QCI Connect SDK

A Flexible Toolbox for Bringing Applications to Quantum Computers

David da Costa, Thomas Keitzl, Elisabeth Lobe, Johannes Renkl, Gary Schmiedinghoff, Thomas Stehle, Lukas Windgätter

Overview

The **QCI Connect SDK** is an open-source Python package that provides programmatic access to the *QCI Connect* platform for both experts and non-experts in quantum computing (QC). It is developed within the DLR Quantum Computing Initiative (QCI) of the German Aerospace Center (DLR).

Rather than duplicating existing quantum libraries, the software development kit (SDK) eases integration of existing solutions into the *QCI Connect* ecosystem, following a hardware-software co-design principle [1]. The source code is available under the Apache license to support community-driven development.

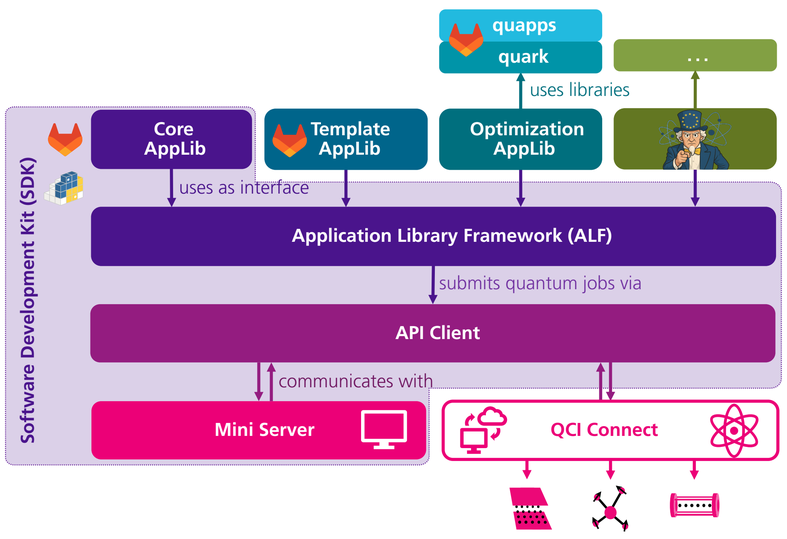


Fig. 1: Illustration of the SDK architecture for developing quantum applications.

The QCI Connect SDK Architecture

The SDK architecture for developing quantum applications is illustrated in Fig. 1. The application programming interface (API) Client can be used to communicate with both *QCI Connect* and Mini Server in the same manner. The application library framework (ALF) uses this interface to run quantum jobs and provides the interface for applications, while the core application library (AppLib) provides reference implementations.

Partners can build extensions to the AppLib (for example starting from the Template AppLib) in their own repository using ALF interfaces and their own existing libraries. An example of this is the Optimization AppLib that brings optimization application from the packages `quark` and `quapps` [3] to *QCI Connect*.

API Client

The API Client manages the communication with the *QCI Connect* platform, such as sending quantum jobs. It is a simple Python library that provides basic **programmatic access** to the *QCI Connect* platform, which in turn provides access to DLR's **QC resources**, i.e. hardware, simulators, and compilers, as well as high-performance computing (HPC).

Mini Server

The Mini Server provides an execution environment independent of the cloud that allows to **locally execute** compilers and simulators of the *QCI Connect* platform. It can interface as a server with the API Client, substituting the *QCI Connect* platform. It allows to locally develop algorithms with the goal to integrate them to the platform, **prototype quickly**, and run algorithms on the local computer using the **same interfaces** of the *QCI Connect* API Client.

Application Library Framework (ALF)

The ALF is a hardware-agnostic Python framework for **developing applications** and integrating compilers and simulators for the *QCI Connect* platform. It defines **interfaces** for applications, which submit quantum jobs via the API Client. It contains classes to streamline implementing quantum algorithms, error mitigation schemes, and circuit building strategies, and to unify problems and results for **cross-compatibility** of applications.

Application Library (AppLib)

The AppLib is a Python package containing the core **application library** of the *QCI Connect* platform. It provides reference implementations of standard applications and algorithms **based on ALF** and is **expandable** with other packages based on ALF.

