Interface between Quantum and Classical Computers

A study in Cloud-Based Quantum Computing

Ioannis Savvaidis

School of Engineering
Department of Electrical and Computer Engineering
Sector of Electronics and Information Technology Systems
Master in Quantum Computing and Quantum Technologies

Athens, July 2026

This page intentionally left blank.

Interface between Quantum and Classical Computers

A study in Cloud-Based Quantum Computing

Ioannis Savvaidis
Student No. 60663

Supervisor: Ioannis Karafyllidis
Professor, Democritus University of Thrace

Co-supervisor: Georgios Sirakoulis
Professor, Democritus University of Thrace

Panagiotis Dimitrakis
Director of Research, Institute of Quantum Computing & Quantum Technology, NCSR "Demokritos"

School of Engineering
Department of Electrical and Computer Engineering
Sector of Electronics and Information Technology Systems
Master in Quantum Computing and Quantum Technologies

Thesis

Athens, July 2026

This page intentionally left blank.

Report on Compliance with Academic Ethics

This work, entitled "Interface between Quantum and Classical Computers", is original and was conducted by Ioannis Savvaidis, Department of Electrical and Computer Engineering, AEM Registration No. 60663 in the Electronics and Information Systems Technology sector, under the supervision of Ioannis Karafyllidis. The thesis was written solely by the student, under the guidance of the supervisor.

It is confirmed that the student complied with all legal provisions and Departmental regulations, adhered to the Principles of Academic Ethics and the Code of Conduct, and refrained from falsification of results, reporting of false data, misuse of intellectual property, and plagiarism.

This page intentionally left blank.

Interface between Quantum and Classical Computers

Preparation of this work was facilitated by the use of the *IPLeiria-Thesis* template.

This page intentionally left blank.

Περίληψη

Οι κβαντικοί υπολογιστές χρησιμοποιούνται ολοένα και περισσότερο όχι ως αυτόνομες μηχανές, αλλά ως επιταχυντές μέσα σε υβριδικές υπολογιστικές ροές εργασίας. Σε αυτές, ένα κλασικό υπολογιστικό σύστημα προετοιμάζει ένα κβαντικό κύκλωμα, το υποβάλλει σε έναν κβαντικό επεξεργαστή και στη συνέχεια ερμηνεύει τα πιθανοκρατικά αποτελέσματα που αυτός επιστρέφει. Η παρούσα διατριβή αντιμετωπίζει τη διεπαφή μεταξύ κλασικού και κβαντικού υπολογισμού ως πρόβλημα αρχιτεκτονικής και διαχείρισης συστημάτων και όχι ως ζήτημα φυσικής. Πρακτικές όπως η ενορχήστρωση, ο προγραμματισμός εκτέλεσης και η παρατηρησιμότητα, οι οποίες θεωρούνται δεδομένες στα σύγχρονα κλασικά υπολογιστικά νέφη (cloud-native), απουσιάζουν σε μεγάλο βαθμό από το σημείο όπου εκτελείται ο κβαντικός υπολογισμός.

Με αφετηρία μια πολυεπίπεδη ανάλυση αυτής της διεπαφής και μια επισκόπηση των υπαρχουσών προσεγγίσεων στην ενορχήστρωση και την παρατηρησιμότητα, η διατριβή διατυπώνει ένα σύνολο απαιτήσεων και προτείνει ως απόδειξη της ιδέας (proof of concept) τον Quantum Circuit Controller (QCC), ένα επίπεδο ελέγχου (control plane) για τη διεπαφή μεταξύ κλασικού και κβαντικού υπολογισμού. Ο QCC εισάγει στο σημείο εκτέλεσης του κβαντικού υπολογισμού τεχνικές ενορχήστρωσης εμπνευσμένες από τα cloud-native συστήματα, καθώς και ουδέτερη ως προς τον προμηθευτή (vendor-neutral), βασισμένη σε ανοικτά πρότυπα, παρατηρησιμότητα. Έτσι, ο κβαντικός υπολογιστής, ως υπόβαθρο (quantum backend), αντιμετωπίζεται όχι ως ένας αδιαφανής απομακρυσμένος προορισμός εκτέλεσης, αλλά ως ένας διαχειρίσιμος και πλήρως παρατηρήσιμος υπολογιστικός πόρος.

Η προτεινόμενη αρχιτεκτονική δοκιμάστηκε και αξιολογήθηκε καθ'ολοκληρίαν με έναν αντιπροσωπευτικό κβαντικό αλγόριθμο, τόσο σε προσομοιωμένο όσο και σε πραγματικό κβαντικό υπολογιστή. Τα αποτελέσματα δείχνουν ότι ικανοποιεί τις απαιτήσεις που είχαν τεθεί αρχικά, και καθιστά ορατό τόσο το υπολογιστικό κόστος όσο και τη συμπεριφορά της διεπαφής μεταξύ των δύο κόσμων. Η διατριβή προτείνει επίσης δύο κατευθύνσεις τυποποίησης: μια ουδέτερη διεπαφή εκτέλεσης, ανεξάρτητη από προμηθευτές κβαντικών υπολογιστών, και ένα σύνολο προτύπων για την παρατηρησιμότητα των κβαντικών συστημάτων. Συνολικά, μεταφέρει τις αρχές της Μηχανικής Αξιοπιστίας Συστημάτων (Site Reliability Engineering – SRE) στον χώρο της συνεργασίας μεταξύ κλασικού και κβαντικού υπολογισμού.

Λέξεις-κλειδιά: διεπαφή κβαντικών και κλασικών υπολογιστών, υβριδικά κβαντικά–κλασικά υπολογιστικά φορτία, Kubernetes, OpenTelemetry, παρατηρησιμότητα, cloud-native ενορχήστρωση, Qiskit

Abstract

Quantum computers are increasingly used not as standalone machines but as accelerators within hybrid workflows, in which a classical system prepares a circuit, submits it to a quantum processor, and interprets probabilistic results. This thesis treats the quantum–classical interface as a systems problem rather than a physics one. The orchestration, scheduling, and observability taken for granted in classical cloud-native operations are largely absent at the quantum execution boundary.

From a layered analysis of the interface and a survey of existing orchestration and observability work, the thesis derives a set of requirements and answers them with a proof-of-concept, the Quantum Circuit Controller (QCC), a control plane for the quantum–classical interface. It brings cloud-native orchestration and vendor-neutral, standards-based observability to the quantum execution boundary, treating a quantum backend as a managed, observable resource rather than an opaque remote endpoint. Exercised end-to-end on a representative quantum workload across both simulated and real hardware, it satisfies these requirements and makes the computational cost and behaviour of the interface visible. The thesis proposes two standardisation pathways, a vendor-neutral execution interface and a set of quantum observability conventions. Throughout, the work brings the discipline of site reliability engineering to the quantum–classical space.

Keywords: quantum–classical interface, hybrid quantum–classical workloads, Kubernetes, OpenTelemetry, observability, cloud-native orchestration, Qiskit

Acknowledgements

I would like to thank my supervisor, Professor Ioannis Karafyllidis, for entrusting me with this thesis topic and for his constant support and inspiration. I am equally grateful to Professor Konstantinos Margaritis, who supervised my undergraduate dissertation over a decade ago and mentored me at the start of my career.

On a personal note, I thank my friend Aemilia Papaphilippou, an artist working across physics and mathematics, whose perspective and encouragement continually inspire me, and my partner, Clio, for her patience and support throughout these years.

This page intentionally left blank.

Contents

This page intentionally left blank.

List of Figures

This page intentionally left blank.

List of Tables

This page intentionally left blank.

1 Introduction

1.1 Motivation and Problem Statement

This thesis is motivated by the intersection of cloud-native infrastructure engineering and quantum computation. A professional background in Site Reliability Engineering (SRE) makes the absence of established distributed systems paradigms, robust observability, and infrastructure reliability frameworks within contemporary quantum environments immediately apparent. Automated scheduling, orchestration, and telemetry are foundational in classical cloud-native ecosystems, yet they remain largely unintegrated into the quantum execution layer.

This operational gap surfaced during the practical evaluation of Deutsch's algorithm (Deutsch, 1985). From a systems engineering perspective, executing this elementary circuit requires a multi-stage lifecycle: the abstract circuit must be transpiled into the backend's native gate set, routed across physical qubits, dispatched through a cloud runtime layer, and wait for the results. When executed on the IBM Quantum Platform, with `ibm_sherbrooke` backend using Qiskit, the resulting wall-clock execution time was approximately 3 seconds, as reported from job's QPU usage time. For a two-qubit workload, this latency appeared disproportionately high, serving as an empirical catalyst to systematically profile the underlying execution and compilation characteristics.

To understand where this time was spent, two diagnostic functions were developed: `metrics()`, which plots circuit depth, gate counts, scheduling timelines, and per-gate durations for a given backend; and `aer_mimic_simulation()`, which transpiles and executes circuits across different backends using `SamplerV2` with 4096 shots (the primitive's default). These tools enabled a systematic comparison across three execution environments: an ideal `AerSimulator`, a noise-mimicking `FakeSherbrooke`, and the real `ibm_sherbrooke` backend. The source code for both functions is provided in Appendix A.

The analysis revealed several findings. First, transpilation for the target backend

substantially restructured the circuit, although its raw counts are easily misread. The original Deutsch circuit had a depth of 5, contained 6 gates, and included a single non-local gate; after transpilation for `ibm_sherbrooke` the depth increased to 15, the reported gate count (`size`) rose from 6 to 151, and the total operation count grew from 9 to 154 (Figure 1.1, Figure 1.2). These totals are dominated, however, not by gates but by `delay` instructions. Transpilation embeds the two-qubit circuit into the backend's full 127-qubit register, and the `asap` scheduling pass then pads each idle qubit with a delay so that the execution timeline is fully specified. On `FakeSherbrooke`, the identical pipeline yields 129 `delay` operations against only 16 actual gates (9 `rz`, 4 `sx`, 2 `x`, and 1 `ecr`); the totals reported for `ibm_sherbrooke` arise in the same manner. The genuine effect of decomposition is therefore modest, expanding the circuit from 6 to of order twenty native gates drawn from the backend's set (`rz`, `sx`, `x`, `ecr`), mapped onto physical qubits 115 and 116; the headline counts instead reflect scheduling scaffolding distributed across the device. Consistent with this, the transpiled circuit (Figure 1.3), rendered with idle wires suppressed, displays only the two active qubits, on which the original circuit becomes a short sequence of rotation, $\sqrt{X}$, and `ecr` gates interspersed with synchronisation delays.

