



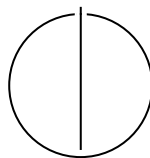
SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY —
INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

**Connecting Quantum Software Stacks -
MQSS and QCI Connect**

Florea Ioan-Albert





SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY —
INFORMATICS

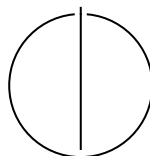
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

**Connecting Quantum Software Stacks -
MQSS and QCI Connect**

**Verbindung von Quantensoftware-Stacks -
MQSS und QCI Connect**

Author:	Florea Ioan-Albert
Examiner:	Robert Wille
Supervisor:	Lukas Burgholzer
Submission Date:	14.07.2026



I confirm that this master's thesis is my own work, and that I have documented all sources and materials used.

Munich, 14.07.2026

Florea Ioan-Albert

Acknowledgments

I want to thank Lian Remme and Andre Waschke from the German Aerospace Center (DLR) for their help over the last couple of months.

I would also like to thank Lukas and Robert for their constant support and constructive feedback over the last few years and for allowing me to be part of their research group and contribute to many interesting topics.

Lastly, I want to thank my family and my girlfriend for supporting me throughout this process. Without them, I wouldn't be here. Therefore, I am forever grateful.

Abstract

Quantum computing systems are increasingly accessed through software stacks that hide hardware-specific details behind higher-level abstractions. This is necessary because current quantum resources differ in their physical realisation, supported program formats, execution parameters, calibration data, status models, and result representations. At the same time, independent software stack initiatives create an interoperability challenge: even systems with similar goals may not communicate directly if their APIs encode different assumptions.

This thesis studies this problem for the Munich Quantum Software Stack (MQSS), whose backend abstraction is the Quantum Device Management Interface (QDMI), and QCI Connect, a modular full-stack quantum computing platform developed within the DLR Quantum Computing Initiative. The goal is to connect these systems bidirectionally while preserving their architectural abstractions. In one direction, MQSS resources are exposed as QCI Connect QPUs via a dedicated QPU connector. In the other direction, QCI Connect is exposed as a QDMI-compatible device that translates QDMI session, query, job, status, cancellation, and result calls into QCI Connect API requests.

The thesis compares the relevant API concepts and implements mappings for resource discovery, job submission, status handling, cancellation, and result retrieval. The implementation shows that bidirectional communication is feasible through existing extension points in both stacks. It also reveals semantic mismatches, including incomplete metadata equivalence, differences in status granularity, assumptions about circuit formats, and result fields requiring custom handling. These findings show that interoperability between quantum software stacks is possible, but long-term portability requires clearer standardisation of capabilities, execution parameters, status models, and result schemas.

Contents

Acknowledgments	iii
Abstract	iv
1 Introduction	1
2 Background	4
2.1 Quantum Software Stacks	4
2.2 The Munich Quantum Software Stack	6
2.3 QCI Connect	10
3 Enabling Bidirectional Communication	14
3.1 Common Concepts Required by Both Software Stacks	15
3.2 QCI Connect to MQSS	17
3.2.1 Resource Discovery and QPU Representation	18
3.2.2 Job Submission	21
3.2.3 Status Handling and Result Translation	23
3.2.4 Discussion	23
3.3 QDMI to QCI Connect	25
3.3.1 Session Management and Resource Selection	25
3.3.2 Job Configuration and Submission	28
3.3.3 Status Handling, Cancellation, and Results	30
3.3.4 Discussion	34
4 Limitations and Future Work	35
4.1 Limitations	35
4.2 Future Work	37
5 Conclusion	40
List of Figures	42
List of Tables	43
Bibliography	44

1 Introduction

Quantum computing is increasingly transitioning from isolated laboratory prototypes to components of broader computing infrastructures. Its long-term promise lies in solving selected classes of problems that are difficult or impractical for classical computers, especially in the simulation of quantum systems and in hybrid optimisation workflows. At the same time, the current generation of devices remains constrained by noise, limited qubit counts, and imperfect control, which is why they are commonly described as belonging to the Noisy Intermediate-Scale Quantum (NISQ) era. In this period, quantum computers should not be understood as stand-alone replacements for classical systems, but rather as specialised resources whose practical value depends heavily on their integration into larger software and hardware environments [1, 2].

This situation leads to an important software challenge. Over the last few years, the quantum computing landscape has seen the development of a broad range of hardware technologies and QPUs, and access to many of these resources is offered through cloud platforms [3]. At the same time, the user base is expanding beyond small groups of hardware specialists to include application scientists and HPC users who require portable, efficient access to such systems. This shift calls for software abstractions that decouple high-level program development from backend-specific execution details while enabling compilation, scheduling, and optimisation across heterogeneous devices [2, 4].

Recent work on the Munich Quantum Software Stack (MQSS) argues that quantum systems should be treated as accelerators within hybrid classical-quantum workflows and that achieving this vision requires a comprehensive software stack with clear separation between user-facing programming models, compilation and orchestration middleware, and backend-specific execution layers [4]. A major obstacle to this vision, however, is the diversity of interfaces exposed by today’s quantum platforms. Quantum backends differ not only in their physical realisation, but also in execution semantics, available device metadata, authentication mechanisms, and supported program representations. Without a unifying abstraction, each new backend or cloud service requires dedicated integration logic, reducing portability, increasing maintenance effort, and hindering interoperability across software stacks. Standardised backend abstractions are therefore becoming a prerequisite for scalable multi-platform quantum software environments [3, 4].

This need for interoperability is also reflected in current standardisation efforts. The DIN SPEC [5] on interfaces between quantum computer backends and software frameworks explicitly identifies the variety of both software frameworks and backend providers as a source of fragmentation and points out that the absence of unified interfaces creates a risk of vendor lock-in. It further emphasises that standardisation can reduce redundant interface development and support interoperability in an emerging ecosystem. Against this background, backend interfaces are not merely implementation details, but foundational building blocks for an open and sustainable quantum software landscape.

One concrete response to this challenge is the Quantum Device Management Interface (QDMI). QDMI is designed as a backend-agnostic interface for device discovery, capability queries, job submission, monitoring, result retrieval, and lifecycle management. Within MQSS, it serves as the abstraction boundary between the middle layer, where compilation, scheduling, and resource management take place, and the backend layer, where concrete quantum devices, simulators, and even cloud-hosted services are accessed [3, 4]. Recent work has further shown that QDMI can be used not only for individual devices, but also for cloud services such as Amazon Braket, highlighting its suitability as a unifying abstraction across different deployment models [3].

At the same time, QDMI is not the only effort pursuing this goal. QCI Connect, developed at the German Aerospace Center (DLR), represents another approach to connecting quantum software developers to quantum devices via its own API [6, 7]. The coexistence of such interfaces reflects the broader heterogeneity of the quantum software ecosystem: different platforms expose different abstractions, interaction models, and assumptions about how users, software frameworks, and backend resources are connected. As a consequence, interoperability between software stacks cannot be taken for granted. Even when two approaches pursue similar goals, differences in their architectural organisation and in the concepts encoded by their APIs can make direct communication difficult. This creates a practical barrier to portability and reuse across infrastructures and motivates the need for a systematic analysis of how such systems relate to one another.

The central objective of this thesis is therefore to connect QDMI, the backend management interface used within MQSS, with QCI Connect, which provides a software stack for accessing and managing quantum devices. This connection enables requests to be forwarded between the two systems while preserving their respective architectural abstractions. To achieve this, the work not only considers the translation between their interfaces but also compares the architectures of the two systems to identify how their respective components and abstractions relate to one another. Based on this analysis, the thesis investigates where both approaches are conceptually aligned, where semantic mismatches arise, and which extensions or adaptations are required for meaningful

integration. In this way, this work also addresses the broader question of how different backend management architectures can be systematically interconnected.

There are several reasons this problem is relevant. First, it addresses interoperability at the backend-management level, where incompatibilities directly affect whether software tools can be reused across providers and infrastructures. Second, it complements the broader direction of MQSS, which aims to decouple frontends from backends, thereby supporting many users, many systems, and many deployment models through shared abstractions and standardised interfaces [2, 4]. Third, it provides a concrete case study for understanding how different interface designs encode concepts such as sessions, jobs, properties, execution parameters, and results, and whether these concepts can be translated without semantic loss. In this sense, the work is not only about implementing a wrapper or adapter, but also about analysing how backend abstractions can be aligned in a quantum ecosystem.

The importance of such alignment is further underscored by recent extensions of quantum software stacks toward richer forms of hardware access. For example, work on adding pulse-level support to MQSS shows that future backend interfaces must increasingly accommodate more detailed descriptions of hardware capabilities, including low-level notions such as ports, frames, and waveforms [8]. This illustrates that backend abstractions are expected to evolve beyond simple gate-based job submission toward more expressive models of device interaction. Any attempt to interconnect two management interfaces, such as QDMI and QCI Connect, should therefore consider not only present-day compatibility but also extensibility toward future requirements.

This thesis is situated at the intersection of quantum software engineering, interoperability, and systems integration, and it makes three main contributions to this field. First, it compares the relevant backend-management concepts of MQSS/QDMI and QCI Connect, providing insights into their similarities and differences. Second, it implements a QCI Connect QPU connector that exposes MQSS resources as QCI Connect QPUs, facilitating seamless integration. Third, it develops a QDMI-compatible device that allows QCI Connect resources to be accessed through the QDMI lifecycle. Ultimately, this work addresses the concrete problem of bridging QDMI and QCI Connect while contributing to the broader question of how standardised, portable, and extensible interfaces can enable access to heterogeneous quantum resources. By comparing the two APIs, identifying common abstractions and semantic differences, and implementing a translation layer between them, this thesis aims to support the development of a more open and interoperable quantum software ecosystem.

2 Background

In this chapter, we introduce the technical background necessary to understand how MQSS, QDMI, and QCI Connect relate to one another. First, we motivate the need for quantum software stacks and backend interfaces. This motivation is important because practical quantum computing requires not only quantum hardware, but also software layers that make heterogeneous quantum resources accessible, programmable, and manageable. We then present the Munich Quantum Software Stack (MQSS) as the broader software-stack architecture that embeds QDMI. Afterwards, we describe QDMI in more detail, focusing on its role as a backend abstraction and on its main concepts for sessions, device queries, and jobs. Finally, we introduce QCI Connect as the second quantum software stack considered in this thesis and discuss the architectural concepts relevant to establishing communication between MQSS, QDMI and QCI Connect.

2.1 Quantum Software Stacks

Quantum computers are increasingly moving from isolated laboratory prototypes toward resources that are integrated into larger computing infrastructures. However, practical use of these systems requires more than the availability of quantum hardware. Users need programming environments, compilers, runtime systems, schedulers, backend interfaces, and control software that together transform a high-level problem description into an executable program for a concrete quantum device. This collection of layers is commonly referred to as a quantum software stack.

The motivation for such a stack is illustrated in Figure 2.1. The left-hand side shows a direct-access setting in which users or software tools interact with quantum devices through separate, backend-specific paths. However, this approach has several drawbacks. In particular, each user-facing tool must either implement support for every backend individually or rely on provider-specific access mechanisms. Furthermore, concurrent access to the same hardware requires explicit coordination, since multiple users or tools may otherwise compete for scarce quantum resources without a common scheduling or resource-management layer.

A quantum software stack addresses this problem by introducing shared layers between users and devices. Instead of exposing every backend-specific detail directly to the user-facing software, the stack provides common abstractions for compilation,

resource management, execution, and backend interaction. This allows applications to be developed against stable interfaces while backend-specific details are handled by lower layers of the stack. In this sense, Figure 2.1 not only shows a different access pattern, but it also illustrates why decoupling is necessary for scalable access to many different quantum resources.

A second motivation for quantum software stacks is the variety of current quantum hardware and access models. Quantum devices differ in their physical realisation, native gate sets, qubit connectivity, calibration data, noise characteristics, supported program formats, and access mechanisms. Consequently, a program that can be executed on one backend generally cannot be transferred to another backend without further adaptation.

A software stack can support this adaptation by combining intermediate representations, compiler passes, and backend descriptions. Based on information about the available backends, the stack can transform a workload into a representation that matches the constraints of a selected execution target, for example, by decomposing operations into the backend’s native gate set, mapping logical qubits to the device topology, or adapting execution parameters. Moreover, the stack can use resource management functionality to select a suitable backend in the first place, or to dispatch a job to an alternative backend if the originally intended target is unavailable or unsuitable. This does not eliminate backend differences, but it shifts much of the required adaptation from the application developer to dedicated compilation, resource management, and backend interface layers.

This need for common abstractions is also reflected in DIN SPEC 91520 [5], a specification for interfaces between quantum computer backends and software frameworks. The specification describes diversity on both the user and provider sides as a source of fragmentation and links the lack of interoperability to the risk of vendor lock-in. It therefore motivates the early standardisation of backend interfaces to reduce redundant interface development and support interoperability across the emerging ecosystem. The specification was published by DIN Media in 2025 and developed according to the PAS procedure, in which a DIN SPEC is created by a dedicated consortium rather than through a full standardisation process involving all stakeholders.

