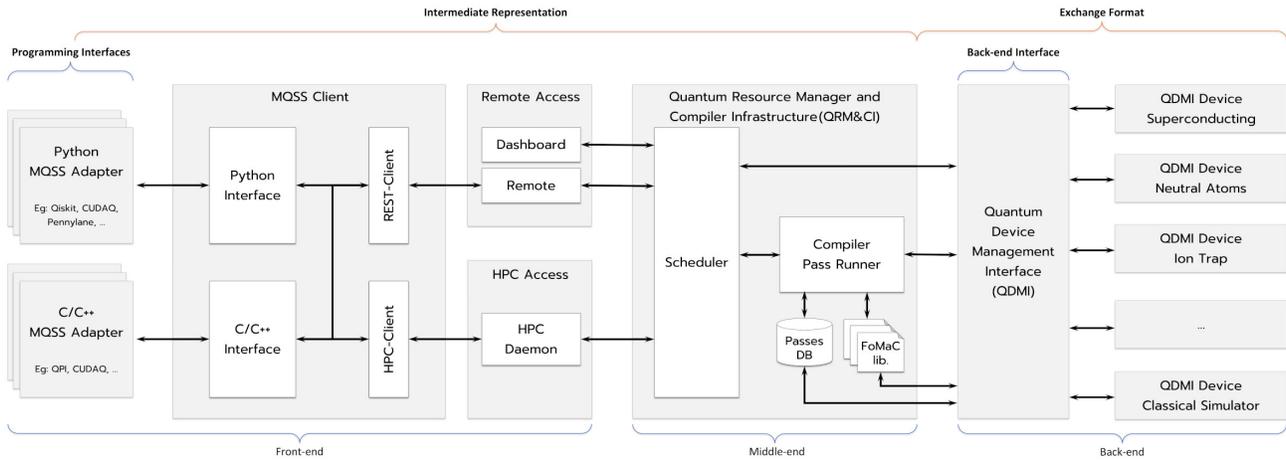
In addition to supporting the Python-based frameworks that are prominent in quantum computing, the MQSS offers programming models tailored to HPC environments, such as the Quantum Programming Interface (QPI; [14]), enabling a more natural integration into native HPC languages, like C and C++, for developers accustomed to that ecosystem.

By supporting multiple adapters, the MQSS also enables users to take advantage of the distinctive strengths and specialized features offered by different libraries, thereby expanding the capabilities available to them. For links and documentation to the available adapters, see:

📄 https://munich-quantum-software-stack.github.io/
   MQSS-Interfaces

4 Instantiations and Deployment

The MQSS has moved beyond conceptual design to a deployed, production-ready system. Its architecture has been validated in a live HPC environment as part of the MQV initiative, where it serves as the unified access point to a heterogeneous set of quantum devices. Operating within the production infrastructure of one of Europe's leading supercomputing centers for HPCQC, the stack has already been adopted by a diverse user base through multiple access modalities—from a user-friendly web portal to tightly-coupled hybrid access from within HPC jobs. Experiences gathered during

Figure 7: Low-level overview of the compilation and execution pipeline of the MQSS.

early user phases have provided valuable feedback for refining components and confirming that the stack can meet the demands of both research prototypes and real-world workloads.