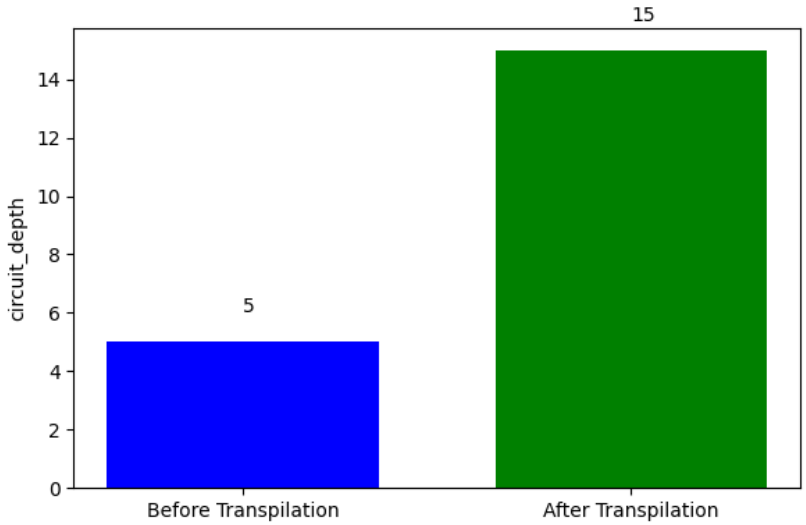


Figure 1.1: *Circuit depth before and after transpilation for `ibm_sherbrooke`.*

Second, an examination of the scheduling behaviour using `timeline_drawer()` function and the system cycle time ($dt = 0.2\,\text{ns}$) showed that measurement operations and synchronisation delays dominated the scheduled duration, while the quantum gates themselves occupied only a small fraction of the total timeline (Figure 1.4). The total scheduled time was $9466\,dt$, corresponding to approximately $1.89\,\mu\text{s}$. Separately, the linear sum of aggregate per-gate durations was approximately $14.8\,\mu\text{s}$ (Figure 1.5); this sum does not account for parallel execution in the scheduled timeline. Both estimates fall six orders of magnitude below the observed 3-second wall-clock time. The remaining time was consumed by the `SamplerV2` primitive, the IBM Runtime layer, job

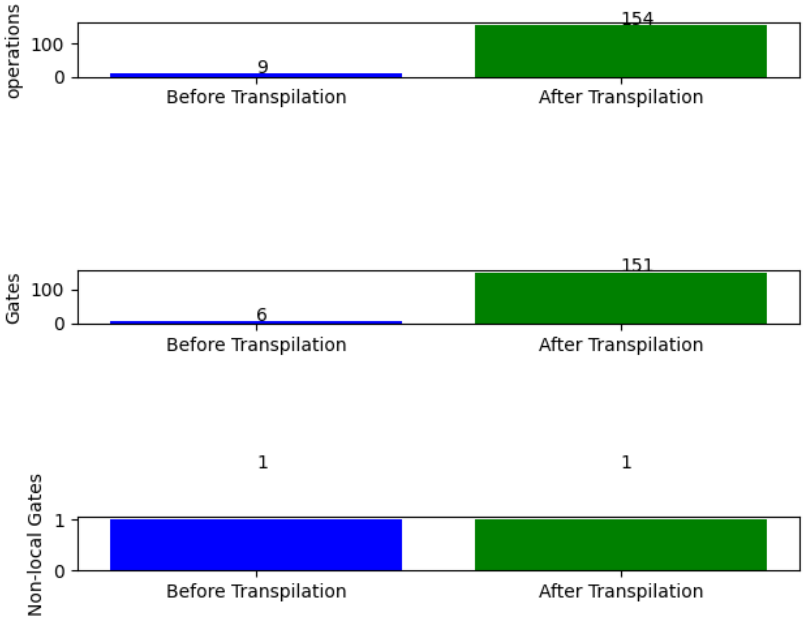


Figure 1.2: *Operation counts before and after transpilation for* `ibm_sherbrooke`.

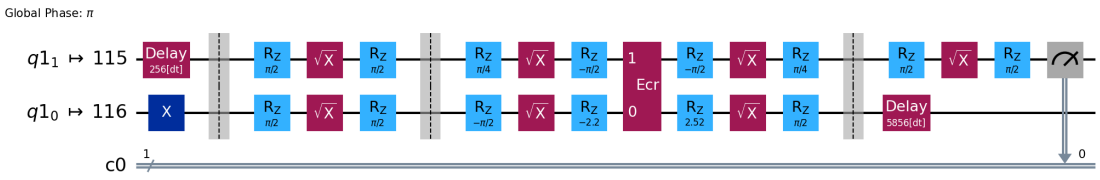


Figure 1.3: *Transpiled Deutsch circuit for* `ibm_sherbrooke`.

queuing, and classical–quantum communication overhead. None of this was visible or instrumentable by the user in Qiskit.

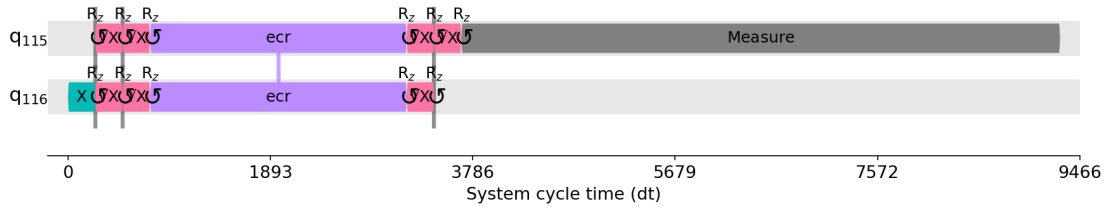


Figure 1.4: *Scheduling timeline of the transpiled Deutsch circuit on* `ibm_sherbrooke`*, generated with Qiskit's* `timeline_drawer()`*. Time is expressed in system cycle time (dt) units.*

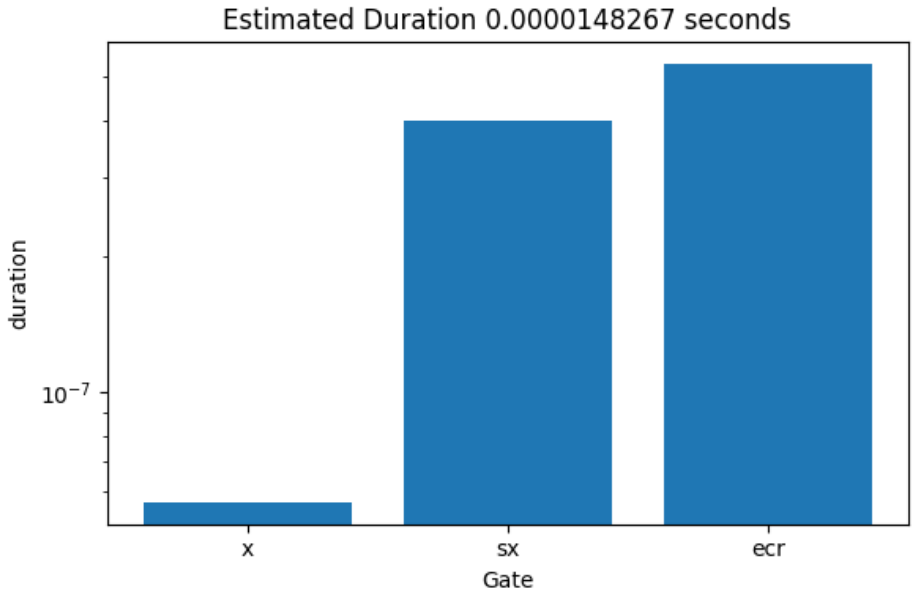


Figure 1.5: *Per-gate duration on* `ibm_sherbrooke` *for the three native gate types:* `x`*,* `sx`*, and* `ecr`*.*

The comparison across execution environments also revealed the limitations of simulated noise models. The ideal `AerSimulator` returned the correct output for every shot with zero errors. The `FakeSherbrooke` noise simulation introduced 38 erroneous counts of $|0\rangle$ out of 4096 shots. However, the real `ibm_sherbrooke` backend produced 140 erroneous counts of $|0\rangle$, nearly four times more than the simulated noise model predicted. This demonstrated that simulated noise models provide only a partial approximation of real hardware behaviour, reinforcing the need for runtime observability on actual quantum processors.

These findings from a simple two-qubit circuit raised the question of how they scale with circuit complexity. The same pattern returned, more sharply, in a later coursework assignment that implemented Shor's algorithm (Shor, 1994) for integer factoring, a standard benchmark for quantum period-finding (LaPierre, 2021). With the larger circuit, the choice of backend now determined whether the computation produced any usable signal at all, and the available tooling offered no way to judge that choice in

advance.

Factoring $N = 15$ with $a = 4$, a correct run should recover the period $r = 2$, visible as two peaks in the histogram at the bitstrings `0000` and `1000`, from which the factors 3 and 5 follow. On `ibm_brisbane`, selected for its immediate availability, the measurement counts were instead scattered across all 16 output states with no discernible signal at those outcomes (Figure 1.6), rendering the computation useless. Transpilation expanded the circuit from a depth of 9 to 1,727 and increased the gate count from 14 to over 3,400 on both backends. Executing the same circuit on `ibm_sherbrooke` produced a measurable, though noisy, signal at the expected outcomes. The `FakeSherbrooke` noise simulation proved more reliable than the actual `ibm_brisbane` hardware.