Backend interfaces form the boundary between the higher software layers and concrete quantum devices or services. They transfer programs, expose device information, manage jobs, and return results. The DIN SPEC identifies several aspects that are particularly important for such interfaces. A backend interface should operate at an abstraction level that represents quantum circuits independently of physical topology and native gates, use a common representation for quantum programs, integrate with classical information technology, and structure exchanged data into job-specific, device-specific, and results data. These categories are directly relevant to this thesis, since

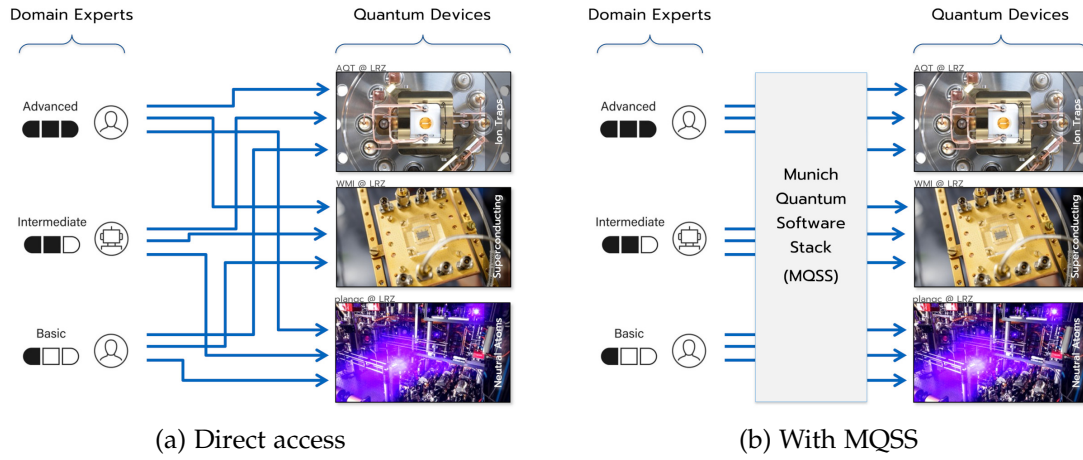


Figure 2.1: Comparison of access models for quantum devices. Figure taken from [4], figure 5.

connecting QDMI and QCI Connect requires analysing how both interfaces represent jobs, devices, execution parameters, status information, and results.

Consequently, the problem addressed in this thesis is not only a technical forwarding problem. It is also an architectural mapping problem between two backend management interfaces. Even if QDMI and QCI Connect provide similar high-level functionality, such as job submission or result retrieval, they may differ in how they model devices, sessions, authentication, execution parameters, job states, and returned data. Establishing communication between them, therefore, requires identifying which concepts correspond directly, which require transformation, and which cannot be translated without additional assumptions or extensions.

2.2 The Munich Quantum Software Stack

The Munich Quantum Software Stack (MQSS) is a modular software ecosystem for hybrid quantum-classical computing. It is developed in the context of the Munich Quantum Valley (MQV) [9] and aims to connect users, software tools, and heterogeneous quantum hardware through a unified stack [4, 9]. The central motivation behind MQSS is that quantum computers should not be treated as isolated systems accessible only through vendor-specific tools. Instead, they should be integrated as specialised accelerators into larger classical computing environments, especially high-performance computing infrastructures [2, 4].

As shown in Figure 2.2, MQSS addresses this goal by separating the software stack

into multiple layers. At the top, user-facing interfaces allow different user groups to express quantum or hybrid applications without depending directly on a specific quantum device. In the middle, compilation, optimisation, scheduling, and resource management components transform these programs into intermediate representations (IRs) that can be executed efficiently. At the bottom, backend-facing components connect the stack to concrete quantum processing units, simulators, or cloud-based quantum services. This layered structure allows MQSS to decouple application development from hardware-specific execution details, thereby enabling support for a large number of quantum resources.

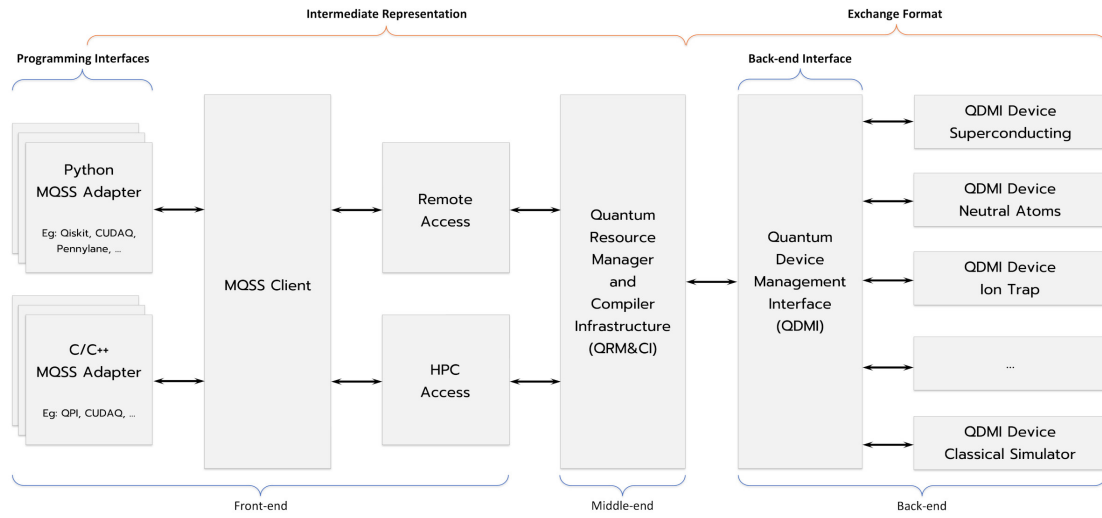


Figure 2.2: The high-level architecture of the MQSS taken from [4], suggesting a layered approach.

The frontend layer provides the entry point for users and application developers. Since MQSS targets users with different levels of expertise, this layer is not restricted to a single programming model. Instead, it is designed to support several access paths, such as high-level libraries, established quantum programming frameworks, and portal-based access. This is important because quantum computing users range from domain experts who want to formulate applications at a high level to quantum software developers who need more control over the generated circuits. The frontend layer, therefore, hides many details of the target backend while still allowing advanced users to interact with the stack at a lower level when required.

The middle layer forms the core of MQSS. It is responsible for transforming user-level programs into executable workloads that can run on available quantum resources. This includes compiler components, optimisation passes, mapping and routing, verification

tools, and scheduling mechanisms. Since quantum devices differ in their native gate sets, connectivity, noise behaviour, and calibration data, this layer cannot compile programs purely statically. Instead, it must take the current backend information into account. MQSS, therefore, follows a dynamic compilation approach in which backend properties and constraints can influence the generated executable representation. This enables the stack to adapt a quantum program to the concrete device that will execute it.

A central component of the middle layer, as seen in Figure 2.2, as well is the quantum resource-management functionality. The resource manager coordinates the interaction between compiled programs and available quantum resources. It must select suitable backends, consider device capabilities, handle execution requests, and interact with backend-facing interfaces. In this role, it acts as a bridge between the compiler-oriented parts of MQSS and the hardware-facing parts of the stack. This component is especially important because communication with other device-management systems ultimately affects how resources are discovered, selected, and accessed. As we will see in chapter 3, communication with an external system, such as QCI Connect, can be enabled by creating a QDMI Device tailored specifically for it.

The backend layer provides the abstraction boundary between MQSS and the concrete execution targets (in MQSS, these are often referred to as "backends" or "devices"). These targets may be physical quantum devices, simulators, or cloud-hosted services. Instead of requiring each middle-layer component to implement custom logic for each backend, MQSS introduces a common backend management abstraction. This abstraction is provided by the Quantum Device Management Interface (QDMI), which serves as the communication interface between MQSS software components and connected quantum resources [10]. QDMI exposes operations to query device information, manage sessions, submit jobs, monitor execution, and retrieve results. In this way, it allows MQSS to treat different backends through a common interface while leaving backend-specific details inside the corresponding QDMI implementation. A visualisation of how one should imagine communication with these interfaces is shown in Figure 2.3 above.

This separation of concerns is one of the main architectural strengths of MQSS. The frontend layer can focus on usability and programming models, the middle layer on compilation and orchestration, and the backend layer on device access. Each layer depends on well-defined abstractions rather than on the layers below it. As a result, MQSS can support new user interfaces, compiler components, and backend technologies without redesigning the entire stack. For example, a new quantum device can be integrated by providing a suitable backend implementation, while existing frontend and middle-layer components can continue to use the same abstract interface.

Another important aspect of MQSS is its focus on heterogeneous hardware. Quantum hardware platforms differ not only in their physical realisation, such as superconducting

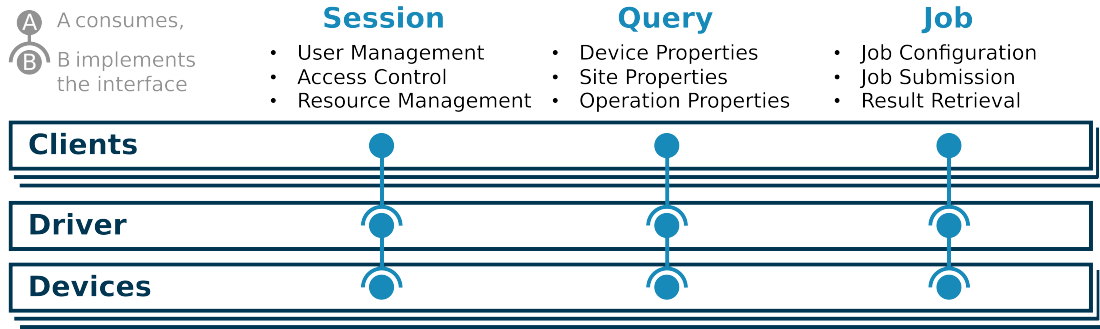


Figure 2.3: The rationale of QDMI taken from [11], where the clients, driver and devices communicate via session, query and job interfaces, each with specific purposes.

qubits, trapped ions, neutral atoms, or photonic systems, but also in their execution models and available control parameters. A software stack that aims to support such systems must therefore expose sufficient backend information to enable informed decisions about compilation and scheduling. MQSS addresses this by allowing backend properties and constraints to flow upward through the stack. These properties may include information about the number of qubits, connectivity, supported operations, timing constraints, calibration data, and other figures of merit. The compiler and resource-management components can then use this information to generate programs that better match the selected execution target.

MQSS is also designed with hybrid classical-quantum workflows in mind. In such workflows, quantum computation is only one part of a larger computation that may include classical preprocessing, iterative optimisation, result analysis, and repeated communication between classical and quantum resources. This makes integration with classical high-performance computing environments particularly relevant. MQSS, therefore, not only provides access to quantum devices but also aims to embed quantum execution into broader classical workflows. This perspective distinguishes a software stack from a simple backend API: the stack must coordinate programming, compilation, resource management, and execution across both classical and quantum components.

In summary, MQSS provides a layered architecture for connecting high-level quantum applications to heterogeneous quantum backends. Its frontend layer offers access paths for different users, its middle layer performs compilation, optimisation, scheduling, and resource management, and its backend layer uses QDMI to abstract concrete quantum resources. This structure enables portability, extensibility, and integration with classical computing environments. In the context of this thesis, MQSS defines the architectural

environment in which QDMI operates. After introducing QCI Connect in the next chapter 3, we will need to align the two software stacks to ensure seamless bidirectional communication.

2.3 QCI Connect

QCI Connect is the access and integration platform of the DLR Quantum Computing Initiative (DLR QCI) [12]. DLR QCI provides quantum-computing resources at its innovation centres in Hamburg and Ulm and aims to make these resources usable by DLR teams, research partners, start-ups, industry partners, and, in the future, a wider user community [13]. The QCI Connect platform is therefore not only a technical interface to a single quantum device, but a broader Quantum-as-a-Service platform through which users can access quantum computers, digital twins, emulators, simulators, and other compute resources.

The broader context of QCI Connect is the DLR QCI software ecosystem. DLR QCI combines hardware, software, and application projects to bring quantum computers into use more quickly. The DLR QCI website describes this ecosystem as consisting of hardware projects across several qubit technologies and more than sixty application projects, while the software and application projects develop both future use cases and the necessary software stack [12]. This ecosystem perspective is also reflected in the work on quantum software ecosystem design by Basermann et al. [14]. Their work emphasises that a quantum software ecosystem must connect user-facing access, hardware integration, and the availability of quantum resources for different end users. From this perspective, a usable quantum-computing environment requires not only algorithms and compilers, but also user interfaces, orchestration, data management, simulators, and integration with classical high-performance computing resources. QCI Connect can be understood as an implementation-oriented realisation of these ideas in the DLR QCI infrastructure.

The development of QCI Connect has started with the CLIQUE project (*Classical Integration of Quantum Computers*) [15]. The project focuses on integrating quantum computers into usable software environments and on supporting the path from user applications to executable quantum workloads. Its goals include accessible development environments, compilation support for quantum applications, and reusable algorithmic components. These goals are also visible in QCI Connect: the platform is not limited to forwarding quantum circuits to a device but also combines user interaction, compilation, execution management, and access to various resources [6].