To make this more tangible, a production instantiation is described next in the form of a representative end-to-end workflow, followed by a summary of a key study conducted during the early user phase.
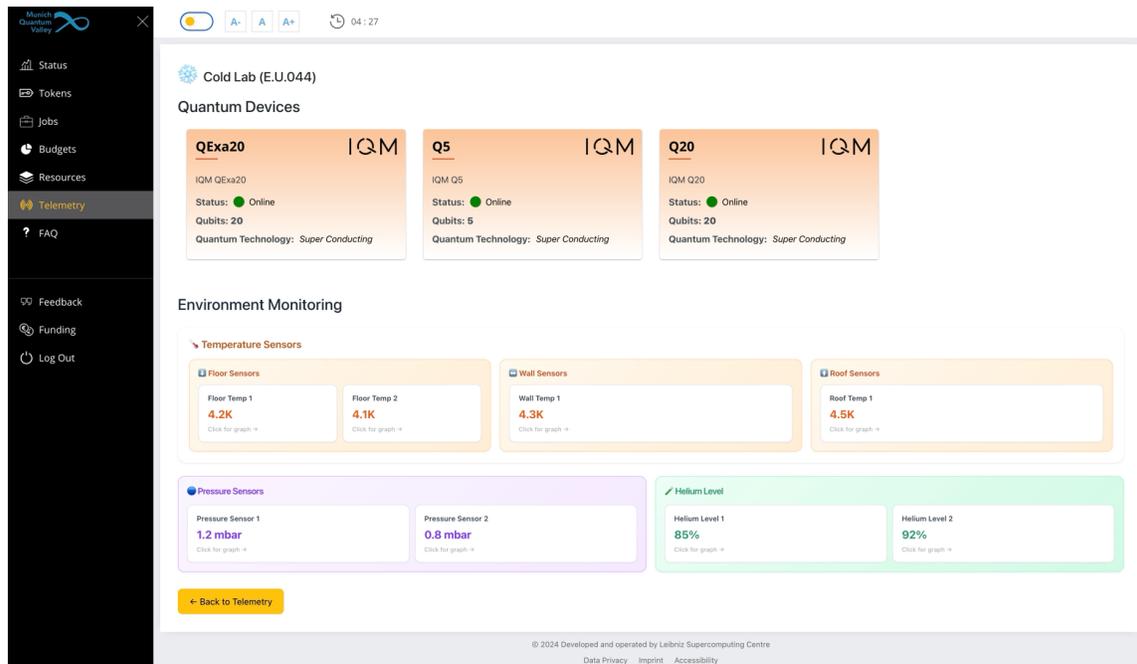
4.1 Production Instantiation at LRZ

The MQSS is deployed at the Leibniz Supercomputing Centre (LRZ) as the single entry point to a heterogeneous set of quantum back-ends, including trapped-ion systems from AQT and superconducting systems from IQM. In this setup, the MQSS components are deployed across different network zones to ensure security and manageability. The user-facing components, such as the MQSS Dashboard (a web portal) and the MQSS Client endpoints, are accessible from the general internet and the HPC login nodes, respectively. The middle-end layer of the stack, including the Quantum Resource Manager & Compiler Infrastructure, runs in a dedicated, secured zone. The QRM&CI communicates with the various quantum systems via QDMI, which provides a standardized interface to the vendor-specific control software running in isolated, hardware-proximate environments. This distributed deployment demonstrates the flexibility of the MQSS architecture, allowing it to be adapted to the specific security and networking requirements of a production HPC center.

4.2 End-to-End Workflow through the MQSS

Once installed and deployed, users can access the system in multiple ways that map directly onto the layers shown in Figure 2. The following representative end-to-end workflow demonstrates the interplay of these components in action, as shown in greater detail in Figure 7. The numbered steps below correspond to the labels in that figure.

Consider the scenario of a user who wishes to execute a hybrid quantum-classical algorithm.

(1) **Job Submission:** The user defines and implements their kernel in their favorite language or model and then submits it through corresponding Front-end Adapter. For interactive use, they might use a Python SDK, like Qiskit from a Jupyter notebook connected to the MQSS Dashboard (see Figure 8). For a large-scale hybrid workflow, they could code the kernel as a C++ application using QPI and execute that from a Slurm login node at LRZ. The MQSS Client receives the submission, handles authentication, and forwards the quantum part of the workflow to the QRM&CI.

(2) **Scheduling and Device Selection:** The QRM&CI's scheduler receives the quantum kernel. It first analyzes the circuit's requirements (e.g., qubit count, connectivity needs). Then, it queries the FoMaC Libraries for the current status and capabilities of all available QPUs. Based on this information and a user-defined (or default) policy, the scheduler performs a Pareto-optimal selection, identifying the most suitable back-end that balances various factors like queue time, expected fidelity, and execution speed. This integrated selection and scheduling step ensures that resources are allocated efficiently.