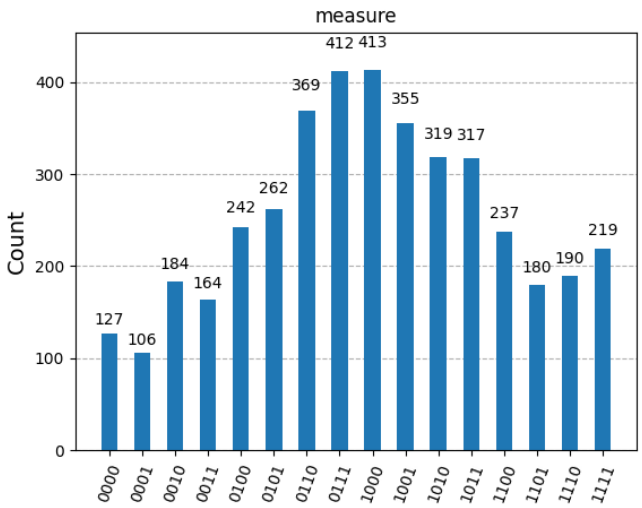
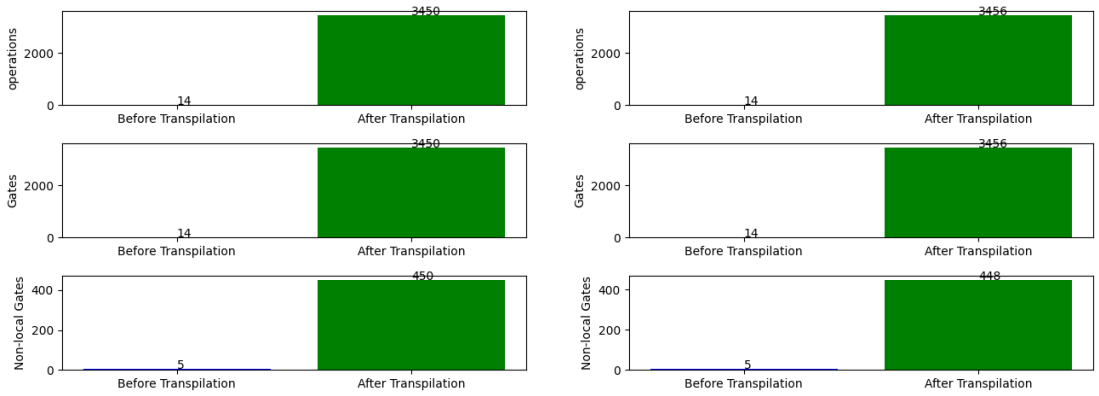


Figure 1.6: *Measurement counts for Shor's algorithm on `ibm_brisbane`.*

Counterintuitively, `ibm_brisbane` had fewer total operations after transpilation than `ibm_sherbrooke` (3,450 versus 3,456), yet produced significantly worse results (Figure 1.7). This suggested that the quality of qubit mapping and the physical topology of the backend mattered more than the raw operation count.



`ibm_brisbane` (3,450 ops) *`ibm_sherbrooke` (3,456 ops)*

Figure 1.7: *Operations comparison for Shor's algorithm across two backends.*

The experiments were conducted in August 2024 with the IBM Quantum Open Plan,[1] which provided access to three backends, most with significant job queues. The only programmatic selection mechanism available was Qiskit's `least_busy()` function, which considers queue depth but not calibration quality or circuit suitability. Beyond that, backend selection required manually inspecting calibration data, submitting trial jobs, and comparing outputs. This process was neither reproducible nor scalable. The execution model itself blocked a Python process until a backend became available, far removed from the asynchronous, event-driven patterns standard in modern cloud infrastructure.

By the standards of SRE practice, quantum cloud platforms lack operational maturity. Backends have no health checks or readiness probes. Fidelity issues surface only after job completion, with no programmatic signal indicating whether a backend's calibration state is suitable for a given circuit. Standard practices such as service level indicators have not been adapted for quantum execution: metrics like queue latency and job completion rate would transfer directly, but execution fidelity requires new definitions that account for the probabilistic nature of quantum measurement and the temporal drift of hardware calibration. Without such indicators, there are no error budgets and no systematic way to assess backend performance.

Taken together, these observations point to a structural gap. Quantum processors remain isolated, special-purpose resources rather than accelerators in a heterogeneous cloud platform managed with the orchestration, scheduling, and observability that classical infrastructure has long provided. Realising the full potential of quantum computing requires not further hardware refinement alone but an "integrated Quantum-Classical systems architecture" (Seelam et al., 2026).

This is not an artefact of one set of experiments but a reflection of where the field currently stands. Quantum computing has matured from a theoretical concept to a technology accessible through commercial cloud platforms. Providers such as IBM Quantum, Amazon Braket, and Azure Quantum now offer remote access to quantum processors, and major HPC centres are beginning to deploy quantum hardware alongside their classical infrastructure (Beck et al., 2024; Rallis et al., 2025; Seelam et al., 2026). However, current quantum devices remain constrained by noise, limited qubit counts, and short coherence times, a regime commonly referred to as the NISQ era (LaPierre, 2021; Preskill, 2025).

These constraints make hybrid quantum–classical computing the dominant execution model. The quantum processor handles specific subroutines while classical systems manage the surrounding workflow: preparing inputs, configuring the circuit, mitigating errors, and processing results. Even an algorithm that issues a single circuit reflects this division of labour. Shor's algorithm pairs a quantum period-finding circuit with classical pre-processing and a classical continued-fraction step that recovers the factors (LaPierre, 2021).

[1] Qiskit 1.1.1, Qiskit IBM Runtime 0.27.0, Qiskit Aer 0.14.2, Python 3.11.1.

This hybrid model is gaining institutional momentum. In 2024–2025, several HPC centres integrated quantum processors into their infrastructure: Quantinuum's trapped-ion system was coupled with the Fugaku supercomputer in Japan (Quantinuum, 2025), the EuroHPC programme deployed quantum simulators at Tier-0 sites in Germany and France (EuroHPC Joint Undertaking, 2025), and the University of Innsbruck integrated a trapped-ion system into its LEO5 cluster (University of Innsbruck and AQT, 2024). A 2025 perspective from the ADAC Institute, a consortium of more than twenty leading HPC centres, draws on a member survey to report that only a subset currently operate their own quantum systems, while many are building testbeds or accessing quantum hardware through the cloud and anticipate rising demand (ADAC Institute QC Working Group, 2025). This momentum extends to the software layer, where Kubernetes-based orchestration platforms such as Qubernetes (Stirbu et al., 2024) and Qonductor (Giortamis et al., 2025b), along with vendor-neutral middleware such as QRMI (Bacher et al., 2025), have been proposed to manage hybrid workloads.

Despite this momentum, a significant gap remains between quantum algorithm research and the infrastructure needed to deploy hybrid workloads in practice. Quantum SDKs such as Qiskit provide tools for circuit construction and execution (Javadi-Abhari et al., 2024), and compilation toolchains address instruction-level coordination (Seitz et al., 2023; Elsharkawy et al., 2024). The cloud-native efforts noted above, however, remain early-stage; the infrastructure layer that deploys, schedules, monitors, and manages hybrid workloads is still largely bespoke, with no common standards yet in place (ADAC Institute QC Working Group, 2025). A review of 107 publications on HPC–QC integration identifies the standardisation of interfaces and tools as the field's principal next step (Döbler et al., 2025).

This thesis addresses that gap by bringing cloud-native infrastructure practices to the classical layer surrounding quantum execution for workload orchestration, declarative deployment, and standards-based observability.

1.2 Research Objectives and Guiding Questions

1.2.1 Objectives

O1. Interface classification. Survey existing approaches for quantum–classical interfacing across architectural layers, establishing a classification that organises the current landscape.

O2. Representative interface analysis. Analyse interface models at the programming-framework level and the infrastructure level, identifying their design choices and limitations.

O3. Operational gap identification. Identify gaps related to production deployment, workload orchestration, and observability.

O4. Cloud-native workflow design. Design and demonstrate a cloud-native hybrid execution workflow using K8s for orchestration and OTel for telemetry.

1.2.2 Guiding Questions

Q1. Layers and classification. At which architectural layers does quantum–classical interfacing occur, and how can these layers be classified?

Q2. Framework versus infrastructure. How do framework-level interfaces differ from infrastructure-oriented approaches?

Q3. Production and observability limits. What limitations exist with respect to production deployment and observability?

Q4. Cloud-native transfer. How can cloud-native practices be adapted to hybrid quantum–classical workloads?

1.3 Scope and Boundaries

1.3.1 In Scope

- Software-based interfaces for gate-based quantum computing.
- Cloud-accessible quantum execution models.
- Hybrid workflows in which a quantum circuit executes within a classical orchestration layer, with Shor's algorithm ($N = 15$) as the reference workload.
- Orchestration, execution lifecycle, and observability for hybrid workloads.

1.3.2 Out of Scope

- Hardware-level control (pulse engineering, control electronics).
- New quantum algorithm design or claims of quantum advantage.
- Quantum error correction as a primary topic (Cai et al., 2023).
- Quantum networking and distributed entanglement (Caleffi et al., 2024).

1.3.3 Implementation Boundaries

Qiskit was selected as the primary SDK due to its maturity, extensive documentation, and the availability of cloud-accessible backends through IBM Quantum, used extensively throughout the MSc coursework. Shor's algorithm factoring $N = 15$ was chosen as the reference workload because it is a standard period-finding benchmark with a known-correct output, so result quality can be scored against ground truth, and its post-transpilation depth is large enough to exercise the full compile–route–execute–measure pipeline and to expose differences in backend quality across substrates.

1.4 Thesis Structure

Chapter 2: Information and Computation. Establishes computation as a physical process: Shannon entropy, Landauer's principle, the abstract model of computation, and the conditions under which a physical system can be said to compute.

Chapter 3: Classical Computing. Traces the classical infrastructure stack: hardware, operating systems, distributed systems, HPC, cloud computing and containers, and observability.

Chapter 4: Quantum Computing. Covers the qubit, multi-qubit systems and entanglement, quantum gates and reversibility, the circuit model, and the physical architecture of the QPU under the constraints of the NISQ era.

Chapter 5: Literature Review. Surveys the quantum–classical interface literature across four software-interface layers (applications, programming frameworks, compilation and runtime, and infrastructure and orchestration), with observability as a cross-cutting concern.

Chapter 6: Architecture. Proposes a cloud-native architecture for quantum–classical interfacing, with K8s-based orchestration and an OTel-based observability model.

Chapter 7: Implementation and Demonstration. Details the proof-of-concept: Shor's algorithm ($N = 15$) executed through K8s custom resources reconciled by an operator, with distributed tracing and metrics collected across the quantum–classical boundary, and the workload compared across simulated and real backends.

Chapter 8: Discussion. Assesses the results, discusses production readiness, and considers implications for researchers and platform engineers.

Chapter 9: Conclusions and Future Work. Summarises the contributions and outlines future work.

2

Information and Computation

The interface this thesis treats as a systems problem is an interface between two physical substrates. Each encodes, transforms, and decodes information in a different way. Treating that interface as an engineering problem first requires establishing what it means for a physical system to compute.