Installation

The easiest way to install the SDK is via

```
> pip install qciconnect
```

The SDK components can also be installed separately via, e.g.,

```
> pip install qciconnect-client
```

Example: Submit Circuit over API Client

```
[1]: from qciconnect.client import Client

[2]: client = Client(token = "my_qci_connect_token")

[3]: client.qpus.show()
```

```
Alias              Qubits  Status       Technology         Accessible to me
-----------------  ------  -----------  -----------------  ----------------
statevector_simulator  24  running      statevector_simulator          True
xq1i                    4  idle         vacancy_centres                True
toccata                50  coming_soon  trapped_ions                   True
[...]
```

```
[4]: qasm_circuit = """OPENQASM 3.0;
                       include 'stdgates.inc';
                       qubit[2] q;
                       h q[0];
                       cx q[0], q[1];"""
     result = client.qpus.statevector_simulator.submit(
         circuit = qasm_circuit,
         primitive = "quantum_state",
         name = "Example Job")
     result.data
```

```
Job submitted with ID: a94b6db6-6e84-48ce-968f-d178a2018989
```

```
[4]: {'[0, 0]': (0.7071067811865476+0j), '[1, 1]': (0.7071067811865475+0j)}

[5]: result.aggregated_data

[5]: {'executed_shots': 1}
```

Vision: Provision Applications via QCI Connect Platform

The *QCI Connect* platform, its architecture, the backend and the web frontend, are developed within the QCI of the DLR together with d-fine and PlanQC. The platform follows the three-layer architecture – physical, system, and application – suggested in [2], where a central orchestrator turns jobs into task workflows and routes them to compilers, simulators, quantum processing units (QPUs), or HPC systems through standardized RESTful APIs.

We refer to the paper on the right for more details on the design rationale and key lessons learned, such as the importance of modularity, open interfaces, local development environments, and intellectual property (IP) protection for industrial adoption through granular access controls. It further addresses the fragmentation of quantum software stacks, vendor lock-in, and incompatible APIs across hardware ecosystems.

With such a diverse ecosystem as the QCI and their different partners within and outside of DLR, a variety of hardware and use cases is brought together. The standard interfaces provided by the SDK allow to not only program internally, but also host applications on the platform for showcasing while keeping IPs protected. In Fig. 2, we show how applications shall be made available.

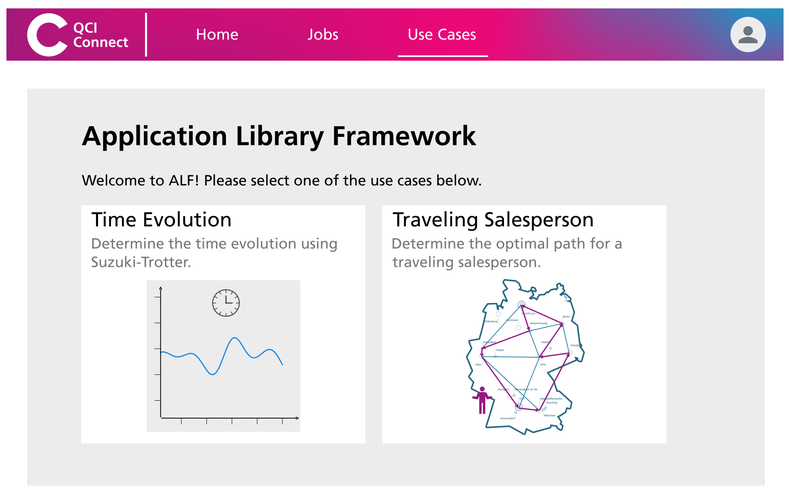


Fig. 2: Mockup of envisioned application selection in *QCI Connect* web frontend.

References

[1] Achim Basermann et al. "Quantum Software Ecosystem Design". In: **Quantum Software: Aspects of Theory and System Design**. Ed. by Iaakov Exman et al. Springer, Cham, 2024. DOI: 10.1007/978-3-031-64136-7_7.

[2] Philipp Kunst et al. **Schlussbericht Innovationsprojekt QC Next**. BMWK, 2025. URL: https://www.digitale-technologien.de/DT/Redaktion/DE/Downloads/Publikation/Quantencomputing/250820_qc_next_software_praxis.html (visited on 03/18/2026).

[3] Lukas Windgätter et al. "Quantum Optimization Applications With Quark and Quapps: Bridging the Gap Between Application and Hardware". In: **IEEE Software** 42.5 (2025). DOI: 10.1109/MS.2025.3564146.

Central Building Block: Algorithm

`Algorithms` are the central objects of the ALF. They consume a `ProblemInput` and return a corresponding `Result` as illustrated in Fig. 3. The problem input is logically separated from the `AlgorithmParams`, just providing options for e.g. the construction of circuits or their execution. Analogously, the `AlgorithmStats` return the gathered information from the execution.