(3) **Device-Agnostic Compilation:** Once a set of candidate devices is known, the compiler infrastructure applies a series of device-agnostic optimization passes. A pass selector, potentially guided by machine learning heuristics, chooses transformations like gate cancellation, commutation-based reordering, or gate fusion to reduce the overall complexity of the circuit.

(4) **Device-Specific Compilation:** With a final target device selected by the scheduler, a second set of passes is applied. It uses detailed, live information from FoMaC Libraries (e.g., the native gate set, coupling map, or specific gate fidelities) to perform basis translation, qubit placement (mapping), and routing.

Figure 8: The MQSS Dashboard at LRZ, offering a web-based interface for job submission, monitoring, and management. Users can oversee token and budget usage, manage jobs and resources, and access detailed environment telemetry.

(5) **Code Generation:** The final, optimized circuit is lowered to an execution-ready representation. This could be a low-level IR, like QIR, or a vendor-specific format, potentially including pulse-level definitions, if supported by the back-end.

(6) **Execution:** The executable kernel is dispatched via QDMI to the selected back-end. QDMI manages the asynchronous execution and retrieves the results once the job is complete.

(7) **Result Handling:** The MQSS Client receives the raw measurement outcomes (e.g., bitstrings) from QDMI. It can perform initial post-processing, such as building histograms or applying measurement error mitigation, before returning the refined results and optional metadata (e.g., device used, calibration snapshot) to the user's application, again via the Front-end Adapter, enabling reproducibility and further analysis.

The example illustrates the focus on extensibility and modularization, with each stage in the pipeline being policy-driven and replaceable. Device-agnostic and device-specific pass selectors can be replaced or augmented with site-specific heuristics or ML models; mapping and routing passes can be tailored for different hardware technologies; scheduling policies can be adapted to specific operational priorities; and cost models can be tuned to optimize for metrics that matter in a given deployment. Ultimately, this modularity ensures that, while the user experience and APIs remain stable, the underlying strategies for compilation and scheduling can evolve with hardware, workloads, and optimization objectives.

4.3 Early-user Study

As part of an early-user phase on the Q-Exa demonstrator, a hybrid quantum-classical workflow was executed through the MQSS on one of the 20-qubit superconducting devices available at LRZ [1, 17]. The study targeted a chemically relevant problem using a multiscale Quantum Mechanics/Molecular Mechanics (QM/MM) approach, in which a quantum computation was embedded within a larger molecular simulation. The workflow combined classical molecular dynamics and embedding methods on HPC nodes with quantum execution on the superconducting back-end, accessed seamlessly through the MQSS.

The problem was reduced to a 13-qubit Hamiltonian, enabling execution within the device's constraints. On the QPU, a quantum algorithm was used to sample determinant subspaces, with the projected Hamiltonian solved classically. Results achieved near chemical accuracy and consistently improved on baseline methods, demonstrating that embedded, noise-tolerant quantum algorithms can be effectively deployed within production HPC workflows.

This study illustrates how the MQSS supports tightly-coupled quantum acceleration in a real HPC environment, with device access, scheduling, and execution fully abstracted from the user's scientific problem while preserving flexibility in algorithmic and compilation strategies.

5 Discussion / Related Work

The development of quantum software stacks is a critical area of research and development worldwide, with several major initiatives pursuing different strategies. The MQSS distinguishes itself through its emphasis on deep HPC integration, modularity, and open, community-driven standards, directly addressing the challenges of heterogeneity, integration, and the lack of practical community standards, as identified in subsection 1.1.

Major hardware vendors, such as *IBM*, have developed comprehensive, vertically integrated software stacks (e.g., Qiskit [12]) that are tightly coupled to their own hardware offerings. While powerful and feature-rich, these stacks are naturally centered around a single vendor's ecosystem. In contrast, the MQSS is designed from the ground up to be hardware- and vendor-agnostic, aiming to provide a unified interface to a diverse and heterogeneous landscape of quantum hardware from various providers. Its modular architecture allows for the integration of any front-end framework or back-end system that adheres to its open interfaces.