2.1 Information as a Physical Quantity

Information is always tied to a physical representation. Establishing this connection required resolving a paradox that persisted for over a century. The resolution revealed a link between information, energy, and entropy.

Entropy spans multiple domains of science. In thermodynamics, it quantifies the disorder of a physical system, the number of microscopic configurations (microstates) consistent with its observed macroscopic state (Boltzmann, 1877). A gas dispersed uniformly through a container has high entropy while the same gas compressed into one corner has low entropy. The second law of thermodynamics states that the total entropy of an isolated system can only remain constant or increase, but it never spontaneously decreases. It defines the arrow of time for physical processes. The heat flows from hot to cold, ordered states evolve toward disorder, and certain processes cannot be reversed.

In information theory, Shannon introduced an analogous measure to address the efficient communication of messages (Shannon, 1948). Shannon entropy quantifies the average uncertainty of a random variable's possible outcomes. Equivalently, it measures the expected information gained when the outcome is observed, or the minimum average number of bits needed to encode a source. A fair coin has maximum entropy of one bit. Each flip is maximally uncertain, so each outcome conveys exactly one binary digit of information. A coin that always lands heads has zero entropy because the outcome is certain, and observing it reveals nothing. Shannon's formula is identical in form to Boltzmann's thermodynamic entropy, with the probabilities of message symbols replacing the probabilities of physical microstates.

In his 1871 *Theory of Heat*, Maxwell proposed an experiment that appeared to violate the second law (Maxwell, 1871). He imagined a container of gas divided into two halves by a partition with a small door, operated by an intelligent agent, the *demon* (Figure 2.1). The demon observes individual molecules approaching the door. When a fast molecule approaches from one side, it opens the door to let the molecule pass and when a slow molecule approaches from the other side, it lets the molecule through in the opposite direction. Over time, fast molecules accumulate on one side and slow molecules on the other. A temperature difference emerges from an initially uniform gas, an apparent decrease in entropy with no work performed on the gas.

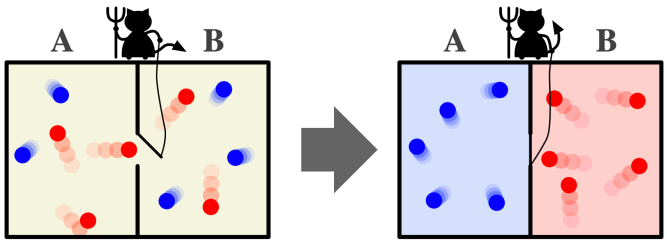


Figure 2.1: *Maxwell's demon. Adapted from Htkym, Wikimedia Commons, licensed under CC BY-SA 3.0.*

The paradox stood for over a century because Maxwell's formulation did not account for the demon itself as a physical system. The demon observes, decides, and acts, but each operation has a physical cost. Landauer addressed this cost from the perspective of computation. He showed that any logically irreversible operation, such as erasing a bit of memory, must dissipate at least $kT \ln 2$ of energy as heat, where k is Boltzmann's constant and T is the temperature of the environment (Landauer, 1961). This result, known as Landauer's principle, defined the energy cost of irreversible computation and unified thermodynamics and information with entropy. Erasing one bit of Shannon entropy must increase the thermodynamic entropy of the environment by a corresponding amount.

Bennett completed the resolution by showing that the demon's measurements can in principle be performed reversibly, without thermodynamic cost (Bennett, 1982). The entropy cost arises elsewhere, because the demon has finite memory. After enough observations, it must erase its recorded data to continue operating. That erasure, by Landauer's principle, dissipates at least as much energy as the sorting saved. The total entropy of the combined system, gas and demon together, never decreases. The second law is preserved.

This chain of reasoning leads to a foundational conclusion: information is not an abstract mathematical construct but a physical entity, subject to the same laws that govern energy and matter. Recording information requires a physical medium. Processing information involves physical operations. Erasing information has a measurable, irreducible energy cost. Landauer captured this view by arguing that the laws of physics are themselves algorithms for handling information (Landauer, 1996). Physical systems are therefore not only carriers of information but also computational entities.

2.2 The Abstract Model of Computation

While Landauer established that information processing is a physical act, the question of what can be computed was formalised decades earlier. In 1936, Turing addressed the *Entscheidungsproblem*, the question of whether a mechanical procedure can decide the truth or falsity of any mathematical statement, by introducing an abstract machine now known as a Turing machine (Turing, 1936).

A Turing machine has three components: an infinite tape divided into cells, each holding a symbol from a finite alphabet; a read/write head that moves one cell at a time in either direction; and a finite state register with a transition function. The transition function determines, for each combination of current state and symbol read, which symbol to write, which way to move, and which state to enter next. Despite this simplicity, Turing proved that such a machine can simulate any algorithmic process. The Church–Turing thesis generalises the result. It asserts that any function which can be effectively computed by any procedure can be computed by a Turing machine. The thesis has never been proven in the mathematical sense, but it remains the benchmark for computational universality.

Modern digital computers are practical realisations of this principle. They replace the infinite tape with finite memory and the transition rules with instruction sets, and they add parallelism, input/output systems, and layered software abstractions. The Turing model defines what is computable in the abstract, but it does not address the physical substrate that performs the computation.

2.3 When Does a Physical System Compute?

Recognising the gap between abstract computability and physical realisation, Horsman et al. developed Abstraction/Representation (AR) theory to formalise when a physical process constitutes computation (Horsman et al., 2014).

In AR theory, a computer is a physical system with constituent parts and internal interactions that can move from one physical state to another. To qualify as computing, the system must satisfy three requirements: it must encode abstract data into physical states, it must evolve those states through at least one dynamical operation, and it must allow the results to be decoded back into abstract form. The element that distinguishes computation from physical evolution alone is the *representation relation*, a mapping between physical states and mathematical objects that lets the physical dynamics be interpreted as an abstract calculation.

AR theory introduces the concept of *computational entities*, the physical components that locate and maintain the representation relation within the system. These entities perform the encoding and decoding steps, translating abstract data into physical states and back. In classical computing, computational entities include processors, memory controllers, and input/output interfaces. In quantum computing, they include the control electronics that prepare quantum states and the measurement apparatus that

extracts classical results. A key insight of AR theory is that computation is predictive, since a computer is used to predict the outcome of an abstract evolution. This predictive role distinguishes computing from other physical processes, because the system must reliably produce results that correspond to the abstract calculation.

Stepney and Kendon extended the AR model to include a Representational Entity (RE), the agent that establishes and maintains the representation relation between physical and abstract domains (Stepney et al., 2021). The full compute cycle, shown in Figure 2.2, proceeds through several steps. The RE represents its desired physical state P_{RE} as an abstract model $m_{P_{\mathrm{RE}}}$, encodes this into a computational model m_{P_c}, and instantiates it into the physical computer state P_c. The physical computer evolves to a new state P'_c through its internal dynamics H_c. The resulting state is represented as an abstract computational solution m'_{P_c}. This is finally decoded to the abstract problem solution $m'_{P_{\mathrm{RE}}}$, the result of the abstract evolution C_c that models the final state of the RE. Each step in this cycle must approximately commute, keeping the representational entity and the physical computer consistent throughout the computation.

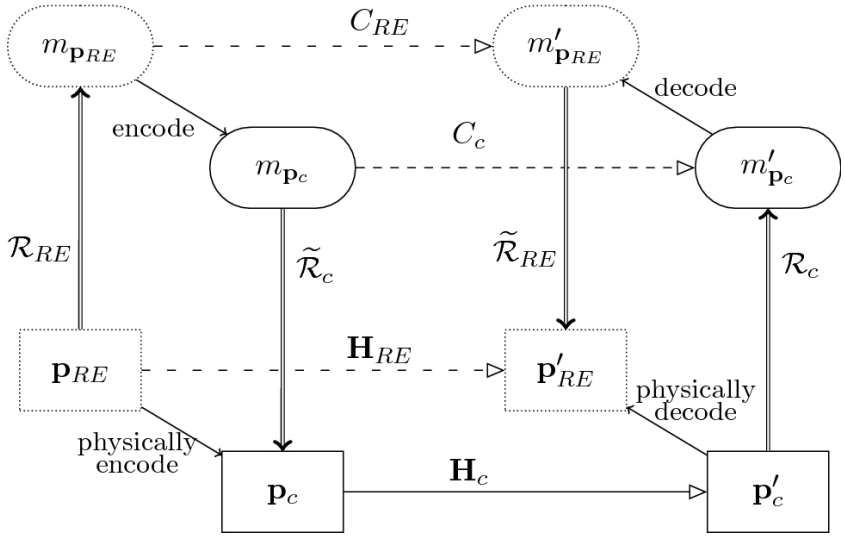


Figure 2.2: *The full compute cycle with the Representational Entity extension. Reproduced from Stepney et al. (2021) Fig. 5.*

2.4 Implications for Classical-Quantum Interfacing

The Turing model and AR theory together provide the lens for this thesis. The Turing model defines the boundary of what is computable and AR theory defines what it means for a physical system to realise that computation. Both frameworks apply to classical and quantum computers alike, since a quantum computer follows the same encode–evolve–decode cycle described by AR theory.

The concrete realisation of each step differs between the two substrates. In a classical system, encoding is the familiar process of writing data to memory, evolution is deterministic gate-level logic, and decoding is reading a definite result. The supporting

infrastructure (operating systems, schedulers, monitoring systems) has been refined over decades. In a quantum system, encoding becomes quantum state preparation, evolution becomes the application of unitary gates to qubits in superposition, and decoding becomes projective measurement. Measurement is probabilistic and collapses the quantum state, yielding to a statistical distribution rather than a definitive state.

3
Classical Computing

Chapter 2 established computation as a physical process, an encode–evolve–decode cycle that any substrate, classical or quantum, must realise. On classical hardware the cycle is concrete and familiar. Encoding writes data to memory, evolution applies deterministic logic, and decoding reads a definite result. The infrastructure that makes this cycle reliable, repeatable, and observable was refined over almost five decades. This chapter traces that infrastructure as a stack of abstractions across those decades.

