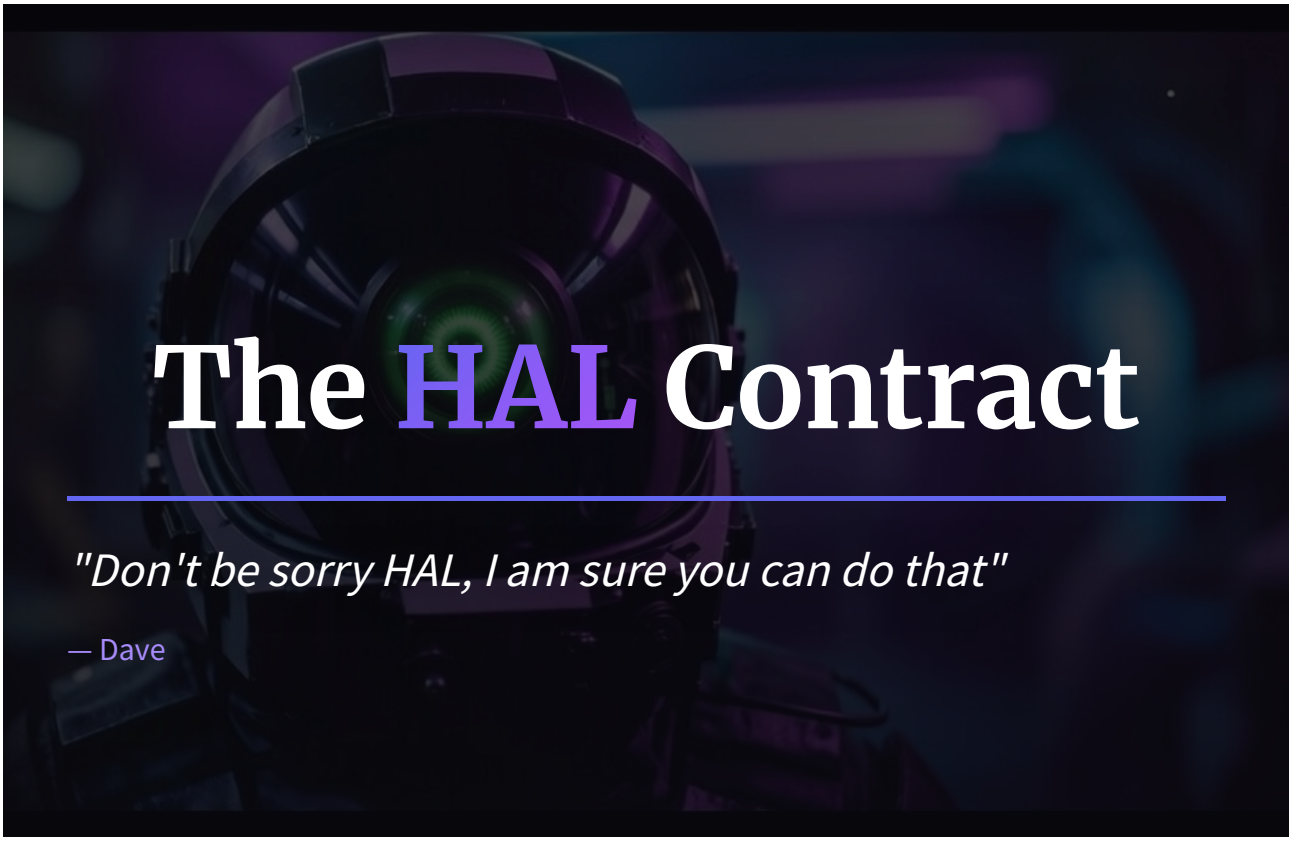

HAL Contract and HIQ Implementation

Version 1.0 — HIQ v1.0.0

Daniel Hinderink

February 2026

Contents

Abstract

The quantum computing landscape is fragmented across multiple hardware vendors, each with proprietary SDKs and APIs. This fragmentation creates vendor lock-in, increases development costs, and slows the adoption of quantum computing in production environments. We present the HAL Contract, a vendor-neutral interface specification that enables quantum algorithms to run on any compliant backend without modification. The contract defines six core interfaces: Backend, Capabilities, Job, Result, Topology, and GateSet. We demonstrate the practical applicability of this approach through HIQ v1.0, a Rust-native reference implementation that supports IQM, IBM Quantum, QDMI (Munich Quantum Software Stack), and local simulation backends. HIQ also provides Qrisp-inspired quantum types, automatic uncomputation, and first-class HPC integration with both SLURM and PBS schedulers. The HAL Contract enables true hardware portability while maintaining the flexibility needed for backend-specific optimizations.