Internally, algorithms describe quantum programs using `Circuits`. Circuit builder objects aid in the construction of circuits. Algorithms can compute bit strings on circuits by submitting them to a `Sampler`, which returns a `Measurement` object containing a histogram of the measured bit strings. Algorithms can also compute `ExpectationValues` of `Observables` for a given circuit in terms of their `MeasurementBasis`. They can create an `EstimationJob` that wraps the necessary information and send it to an `Estimator`. Finally, the estimator internally calls its `Sampler` and returns an `Estimations` object containing the estimated expectation values.

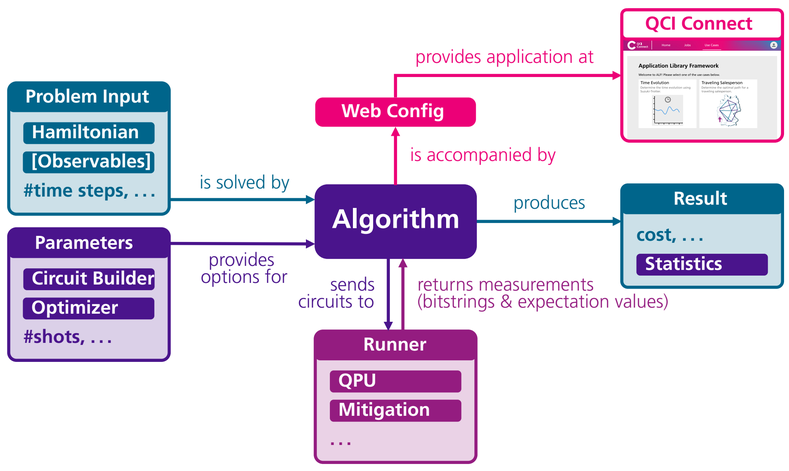


Fig. 3: Illustration of the workflow around the central algorithm.

Example: Execute Time Evolution Algorithm

```
[6]: from qciconnect.applib.algorithm import TimeEvolution, \
                                             TimeEvolutionParams

[7]: from qciconnect.alf.base import MeasurementBasis, PauliSuperposition
     from qciconnect.alf.execution import Observable, Runner
     from qciconnect.alf.problem.problem_input import Hamiltonian

[8]: observables = [Observable.from_string("Z0 + Z1"),
                    Observable.from_string("X0 + X1")]
     bases = [MeasurementBasis("ZZ"), MeasurementBasis("XX")]
     params = TimeEvolutionParams(time_grid = [0.1 * i for i in range(32)],
                                  observables = observables,
                                  bases = bases,
                                  n_shots = 1000)

[9]: runner = Runner(qpu = client.qpus.statevector_simulator)

[10]: time_evolution = TimeEvolution(params = params, runner = runner)

[11]: hamiltonian = Hamiltonian(PauliSuperposition.from_string("X0 + X1"))

[12]: result = time_evolution(hamiltonian)
```

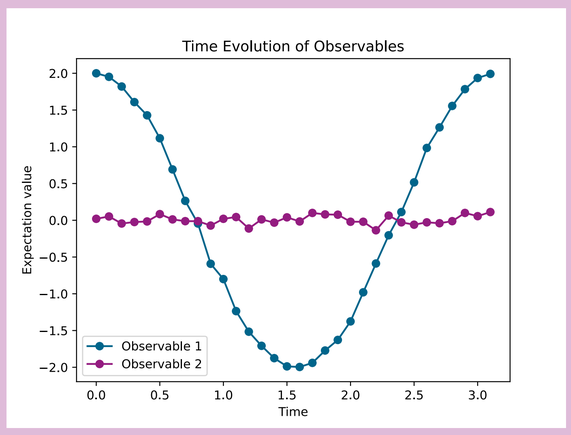
```
Job submitted with ID: f469f57e-7933-4c52-ad22-31b480ab22c9
[...]
```

```
[13]: result.algorithm_statistics.runtime_total_secs

[13]: 203.67975909996312

[14]: from qciconnect.applib.app.time_evolution import write_output

[15]: _ = write_output.prepare_plots(params, None, result = result)
```



Links

Release
QCI Connect SDK - version 0.9
DOI:10.5281/zenodo.20555800

Web Documentation
QCI Connect SDK - Landing Page
dlr-sc-qc.gitlab.io/qciconnect-sdk/landing-page/

Paper
QCI Connect: A Modular Full-Stack Quantum Computing Platform, arXiv:2606.14456 [quant-ph]

DLR Institute of Software Technology
dlr.de/en/sc
qc-software@dlr.de

DLR QCI
DLR
Quantum Computing Initiative
qci.dlr.de

supported by
Bundesministerium für Forschung, Technologie und Raumfahrt