National laboratories and other large-scale initiatives are also developing software solutions. For example, the work at *Oak Ridge National Laboratory* focuses on integrating quantum resources into their leadership computing facility [29]. While sharing the goal of HPCQC integration, the MQSS places a stronger emphasis on creating a portable, open-source stack that is not tied to a specific facility's infrastructure and can be deployed elsewhere. Similarly, initiatives by organizations like the German Aerospace Center (*DLR*) are building their own software platforms. The MQSS differentiates itself through its long-standing, community-focused co-design process, involving a wide range of stakeholders from hardware physicists to domain-specific end-users, and its promotion of cross-platform standards, like QDMI, to foster a broader, more collaborative ecosystem. Another related effort, the Quantum Resource Management Interface (QRMI; [31]), focuses primarily on standardized batch job submission to quantum systems, whereas QDMI provides a complete and comprehensive device management interface that also, next to its job management functionality, includes fine-grained, stateful session management and detailed, real-time query capabilities essential for dynamic compilation and optimization.

The MQSS's approach can be seen as a community-driven alternative to more monolithic or proprietary stacks. By focusing on modularity, interoperability, and open standards, it aims at creating a flexible and sustainable software ecosystem that can adapt to the rapid evolution of quantum hardware and serve as a common platform for the entire HPCQC community, rather than a single organization or vendor. This federated approach is crucial for tackling the challenges of integrating diverse hardware and software components, as highlighted in recent studies on the topic [11].

6 Conclusions and Outlook

In this paper, we have introduced the Munich Quantum Software Stack (MQSS), a comprehensive, open-source software ecosystem designed to bridge the gap between high-performance and quantum computing. We have motivated the need for such a stack in enabling the vision of quantum acceleration, detailed the significant challenges in its development, and presented our community-driven approach to overcoming them. The modular, multi-layered architecture of the MQSS, from the hardware-facing QDMI to the user-facing Front-end Adapters, provides a flexible and extensible platform for the development and execution of hybrid quantum-classical workflows.

The MQSS is not only a concept and academic prototype, but a tangible and evolving system, already deployed in a production HPC environment and gaining traction within the broader quantum community. Our work demonstrates a viable path towards the deep integration of quantum processing units into classical supercomputing infrastructure, a critical step for unlocking the practical potential of quantum computing.

The future development of the MQSS will proceed along three main pillars. First, **Stabilization**: we will continue to harden the existing components, improve interfaces, and provide robust support and documentation to maintain a production-ready, easy-to-use software stack. Second, **Evolutionary expansion**: we will continuously expand the stack's capabilities, integrating new hardware back-ends, developing more sophisticated compiler optimizations, and enhancing support for NISQ-era applications. We will also integrate community contributions towards an open ecosystem. Third, **Revolutionary transformation**: we will proactively address the long-term challenges of fault-tolerant quantum computing, ensuring that the stack's architecture and interfaces are prepared for the paradigm shift to error-corrected quantum systems. Through these efforts, the MQSS will remain at the forefront of quantum software, providing a vital platform for the future of integrated quantum and high-performance computing.

Acknowledgments

References

[1] Thomas M. Bickley, Angus Mingare, Tim Weaving, Michael Williams de la Bastida, Shunzhou Wan, Martina Nibbi, Philipp Seitz, Alexis Ralli, Peter J. Love, Minh Chung, Mario Hernández Vera, Laura Schulz, and Peter V. Coveney. 2025. Extending Quantum Computing through Subspace, Embedding and Classical Molecular Dynamics Techniques. arXiv:2505.16796 [quant-ph] https://arxiv.org/abs/2505.16796

[2] Dolev Bluvstein, Alexandra A. Geim, Sophie H. Li, Simon J. Evered, J. Pablo Bonilla Ataides, Gefen Baranes, Andi Gu, Tom Manovitz, Muqing Xu, Marcin Kalinowski, Shayan Majidy, Christian Kokail, Nishad Maskara, Elias C. Trapp, Luke M. Stewart, Simon Hollerith, Hengyun Zhou, Michael J. Gullans, Susanne F. Yelin, Markus Greiner, Vladan Vuletic, Madelyn Cain, and Mikhail D. Lukin. 2025. Architectural Mechanisms of a Universal Fault-Tolerant Quantum Computer. arXiv:2506.20661 [quant-ph]

[3] 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. doi:10.21105/joss.07478

[4] Michel H. Devoret and John M. Martinis. 2004. Implementing qubits with superconducting integrated circuits. (2004).