Figure 2.4 illustrates this role at a high level. On the user side, QCI Connect serves as the common entry point for different groups seeking to access DLR QCI

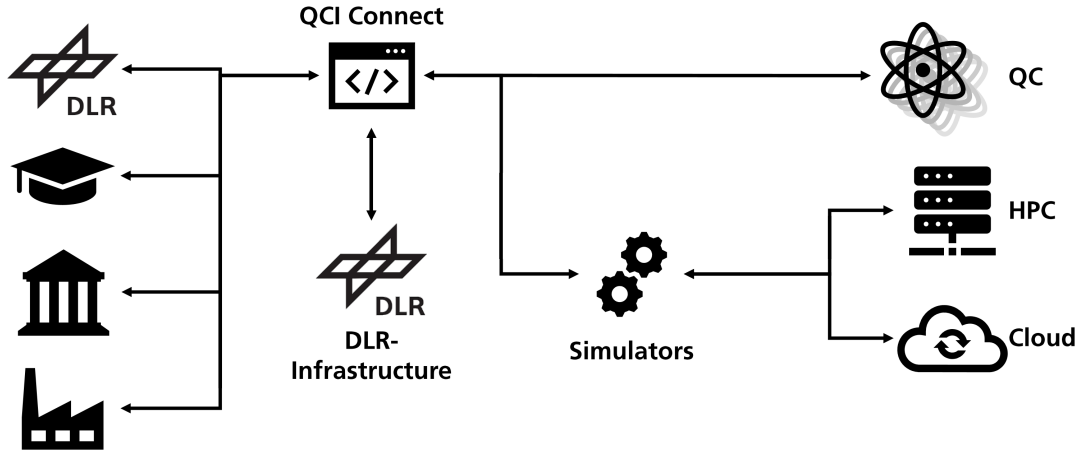


Figure 2.4: High-level interaction model of QCI Connect within the DLR QCI infrastructure. The figure shows QCI Connect as an access layer between different stakeholder groups and available compute resources, including quantum computers, HPC resources, simulators, and cloud services. Figure taken from [16].

resources. On the resource side, it connects to several types of compute infrastructure, including quantum computers, HPC resources, simulators, and cloud services [16]. This corresponds to the public description of QCI Connect as a service that enables partners from industry, start-ups, and research to obtain computing time on quantum computers and other computing resources. The platform currently exposes various execution targets, including physical quantum computers and other compute resources, such as digital twins, emulators, and simulators [13]. This diversity is directly relevant to communication among MQSS, QDMI, and QCI Connect, because an integration layer must treat QCI Connect as a platform-level access point rather than a single fixed backend with a single device model.

The internal structure considered in this thesis is shown in Figure 2.5. The central element is the QCI Connect platform, which offers the services needed to accept user requests, prepare them for execution, and forward them to suitable compute resources. Users can interact with the platform either through a web-based interface or through a Python Software Development Kit (SDK), called the QCI Connect SDK: API Client [17]. The web interface provides interactive access, whereas the SDK provides programmatic access for application developers and software components [18, 19]. The API Client allows users to authenticate with QCI Connect, submit quantum circuits for compilation and execution, and retrieve execution results. Both access paths ultimately interact with the platform backend, where jobs are represented, processed, and routed to the

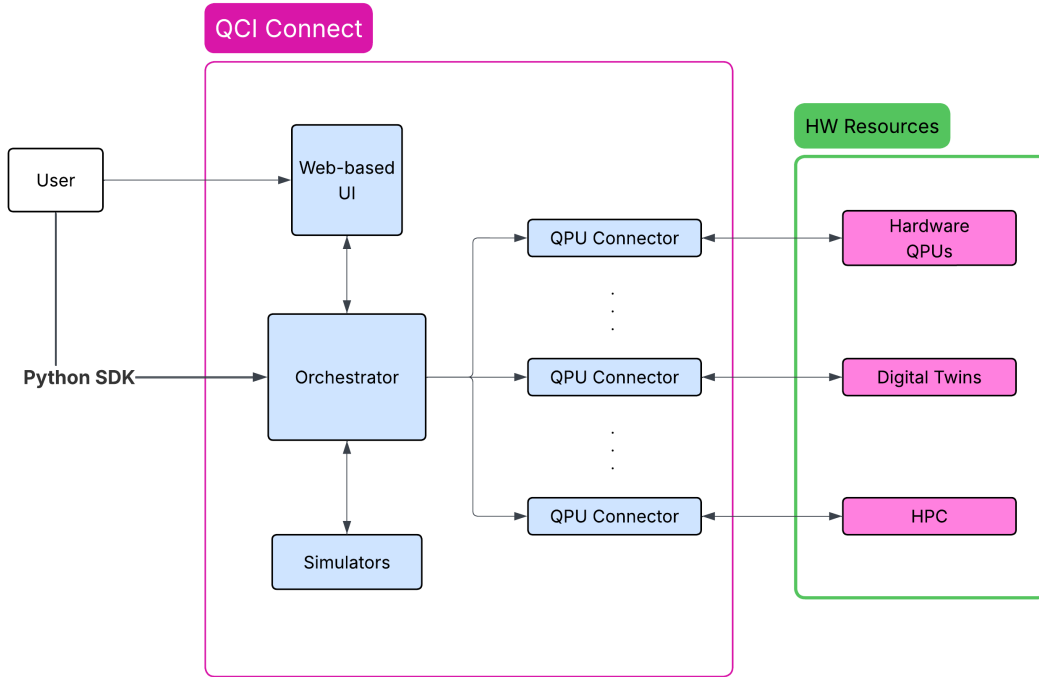


Figure 2.5: QCI Connect high-level architecture illustrating communication between the main components of the platform and the external compute resources.

appropriate execution target. These components demonstrate that QCI Connect exposes concepts central to QDMI, including authentication, job submission, execution, status handling, and result retrieval.

Within this architecture, the orchestrator is the central coordination component. It exposes the QCI Connect API to external clients and controls the flow of quantum circuits and execution requests inside the platform. In particular, it forwards circuits to the components responsible for compilation, simulation, or execution on quantum hardware, and it keeps track of the state of submitted jobs. It is also responsible for platform-level execution management, such as ordering jobs, handling priorities and providing information about job and backend status. From a client's perspective, the orchestrator is therefore the primary service endpoint of QCI Connect. On the other hand, from the platform's perspective, it is the component that connects user-facing access paths with the available execution resources.

Simulators are attached to the orchestrator as execution backends. They can be used in a similar role to physical quantum devices, because they accept quantum

workloads from the platform and return execution results without requiring direct access to a real QPU. Users can access not only quantum hardware, but also emulators and simulators through the platform [13, 18]. Simulators are therefore important for testing, development, and execution workflows in which either real hardware is not necessary or an emulated environment is more appropriate.

Physical and hardware-related resources are accessed through QPU connectors. These connectors provide the bridge between the orchestrator and concrete execution targets. Several connectors can be attached to the platform, allowing QCI Connect to address different QPUs or hardware resources through a single access model. In this way, the connector layer separates platform-level job handling from the details of a concrete backend. This separation is relevant because it shows that QCI Connect already contains an internal backend abstraction layer: the platform-facing interface is provided by the orchestrator, while backend-specific communication is delegated to the corresponding connector component.

The concepts exposed by QCI Connect also overlap with the interface categories identified in DIN SPEC 91520 [5]. The specification distinguishes among job-specific data, device-specific data, and result data, including device identifiers, execution parameters, quantum programs, authentication tokens, job status, supported input formats, availability, and results. These categories provide a useful reference point for comparing QCI Connect and QDMI because both systems must represent jobs, resources, execution parameters, status information, and results, but organise them around different architectural assumptions.

In summary, QCI Connect provides a platform for heterogeneous DLR QCI resources, while QDMI provides the backend-management abstraction used within MQSS. Although both systems involve similar concepts, they organise them according to different architectural models. We will address these aspects in the next chapter and show how to connect both quantum software stacks without requiring fundamental adaptations to the current architectures.

3 Enabling Bidirectional Communication

The previous chapter introduced MQSS and QCI Connect as two software-stack architectures that address the same general problem from different perspectives: making heterogeneous quantum resources accessible through stable software abstractions. This chapter now turns this architectural comparison into an integration strategy. The goal is not only to forward requests between two APIs, but to preserve the abstractions of both systems while enabling job submission in both directions.

The central design decision of this work is to avoid defining an additional interface between the two software stacks. Instead, the integration is built on the existing extension points in both systems. On the QCI Connect side, new execution targets are integrated via QPU connectors (see Figure 2.5). Such connectors are independent services that communicate with the orchestrator through the backend API and hide the details of the concrete execution target behind this platform-facing interface [7]. On the MQSS side, new execution targets are integrated through QDMI devices. In the context of QDMI, these devices are referred to as backends, but to avoid confusion, we will use the term devices. A QDMI device implements the session, query, and job functions required by the QDMI specification [10], thereby hiding backend-specific communication behind the MQSS backend-management boundary.

Consequently, the two directions of communication are implemented symmetrically but not identically. In the first direction, QCI Connect is extended with a QPU connector that treats MQSS as the execution backend. From the perspective of QCI Connect, MQSS therefore appears as an additional quantum resource that can be selected, submitted to, monitored, and queried for results through the usual QCI Connect workflow. In the second direction, MQSS is extended with a QDMI device that wraps QCI Connect. From the perspective of MQSS, the QCI Connect platform appears as a QDMI-compatible backend whose resources can be queried and whose jobs can be managed through the regular QDMI lifecycle.

This design is shown conceptually in Figure 3.1. The important point is that neither software stack has to expose its internal components to the other. QCI Connect does not need to understand the internal organisation of the MQSS compiler and resource manager. Similarly, MQSS does not need to understand the QCI Connect orchestrator’s internal workflow, as it will use only the orchestrator’s API for communication. Both systems interact only through mechanisms provided by their own architecture.

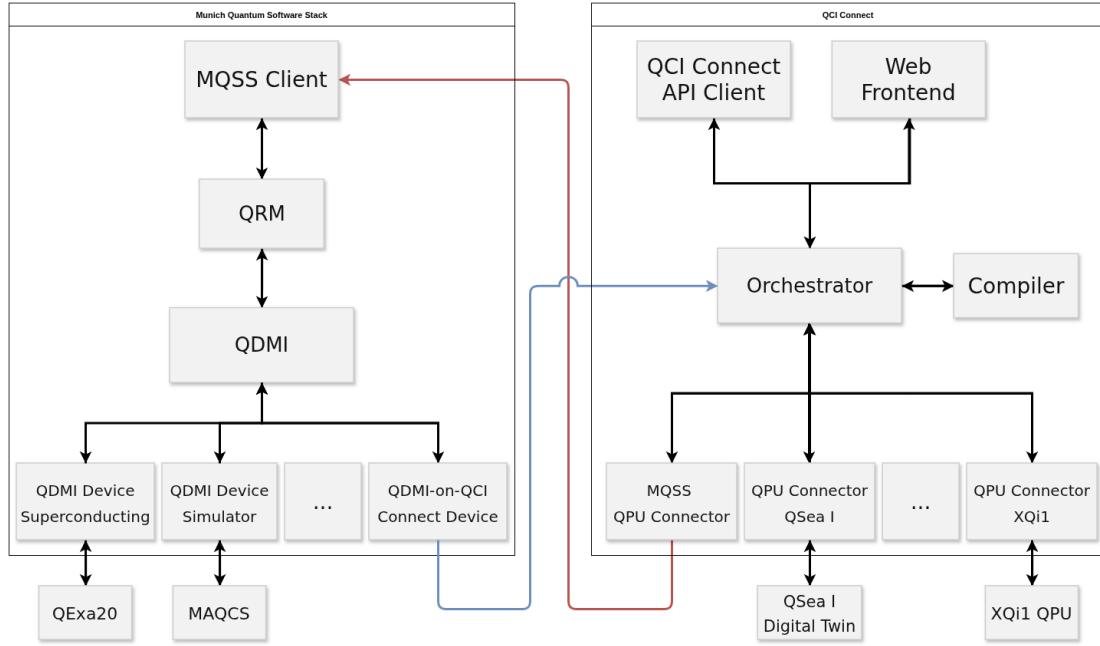


Figure 3.1: High-level architecture of how the implementation bridges the two software stacks. In both cases, the integration extends the lowest layer in their stacks, closest to the external hardware resources.

3.1 Common Concepts Required by Both Software Stacks

Both QCI Connect and MQSS provide abstractions that make heterogeneous quantum resources accessible via a software stack. Although their architectures differ, a successful integration requires a shared set of backend-management concepts: an authenticated interaction context, an execution target, a program representation, execution parameters, an asynchronous job model, status reporting, and a classical result representation. These concepts are not specific to one concrete quantum algorithm or one particular backend. Instead, they form the minimal vocabulary that both systems must expose to submit work to a quantum resource and retrieve the outcome after execution.

This section, therefore, abstracts from the two concrete communication directions and identifies the concepts that appear on both sides of the integration. The purpose is to make the later mappings easier to follow. In the following sections, these concepts are instantiated twice: first when MQSS resources are exposed as QCI Connect QPUs, and second when QCI Connect is exposed through a QDMI device. In both cases, the implementation does not introduce a new universal interface between the two systems.

Instead, it translates between concepts that already exist in the respective APIs. This is important because the integration should preserve the architectural assumptions of both systems rather than forcing one stack to adopt the other's internal model.

The first concept observed in both stacks is an authentication context. A user or software component (often referred to as a client) must be authenticated before it can access backend resources. In QCI Connect, this context is managed at the platform level by the orchestrator and its authentication infrastructure (see [6]). In MQSS, the corresponding context is represented either by the MQSS Client configuration or, at the backend boundary, by a QDMI device session. QDMI explicitly models this through session functions that create and initialise an authenticated context for a selected backend. The QDMI interface is organised around session, query, and job functions, which respectively cover authenticated access, device-property access, and execution management [20].