1. Introduction

Quantum computing has transitioned from theoretical curiosity to practical reality. Organizations worldwide are deploying quantum processors for research, optimization, and machine learning workloads. However, the current ecosystem suffers from a fundamental problem: **each quantum hardware vendor provides its own SDK, API, and execution model**.

A researcher developing a variational quantum eigensolver (VQE) algorithm faces a difficult choice. Should they target IBM's Qiskit Runtime? IQM's native API? Google's Cirq? Each choice locks them into a specific ecosystem, making it expensive to switch providers or leverage multiple backends.

This whitepaper introduces the **HAL Contract**—a Hardware Abstraction Layer specification designed to solve this fragmentation. Inspired by successful abstraction patterns in classical computing (JDBC for databases, POSIX for operating systems), the HAL Contract provides a stable interface that decouples quantum algorithms from their execution environment.

1.1 Goals

- **Portability**: Write quantum code once, run it on any compliant backend
- **Testability**: Develop against simulators, deploy to real hardware

- **HPC Integration**: First-class support for job schedulers (SLURM, PBS)
- **Extensibility**: Allow backend-specific features without breaking compatibility

1.2 Non-Goals

The HAL Contract intentionally does not address:

- Circuit representation or gate definitions (use OpenQASM 3.0)
- High-level algorithm frameworks (use Qiskit, Cirq, etc.)
- Compilation and optimization passes (implementation-specific)

2. The Fragmentation Problem

Today's quantum computing ecosystem resembles the database landscape of the 1980s—before SQL and JDBC established common interfaces. Each vendor's solution is an island, requiring specialized knowledge and tooling.

2.1 Current State

Vendor	SDK	API Style	Job Model
IBM Quantum	Qiskit	REST + Primitives	Qiskit Runtime
IQM	IQM Client	REST	Resonance API
Google	Cirq	gRPC	Quantum Engine
Rigetti	pyQuil	REST	QCS
Amazon	Braket SDK	AWS API	Braket Tasks

2.2 Consequences

Vendor Lock-in. Once an organization commits to a specific SDK, switching costs are substantial. Code must be rewritten, workflows redesigned, and teams retrained.

Duplicated Effort. Research groups implement the same algorithms multiple times for different backends, wasting resources that could advance quantum computing.

Limited Benchmarking. Comparing quantum hardware requires running identical workloads, which is impossible when each backend speaks a different language.

HPC Integration Challenges. High-performance computing centers with SLURM or PBS schedulers must build custom integrations for each quantum backend they support.

2.3 The Cost of Fragmentation

Consider a pharmaceutical company running quantum chemistry simulations. They start with IBM Quantum for initial development, but IQM offers better gate fidelities for their specific workload. Without a hardware abstraction layer, migration requires:

- Rewriting all circuit submission code
- Adapting job status polling and result retrieval
- Updating error handling for different failure modes
- Modifying CI/CD pipelines and monitoring

With the HAL Contract, migration is a configuration change.

3. The HAL Contract

The HAL Contract is a minimal, stable interface specification that quantum backends implement to achieve interoperability. It follows three design principles:

3.1 Design Principles

Principle 1: Minimal Surface Area. The contract defines only what is necessary for portability. Backend-specific features are accessible through extension points, not modifications to core interfaces.

Principle 2: Async-First. Quantum job execution is inherently asynchronous. Jobs are submitted, polled, and retrieved—the contract embraces this model rather than hiding it.

Principle 3: Capability Discovery. Backends advertise their capabilities (qubit count, gate set, topology) through a standard interface. Applications can query and adapt at runtime.