[5] Jorge Echavarria, Muhammad Nufail Farooqi, Amit Devra, Santana Lujan, Léo Van Damme, Hossam Ahmed, Martín Letras, Ercüment Kaya, Adrian Vetter, Max Werninghaus, Martin Knudsen, Felix Rohde, Albert Frisch, Eric Mansfield, Rakhim Davletkaliyev, Vladimir Kukushkin, Noora Färkkilä, Janne Mäntylä, Nikolas Pomplun, Andreas Spörl, Lukas Burgholzer, Yannick Stade, Robert Wille, Laura B. Schulz, and Martin Schulz. 2025. Tackling the Challenges of Adding Pulse-level Support to a Heterogeneous HPCQC Software Stack: MQSS Pulse. In *Proceedings of the SC '25 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25)*. Association for Computing Machinery, New York, NY, USA, 1868–1878. doi:10.1145/3731599.3767552

[6] Laird Egan, Dripto M. Debroy, Crystal Noel, Andrew Risinger, Daiwei Zhu, Debopriyo Biswas, Michael Newman, Muyuan Li, Kenneth R. Brown, Marko Cetina, and Christopher Monroe. 2021. Fault-Tolerant Control of an Error-Corrected Qubit. *Nature* 598, 7880 (Oct. 2021), 281–286. doi:10.1038/s41586-021-03928-y

[7] 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. 6, 3 (2025), 1–46. doi:10.1145/3743149

[8] H. Haeffner, C. F. Roos, and R. Blatt. 2008. Quantum computing with trapped ions. 469, 4 (2008), 155–203. doi:10.1016/j.physrep.2008.09.003

[9] P. Hopf, E. Ochoa Lopez, Y. Stade, D. Rovara, N. Quetschlich, I. A. Florea, J. Izaac, R. Wille, and L. Burgholzer. 2026. Integrating Quantum Software Tools with(in) MLIR. In *SCA/HPCAsia: Supercomputing Asia and International Conference on High Performance Computing in Asia Pacific Region*. doi:10.1145/3773656.3773658

[10] Patrick Hopf, Nils Quetschlich, Laura Schulz, and Robert Wille. 2025. Improving Figures of Merit for Quantum Circuit Compilation. In *2025 Design, Automation Test in Europe Conference (DATE)*. IEEE, 1–7. doi:10.23919/date64628.2025.10992761

[11] Travis S. Humble, Alexander McCaskey, Dmitry I. Lyakh, Meenambika Gowrishankar, Albert Frisch, and Thomas Monz. 2021. Quantum Computers for High-Performance Computing. 41, 5 (2021), 15–23. doi:10.1109/MM.2021.3099140

[12] Ali Javadi-Abhari, Matthew Treinish, Kevin Krsulich, Christopher J. Wood, Jake Lishman, Julien Gacon, Simon Martiel, Paul D. Nation, Lev S. Bishop, Andrew W. Cross, Blake R. Johnson, and Jay M. Gambetta. 2024. Quantum computing with Qiskit. arXiv:2405.08810 [quant-ph] doi:10.48550/arXiv.2405.08810

[13] Morris Jette, Andy Yoo, and Mark Grondona. 2003. SLURM: Simple linux utility for resource management. *LECTURE NOTES IN COMPUTER SCIENCE*. doi:10.1007/10968987_3

[14] Ercümeant Kaya, Burak Mete, Laura Schulz, Muhammad Nufail Farooqi, Jorge Echavarria, and Martin Schulz. 2024. QPI: A Programming Interface for Quantum Computers. In *Int'l Conf. on Quantum Computing and Engineering*, Vol. 02. 286–291. doi:10.1109/QCE60285.2024.10293

[15] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *Int'l Symp. on Code Generation and Optimization*. 2–14. doi:10.1109/CGO51591.2021.9370308