A second common concept is the execution target. QCI Connect represents execution targets as QPUs or QPU-like resources managed by the platform. These resources may correspond to physical quantum computers, simulators, digital twins, or other compute services. QCI Connect follows a modular connector model in which such resources are integrated through backend connectors, keeping vendor-specific complexity outside the platform core. MQSS, on the other hand, exposes quantum resources through its own resource model and, ultimately, via QDMI-compatible device interfaces. For the integration, this means that a resource identifier must always be translated: an MQSS resource name must become a QCI Connect QPU identifier in one direction, while a QCI Connect QPU identifier or alias must become the selected resource inside a QDMI session in the other direction.

Further, we note that both quantum software stacks handle resource metadata in their own ways. Before a job can be submitted, the software stack needs to know what a target resource supports. Typical metadata includes the number of qubits, current availability, supported program formats, supported operations, connectivity restrictions, shot limits, and, where available, quality metrics such as gate fidelities. QCI Connect exposes such information through its QPU model, including fields such as status, supported circuit formats, gate sets, fidelities, and qubit connectivity. QDMI exposes the same kind of information through device, site, and operation property queries. This overlap is one of the reasons why a translation is possible, but it also exposes mismatches: not every field in one API has an equally precise counterpart in the other.

Moreover, each stack must accept a representation of the quantum program that should be executed. In the current implementation, the focus is on textual circuit representations, in particular, OpenQASM 2, OpenQASM 3, and QIR were supported. However, the concept is more general than gate-based circuits. Future versions of

the stacks may also expose pulse-level programs, calibration jobs, or other execution modes. The integration, therefore, treats the program format as an explicit part of the job description rather than assuming that all resources accept the same representation.

Another concept common to both implementations is the execution configuration. A quantum job is not fully described by the program alone. It also requires execution parameters, such as the number of shots, and, depending on the backend, additional options such as whether the compiler may modify the circuit, whether the job should be queued, or whether backend-specific options should be passed along. In QCI Connect, these parameters are part of the job submission and QPU task. In MQSS, they are part of the circuit-job request submitted through the MQSS Client. In QDMI, they are set through typed job parameters and, where no standard parameter exists, through custom parameters defined by the device implementation.

The job lifecycle also needs to be mapped between the two architectures. Quantum execution is inherently asynchronous. A submitted job may be accepted, queued, executed, completed, failed, or cancelled, depending on its status. QCI Connect models this at the platform level: jobs are accepted by the orchestrator, processed as tasks, and then dispatched to workers or backend connectors. QDMI models the same lifecycle through job functions for creating a job handle, setting parameters, submitting the job, checking or waiting for its status, cancelling it, retrieving results, and freeing the handle. QCI Connect exposes its QPU-facing API through endpoints for retrieving QPU information, submitting jobs, and retrieving results, as well as additional endpoints for status and related operations. The integration must therefore translate both job identifiers and job states while preserving the asynchronous execution model.

The final common concept is the result representation. For sampling jobs, the most important result is a distribution over measured bitstrings. However, APIs differ in how they encode this distribution. One system may use compact bitstrings such as 00, while another uses array-like keys such as [0,0]. Some APIs expose raw shots, others expose histograms, probabilities, or a structured JSON object containing several result views. The implementation, therefore, normalises result payloads so that the receiving stack observes the format it expects.

3.2 QCI Connect to MQSS

The first communication direction exposes MQSS resources inside QCI Connect. In this direction, QCI Connect remains the user-facing platform. Users still submit jobs through the QCI Connect GUI, SDK, or frontend API, and the QCI Connect orchestrator continues to control authentication, job creation, task scheduling, and result persistence. MQSS is not exposed directly to the user. Instead, it is integrated as an additional

execution backend behind a QCI Connect QPU connector.

This design follows QCI Connect’s modular architecture. As described in the previous chapter, QCI Connect integrates concrete execution resources through dedicated connectors. These connectors isolate backend-specific logic from the orchestrator and communicate with the rest of the platform through the existing task and backend API model. The MQSS connector developed in this thesis uses exactly this extension point. From the perspective of QCI Connect, an MQSS resource, therefore, appears as a normal QCI Connect QPU: it can be discovered, displayed, selected for execution, monitored, and queried for results. From the connector’s perspective, however, this QPU is only a local representation of a resource that is actually accessed via the MQSS Client [21].

The integration has two main responsibilities. First, the connector must synchronise MQSS resources into the QCI Connect resource model. Second, it must translate QCI Connect QPU tasks into MQSS job requests and translate the MQSS result back into the format expected by QCI Connect. These two responsibilities correspond to the two main runtime flows of the connector: a resource-discovery flow and a job-execution flow.

3.2.1 Resource Discovery and QPU Representation

The resource-discovery flow starts from the MQSS QPU connector, which queries MQSS via its client for available resources. As a response, it receives resource descriptions containing, among others, the resource name, the number of qubits, online status, supported instructions, and connectivity information. Each such resource is converted into a QCI Connect QPU entry. This conversion is necessary because the orchestrator and frontend do not operate on MQSS resource objects directly. They expect QCI Connect’s own QPU representation.

The MQSS resource name is the most important identifier on the MQSS side. It is used again later when a job is submitted. QCI Connect, however, uses an integer QPU identifier together with an alias. The connector, therefore, maps each MQSS resource name to a stable QCI Connect QPU ID and stores the original MQSS resource name in the QPU metadata. This additional metadata is important because the mapping has to be reversible: during job execution, the connector receives only the QCI Connect QPU task and must recover the corresponding MQSS resource name from the selected QPU entry.

The connector also maps the MQSS resource status to a QCI Connect QPU status. Resources reported as online by MQSS are exposed as available QCI resources, while offline or unreachable resources are marked as unavailable. If the connector cannot reach MQSS during a status refresh, previously known MQSS-backed QPUs are marked

3 Enabling Bidirectional Communication

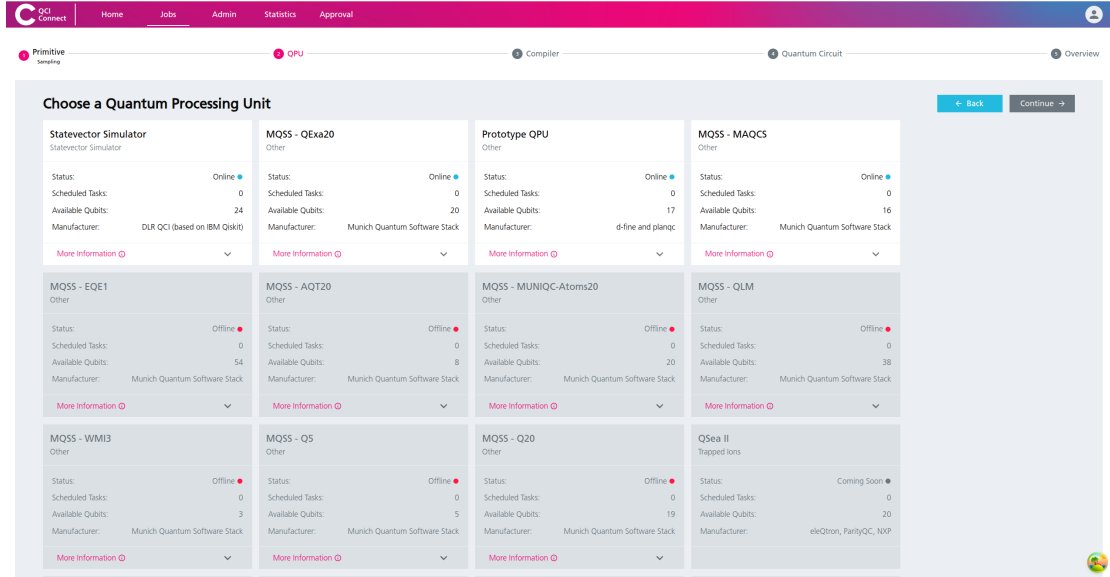


Figure 3.2: Screenshot taken from the QCI Connect web interface showing the availability of the QPUs. Resources that are queried from MQP are marked with MQSS beforehand.

offline. This prevents QCI Connect from offering resources that may no longer be reachable. At the same time, the connector adheres to QCI Connect’s platform-level status handling. For example, if an administrator has set an active maintenance event for a QPU, this platform-level status is preserved rather than overwritten by the MQSS status.

Resource metadata is mapped conservatively. The number of qubits reported by MQSS is used as both the available and maximum qubit count in QCI Connect. Instructions reported by MQSS are mapped to the accepted and native gate sets of the QCI Connect QPU entry. Connectivity information is mapped to QCI Connect’s coupling map and qubit-connectivity fields, where possible. In addition, MQSS-specific information, such as the original resource name, pending job count, raw connectivity, and serialised instruction metadata, is stored in the additional QPU information field. This keeps the QCI Connect resource model usable for existing platform components while preserving the information needed by the connector.

A relevant mismatch appears for circuit formats. QCI Connect expects each QPU to advertise the circuit formats it supports, because this information is used during job creation and validation. In the current implementation, however, the MQSS resource metadata exposed through the MQSS Client does not specify supported circuit formats

for each resource. The connector, therefore, uses a single connector-wide circuit-format setting and advertises this format for all discovered MQSS resources. This is a pragmatic solution that allows the integration to work, but it is less precise than a per-resource format description. If future MQSS Client versions expose circuit-format metadata for each resource, this part of the discovery flow can be extended without changing the overall connector architecture.

In Figure 3.2, we can see how these concepts come together and what they should look like in the QCI Connect platform. In this example, on the Munich Quantum Portal [22], two online resources were available: the QExa20 superconducting QPU and the MAQCS simulator. Resources that are not online for some reason, such as maintenance, appear offline. However, other attributes, such as qubit connectivity, remain available even if they are not online. In the QCI Connect web interface, these can be visualised by clicking the "More Information" button visible in Figure 3.2.

At the API level, we can observe in Table 3.1 how the two APIs exposed by QCI Connect [7] and the MQSS Client [21] can be mapped in order to allow information retrieval about each resource exposed in MQP. The mapping presented in this table is just an example of how a proof-of-concept implementation, as in our case, could be done. However, a production-ready implementation would require addressing at least the limitations and future work recommendations discussed in Chapter 4.

QCI Connect API field	MQSS / mqss-client API
QPUInfo.qpu_id	Derived from ResourceInfo.name by computing a stable identifier
QPUInfo.alias	mqss_<ResourceInfo.name>
QPUInfo.manufacturer	Not supported. Set by default to Munich Quantum Software Stack.
QPUInfo.name	MQSS - <ResourceInfo.name>.
QPUInfo.version	Not supported. Set to unknown.
QPUInfo.technology	Not supported. Set to QPUTechnology.OTHER.
QPUInfo.description	Set to a fixed description indicating that the resource was discovered through MQSS.
QPUInfo.status	ResourceInfo.online
QPUInfo.number_of_qubits_available	ResourceInfo.qubits.
QPUInfo.cost_center_required	No equivalent value available. Set to false.
QPUInfo.supported_circuit_formats	No per-resource value available.

QCI Connect API field	MQSS / mqss-client API
<code>QPUInfo.restricted_fields</code>	No direct attribute, but many of these are deduced based on the <code>ResourceInfo</code> object and its attributes (such as <code>ResourceInfo.connectivity</code> and <code>ResourceInfo.instructions</code>).

Table 3.1: Mapping of QCI Connect API fields relevant for resource discovery to MQSS and `mqss-client` resource information.

3.2.2 Job Submission

Once MQSS resources have been synchronised with QCI Connect, users can select them as they would other QCI Connect QPUs. The submitted job first follows the normal QCI Connect workflow. The orchestrator validates the job, stores it, creates the corresponding tasks, and sends the QPU task to the appropriate Celery queue. The MQSS connector is selected by the QPU ID range reserved for MQSS-backed resources. This means that the orchestrator does not require MQSS-specific submission logic. It only has to route the task to the correct connector.

The connector currently maps QCI Connect sampling jobs to MQSS circuit jobs. This choice reflects the common denominator of the two APIs in the implemented version: a circuit is executed on a selected resource for a fixed number of shots, yielding a distribution over measured bitstrings. Other primitives, such as quantum-state results or Hamiltonian-style execution, are not supported by the current connector.

When the MQSS connector receives a QPU task, it first checks that the task has not already been executed. This is important because Celery may retry tasks after timeouts or worker interruptions. Re-submitting the same task to a quantum backend could create duplicate executions. To prevent this, the connector uses the QCI Connect task status to block accidental resubmission after the task has already been sent to MQSS.

The connector then resolves the target resource. The QPU task contains the QCI Connect QPU ID, but MQSS expects the original MQSS resource name. This name was stored during resource synchronisation and is read from the selected QPU's additional metadata. The connector also extracts the number of shots from the QCI Connect task and selects the circuit to be executed. If the job contains the result of a previous compiler task, the compiled circuit is preferred. Otherwise, the originally submitted circuit is used. This preserves the normal QCI Connect workflow in which a job may contain both compiler tasks and QPU tasks.

After these fields are extracted, the connector constructs an MQSS circuit job request. The request contains the circuit payload, the connector-wide circuit format, the MQSS resource name, the number of shots, and two execution flags that control whether

MQSS may modify the circuit and whether the job should be queued. The request is then submitted through the MQSS Client. MQSS returns a job identifier, which the connector keeps as execution metadata so that the forwarded job can later be traced back to its MQSS execution.