3.1 Hardware Foundations and Binary Encoding

The classical representation relation is *binary encoding*. Every abstract value, whether an integer, a character, or an instruction, maps to a sequence of bits, and each bit is realised physically as a voltage held above or below a threshold in a transistor circuit (Silberschatz et al., 2018). Two properties of this encoding shape every layer above it. It is *deterministic*, since one abstract value always maps to the same bit pattern, and that pattern always decodes back to the same value. It is also *non-destructively readable*, since any bit can be inspected at any time without disturbing it. These properties are so basic to classical computing that they are rarely stated. They become significant only by contrast, with Unconventional Computing paradigms, such as Quantum Computing.

Bits compose upward into a hierarchy of digital logic (Silberschatz et al., 2018; Matthews et al., 2022). A *register* is a small, fast group of bits that the processor treats as a single unit with the width of a specific *word* size. *Logic gates* transform bits according to fixed Boolean rules such as AND, OR, and NOT, and gates wired in sequence form the *circuits* that carry out operations such as integer addition. Most classical gates are irreversible. An AND gate maps two input bits to a single output bit, and the inputs cannot be recovered from that output alone. Chapter 2 gave this irreversibility its thermodynamic weight through Landauer's principle. Quantum gates, by contrast, must be reversible, a difference taken up in Chapter 4.

These components are organised by the von Neumann architecture (Figure 3.1), the model underlying most modern computers (Matthews et al., 2022). It defines five

units: a *processing unit* that executes instructions, a *control unit* that drives their execution, a *memory unit* that holds both data and instructions, and input and output units that move data in and out. The processing and control units together form the CPU. The processing unit contains the *arithmetic/logic unit* (ALU), which performs the arithmetic and logical operations, together with a bank of registers holding the operands and results currently in use. The control unit tracks execution through two special-purpose registers. The *program counter* (PC) holds the address of the next instruction, and the *instruction register* (IR) holds the instruction currently being executed.

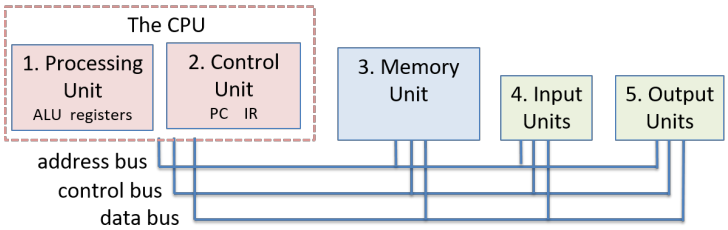


Figure 3.1: *The five units of the von Neumann architecture. Reproduced from Matthews et al. (2022), licensed under CC BY-NC-ND 4.0.*

The five units repeat a four-stage *fetch–decode–execute–store* cycle (Matthews et al., 2022). The control unit fetches the next instruction from the address in the PC, decodes it to determine the operation and the location of its operands, the processing unit executes it in the ALU, and the result is written back to a register or to memory. The PC then advances and the cycle repeats. Modern processors overlap consecutive instructions through *pipelining*, fetching one while another is decoded and a third is executed, which raises throughput by keeping the stages busy at once (Matthews et al., 2022). Each pass is one step of the physical system evolving from one state to the next, the hardware-level realisation of the encode–evolve–decode cycle of Chapter 2.

The CPU does not work in isolation. It reads instructions and data from main memory, the random-access memory (RAM) that is fast but volatile, while persistent storage such as solid-state and hard drives retains data across power cycles at far lower speed. Buses connect the units: a data bus transfers values, an address bus carries the location of each read or write, and a control bus signals the requested action. The *stored-program* model places instructions and data in the same memory (Matthews et al., 2022). This is what makes a computer programmable, because loading a different program, rather than rewiring hardware, changes what the machine computes. It also imposes a structural constraint. The shared path between processor and memory must carry both instruction fetches and data transfers, and the processor can consume data faster than that path can supply it (Silberschatz et al., 2018). This bottleneck is the first instance of a pattern that recurs throughout the chapter, in which a resource limit at one layer drives the design of the next, from on-chip caches to computation distributed across thousands of machines.

A bare machine can execute only one program at a time. Abstracting and mediating access to the physical resources is the role of the operating system.

3.2 Operating Systems and the Kernel

An operating system is the software layer that manages a computer's hardware and provides the environment in which programs run (Silberschatz et al., 2018). It is made up of the kernel together with system libraries and utilities. The *kernel* is the part that matters most for this thesis. It is the lowest layer of software, runs in privileged mode with direct access to the hardware, and stays resident in memory for as long as the system is running. Everything an application does, from reading a file to sending a network packet, ultimately passes through it. The kernel's way of organising computation is worth establishing carefully, because the architecture proposed in Chapter 6 reuses its core concerns for quantum workloads: scheduling, lifecycle management, resource isolation, and observability.

The kernel's central job is to manage resources so that many programs can share one set of physical components without interfering with each other (Silberschatz et al., 2018; Gregg, 2020). This work falls into a handful of areas that recur, in different forms, at every level above it. Process management creates, schedules, and terminates the running programs that compete for the processor. Memory management gives each of them a private address space, backed by virtual memory and page tables, so that one program cannot reach into another's memory. Storage is reached through the virtual file system (VFS), a uniform interface that hides the differences between the many file-system implementations beneath it. Device drivers do the same for hardware, presenting disks, network adapters, and accelerators to the rest of the kernel through standard internal interfaces. Hardware interrupts let those devices signal the processor asynchronously when they need attention, and a periodic timer interrupt lets the kernel preempt a running process to reclaim the processor. Two further mechanisms handle communication. The networking stack implements the TCP/IP protocols and the sockets that let processes talk across machines, while inter-process communication (IPC) lets processes on one machine coordinate through pipes, signals, and shared memory, the kernel-mediated channels by which separate processes combine into a larger working system.

Because user programs cannot touch the hardware directly, they ask the kernel to act for them through *system calls* (Silberschatz et al., 2018; Gregg, 2020). A system call is the defined entry point for a privileged operation such as reading a file, allocating memory, or opening a network connection. Invoking one switches the processor from user mode into kernel mode, the kernel carries out the request, and control returns to the program. This controlled crossing between user space and kernel space is the interface through which an application reaches everything the operating system manages.

The unit the kernel actually schedules and isolates is the *process* (Silberschatz et al., 2018). A program sitting on disk is only data, a stored sequence of instructions and constants, and it does nothing until the kernel turns it into a process by giving it memory, loading its binary, assigning it an identifier, and starting execution at its entry point. The same program can be launched many times over, each instance a separate

process with its own state. The kernel records that state in a per-process structure, the Process Control Block, called the `task_struct` in Linux, which holds the process's registers, program counter, memory mappings, open files, and scheduling priority.

That isolation rests on the virtual address space (Silberschatz et al., 2018). Each process sees a private range of memory addresses, and the memory management unit (MMU) translates those virtual addresses into physical ones through per-process page tables. Because no process's page tables map another's memory, one process cannot read or corrupt the memory of another, even by accident. The same boundary contains failures. When a process crashes, the kernel reclaims its memory and other resources, and the rest of the system carries on.

Creating and running a process on Unix-like systems, which underpin almost all modern servers, HPC clusters, and cloud platforms, follows a distinctive two-step pattern (Silberschatz et al., 2018; Gregg, 2020). A process first duplicates itself with `fork`, producing a child that is a copy of the parent, and the child then replaces its own program image with a new binary through `exec`. The parent waits for the child to finish and collects its exit status when it does. This shape, in which a unit of work is created, run to completion, and then has its result collected, is one of the most durable patterns in computing. It reappears at every scale, from a single Unix process up to a K8s Job that runs a quantum circuit on a remote backend and reports back when the run completes (Chapter 7).

Among these responsibilities, scheduling is the one this thesis leans on most heavily. At any instant a machine may have far more runnable processes than processor cores, so the kernel's scheduler must decide which process runs where, and for how long (Silberschatz et al., 2018). Switching the processor from one process to the next is a *context switch*, in which the kernel saves the registers and program counter of the outgoing process into its `task_struct` and restores those of the incoming one, a small but real cost the scheduler must weigh.

It does so while balancing goals that pull in different directions: responsiveness for interactive tasks, throughput for batch work, and fairness across everything competing for the processor. Linux has revised this component repeatedly.

A final design sensibility runs through all of this. The Unix tradition favours small, focused tools that combine through simple, standard interfaces such as pipes, files, and sockets. Its guiding principles carry forward into every layer that follows: modularity, composability, and separation of concerns.

Underlying every one of these mechanisms is a single principle. The kernel manages the resources a computation needs and leaves the computation itself to the application. It allocates processor time, isolates memory, schedules competing work, and exposes the interfaces through which programs reach the hardware, yet the logic being computed is never its concern. This separation between a computation and the infrastructure that runs it is one of the most productive ideas in systems engineering, and it is the separation this thesis carries into the quantum setting, where a classical control plane manages execution and the quantum processor performs the computation.

3.3 Distributed Systems

Distributed computing extends the single-machine model by coordinating many independent computers to solve problems that no single processor and memory could hold (Kleppmann et al., 2026). Workloads such as weather simulation, molecular dynamics, genomics, and large-scale data analysis routinely outgrow one machine, and the only way forward is to spread the computation across many.

Spreading a computation across a network breaks three assumptions that held on a single machine (Kleppmann et al., 2026). Processes on different machines share no memory, so every exchange of information has to be made explicit. They share no clock, so events on different machines cannot be placed in a definite order without extra coordination. And they fail independently, so any one machine can crash, become unreachable, or simply slow down while the others keep running.

The distinction between a distributed system and a parallel one turns on exactly these points. Figure 3.2 contrasts the two. A distributed system is a set of networked computers, each with its own private local memory, that coordinate only by passing messages across the links between them. A parallel system instead places its processors behind a single shared memory, through which they exchange information directly.