3.2 Architecture Overview

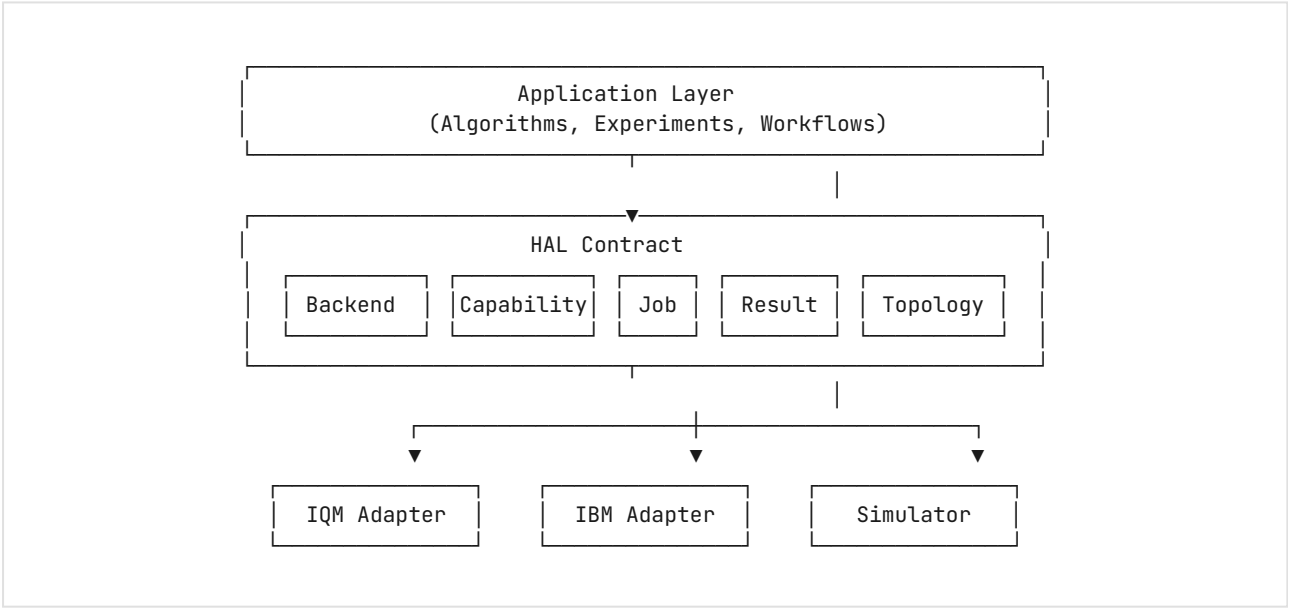

Figure 1: HAL Contract Architecture

3.3 Contract Versioning

The HAL Contract follows semantic versioning. Minor versions add optional capabilities; major versions may introduce breaking changes. Backends declare which contract version they implement.

4. Core Interfaces

The HAL Contract defines six core interfaces. Together, they provide everything needed to submit quantum circuits, monitor execution, and retrieve results.

4.1 Backend

The `Backend` trait is the primary interface for quantum execution. It defines methods for job submission, status checking, result retrieval, and cancellation.

```
trait Backend: Send + Sync {
    /// Get the name of this backend
    fn name(&self) → &str;

    /// Get the capabilities of this backend
    async fn capabilities(&self) → HalResult<Capabilities>;

    /// Check if the backend is available
    async fn is_available(&self) → HalResult<bool>;

    /// Submit a circuit for execution
    async fn submit(&self, circuit: &Circuit, shots: u32) → HalResult<JobId>;

    /// Get the status of a job
    async fn status(&self, job_id: &JobId) → HalResult<JobStatus>;

    /// Get the result of a completed job
    async fn result(&self, job_id: &JobId) → HalResult<ExecutionResult>;

    /// Cancel a running job
    async fn cancel(&self, job_id: &JobId) → HalResult<()>;

    /// Wait for a job to complete (default implementation provided)
    async fn wait(&self, job_id: &JobId) → HalResult<ExecutionResult>;
}
```