QCI Connect API field	MQSS / mqss-client API
<code>QPUTask.qpu_id</code>	Used to identify the selected QCI Connect QPU. The connector resolves this QPU to the original MQSS resource name stored during resource discovery.
<code>additional_qpu_info["mqss_resource_name"]</code>	Mapped to <code>CircuitJobRequest.resource_name</code> . This is the MQSS resource on which the job is executed.
<code>QPUTask.total_shots</code>	<code>CircuitJobRequest.shots</code>
<code>JobSubmission.circuit</code>	Mapped to <code>CircuitJobRequest.circuits</code> if no previous compiler result is available.
<code>CompilerTaskResult.compiled_circuit</code>	Preferred over the originally submitted circuit and mapped to <code>CircuitJobRequest.circuits</code> .
<code>CompilerTaskResult.input_circuit</code>	Used as a fallback for <code>CircuitJobRequest.circuits</code> if the compiled circuit is not available.
<code>JobSubmission.circuit_format</code>	<code>CircuitJobRequest.circuit_format</code>
<code>QPUTask.qpu_options</code>	No direct mapping in the current implementation. MQSS-specific execution behaviour is controlled by connector-level settings.

Table 3.2: Mapping of QCI Connect job and task API fields to MQSS circuit-job parameters necessary for job submission.

In Table 3.2, we can see how the mandatory API fields are needed on both sides in order to submit a job. The exception in this case is `CircuitJobRequest.no_modify` on the MQSS side, because this parameter specifies if a job needs modification, i.e. compilation, before running it on the actual hardware or simulators. This is very important for two reasons. One would be that if, by mistake, the QCI Connect platform submits a circuit in OpenQASM 3 format instead of OpenQASM 2, by setting the parameter to *false*, this would prevent any failure on the MQSS side because the job would be compiled to the specific architecture by its internal QRM before running. Moreover, this parameter is of great importance because, for example, if a user wants to compare the hardware capabilities of two resources from different platforms, it prevents optimisations from being applied to the circuit, ensuring accurate benchmarking results.

3.2.3 Status Handling and Result Translation

After submission, the connector waits for the MQSS job to reach a terminal state. It periodically polls MQSS through the MQSS Client. If MQSS reports completion, the connector retrieves the result. If MQSS reports failure or cancellation, the connector translates this into a QCI Connect task failure. If no result is returned before the configured timeout, the connector also reports an execution error to QCI Connect.

The most important part of the result translation is converting MQSS counts into the QCI Connect sample-distribution format. MQSS results may encode counts in several shapes depending on the resource or simulator. For example, the result may be a dictionary from bitstrings to counts, a list of sample strings, or a nested object that contains a counts field. The connector normalises these variants into a single QCI Connect representation. In particular, compact bitstrings such as 10 are converted into QCI Connect's sample-key convention [1,0]. The executed-shot count is then computed as the sum of the normalised distribution.

The final result returned to QCI Connect is a normal QCI Connect QPU task result. It contains the sample distribution, the number of executed shots, timing information (where available), the original or compiled circuit, and additional metadata including the MQSS job identifier and MQSS resource name. From the perspective of the QCI Connect orchestrator, the result therefore looks like that of any other QPU connector. This is the central benefit of placing the integration at the connector boundary: MQSS-specific execution details remain inside the connector, while the rest of QCI Connect can reuse its existing result-storage and result-display mechanisms.

3.2.4 Discussion

The QCI Connect to MQSS direction demonstrates that MQSS can be exposed as an execution backend without changing the platform frontend, the QCI Connect SDK: API Client, or job model. The integration reuses QCI Connect's existing connector abstraction and maps MQSS resources into QCI Connect QPU entries. This keeps the user experience consistent: users select an MQSS-backed resource in the same way as any other QCI Connect resource.

At the same time, the implementation reveals several semantic mismatches. The most important one is that QCI Connect expects per-resource circuit-format metadata, while the currently used MQSS Client resource descriptions do not expose this information. The connector, therefore, uses a connector-wide circuit-format setting. A second limitation is that the implementation currently supports only sampling jobs. This is sufficient for demonstrating end-to-end execution of circuit-sampling workloads, but it does not cover the full range of possible quantum workloads. A third limitation is

3 Enabling Bidirectional Communication

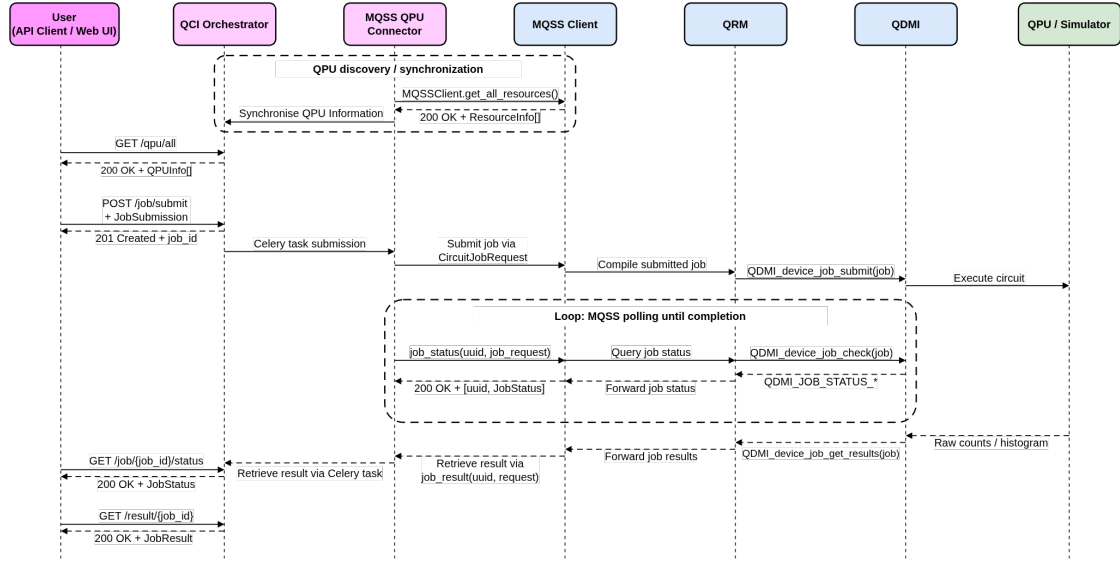


Figure 3.3: High-level overview of an end-to-end execution flow between QCI Connect and MQSS.

that resource metadata is mapped only where a clear correspondence exists. Some MQSS details have no direct QCI Connect field and are therefore stored as additional metadata rather than as first-class platform properties.

All these concepts described in the previous subsections are brought together in Figure 3.3. It shows an example of how the interaction between QCI Connect and MQSS can look. The process starts with discovering MQSS resources and representing them as QCI Connect QPUs. When a user submits a job, the connector resolves the selected QPU to the corresponding MQSS resource, forwards the circuit together with the relevant execution parameters, and monitors the resulting asynchronous MQSS job. Once execution has finished, the connector translates the final status and normalises the returned counts into the sample-distribution format expected by QCI Connect. The figure, therefore, summarises the full path from resource discovery and job submission to status handling and result retrieval.

Overall, this direction shows that the two software stacks can be connected without collapsing their architectures into a single one. QCI Connect remains responsible for users, jobs, scheduling, and results, while MQSS remains responsible for the actual execution behind the connector. The translation layer is therefore narrow and localised.

3.3 QDMI to QCI Connect

The second communication direction exposes QCI Connect through a QDMI device. In this direction, MQSS or any other QDMI client remains the caller. The client interacts only with the QDMI interface and does not need to know that the underlying execution is handled by QCI Connect. The QCI Connect platform is hidden behind a QDMI device implementation that translates QDMI session, query, and job calls into QCI Connect REST requests.

This direction is different from the previous one. In the QCI Connect to MQSS direction, the integration is a QCI Connect connector. In the QDMI-to-QCI Connect direction, the integration is a shared library that implements the QDMI C interface. The outer interface is therefore QDMI, and QCI Connect becomes the backend service wrapped by the device implementation. This is consistent with QDMI's role in MQSS: QDMI is the backend management boundary, and a concrete QDMI device hides the details of the system that actually executes the job.

The implementation follows the QDMI lifecycle. A client first initialises the device, allocates a session, sets session parameters, initialises the session, queries device properties if needed, creates a job, configures the job, submits it, checks or waits for completion, retrieves results, and finally frees the job and session. Each of these steps is mapped to local state changes or to QCI Connect API requests.

3.3.1 Session Management and Resource Selection

In QDMI, a session represents the context in which a client communicates with a device. For the QCI-backed QDMI device, this context contains the QCI Connect API base URL, the authentication token, and the selected QCI Connect QPU. The base URL is set through the standard QDMI session parameter for backend URLs. The token is set through the standard QDMI token parameter and is used as the API key when sending requests to QCI Connect.

The complete session-initialisation path can therefore be understood as a sequence of translation steps between the QDMI session model and the QCI Connect resource model, as illustrated in Figure 3.4. The QDMI client first provides the connection context through the base URL and authentication token, then selects a QCI Connect resource by ID or alias. During session initialisation, the device queries the available QPUs, verifies the selected target, and stores the relevant QPU metadata locally. This cached information then serves as the basis for subsequent QDMI query calls, allowing device, site, and operation properties to be returned through the QDMI interface while the underlying resource description is provided by QCI Connect.

Resource selection is represented through custom QDMI session parameters. The

3 Enabling Bidirectional Communication

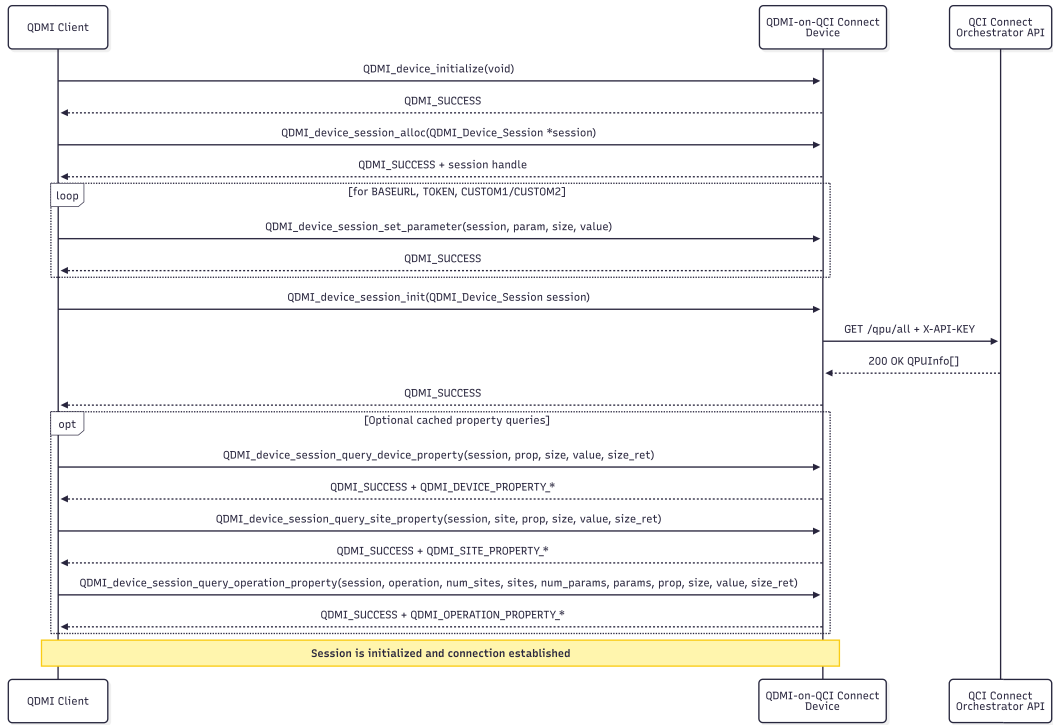


Figure 3.4: QDMI session initialisation and resource selection for the QDMI-on-QCI Connect device.

implementation allows the client to select a QCI Connect resource either by QPU ID or by alias. This is necessary because QDMI does not prescribe QCI Connect’s concrete resource-identification model. During session initialisation, the device implementation calls the QCI Connect endpoint, which returns the available QPUs, selects the requested resource, checks whether the current user has access to it, and caches the relevant metadata in the QDMI session.

This cached metadata is used for later query calls. The QDMI client can request device, site, and operation properties, while the implementation answers these queries using the selected QCI Connect QPU description. This avoids repeated network requests for each property query and provides the QDMI session with a stable view of the selected resource. The trade-off is that metadata changes in QCI Connect are not automatically visible until a new session is initialised.

The mapping from QCI Connect resource metadata to QDMI properties follows the same conceptual categories introduced in Section 3.1. The QCI Connect QPU name, version, status, number of qubits, coupling map, supported circuit formats, and gate-set

information are translated into the corresponding QDMI device properties. Qubit sites are generated from the reported qubit count and named `q[0]`, `q[1]`, and so on. Operation objects are derived from the QCI Connect gate-set metadata. Where QCI Connect provides gate fidelities, they are exposed as QDMI operation fidelities.

Some information has to be inferred. For example, the arity and number of parameters of an operation are not always represented in exactly the form required by QDMI. The implementation, therefore, infers these values for common gate names. For single-qubit operations, all sites are considered possible operation sites. For two-qubit operations, the applicable sites are derived from the coupling map. Pulse-level support is reported as unsupported because the current QCI Connect mapping exposes only the gate-level execution path for this QDMI device.