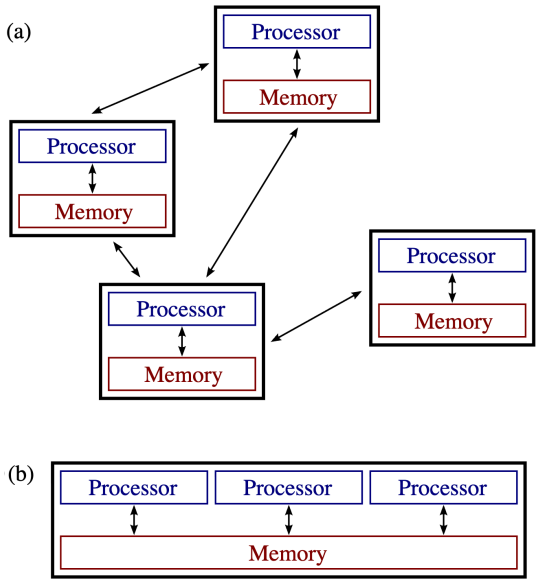


Figure 3.2: *Distributed (a) and parallel (c) systems contrasted. Adapted from Miym, Wikimedia Commons, licensed under CC BY-SA 3.0.*

Because there is no shared memory, communication itself becomes a first-class design problem (Burns, 2024). On one machine, processes exchange data through the kernel-mediated inter-process communication of Section 3.2, with negligible latency and guaranteed delivery. Across a network none of that holds, since messages can be delayed, reordered, duplicated, or lost. Two models dominate the response. *Message passing* keeps the network in plain view, with nodes sending and receiving messages directly, as MPI (Message Passing Interface) does for HPC workloads and HTTP does

for web services. *Remote procedure calls* (RPCs) take the opposite approach, hiding the network behind an ordinary function-call interface so that a remote invocation reads like a local one. gRPC, built by Google on HTTP/2, has become the dominant RPC framework in cloud-native infrastructure. Kubernetes (K8s) uses it for internal communication between components, and it is the usual choice for service-to-service calls between microservices (Burns, 2024).

A local function call either returns or fails, whereas a remote call can also hang indefinitely, or return only after an unpredictable delay due to unreliable network. In addition, a request may succeed on the server while its response is lost on the way back, leaving the caller unable to tell whether the operation took effect at all. Distributed systems manage reliability with a standard set of techniques: replication keeps copies of data on several nodes, timeouts bound how long a caller waits before assuming the worst, retries paired with idempotency let a request of uncertain outcome be reissued safely, and health checks probe components continuously so that failures are caught early (Burns, 2024).

Beyond tolerating failure, a distributed system needs ways to have consensus over the distributed states. Communication protocols let a set of nodes agree on shared state even when some of them fail. Distributed schedulers decide which node runs which piece of work. Across a cluster they play the role that the kernel's process scheduler plays on a single machine (Section 3.2), multiplexing a pool of shared resources among more consumers than it can serve at once. How they do so depends on the workload. Tightly coupled clusters rely on batch schedulers, also known as HPC schedulers, while loosely coupled services are placed by orchestrators (Section 3.4).

High-performance computing is distributed computing specialised for compute-intensive scientific and engineering work (Gropp et al., 2020). It sets aside the assumption of an unreliable, open network and builds instead on dedicated hardware, tightly coupled clusters joined by high-bandwidth low-latency interconnects, and a single administrative domain over every resource. Its two programming models mirror the memory distinction of Figure 3.2. MPI, introduced above, provides distributed-memory parallelism by passing messages between nodes; OpenMP provides shared-memory parallelism within a node through compiler directives; and large codes combine the two (Gropp et al., 2020).

Work reaches an HPC cluster through a batch scheduler, most often Slurm, to which users submit jobs declaring the resources they need (nodes, cores, memory, runtime, and accelerators) for the scheduler to place by availability and priority (Gropp et al., 2020). The model suits large, predictable jobs, but it assumes that resources are co-located and that requirements are known in advance. Modern clusters are also heterogeneous, pairing general-purpose CPUs with accelerators such as GPUs for the heaviest numerical work (Gropp et al., 2020). Co-located hardware, tight interconnects, and batch scheduling served large-scale scientific computing for decades.

From a single kernel to clusters of thousands of machines, these principles set the stage for the next shift: computing delivered on demand as a service.

3.4 Cloud Computing, Virtualisation, and Containers

Cloud platforms turned computing into a utility. Instead of buying and running physical hardware, organisations and users provision resources on demand through APIs and pay only for what they use. This changed the economics of the technology industry. Infrastructure shifted from a capital expense, bought and maintained ahead of need, to an operating cost billed by the hour, and capacity could grow or shrink with demand rather than being provisioned for the long term (Armbrust et al., 2010).

Commercial providers such as Amazon Web Services, Google Cloud Platform, and Microsoft Azure built large businesses on this model, while open-source platforms such as OpenStack let organisations run private clouds on their own hardware. Infrastructure as a service (IaaS) provides raw virtual machines, storage, and networking, leaving the operating system and everything above it to the customer. Platform as a service (PaaS) adds a managed runtime, so the customer deploys code rather than machines. Software as a service (SaaS) delivers an application over the network. A further level, serverless or function as a service (FaaS), runs short-lived functions on demand and charges only for the time they execute. The more of the stack the provider operates, the less the customer has to run.

Two technical foundations made this possible, virtualisation and containerisation, and both build on the kernel primitives described in Section 3.2.

Virtualisation lets several operating systems share one physical machine, each running inside a *virtual machine* that behaves as though it were dedicated hardware. A *hypervisor*, or virtual machine monitor, creates and runs these virtual machines (Gregg, 2020). The result is two levels of system software. At the lower level the hypervisor controls the real processor, memory, and devices. At the upper level, inside each virtual machine, a full guest operating system runs, its *guest kernel* driving the virtual hardware as though it were real. The guest kernel expects the same control over the processor that a kernel has on bare metal, but the hypervisor cannot grant real control without losing the machine. So the processor runs the guest's ordinary instructions directly and traps any instruction that would touch the real hardware down to the hypervisor, which performs it on the guest's behalf. This is *trap-and-emulate* (Popek et al., 1974).

Doing it in software for every such instruction is slow, so processors gained hardware support, the extensions Intel calls VT-x and AMD calls AMD-V (Uhlig et al., 2005). These run the guest in a dedicated guest mode, where its kernel executes directly on the processor and schedules its own processes without trapping into the hypervisor, at close to native speed. The guest leaves guest mode only when it reaches hardware the hypervisor owns, such as reading or writing a virtual disk or sending a network packet. That transition is a *VM exit*. Like a trap or interrupt, it forces the processor into a more privileged handler, except that it enters the hypervisor rather than the kernel. The hypervisor performs the operation, then resumes the guest with a *VM entry*. Each exit costs time, so a compute-bound workload runs almost as fast as

on bare metal, while an I/O-heavy one pays for its frequent exits.

Hypervisors come in two kinds. A Type 1, or bare-metal, hypervisor runs directly on the hardware with the guests as its only tenants, as in VMware ESXi or Xen. A Type 2, or hosted, hypervisor runs as a program on top of an ordinary operating system, as VirtualBox does on a desktop. Linux blurs the line between the two. There the hypervisor is the host kernel itself. The Kernel-based Virtual Machine (KVM) is a module that gives the running kernel this role, so it runs directly on the hardware like a Type 1 while remaining a full operating system that still schedules ordinary processes like a Type 2 (Gregg, 2020). KVM switches the processor into its virtualisation mode and reuses the scheduler, memory management, and cgroups the kernel already provides. A guest is then an ordinary host process, each of its virtual CPUs a thread that the normal Linux scheduler runs like any other. KVM handles the processor and memory, while a userspace process such as QEMU provides the guest's virtual devices and handles the device I/O that traps out to it. From the host kernel's point of view, a virtual machine is just a process, one that periodically drops into guest mode. The isolation is strong, because each guest keeps its own kernel. That kernel is also the cost, since every VM carries a full operating system and takes gigabytes of memory and seconds or more to boot.

A container is the lighter alternative, and to the kernel it is barely an abstraction at all. Where a VM gives each guest its own kernel, every container on a host shares the host's single kernel and runs as an ordinary process, isolated by features that kernel already provides (Gregg, 2020). A container starts with the `clone` system call, the general form of the `fork` from Section 3.2, issued with namespace flags. The new process lands in fresh *namespaces*, each giving it a private view of one global kernel resource. A PID namespace makes the process believe it is PID 1, the root of its own process tree. A network namespace gives it its own interfaces, routes, and ports, and a mount namespace its own filesystem tree. Further namespaces isolate the hostname, inter-process communication, and the user and group IDs. *Cgroups*, or control groups, then meter and cap the process's CPU time, memory, and I/O, while seccomp filters the system calls it may issue and the Linux security modules confine what it may access.

The container's root filesystem comes from its image rather than a disk. A union filesystem such as overlayfs stacks the read-only layers of the image into one tree, adds a thin writable layer on top, and presents that tree as the process's root. From the kernel's point of view, a container is just a process too, running with a restricted view of the system. The kernel never creates a thing called a container. It creates a process, scheduled no differently from any other, and limits what that process can see and use. This is why a container starts in milliseconds and costs megabytes rather than gigabytes. It is also why its isolation is weaker, since the shared kernel means a single kernel vulnerability crosses the boundary a separate guest kernel would have held. A virtual machine and a container are thus two ways of confining the same primitive, a scheduled process, differing only in whether the boundary is a second kernel or the host kernel's own features.

What runs a container is a *container runtime*, split in practice into two layers (Burns et al., 2022). A high-level runtime, containerd with Docker built on top of it, pulls and stores images and manages the container lifecycle. It hands creation to a low-level runtime, runc, the reference implementation that turns the OCI bundle into the running process. The Open Container Initiative (OCI) standardises the two contracts between these pieces, an image format and a runtime specification, so any compliant tool interoperates. A container image is an immutable, content-addressed artifact, identified by a cryptographic digest of its contents. It packages the application binary with the shared libraries and files it needs at run time, so the process brings its own userspace and depends on nothing installed on the host. Images live in a registry, and because each is addressed by its digest, any node that pulls one runs the identical artifact, which makes a container deployment reproducible.

Running containers on one machine is straightforward, but running thousands across a fleet is not, and that is the problem Kubernetes was built to solve (Burns et al., 2022). Kubernetes became the industry-standard orchestrator, and its design descends directly from Borg, the system Google used to manage clusters of tens of thousands of machines through declarative configuration and reconciliation loops (Verma et al., 2015). What it inherited from Borg is a single principle, reconciliation, and it runs through everything that follows. At the scale these clusters run, machines, processes, and network links fail constantly, so the design assumes failure rather than trying to prevent it. The user declares the desired state, and controllers compare it against what is actually running and act to close the difference, again and again. Action follows from observation rather than from a fixed sequence of steps. A dropped event or a crashed process is never fatal, because the next pass of the loop sees the gap and corrects it. Convergence, not choreography, is what keeps the system correct.