4.2 Capabilities

The `Capabilities` structure describes what a backend can do. Applications use this information to select appropriate backends and validate circuits before submission.

```
struct Capabilities {
    name: String,           // Backend identifier
    num_qubits: u32,        // Available qubits
    gate_set: GateSet,      // Supported gates
    topology: Topology,     // Qubit connectivity
    max_shots: u32,         // Maximum shots per job
    is_simulator: bool,     // Hardware or simulation
    features: Vec<String>,  // Optional features
}
```

4.3 Job and JobStatus

Jobs progress through a defined lifecycle: `Queued → Running → Completed/Failed/Cancelled`. The `JobStatus` enum captures these states.

```
enum JobStatus {
    Queued,              // Waiting in queue
    Running,             // Currently executing
    Completed,           // Finished successfully
    Failed(String),      // Failed with error message
    Cancelled,           // Cancelled by user
}
```

4.4 ExecutionResult

Results contain measurement counts, execution metadata, and optional backend-specific information.

```
struct ExecutionResult {
    counts: Counts,                    // Measurement outcomes
    shots: u32,                        // Shots executed
    execution_time_ms: Option<u64>,    // Execution duration
    metadata: Value,                   // Backend-specific data
}
```

4.5 Topology

Qubit topology defines which pairs of qubits can interact directly. This is essential for circuit routing and optimization.

```
struct Topology {
    kind: TopologyKind,        // Linear, Star, Grid, Custom
    edges: Vec<(u32, u32)>,    // Connected qubit pairs
}

enum TopologyKind {
    FullyConnected,    // All-to-all connectivity
    Linear,            // Chain: 0-1-2-3-...
    Star,              // Center connected to all
    Grid { rows, cols }, // 2D lattice
    Custom,            // Arbitrary connectivity
}
```

4.6 GateSet

The gate set defines native operations supported by the hardware. Circuits using non-native gates must be transpiled before execution.

```
struct GateSet {
    single_qubit: Vec<String>,  // e.g., ["rx", "ry", "rz"]
    two_qubit: Vec<String>,     // e.g., ["cz", "cx"]
    native: Vec<String>,        // Preferred gates for this backend
}
```

Backend	Native Single-Qubit	Native Two-Qubit
IQM	PRX (phased rotation)	CZ
IBM	RZ, SX, X	CX (CNOT)
Simulator	Universal	Universal

5. Implementation Guide

Implementing the HAL Contract for a new backend requires implementing the `Backend` trait and mapping vendor-specific APIs to the standard interface.

5.1 Minimal Implementation

A minimal implementation must provide all required methods. Here's a skeleton for a hypothetical "Acme Quantum" backend:

```
struct AcmeBackend {
    client: AcmeClient,
    config: BackendConfig,
}

#[async_trait]
impl Backend for AcmeBackend {
    fn name(&self) → &str {
        "acme-quantum"
    }

    async fn capabilities(&self) → HalResult<Capabilities> {
        let info = self.client.get_device_info().await?;
        Ok(Capabilities {
            name: info.name,
            num_qubits: info.qubit_count,
            gate_set: GateSet::from_acme(&info.gates),
            topology: Topology::from_acme(&info.connectivity),
            max_shots: info.max_shots,
            is_simulator: false,
            features: vec![],
        })
    }

    async fn submit(&self, circuit: &Circuit, shots: u32) → HalResult<JobId> {
        let acme_circuit = convert_to_acme_format(circuit)?;
        let job = self.client.submit(acme_circuit, shots).await?;
        Ok(JobId::new(job.id))
    }

    // ... remaining methods
}
```

5.2 Error Handling

The HAL Contract defines standard error types that backends must map their errors to:

```
enum HalError {
    NotAvailable(String),      // Backend offline
    InvalidCircuit(String),    // Circuit validation failed
    JobFailed(String),         // Execution error
    JobCancelled,              // Job was cancelled
    Timeout(String),           // Operation timed out
    Authentication(String),    // Auth failure
    RateLimited,               // Too many requests
    Internal(String),          // Unexpected error
}
```

5.3 Testing Compliance

The HAL Contract provides a compliance test suite that validates implementations:

```
// Run compliance tests against your backend
#[test]
async fn test_hal_compliance() {
    let backend = AcmeBackend::new(test_config());
    hal_compliance::run_all_tests(&backend).await;
}
```

The test suite verifies:

- Capability reporting is consistent
- Job lifecycle follows the specified state machine
- Error types are properly mapped
- Results are correctly formatted

6. Case Study: HIQ

HIQ is the reference implementation of the HAL Contract. Built in Rust for performance, it provides adapters for multiple quantum backends and integrates with HPC job schedulers.