QDMI parameter	QCI Connect API
QDMI_DEVICE_SESSION_PARAMETER_BASEURL	<code>https://connect.qci.dlr.de/</code> or for local development <code>https://localhost</code>
QDMI_DEVICE_SESSION_PARAMETER_TOKEN	Authentication token (to be created beforehand on the QCI Connect platform)
QDMI_DEVICE_SESSION_PARAMETER_CUSTOM1	<code>QPUInfo.qpu_id</code>
QDMI_DEVICE_SESSION_PARAMETER_CUSTOM2	<code>QPUInfo.alias</code>
QDMI_DEVICE_PROPERTY_NAME	<code>QPUInfo.name</code> ; falls back to the selected alias if no name is available
QDMI_DEVICE_PROPERTY_VERSION	<code>QPUInfo.version</code> ; falls back to unknown
QDMI_DEVICE_PROPERTY_STATUS	<code>QPUInfo.status</code>
QDMI_DEVICE_PROPERTY_QUBITSNUM	<code>QPUInfo.number_of_qubits_available</code>
QDMI_DEVICE_PROPERTY_SITES	Generated from the selected QPU's qubit count. Sites are named <code>q[0]</code> , <code>q[1]</code> , and so on
QDMI_DEVICE_PROPERTY_OPERATIONS	Derived from <code>QPUInfo.restricted_fields.gate_set.native_gate_set</code> ; if unavailable, derived from the accepted gate set
QDMI_DEVICE_PROPERTY_COUPLINGMAP	Derived from <code>QPUInfo.restricted_fields.base_coupling_list</code> ; if unavailable, derived from qubit-connectivity metadata
QDMI_DEVICE_PROPERTY_SUPPORTEDPROGRAMFORMATS	<code>QPUInfo.supported_circuit_formats</code>
QDMI_DEVICE_PROPERTY_PULSESUPPORT	Not supported

Table 3.3: Mapping of QDMI session and query concepts to QCI Connect QPU metadata.

A translation of most of this metadata information appears in Table 3.3. The table shows that the QDMI session and query interfaces can be implemented from the QCI Connect QPU description without exposing QCI Connect’s internal platform model to the QDMI caller. At the same time, it makes the mapping’s boundary explicit. Identifiers, status information, qubit counts, supported formats, sites, operations, and fidelities can be translated directly or derived from cached QPU metadata. However, information that is not present in the selected QCI Connect resource must either be reported as unsupported or represented through implementation-specific extensions.

3.3.2 Job Configuration and Submission

After a session has been initialised, the QDMI client can create a job handle. The job is initially local to the QDMI device implementation. It stores the selected session, the program format, the program payload, the number of shots, and several QCI-specific optional fields. No QCI Connect job is created until the client explicitly calls the QDMI submit function.

The standard QDMI job parameters are mapped to the central fields of a QCI Connect job submission. The QDMI program becomes the QCI Connect circuit payload. The QDMI program format is translated into the corresponding QCI Connect circuit format: QASM2 becomes OPENQASM2, QASM3 becomes OPENQASM3, and QIR base string becomes QIR. The QDMI shot count becomes the total number of shots in the QCI Connect QPU task. The implementation checks that the selected format is supported by the selected QCI Connect QPU and that the number of shots falls within the resource’s advertised limits.

QDMI also provides custom job parameters that expose QCI-specific fields that lack direct equivalents in the standard QDMI. These fields include the QCI Connect job name, primitive, comment, cost bearer, and QPU options. The primitive is mapped to QCI Connect’s job primitive field and is currently restricted to the values accepted by the implementation. The QPU options are stored as JSON because the QCI Connect schema allows a nested and extensible options object.

When the job is submitted, the implementation constructs a QCI Connect job submission. The submission contains the circuit, circuit format, primitive, name, optional comment, optional cost bearer, and a task list. The current implementation always creates a QPU task targeting the QPU selected during session initialisation. If a compiler task has been configured, it is inserted before the QPU task, which allows the QCI Connect workflow to compile the circuit before execution. The job submission is then sent to QCI Connect using the job-submission endpoint. The returned QCI Connect job identifier is stored inside the QDMI job object and becomes the QDMI job ID.

This job-submission flow is summarised in Figure 3.5, which shows how the locally

3 Enabling Bidirectional Communication

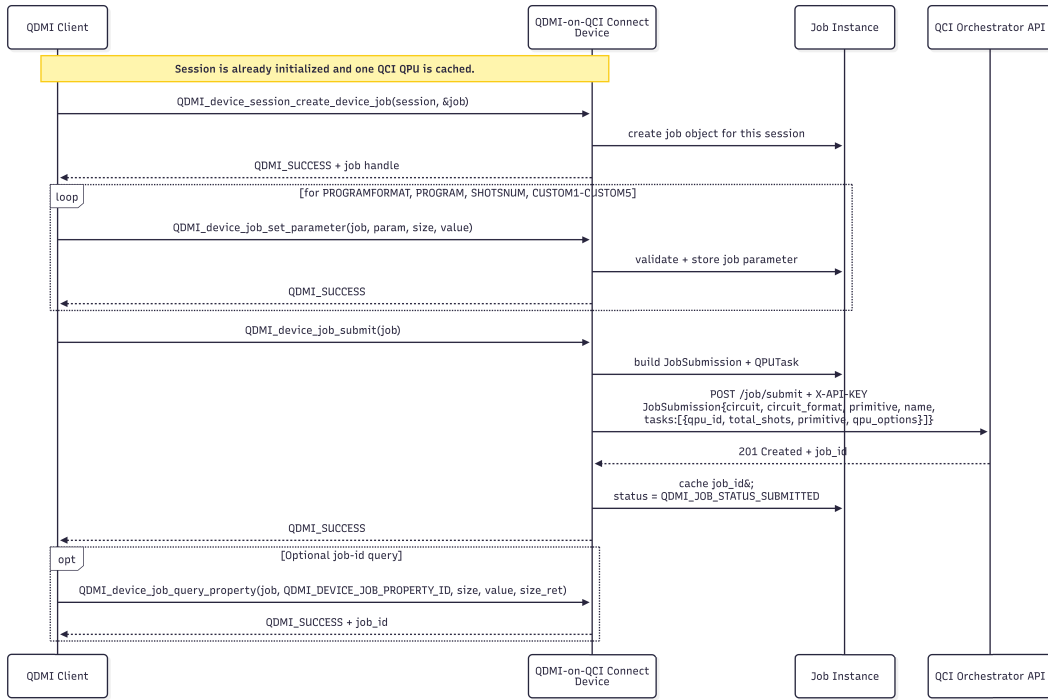


Figure 3.5: QDMI job creation, configuration, and submission through the QDMI-on-QCI Connect device.

configured QDMI job is turned into a platform-level QCI Connect request. Standard QDMI job parameters are stored locally and translated into a QCI Connect job submission containing a QPU task for the selected QCI Connect resource. The program payload, program format, and shot count form the core of this translation, while custom QDMI parameters carry QCI-specific fields such as the job name, primitive, comment, cost bearer, and QPU options. The figure, therefore, illustrates the point at which the QDMI lifecycle leaves the local device implementation and becomes an actual QCI Connect job managed by the platform.

QDMI job parameter or function	QCI Connect API
QDMI_DEVICE_JOB_PARAMETER_PROGRAMFORMAT	Mapped to <code>JobSubmission.circuit_format</code> . QDMI_PROGRAM_FORMAT_QASM2 → OPENQASM2, QDMI_PROGRAM_FORMAT_QASM3 → OPENQASM3, and QDMI_PROGRAM_FORMAT_QIRBASESTRING → QIR.
QDMI_DEVICE_JOB_PARAMETER_PROGRAM	<code>JobSubmission.circuit</code>

QDMI job parameter or function	QCI Connect API
QDMI_DEVICE_JOB_PARAMETER_SHOTSNUM	QPUTask.total_shots
QDMI_DEVICE_JOB_PARAMETER_CUSTOM1	JobSubmission.name
QDMI_DEVICE_JOB_PARAMETER_CUSTOM2	Mapped to both JobSubmission.primitive and QPUTask.primitive
QDMI_DEVICE_JOB_PARAMETER_CUSTOM3	JobSubmission.comment
QDMI_DEVICE_JOB_PARAMETER_CUSTOM4	JobSubmission.cost_bearer_name
QDMI_DEVICE_JOB_PARAMETER_CUSTOM5	QPUTask.qpu_options
QDMI_DEVICE_JOB_PROPERTY_PROGRAMFORMAT	JobSubmission.circuit_format
QDMI_DEVICE_JOB_PROPERTY_PROGRAM	JobSubmission.circuit
QDMI_DEVICE_JOB_PROPERTY_SHOTSNUM	QPUTask.total_shots
QDMI_DEVICE_JOB_PROPERTY_CUSTOM1–CUSTOM5	JobSubmission.name, JobSubmission.primitive, JobSubmission.comment, JobSubmission.cost_bearer_name, and QPUTask.qpu_options

Table 3.4: Mapping of QDMI job configuration and submission parameters to QCI Connect API fields.

The mapping used for job configuration and submission is given in Table 3.4. In contrast to the session and query mapping in Table 3.3, this table concerns values that are provided by the QDMI client at runtime. The standard QDMI parameters cover the program format, program payload, and number of shots, while QCI Connect-specific information, such as a job name, primitive, comment, cost bearer, or backend options, is carried through custom QDMI parameters. This separation keeps the common execution path portable while still allowing the QDMI device to expose QCI Connect fields that lack a standard QDMI equivalent.

The table also highlights a recurring design pattern in the implementation: QDMI remains the outer interface, and QCI Connect-specific fields are introduced only at the translation boundary. A generic QDMI client can therefore submit a simple sampling job without knowing about the QCI Connect schema, while a more specialised client can still use custom parameters to access additional QCI Connect functionality when needed.

3.3.3 Status Handling, Cancellation, and Results

After submission, the QDMI job object contains the QCI Connect job identifier returned by the platform. Status handling is implemented by polling the QCI Connect job-status endpoint and translating the returned platform state into the QDMI job-state

model. This mapping is intentionally coarser than the internal QCI Connect workflow: pending jobs are represented as queued QDMI jobs, running jobs as running QDMI jobs, successful jobs as done, cancelled jobs as cancelled, and failed jobs as failed. The QDMI wait function builds on the same mechanism by repeatedly invoking the status check until a terminal state is reached or the configured timeout expires.

Cancellation is forwarded to QCI Connect through the corresponding cancellation endpoint. If QCI Connect accepts the cancellation request, the local QDMI job state is set to cancelled. This again preserves the QDMI lifecycle, since the client does not need to know how QCI Connect implements cancellation. It only observes the QDMI job state.

Result retrieval is more involved because QDMI exposes several result views, while QCI Connect returns a structured job result. The implementation first retrieves the QCI Connect job result and caches it inside the QDMI job object. It then extracts the last QPU result and, where present, the last compiler result. For sampling jobs, the QCI Connect sample distribution is converted into QDMI result channels. The QDMI client can request raw shots, histogram keys and values, dense probabilities, or sparse probability keys and values. The implementation derives these views from the same QCI Connect distribution.

The interaction between these operations is illustrated in Figure 3.6, which groups status polling, cancellation, and result retrieval within a single post-submission workflow. After a job has been submitted, the device uses the stored QCI Connect job identifier to communicate with the corresponding platform job: it polls the job status while execution is ongoing, forwards cancellation requests when needed, and retrieves available result data once it becomes available. The device maps QCI Connect job states to QDMI job states and converts the structured QCI Connect result into QDMI result channels. For sampling jobs, this means deriving the QDMI result views described above, such as shots, histograms, sparse probabilities, and dense probabilities, from the sample distribution returned by QCI Connect.

The QDMI custom result channels expose QCI-specific result information that does not fit into the standard QDMI sampling result categories. One custom result returns the complete QCI Connect job result as JSON. Other custom results expose the aggregated QPU data, the raw QPU data, population errors, or the compiled circuit returned by a preceding compiler task. These custom channels are useful for preserving QCI Connect information without extending the QDMI standard result enumeration.

Statevector results are currently reported as unsupported. This does not mean that QCI Connect can never represent such data, but only that the implemented QDMI mapping does not expose statevector result channels. The current focus is on the sampling-style result model, which is common to both systems and sufficient for the integration scenario tested in this thesis.