A Kubernetes cluster divides into a control plane, which holds the cluster's desired state and makes global decisions, and a set of worker nodes, which run the workloads (Figure 3.3). Table 3.1 lists the components of each (Burns et al., 2022). Above these components are the abstractions that workloads are built from. A handful recur in the architecture and implementation that follow, and Table 3.2 collects them.

Custom Resource Definitions, Operators, and Device Plugins together make Kubernetes extensible enough to orchestrate far more than the stateless microservices it was first built for. With new object types, controllers to reconcile them, and hardware advertised beyond CPU and memory, one declarative model can govern workloads the platform never shipped with. The same control loop now governs how clusters themselves are operated, an approach the industry has converged on, GitOps[1], in which a Git repository holds the declared cluster state and reconciliation controllers such as Argo CD[2] drive the live infrastructure to match it. Chapter 6 builds on exactly this extensibility, treating a quantum backend as one more schedulable resource and a circuit as a Job that runs to completion.

[1] https://opengitops.dev
[2] https://argoproj.github.io/argo-cd/

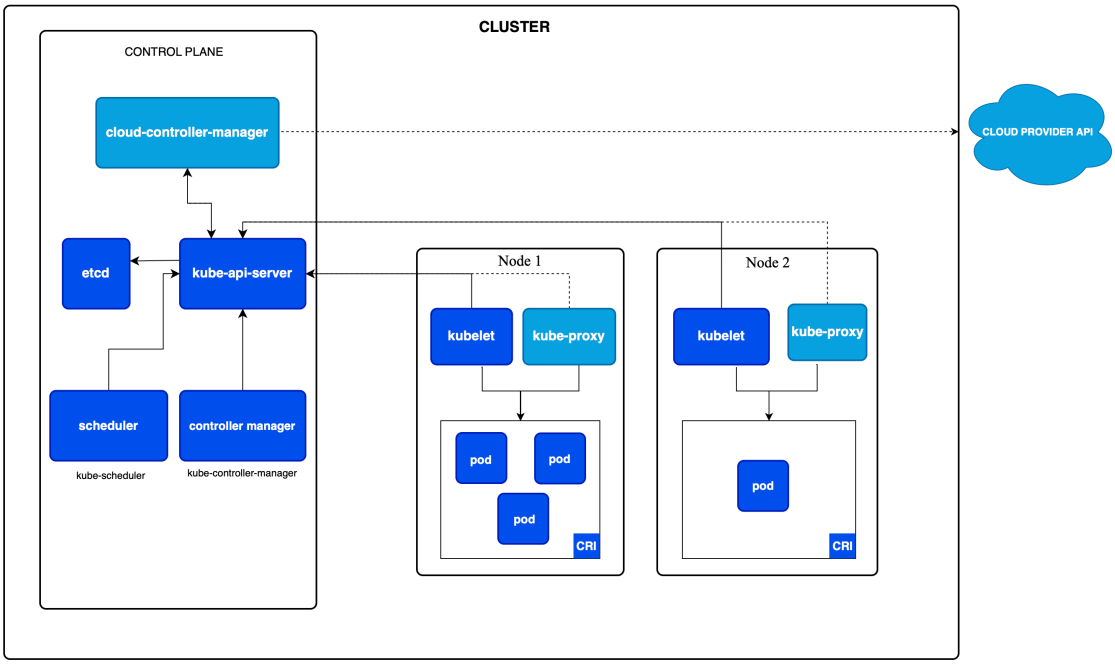


Figure 3.3: *Kubernetes cluster architecture. Reproduced from the Kubernetes documentation, by The Kubernetes Authors, licensed under the CC BY 4.0.*

Table 3.1: *The Kubernetes cluster components.*

Component	Role
Control plane	
API server	The cluster's API surface and single point of entry. All reads and writes of cluster state pass through it, and it is the only component that talks to the datastore.
etcd	The cluster's source of truth, a consistent key-value store that persists the desired and observed state every other component reconciles against.
Scheduler	Decides placement, assigning each unscheduled Pod to a node whose available resources and constraints satisfy the Pod's requirements.
Controller manager	Runs the built-in control loops, among them the node, Deployment, and Job controllers. Each watches one kind of object and drives its observed state toward the declared one.
Cloud controller manager	Connects the cluster to a cloud provider, delegating provider-specific logic such as load balancers and node lifecycle. It runs only on cloud-hosted clusters.
Worker node	
kubelet	The control plane's agent on each node. It drives the node's container runtime to match the Pods assigned to it and reports their status back.
Container runtime	The component that executes containers on a node, invoked by the kubelet through a standard interface.
kube-proxy	Implements the Service abstraction on each node, routing traffic addressed to a Service to one of the Pods currently backing it.

Table 3.2: *Kubernetes building blocks relevant to this thesis.*

Building block	Role
Pods	The smallest unit Kubernetes runs and schedules. It wraps one or more co-located containers that share a single network address and storage.
Services	A stable network identity in front of a set of interchangeable Pods. Because individual Pods are ephemeral, clients address the Service and it forwards to whichever Pods are currently available.
Endpoints	The current list of Pods that back a Service. Kubernetes keeps it up to date automatically as matching Pods come and go.
ConfigMaps & Secrets	External configuration for a workload, kept out of the container image. A ConfigMap holds plain settings and a Secret holds sensitive ones such as credentials. A Pod reads either as environment variables or as files mounted into it.
Deployments	The controller for long-running, stateless services. It manages a ReplicaSet that keeps a declared number of identical Pods running, replacing failures and rolling out new versions without downtime.
Jobs	A task that runs to completion, rather than a service that stays up. The Job starts Pods and finishes once the required number complete successfully.
CRDs	The extension point for new API types. A CRD declares a new kind of object with its own schema, which the API server stores, validates, and serves like a built-in resource. A CRD adds only the type, and its behaviour comes from a controller.
Operators	The controller that gives a CRD its behaviour. It watches the custom resource and manages the application behind it automatically, as a human operator would by hand.
Device Plugins	The bridge between specialised hardware and the scheduler. A node-local plugin registers hardware such as GPUs and FPGAs with the kubelet over a defined gRPC interface, making it schedulable like CPU or memory.
Labels & annotations	Key-value metadata attached to objects. Kubernetes groups and selects objects by their labels, while annotations hold non-identifying notes for humans and tools that the system does not select on.

3.5 Observability

The term *observability* comes from control theory, where Rudolf Kálmán introduced it in 1960 as a formal property of dynamical systems (Kalman, 1960). A system is observable when its complete internal state can be reconstructed from its external outputs over a finite interval. For a linear time-invariant system with state matrix A and output matrix C, this holds when the observability matrix

$$O = \begin{bmatrix} C \\ CA \\ CA^2 \\ \vdots \\ CA^{n-1} \end{bmatrix} \tag{3.1}$$

has full rank, so that its rows span the entire state space. The property has a formal dual, *controllability*, which is the ability to drive a system from any initial state to any desired one through its inputs. A system is controllable exactly when its dual is observable (Kalman, 1960).

The same property matters for software as well. A system is observable to the degree that an operator can tell what is happening inside it from what it emits outside, namely its telemetry (Majors et al., 2022). The reason this matters grows with scale. On a single machine a developer can attach a debugger and read program state directly. A distributed system offers no such vantage point, because its components run on separate machines and fail independently (Section 3.3), so the telemetry they emit is the only practical window into their combined behaviour.

Much of this telemetry is engineered into the operating system itself. The kernel exposes system-call tracing and the `/proc` pseudo-filesystem for resource accounting and statistics (Section 3.2). This is the lowest and always-present source of metrics, the layer on which higher-level instrumentation builds (Gregg, 2020).

How much of that behaviour can be recovered depends on how the telemetry is structured (Majors et al., 2022). Each independent dimension of telemetry is a measurement axis, and the more such axes a system records, the more of its state becomes visible. Within a dimension, *cardinality* (the number of distinct values it can take) sets the resolution. A high-cardinality dimension such as a unique request identifier can pinpoint a single transaction among millions, whereas a low-cardinality one such as an HTTP status code only sorts events into a few buckets. Higher cardinality therefore buys finer insight, at the cost of more data to store and index, which makes observability both powerful and expensive.

These pressures are sharpest in cloud and K8s environments. Workloads there are ephemeral, scheduled onto nodes, rescheduled, and destroyed as load changes (Section 3.4), so there is rarely a stable host left to inspect after a problem. The platform itself runs on the same principle, because a K8s control loop acts on the difference be-

tween observed and desired state. The high-cardinality identifiers that cloud-native systems generate as a matter of course, among them pod names, container identifiers, and per-request trace identifiers, are exactly the dimensions that make a fast-changing system reconstructable. For that reason observability has become a first-class concern in cloud-native operations.

Whatever the environment, telemetry takes a small number of complementary forms:

- *Metrics*: numerical measurements sampled at regular intervals, such as requests per second, CPU utilisation, or tail latency. They are cheap to store and well suited to long-term trends, capacity planning, and alerting.
- *Traces*: the end-to-end path of a single request as it crosses services, assembled from *spans*, the individual timed units of work in a parent–child hierarchy that together show where time is spent and where a request fails.
- *Logs*: timestamped, structured records of discrete events, such as an error with its stack trace or a configuration change.
- *Profiles*: an emerging fourth signal that samples where a program spends its CPU and memory. Correlated with the other three, a profile explains why a request was slow, not only where.

No single signal reveals much on its own. Their power comes from correlation, from linking a latency anomaly in a metric to the trace that experienced it, and from that trace to the log entry or profile that explains the cause (Majors et al., 2022).

OTel has become the open, vendor-neutral standard for producing this telemetry across the cloud-native ecosystem.[3] It standardises how telemetry is produced, how it travels, and how its attributes are named.

Telemetry moves through a pipeline (Figure 3.4). A workload is instrumented to emit signals, either engineered into its own code or through libraries that produce them automatically. A Collector commonly receives this telemetry, processes it, and routes each signal onward to the store suited to it. The component that delivers telemetry to a store is an *exporter*. The exporter is where that destination is chosen, so the workload stays independent of any particular store. Some exporters push to a store, while others expose an endpoint that the store reads from, the model Prometheus uses when it scrapes metrics.

Metrics go to a time-series database such as Prometheus[4], traces to a tracing backend such as Tempo[5], logs to a log store such as Loki[6], and profiles to a profiling backend such as Pyroscope[7]. Monitoring and analysis tools such as Grafana[8] query these stores together, so an operator can move from a metric to the trace, log, or profile that explains it.