6.1 Architecture

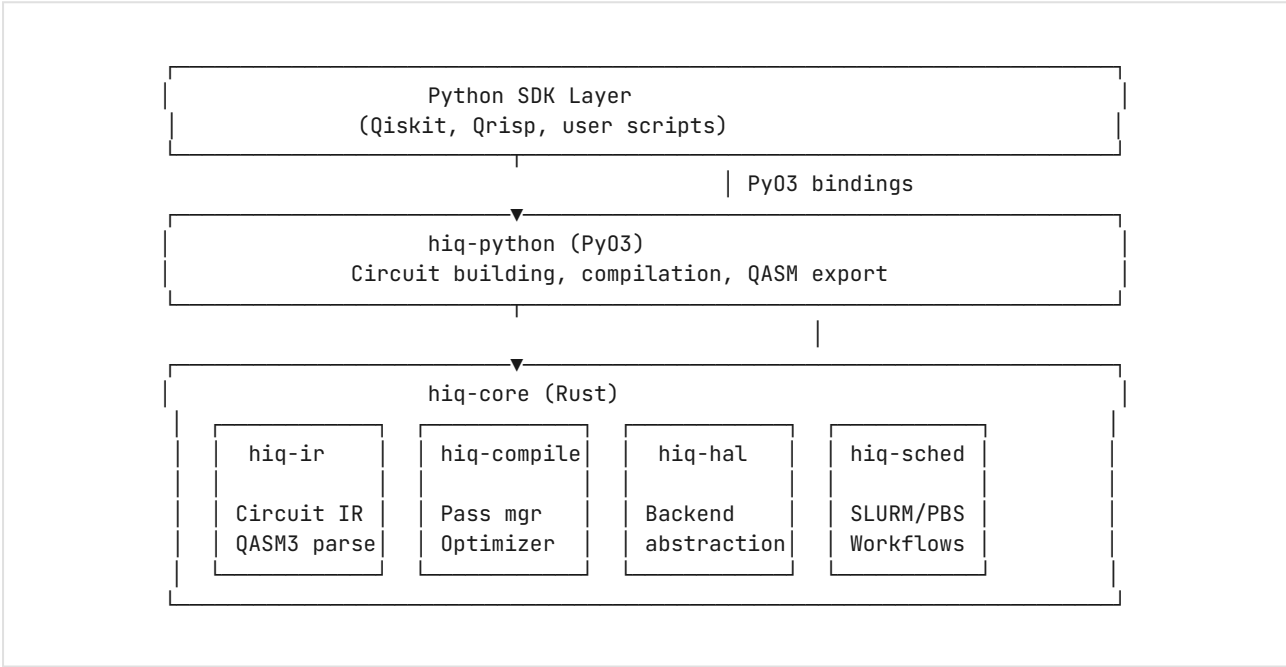

Figure 2: HIQ Architecture

6.2 Supported Backends

Backend	Type	Authentication	Notes
Simulator	Local	None	Statevector, up to ~20 qubits
IQM Resonance	Cloud	API Token	Full Resonance API support
IQM LUMI	HPC	OIDC	CSC Finland integration
IQM LRZ	HPC	OIDC	LRZ Germany integration

Backend	Type	Authentication	Notes
IBM Quantum	Cloud	API Token	Qiskit Runtime primitives
QDMI (MQSS)	HPC/Cloud	Token/OIDC	Munich Quantum Software Stack

6.3 Additional Features

HIQ v1.0 extends beyond the basic HAL Contract with several productivity features:

- **Quantum Types (hiq-types)**: Qrisp-inspired high-level types including QuantumInt, QuantumFloat, and QuantumArray for easier algorithm development
- **Automatic Uncomputation (hiq-auto)**: Framework for automatically uncomputing ancilla qubits through gate inversion
- **HPC Schedulers (hiq-sched)**: Native support for both SLURM and PBS/Torque job schedulers with workflow orchestration
- **LUMI Hybrid Demo**: Complete VQE workflow example for quantum-HPC hybrid computing on LUMI

6.4 Usage Example

```
use hiq_hal::Backend;
use hiq_ir::Circuit;

// Same code works with any backend
async fn run_bell_state(backend: &impl Backend) → Result<()> {
    let circuit = Circuit::bell();

    let caps = backend.capabilities().await?;
    println!("Running on {} ({} qubits)", caps.name, caps.num_qubits);

    let job_id = backend.submit(&circuit, 1000).await?;
    let result = backend.wait(&job_id).await?;

    println!("Results: {:?}", result.counts);
    Ok(())
}

// Switch backends with one line
let sim = SimulatorBackend::new();
run_bell_state(&sim).await?;
```

```
let iqm = IqmBackend::from_env()?;
run_bell_state(&iqm).await?;
```

7. Benefits & Trade-offs

7.1 Benefits

Hardware Independence. Algorithms developed against the HAL Contract run on any compliant backend. This protects investments in quantum software development.

Simplified Testing. Develop and test against local simulators, then deploy to real hardware without code changes. CI/CD pipelines become straightforward.

Multi-Backend Execution. Run the same workload on multiple backends to compare results, benchmark performance, or implement redundancy.

HPC Integration. First-class support for SLURM and PBS means quantum workloads integrate naturally into existing HPC workflows.

Future-Proofing. As new quantum hardware becomes available, adding support requires only implementing the adapter—existing applications work immediately.

7.2 Trade-offs

Lowest Common Denominator. The contract exposes only capabilities shared across backends. Advanced vendor-specific features require extension points or direct API access.

Abstraction Overhead. A thin abstraction layer adds minimal overhead, but high-frequency operations may notice the additional indirection.

Version Coordination. As the contract evolves, backends must update their implementations. Clear versioning mitigates but doesn't eliminate this coordination cost.

7.3 When to Use HAL

Use the HAL Contract when:

- You need to support multiple quantum backends
- You want to test locally and deploy to hardware

- You're building quantum software for HPC environments
- You want to avoid vendor lock-in

Consider direct API access when:

- You need vendor-specific advanced features
- You're optimizing for a single backend
- You're building low-level tooling

8. Conclusion

The quantum computing industry stands at a crossroads. As hardware matures and production deployments increase, the cost of fragmentation will only grow. The HAL Contract offers a path forward—a minimal, stable interface that enables portability without sacrificing flexibility.

We have presented:

- The fragmentation problem facing quantum computing today
- The HAL Contract specification with six core interfaces
- Implementation guidelines for backend developers
- HIQ as a reference implementation demonstrating practical applicability

The HAL Contract is an open specification. We invite quantum hardware vendors, SDK developers, and the research community to adopt, implement, and contribute to its evolution.

8.1 Getting Started

- **Website:** hal-contract.org
- **Reference Implementation:** github.com/hiq-lab/HIQ
- **Specification:** hiq-hal crate

8.2 Call to Action

If you're a quantum hardware vendor, consider implementing the HAL Contract for your platform. If you're a researcher or developer, consider building against the contract to future-proof your work. Together, we can create an interoperable quantum computing ecosystem.

References

1. Qiskit: An Open-source Framework for Quantum Computing. qiskit.org
2. Cirq: A Python framework for creating, editing, and invoking Noisy Intermediate Scale Quantum (NISQ) circuits. quantumai.google/cirq
3. OpenQASM 3.0 Specification. openqasm.com
4. IQM Resonance API Documentation. meetiqm.com
5. XACC: eXtreme-scale ACCelerator programming framework. github.com/eclipse/xacc
6. QDMI: Quantum Device Management Interface (Munich Quantum Software Stack). github.com/Munich-Quantum-Software-Stack/QDMI
7. Qrisp: High-level quantum programming language. qrisp.eu
8. HIQ: Rust-Native Quantum Compilation Stack. github.com/hiq-lab/HIQ

---

hal-contract.org