3 Enabling Bidirectional Communication

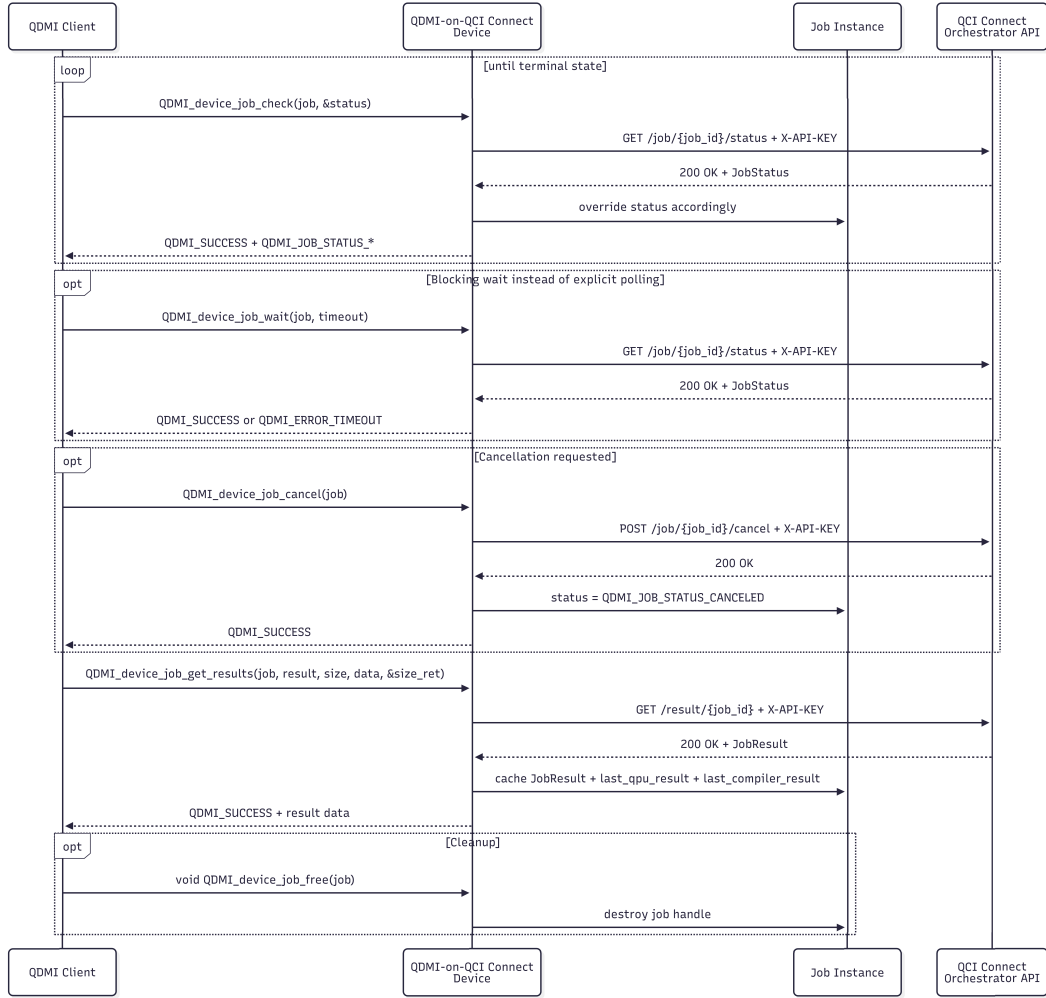


Figure 3.6: QDMI status polling, cancellation, and result retrieval for a QCI Connect job.

The status and result mappings used after a job has been submitted are collected in Table 3.5. The upper part of the table shows how QCI Connect job states are reduced to the QDMI status categories. The lower part shows how the structured QCI Connect result object is exposed through the result channels expected by QDMI clients. This is where most result-level information loss can occur: sampling distributions, histograms, and probabilities can be derived from the QCI Connect result, but richer platform-specific information must be returned through custom result channels. Statevector-

style QDMI result requests are deliberately reported as unsupported in the current implementation because no complete, validated translation for these result channels has been implemented yet.

Together, Tables 3.4 and 3.5 describe the complete QDMI job lifecycle for the QCI-backed device. The first table explains how a QDMI job is turned into a QCI Connect submission, while the second table explains how the resulting QCI Connect execution state and result data are translated back into QDMI. This closes the conceptual loop from job creation to result retrieval.

QDMI function, status, or result	QCI Connect API mapping
QDMI_JOB_STATUS_QUEUED	QCI Connect job status PENDING
QDMI_JOB_STATUS_RUNNING	QCI Connect job status RUNNING
QDMI_JOB_STATUS_DONE	QCI Connect job status SUCCEEDED
QDMI_JOB_STATUS_CANCELED	QCI Connect job status CANCELED
QDMI_JOB_STATUS_FAILED	QCI Connect job status FAILED
QDMI_JOB_RESULT_SHOTS	<code>JobResult.last_qpu_result.data</code> ; If unavailable, shots are reconstructed from <code>JobResult.last_qpu_result.aggregated_data.sample_distribution</code>
QDMI_JOB_RESULT_HIST_KEYS	Keys of <code>JobResult.last_qpu_result.aggregated_data.sample_distribution</code>
QDMI_JOB_RESULT_HIST_VALUES	Mapped from the count values of <code>JobResult.last_qpu_result.aggregated_data.sample_distribution</code>
QDMI_JOB_RESULT_PROBABILITIES_SPARSE_KEYS	<code>JobResult.last_qpu_result.aggregated_data.populations</code> , if populations is unavailable, the keys are derived from <code>sample_distribution</code>
QDMI_JOB_RESULT_PROBABILITIES_SPARSE_VALUES	<code>JobResult.last_qpu_result.aggregated_data.populations</code> , if populations is unavailable, probabilities are computed from <code>sample_distribution</code> and <code>executed_shots</code>
QDMI_JOB_RESULT_PROBABILITIES_DENSE	Constructed by expanding the sparse probability representation into a dense vector indexed by the integer value of each bitstring
QDMI_JOB_RESULT_CUSTOM1	JobResult JSON returned by <code>GET/result/{job_id}</code>
QDMI_JOB_RESULT_CUSTOM2	<code>JobResult.last_qpu_result.aggregated_data</code>
QDMI_JOB_RESULT_CUSTOM3	<code>JobResult.last_qpu_result.data</code>

QDMI function, status, or result	QCI Connect API mapping
QDMI_JOB_RESULT_CUSTOM4	<code>JobResult.last_qpu_result.aggregated_data.population_errors</code>
QDMI_JOB_RESULT_CUSTOM5	<code>JobResult.last_compiler_result.compile_d_circuit</code> , if a compiler result exists
QDMI_JOB_RESULT_STATEVECTOR_*	No QCI Connect result field is mapped in the current implementation. Reported as unsupported.

Table 3.5: Mapping of QDMI job-status and result requests to concrete QCI Connect status and result fields.

3.3.4 Discussion

The QDMI to QCI Connect direction shows that QCI Connect can be represented as a QDMI-compatible device without requiring MQSS to understand the internal structure of QCI Connect. The QDMI client interacts with a standard device, session, query, and job lifecycle. The implementation translates this lifecycle into QCI Connect resource selection, job submission, status polling, cancellation, and result retrieval.

The main advantage of this direction is that it places QCI Connect behind the backend-management abstraction used by MQSS. This is useful because MQSS already expects new execution targets to be integrated through QDMI devices. As a result, QCI Connect can be treated like another QDMI-accessible backend, even though internally, it is a full platform with its own architecture.

The main limitation is that the QDMI device exposes a snapshot of one selected QCI Connect resource per initialised session. This fits the QDMI session model, but it means that switching to another QCI Connect QPU requires another session configuration. A second limitation is that some QCI Connect concepts lack a standard QDMI equivalent and must be represented using custom parameters or custom results. This is acceptable for an implementation-level integration, but it also shows where semantic differences remain between the two APIs. Finally, the current implementation focuses on gate-level circuit execution and sampling-style results. More advanced result types, pulse-level execution, and richer compiler workflows would require additional mapping logic or extensions in future work.

Together with the QCI Connect to MQSS connector, this QDMI device completes the bidirectional integration. In both directions, the implementation preserves the outer software stack’s native abstraction: QCI Connect users see MQSS resources as several QCI Connect QPUs, while QDMI clients see QCI Connect as a QDMI device.

4 Limitations and Future Work

The implementation presented in this thesis demonstrates that bidirectional communication between MQSS and QCI Connect is possible without introducing a new interface between the two software stacks. Nevertheless, the implementation should be understood as a proof-of-concept integration rather than as a production-ready standardisation layer. Its purpose is to show where the two APIs align, where translation is possible with limited assumptions, and where semantic gaps remain. This chapter summarises the main limitations of the current work and outlines future work needed to make such integrations more complete, robust, and maintainable.

4.1 Limitations

A first limitation concerns the supported workload model. The current QCI Connect to MQSS connector focuses on sampling-style circuit execution. This is the common denominator between the implemented QCI Connect QPU task model and the currently used MQSS Client circuit-job interface. Other possible execution modes, such as quantum-state results, Hamiltonian-based workflows, calibration jobs, or pulse-level programs, are not covered in this direction. In the opposite direction, the QDMI device for QCI Connect can forward the selected primitive to QCI Connect, but statevector-style QDMI result channels are not fully mapped. Consequently, the integration demonstrates end-to-end execution for the most relevant gate-based sampling scenario, but it does not yet cover all workload types that future quantum software stacks may require.

A second limitation is the incompleteness of metadata translation. QCI Connect expects each QPU to provide detailed resource information, including supported circuit formats, qubit counts, connectivity, gate sets, shot limits, and restricted fields. MQSS resources expose a partially overlapping set of information through the MQSS Client, but not every QCI Connect field has a direct MQSS equivalent. In particular, per-resource circuit-format metadata is not available in the currently used MQSS Client interface. The connector, therefore, uses a connector-wide format setting. This is sufficient for the proof-of-concept implementation, but it is less precise than a real per-resource capability description and may become problematic once different MQSS resources accept different program representations.

A related limitation occurs in the QDMI to QCI Connect direction. A QDMI session represents one selected QCI Connect resource. During session initialisation, the implementation queries QCI Connect, selects the requested QPU, and caches the relevant metadata. This fits the QDMI lifecycle, but it also means that changes in QCI Connect resource metadata are not automatically visible inside an already initialised session. A production implementation should define a clearer policy for metadata freshness, especially for dynamic properties such as availability, calibration-dependent metrics, queue information, or maintenance states.

Another limitation is the granularity of status handling. QCI Connect and QDMI both model asynchronous execution, but their status vocabularies differ. The implementation maps QCI Connect states, such as pending, running, succeeded, cancelled, and failed, to the corresponding QDMI job states. This preserves the high-level lifecycle, but it may hide intermediate workflow details relevant to debugging or platform operation. The same problem appears in the other direction when MQSS execution information is translated back into a QCI Connect task result. A future integration should preserve additional status details as structured metadata without breaking the generic status model expected by the outer software stack.

Result translation is another source of semantic loss. Sampling results can be represented as bitstring distributions in both systems, but the concrete encodings differ. The current implementation normalises these encodings and derives histograms, probabilities, and raw-shot-like views where possible. However, not all result information has a standard counterpart. For this reason, QCI Connect-specific result data is exposed through custom QDMI result channels in the QDMI device, while MQSS-specific metadata is stored as additional information in the QCI Connect result in the connector direction. This approach preserves information for specialised clients, but it is not as portable as a standardised result schema.

The implementation also depends on APIs that are still evolving. QDMI is designed as an extensible backend-management interface, while QCI Connect is an actively developed platform with an open API specification [7, 10]. The mappings implemented in this thesis are therefore tied to the versions and behaviours available during the thesis work. Future API changes may require updates to field names, supported status values, authentication behaviour, or result formats. This is not a weakness of either system, but it shows that long-term interoperability requires explicit versioning, capability negotiation, and backwards-compatible extension mechanisms.

Finally, the integration was validated primarily as a functional prototype. It demonstrates the feasibility of both communication directions and provides a concrete basis for comparison, but it does not yet include a complete production test suite, systematic performance measurements, or extensive fault-injection experiments. In particular, large-scale concurrent submissions, long-running workloads, network failures, expired

credentials, platform restarts, and evolving backend availability were not exhaustively evaluated. These aspects are essential before such an integration can be used as a reliable operational component.

4.2 Future Work

The most immediate future work is to align the API concepts used by QDMI and QCI Connect. The mappings in Chapter 3 show that both systems share the same central concepts: resources, jobs, execution parameters, status values, and results. However, several fields still require custom parameters, connector-level assumptions, or additional metadata. A useful next step would be to define a small set of shared capability descriptors for circuit formats, primitives, shot limits, compiler requirements, connectivity, and result types. Such descriptors would reduce the need for ad hoc translation and make it easier to add new resources without modifying the integration logic.

A second direction is improved capability discovery. In the QCI Connect to MQSS direction, MQSS should expose more precise per-resource information through the MQSS Client, especially supported program formats and execution modes. In the QDMI-to-QCI Connect direction, the QDMI device should expose more of QCI Connect’s QPU metadata through standard QDMI device, site, and operation properties, where possible. Where no standard property exists, future QDMI extensions could define commonly used fields rather than relying only on custom parameters. This would support the broader goal of reducing fragmentation at the backend interface level, which is also emphasised by current standardisation efforts, such as DIN SPEC 91520 [5].

A third direction is support for richer workload and result types. The current implementation focuses on gate-based sampling because this is the most direct common execution model. Future work should add support for quantum-state primitives provided by QCI Connect resources and define how these results should be exposed through QDMI. Similarly, probability distributions, raw shots, population errors, statevectors, and backend-specific diagnostic data should be represented in a more uniform way. This is especially important for application developers who want to compare results across different platforms without rewriting result-processing code for every backend.

Another important extension concerns compilation and workflow control. QCI Connect can represent workflows that include compiler tasks before QPU execution, whereas MQSS includes compilation and resource management functionality in its middle layer. The current integration keeps these mechanisms mostly separate. Future

work should investigate how to coordinate compiler selection, no-modification flags, benchmarking modes, and backend-specific transpilation requirements across the two systems. For example, if a QCI Connect job is forwarded to MQSS, the connector should be able to decide whether MQSS is allowed to compile the circuit, whether QCI Connect has already compiled it, or whether the job is intended as a benchmarking run where additional optimisation should be disabled.