[16] Ji Liu, Ed Younis, Mathias Weiden, Paul Hovland, John Kubiatowicz, and Costin Iancu. 2023. Tackling the qubit mapping problem with permutation-aware synthesis. In *Int'l Conf. on Quantum Computing and Engineering* (2023-05-04). arXiv:2305.02939 [quant-ph] doi:10.48550/arXiv.2305.02939

[17] Eric Mansfield, Stefan Seegerer, Panu Vesanen, Jorge Echavarria, Muhammad Nufail Farooqi, Burak Mete, and Laura Schulz. 2025. First Practical Experiences Integrating Quantum Computers with HPC Resources: A Case Study With a 20-qubit Superconducting Quantum Computer. In *Proceedings of the SC '25 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25)*. Association for Computing Machinery, New York, NY, USA, 1842–1850. doi:10.1145/3731599.3767551

[18] Durganshu Mishra, Stepan Vanecek, Jorge Echavarria, Xiaolong Deng, Burak Mete, Laura Schulz, and Martin Schulz. 2025. Towards a Unified Architectural Representation in HPCQC: Extending Sys-Sage for Quantum Technologies. In *ISC High Performance*. 1–12.

[19] Prakash Murali, Dripto M. Debroy, Kenneth R. Brown, and Margaret Martonosi. 2020. Architecting Noisy Intermediate-Scale Trapped Ion Quantum Computers. In *Int'l Symposium on Computer Architecture*. 529–542. doi:10.1109/ISCA45697.2020.00051

[20] Jeremy L. O'Brien, Akira Furusawa, and Jelena Vučković. 2009. Photonic quantum technologies. 3, 12 (2009), 687–695. doi:10.1038/nphoton.2009.229

[21] John Preskill. 2018. Quantum Computing in the NISQ Era and Beyond. *Quantum* 2 (Aug. 2018), 79. doi:10.22331/q-2018-08-06-79

[22] QIR Alliance: https://qir-alliance.org 2021. *QIR Specification*. QIR Alliance: https://qir-alliance.org. https://github.com/qir-alliance/qir-spec

[23] Nils Quetschlich, Lukas Burgholzer, and Robert Wille. 2024. MQT Predictor: Automatic Device Selection with Device-Specific Circuit Compilation for Quantum Computing. *ACM Transactions on Quantum Computing* (June 2024). doi:10.1145/3673241

[24] C. Ryan-Anderson, N. C. Brown, M. S. Allman, B. Arkin, G. Asa-Attuah, C. Baldwin, J. Berg, J. G. Bohnet, S. Braxton, N. Burdick, J. P. Campora, A. Chernoguzov, J. Esposito, B. Evans, D. Francois, J. P. Gaebler, T. M. Gatterman, J. Gerber, K. Gilmore, D. Gresh, A. Hall, A. Hankin, J. Hostetter, D. Lucchetti, K. Mayer, J. Myers, B. Neyenhuis, J. Santiago, J. Sedlacek, T. Skripka, A. Slattery, R. P. Stutz, J. Tait, R. Tobey, G. Vittorini, J. Walker, and D. Hayes. 2022. Implementing Fault-tolerant Entangling Gates on the Five-qubit Code and the Color Code. arXiv:2208.01863

[25] Nishant Saurabh, Shantenu Jha, and Andre Luckow. 2023. A Conceptual Architecture for a Quantum-HPC Middleware. In *Int'l Conf. on Quantum Software* (2023-07). 116–127. doi:10.1109/QSW59989.2023.00023

[26] Ludwig Schmid, David Locher, Manuel Rispler, Sebastian Blatt, Johannes Zeiher, Markus Müller, and Robert Wille. 2024. Computational Capabilities and Compiler Development for Neutral Atom Quantum Processors - Connecting Tool Developers and Hardware Experts. *Quantum Science and Technology* (2024). doi:10.1088/2058-9565/ad33ac

[27] Daniel Schoenberger, Stefan Hillmich, Matthias Brandl, and Robert Wille. 2024. Shuttling for Scalable Trapped-Ion Quantum Computers. *IEEE Trans. on CAD of Integrated Circuits and Systems* (2024). arXiv:2402.14065 [quant-ph] doi:10.1109/TCAD.2024.3513262