The integration should also be hardened for production use. This includes a more complete automated test suite for both directions, including unit tests for individual mapping functions and integration tests against local platform deployments. It also includes better error reporting, retry policies, timeout handling, cancellation behaviour, and idempotence guarantees. Since both systems manage asynchronous jobs, failures can occur at several points: before submission, after submission but before the external job identifier is stored, while polling status, or during result retrieval. A robust implementation should handle each of these cases explicitly and expose meaningful errors to the outer stack.

Authentication and authorisation should also be revisited. The proof-of-concept implementation uses the available token and API-key mechanisms to authenticate requests between the systems. In a production setting, user identity, access rights, quotas, and cost accounting may need to be propagated more carefully. This is especially relevant because QCI Connect is a multi-user platform with its own identity integration, whereas QDMI sessions are lower-level execution contexts. A future integration should clarify whether forwarded jobs are executed under a service identity, under the original user identity, or under a delegated credential model.

A further direction is performance evaluation. The integration introduces additional software layers, network calls, and polling loops. Although these overheads are expected to be small compared to queueing and execution times for many quantum workloads, they should still be measured. Useful metrics include job-submission latency, status-polling overhead, result-retrieval latency, throughput under concurrent submissions, and the cost of metadata synchronisation. Such measurements would help identify whether the connector and QDMI device are suitable for interactive use, batch execution, or larger hybrid workflows.

In the longer term, the integration should be extended beyond gate-level circuits. Recent work on MQSS Pulse shows that future HPCQC software stacks may need to support pulse-level programs and lower-level hardware descriptions such as ports, frames, and waveforms [8]. QCI Connect also aims to evolve with future hardware capabilities and more complex hybrid workflows [6]. Supporting such features across both systems will require richer program-format metadata, more expressive backend properties, and result models that can represent calibration and control information. The work in this thesis provides a starting point for such extensions, but the current

implementation does not yet address them.

Finally, future work should generalise the lessons learned from this specific integration. QCI Connect and MQSS are two concrete stacks, but the same questions arise whenever independent quantum software platforms need to interoperate. Identifying shared concepts and those that require translation is crucial. It is also important to recognise any information that may be lost in the process. Additionally, determining which fields should become part of a standard interface can greatly enhance interoperability. Addressing these aspects across multiple platforms would help identify stable backend management abstractions and contribute to the development of more portable, vendor-neutral quantum software ecosystems.

5 Conclusion

This thesis investigated how two quantum software stack approaches, MQSS and QCI Connect, can be connected at their backend management boundaries. The work was motivated by the observation that quantum computing platforms increasingly aim to provide unified access to heterogeneous resources. Still, they often expose different APIs, execution models, and architectural assumptions. Interoperability between such systems is therefore not only a matter of forwarding HTTP requests or wrapping one library with another. It requires understanding which concepts are shared, how they are represented, and where semantic mismatches appear.

The first contribution of this thesis is the conceptual comparison of the two systems. MQSS structures access to quantum resources through a layered architecture in which QDMI serves as the backend-management interface. QCI Connect, in contrast, exposes a full-stack platform model in which users submit jobs through a frontend API, the orchestrator manages workflows, and backend connectors integrate concrete QPUs, simulators, and related services. Despite these architectural differences, both systems share a set of core concepts: authenticated interaction contexts, resource descriptions, program formats, execution parameters, asynchronous jobs, job statuses, cancellation, and classical result data. Identifying these common concepts enabled bidirectional integration without introducing an additional interface between the two stacks.

The second contribution is the QCI Connect to MQSS connector. In this direction, MQSS resources are represented as QCI Connect QPUs. The connector discovers available MQSS resources, maps them into the QCI Connect QPU model, forwards QCI Connect QPU tasks as MQSS circuit jobs, monitors their execution, and translates MQSS results back into QCI Connect result objects. From the QCI Connect perspective, MQSS therefore becomes another execution backend. The rest of the QCI Connect platform can continue to use its existing job management, result storage, and user interface mechanisms.

The third contribution is the QDMI device for QCI Connect. In this direction, QCI Connect is represented as a QDMI-compatible device. A QDMI client can initialise a session, select a QCI Connect resource, query metadata, configure a job, submit it, monitor its status, cancel it, and retrieve results through the standard QDMI lifecycle. Internally, the device implementation translates these calls into QCI Connect API requests. This allows QCI Connect to be accessed from the MQSS side without requiring

MQSS components to understand the QCI Connect platform’s internal structure.

The implementation confirms that the two systems can be connected through their existing extension points. However, it also shows that interoperability is not equivalent to full semantic equivalence. Some concepts map directly, such as shot counts, program payloads, selected resources, and basic job states. Other concepts require assumptions, custom parameters, or additional metadata. Examples include per-resource circuit-format information, detailed status values, QCI Connect-specific job fields, and result channels that lack a standard counterpart in QDMI. These mismatches are not merely implementation details; they indicate where future interface alignment would be useful.

Overall, the thesis demonstrates that QDMI and QCI Connect can interoperate while preserving the abstractions of both systems. QCI Connect users can access MQSS-backed resources through the usual platform workflow, and QDMI clients can access QCI Connect through the usual device-session-job lifecycle. The broader lesson is that backend interoperability in quantum software stacks is achievable when systems expose clear extension points and explicitly map them. At the same time, the work highlights the importance of standardised metadata, capability discovery, status models, and result schemas for future quantum software ecosystems.

The presented implementation is therefore both a concrete integration and a case study in quantum software-stack interoperability. It provides a basis for further work on richer workload types, stronger API alignment, production hardening, and future extensions such as pulse-level execution and more advanced hybrid workflows. As quantum computing platforms continue to evolve, such interoperability layers will be essential to avoid isolated ecosystems and enable portable access to heterogeneous quantum resources.

List of Figures

2.1	Comparison of access models for quantum devices. Figure taken from [4], figure 5.	6
2.2	The high-level architecture of the MQSS taken from [4], suggesting a layered approach.	7
2.3	The rationale of QDMI taken from [11], where the clients, driver and devices communicate via session, query and job interfaces, each with specific purposes.	9
2.4	High-level interaction model of QCI Connect within the DLR QCI infrastructure. The figure shows QCI Connect as an access layer between different stakeholder groups and available compute resources, including quantum computers, HPC resources, simulators, and cloud services. Figure taken from [16].	11
2.5	QCI Connect high-level architecture illustrating communication between the main components of the platform and the external compute resources.	12
3.1	High-level architecture of how the implementation bridges the two software stacks. In both cases, the integration extends the lowest layer in their stacks, closest to the external hardware resources.	15
3.2	Screenshot taken from the QCI Connect web interface showing the availability of the QPUs. Resources that are queried from MQP are marked with MQSS beforehand.	19
3.3	High-level overview of an end-to-end execution flow between QCI Connect and MQSS.	24
3.4	QDMI session initialisation and resource selection for the QDMI-on-QCI Connect device.	26
3.5	QDMI job creation, configuration, and submission through the QDMI-on-QCI Connect device.	29
3.6	QDMI status polling, cancellation, and result retrieval for a QCI Connect job.	32

List of Tables

3.1	Mapping of QCI Connect API fields relevant for resource discovery to MQSS and <code>mqss-client</code> resource information.	21
3.2	Mapping of QCI Connect job and task API fields to MQSS circuit-job parameters necessary for job submission.	22
3.3	Mapping of QDMI session and query concepts to QCI Connect QPU metadata.	27
3.4	Mapping of QDMI job configuration and submission parameters to QCI Connect API fields.	30
3.5	Mapping of QDMI job-status and result requests to concrete QCI Connect status and result fields.	34

Bibliography

- [1] J. Preskill. “Quantum Computing in the NISQ era and beyond.” In: *Quantum* 2 (Aug. 2018), p. 79. ISSN: 2521-327X. DOI: 10.22331/q-2018-08-06-79.
- [2] M. Schulz, L. Schulz, M. Ruefenacht, and R. Wille. “Towards the Munich Quantum Software Stack: Enabling Efficient Access and Tool Support for Quantum Computers.” In: *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*. Vol. 02. 2023, pp. 399–400. DOI: 10.1109/QCE57702.2023.10301.
- [3] P. Hopf, S. Stern, R. Wille, and L. Burgholzer. *Standardizing Access to Heterogeneous Quantum Backends: A Case Study on Cloud Service Integration with QDMI*. 2026. arXiv: 2603.05138 [quant-ph].
- [4] 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. “The Munich Quantum Software Stack: Connecting End Users, Integrating Diverse Quantum Technologies, Accelerating HPC.” In: *Proceedings of the Supercomputing Asia and International Conference on High Performance Computing in Asia Pacific Region*. SCA/HPCAsia 2026. ACM, Jan. 2026, pp. 55–67. DOI: 10.1145/3773656.3773669.
- [5] S. Bock, R. Seidel, F. Zilk, E. Lobe, S. Rubbert, A. Chernoguzov, P. A. Limacher, L. Burgholzer, and D. Georg. *Interface between quantum computer backends and software frameworks*. DIN SPEC. Sept. 2025.
- [6] E. Bertok, H. Busche, F. Drinkler, D. Emmanuel-Costa, D. Herr, S. Hien, T. Keitzl, E. Lobe, A. Paler, J. Renkl, M. Rymarz, G. Schmiedinghoff, P. K. Schuhmacher, T. Stehle, B. Strobel, H. Tschakert, A. Vetter, A. Waschk, A. Weinert, and L. Windgätter. *QCI Connect: A Modular Full-Stack Quantum Computing Platform*. 2026. arXiv: 2606.14456 [quant-ph].
- [7] D. Z. für Luft- und Raumfahrt e. V. (DLR), d-fine (Germany), and planqc GmbH. *QCI Connect API Specification*. Version 0.4.0. June 2026. DOI: 10.5281/zenodo.20553739.

- [8] J. Echavarria, M. N. Farooqi, A. Devra, S. Lujan, L. Van Damme, H. Ahmed, M. Letras, E. Kaya, A. Vetter, M. Werninghaus, M. Knudsen, F. Rohde, A. Frisch, E. Mansfield, R. Davletkaliyev, V. Kukushkin, N. Färkkilä, J. Mäntylä, N. Pomplun, A. Spörl, L. Burgholzer, Y. Stade, R. Wille, L. B. Schulz, and M. Schulz. “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*. New York, NY, USA: Association for Computing Machinery, 2025, pp. 1868–1878. ISBN: 9798400718717. DOI: 10.1145/3731599.3767552.
- [9] Munich Quantum Valley. *Munich Quantum Software Stack (MQSS)*. <https://www.munich-quantum-valley.de/research/research-areas/mqss>. Accessed: 2026-05-12.
- [10] R. Wille, L. Schmid, Y. Stade, J. Echavarria, M. Schulz, L. Schulz, and L. Burgholzer. “QDMI – Quantum Device Management Interface: Hardware-Software Interface for the Munich Quantum Software Stack.” In: *IEEE International Conference on Quantum Computing and Engineering (QCE)*. 2024. DOI: 10.1109/QCE60285.2024.10411.
- [11] R. Wille, L. Schmid, Y. Stade, J. Echavarria, M. Schulz, L. Schulz, and L. Burgholzer. *QDMI – Quantum Device Management Interface: Hardware-Software Interface for the Munich Quantum Software Stack*. Accessed: 2026-05-12.
- [12] DLR Quantum Computing Initiative. *DLR Quantum Computing Initiative – We enable quantum*. URL: <https://qci.dlr.de/en/> (visited on 05/25/2026).
- [13] DLR Quantum Computing Initiative. *QCI Connect*. URL: <https://qci.dlr.de/en/connect/> (visited on 05/25/2026).
- [14] A. Basermann, M. Epping, B. Fauseweh, M. Felderer, E. Lobe, M. Röhrig-Zöllner, G. Schmiedinghoff, P. K. Schuhmacher, Y. Setyawati, and A. Weinert. “Quantum Software Ecosystem Design.” In: *Quantum Software*. Ed. by I. Exman et al. Springer, 2024. DOI: 10.1007/978-3-031-64136-7_7.
- [15] DLR Quantum Computing Initiative. *CLIQUE – Classical Integration of Quantum Computers*. URL: <https://qci.dlr.de/en/cliique/> (visited on 05/25/2026).
- [16] DLR Quantum Computing Initiative. *CLIQUE – Classical Integration of Quantum Computers*. 2025. URL: <https://qci.dlr.de/wp-content/uploads/2025/01/Clique.pdf> (visited on 05/25/2026).
- [17] D. da Costa, T. Keitzl, J. Renkl, and T. Stehle. *QCI Connect SDK: API Client*. 2025. URL: <https://elib.dlr.de/220283/> (visited on 05/25/2026).

- [18] DLR Quantum Computing Initiative. *QCI Connect SDK 0.9.0 released*. May 11, 2026. URL: <https://qci.dlr.de/en/qci-connect-sdk-0-9-0-released/> (visited on 05/25/2026).
- [19] DLR-SC. *qciconnect 0.9.0*. URL: <https://pypi.org/project/qciconnect/> (visited on 05/25/2026).
- [20] QDMI Maintainers. *QDMI Rationale*. URL: https://munich-quantum-software-stack.github.io/QDMI/v1.3.1/md_docs_2rationale.html (visited on 06/30/2026).
- [21] Munich Quantum Software Stack (MQSS) Team. *MQSS Client*. URL: <https://pypi.org/project/mqss-client/> (visited on 06/30/2026).
- [22] Munich Quantum Software Stack (MQSS) Team. *Munich Quantum Portal (MQP)*. URL: <https://portal.quantum.lrz.de/> (visited on 06/30/2026).