[28] Philipp Seitz, Amr Elsharkawy, Xiao-Ting Michelle To, and Martin Schulz. 2023. Toward a Unified Hybrid HPCQC Toolchain. In *Int'l Conf. on Quantum Computing and Engineering* (2023-09-17). 96–102. doi:10.1109/qce57702.2023.10191

[29] Amir Shehata, Peter Groszkowski, Thomas Naughton, Murali Gopalakrishnan Meena, Elaine Wong, Daniel Claudino, and Thomas Beck. 2025. Building a Software Stack for Quantum-HPC Integration. arXiv:2503.01787 [quant-ph]

[30] P.W. Shor. 1996. Fault-Tolerant Quantum Computation. In *Conference on Foundations of Computer Science*. 56–65. doi:10.1109/SFCS.1996.548464

[31] Iskandar Sitdikov, M. Emre Sahin, Utz Bacher, Aleksander Wennersteen, Andrew Damin, Mark Birmingham, Philippa Rubin, Stefano Mensa, Matthieu Moreau, Aurelien Nober, Hitomi Takahashi, and Munetaka Ohtani. 2025. Quantum resources in resource management systems. arXiv:2506.10052 [quant-ph] https://arxiv.org/abs/2506.10052

[32] Yannick Stade, Lukas Burgholzer, and Robert Wille. 2025. Towards Supporting QIR: Steps for Adopting the Quantum Intermediate Representation. In *Proceedings of the SC '25 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25)*. Association for Computing Machinery, New York, NY, USA, 1907–1915. doi:10.1145/3731599.3767546

[33] Yannick Stade, Ludwig Schmid, Lukas Burgholzer, and Robert Wille. 2024. An Abstract Model and Efficient Routing for Logical Entangling Gates on Zoned Neutral Atom Architectures. In *Int'l Conf. on Quantum Computing and Engineering* (Montreal, QC, Canada). IEEE, 784–795. doi:10.1109/QCE60285.2024.00098

[34] Yannick Stade, Ludwig Schmid, Lukas Burgholzer, and Robert Wille. 2025. Optimal State Preparation for Logical Arrays on Zoned Neutral Atom Quantum Computers. In *Design, Automation and Test in Europe*. arXiv:2411.09738 [quant-ph]

[35] John van de Wetering. 2021. Constructing quantum circuits with global gates. 23, 4 (2021), 043015. arXiv:2012.09061 doi:10.1088/1367-2630/abf1b3

[36] Robert Wille, Lucas Berent, Tobias Forster, Jagatheesan Kunasaikaran, Kevin Mato, Tom Peham, Nils Quetschlich, Damian Rovara, Aaron Sander, Ludwig Schmid, Daniel Schoenberger, Yannick Stade, and Lukas Burgholzer. 2024. The MQT Handbook: A Summary of Design Automation Tools and Software for Quantum Computing. In *Int'l Conf. on Quantum Software* (2024). arXiv:2405.17543

[37] Robert Wille, Lukas Burgholzer, and Alwin Zulehner. 2019. Mapping quantum circuits to IBM QX architectures using the minimal number of SWAP and H operations. In *Design Automation Conf.* doi:10.1145/3316781.3317859

[38] Robert Wille, Ludwig Schmid, Yannick Stade, Jorge Echavarria, Martin Schulz, Laura Brandon Schulz, and Lukas Burgholzer. 2024. QDMI – Quantum Device Management Interface: Hardware-Software Interface for the Munich Quantum Software Stack. In *Int'l Conf. on Quantum Computing and Engineering*.

[39] Henry Zou, Matthew Treinish, Kevin Hartman, Alexander Ivrii, and Jake Lishman. 2024. LightSABRE: A Lightweight and Enhanced SABRE Algorithm. arXiv:2409.08368 [quant-ph] https://arxiv.org/abs/2409.08368

[40] Alwin Zulehner, Alexandru Paler, and Robert Wille. 2019. An efficient methodology for mapping quantum circuits to the IBM QX architectures. *IEEE Trans. on CAD of Integrated Circuits and Systems* (2019). doi:10.1109/TCAD.2018.2846658