[3] https://opentelemetry.io
[4] https://prometheus.io
[5] https://grafana.com/oss/tempo/
[6] https://grafana.com/oss/loki/
[7] https://grafana.com/oss/pyroscope/
[8] https://grafana.com/oss/grafana/

Because the exporters and the Collector hold this routing, changing a backend is a configuration change rather than a change to instrumented code.

Figure 3.4: *The telemetry pipeline.*

Metrics, traces, and logs are stable parts of the standard. Its most consequential feature is a set of *semantic conventions*, standardised attribute names for common concepts such as `http.request.method`, `db.system.name`, and `k8s.pod.name`, which let telemetry from different services, languages, and frameworks stay consistent and correlatable across an entire system.

4

Quantum Computing

Chapter 3 traced how classical systems compute, from bits encoded in voltage levels, through the logic gates that transform them, to the circuits and infrastructure that turn those gates into manageable systems. This chapter builds the quantum computer along the same path, starting from the information primitives, developing the circuit model, and then turning to the machine that runs circuits and the toolchain that prepares them.

4.1 Quantum Information

In Section 3.1 the classical bit was defined as a physical system with two stable states. A bit is always exactly 0 or exactly 1, and reading it does not change its value. The qubit is the quantum analogue, and it keeps neither of these guarantees. It can hold a blend of both values at once, and reading it disturbs what it holds.

A qubit is a two-level quantum system. In Dirac notation its state is written as a combination of the two basis states $|0\rangle$ and $|1\rangle$ (LaPierre, 2021):

$$|\psi\rangle = \alpha\ |0\rangle + \beta\ |1\rangle \tag{4.1}$$

The two basis states are represented by column vectors, and a general state by the corresponding pair of coefficients:

$$|0\rangle \rightarrow \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \qquad |1\rangle \rightarrow \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \qquad |\psi\rangle \rightarrow \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \tag{4.2}$$

The coefficients α and β are complex numbers called probability amplitudes. They are not probabilities themselves. The probability of an outcome is the square of the modulus of its amplitude, so a measurement returns 0 with probability $|\alpha|^2$ and 1 with probability $|\beta|^2$. Because one of the two outcomes must occur, the amplitudes satisfy the normalisation condition $|\alpha|^2 + |\beta|^2 = 1$, which makes the state a unit vector in a two-dimensional complex space.

Reading a qubit is therefore unlike reading a bit. Measurement collapses the superposition onto a single basis state, chosen at random with the probabilities the amplitudes prescribe, and the amplitudes themselves are never observed directly. A single measurement yields one bit and discards the rest of the information, so a quantum program returns statistics rather than one definite value. A classical bit, by contrast, can be inspected as often as needed without disturbance. This statistical read model shapes everything above the hardware, because the classical computer never receives a state, only the counts it must interpret.

A qubit with nonzero α and β is in a superposition of $|0\rangle$ and $|1\rangle$. Because the amplitudes are complex, each carries a phase as well as a magnitude, and it is the phase that gives quantum computation its character. Two amplitudes that meet with opposite phase cancel, which is destructive interference, while two that meet in phase reinforce, which is constructive interference. Classical probabilities can only add and never cancel, so nothing of this kind happens in classical computation. The effect is clearest in the two equal superpositions $|+\rangle = (|0\rangle + |1\rangle)/\sqrt{2}$ and $|-\rangle = (|0\rangle - |1\rangle)/\sqrt{2}$, which both give 0 or 1 with equal probability yet behave differently under later gates because their relative phase is opposite. Quantum algorithms exploit exactly this, steering phases so that wrong answers cancel and correct ones reinforce.

Every pure single-qubit state also has a geometric picture. Once the normalisation is applied and the unobservable overall phase is dropped, the state depends on just two angles (LaPierre, 2021):

$$|\psi\rangle = \cos\frac{\theta}{2}\,|0\rangle + e^{i\phi}\sin\frac{\theta}{2}\,|1\rangle \tag{4.3}$$

These angles place the state on the surface of a unit sphere known as the Bloch sphere (Figure 4.1). The poles are the basis states $|0\rangle$ and $|1\rangle$, the equator holds the equal superpositions such as $|+\rangle$ and $|-\rangle$, the polar angle θ fixes the measurement probabilities, and the azimuthal angle ϕ fixes the relative phase. A single-qubit gate acts as a rotation of this sphere, which is why the Bloch picture is the usual way to reason about what a gate does.

A qubit holds its state only for a limited time. Any unwanted interaction with the surrounding environment leaks information out of the qubit and destroys its superposition, a process called decoherence (LaPierre, 2021). Two timescales measure it. The relaxation time T_1 is how long a qubit stays in its excited state before decaying toward $|0\rangle$, and the dephasing time T_2 is how long it keeps a well-defined relative phase. Whichever is shorter sets the window in which a computation must finish, because once coherence is gone the stored state is indistinguishable from noise.

This coherence window varies widely across the physical platforms on which qubits are built. Superconducting transmon qubits, used by IBM and Google, operate near absolute zero and hold coherence for a few hundred microseconds, with gate times of tens of nanoseconds (National Academies of Sciences, Engineering, and Medicine, 2019). Trapped-ion qubits, used by IonQ and Quantinuum, hold coherence for seconds

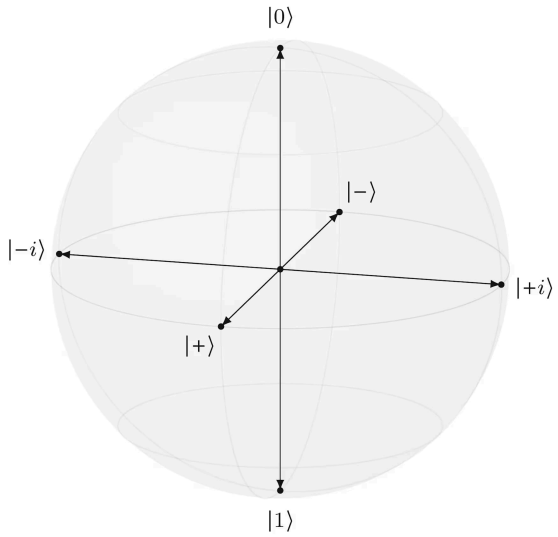


Figure 4.1: *The Bloch sphere. A pure single-qubit state is a point on the surface, fixed by the polar angle θ and the azimuthal angle ϕ.*

and reach higher gate fidelity, at the cost of slower operations. These differences are not incidental to the software above the hardware. Because coherence and error rates differ by platform and drift over time, both the choice of backend and the timing of a run affect the result, a point that recurs throughout the thesis.

A single qubit is limited, and useful computation needs registers of many qubits whose joint state has a structure with no classical parallel. The state of two independent qubits is the tensor product of their individual states (LaPierre, 2021):

$$|\psi_1\rangle \otimes |\psi_2\rangle = \alpha_1\alpha_2 |00\rangle + \alpha_1\beta_2 |01\rangle + \beta_1\alpha_2 |10\rangle + \beta_1\beta_2 |11\rangle \tag{4.4}$$

The two-qubit system lives in a four-dimensional space spanned by $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$. In general a system of n qubits is described by 2^n probability amplitudes:

$$|\psi\rangle = \sum_{x=0}^{2^n-1} \alpha_x |x\rangle , \qquad \sum_{x=0}^{2^n-1} |\alpha_x|^2 = 1 \tag{4.5}$$

The amplitude count doubles with every qubit added, so the state space grows exponentially. A register of three hundred qubits already holds more amplitudes than there are atoms in the observable universe, which is why no classical machine can track it (LaPierre, 2021). Simulating a quantum system on a classical computer is therefore intractable in general, the observation that first motivated quantum computers (Feynman, 1982), and the same exponential growth is the resource that quantum algorithms exploit.

A multi-qubit state is separable when it factors into individual qubit states, so that each qubit has a state of its own. Many states do not factor, and the clearest examples

are the four Bell states, the maximally entangled states of two qubits (LaPierre, 2021):

$$|\Phi^{\pm}\rangle = \frac{1}{\sqrt{2}}\big(|00\rangle \pm |11\rangle\big), \qquad |\Psi^{\pm}\rangle = \frac{1}{\sqrt{2}}\big(|01\rangle \pm |10\rangle\big) \tag{4.6}$$

The Φ pair holds the two qubits in the same value and the Ψ pair holds them in opposite values, with the sign recording the relative phase between the terms. None of the four can be written as a product of two single-qubit states. In $|\Phi^{+}\rangle$, for instance, measuring the first qubit returns 0 or 1 with equal probability, and that outcome immediately fixes the second qubit to the same value, however far apart the two qubits are. States with this property are entangled.

Superposition, interference, and entanglement are the primitives of quantum information. Composing them into circuits and reading out the results is the quantum circuit model.

4.2 The Quantum Circuit Model

Section 3.1 built classical computation from logic gates such as AND, OR, and NOT. Quantum computation has its own gates, with one structural difference. A quantum gate is reversible, which means it has an inverse that undoes it exactly. Most classical gates are not, because an AND gate maps two input bits to a single output and the inputs cannot be recovered from it. As Chapter 2 showed, that erasure carries a thermodynamic cost, and the gates of a quantum circuit avoid it because they discard no information.

A single-qubit gate is an operator on the state vector, and in the computational basis it is a two-by-two matrix. The three Pauli gates and the Hadamard gate are the ones used most often (LaPierre, 2021).

The Pauli-X gate is the quantum NOT. It exchanges the two basis states, and on a superposition it swaps the two amplitudes:

$$X|0\rangle = |1\rangle, \quad X|1\rangle = |0\rangle, \qquad X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{4.7}$$

The Pauli-Z gate leaves $|0\rangle$ unchanged and flips the sign of the $|1\rangle$ amplitude. It changes no measurement probability on its own, yet it flips the relative phase:

$$Z|0\rangle = |0\rangle, \quad Z|1\rangle = -|1\rangle, \qquad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \tag{4.8}$$

The Pauli-Y gate does both at once, a bit flip and a phase flip together:

$$Y|0\rangle = i|1\rangle, \quad Y|1\rangle = -i|0\rangle, \qquad Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \tag{4.9}$$

The Hadamard gate turns a definite state into an equal superposition, sending $|0\rangle$ to $|+\rangle$ and $|1\rangle$ to $|-\rangle$. It is its own inverse, so two Hadamards in a row return the qubit to where it started:

$$H|0\rangle = |+\rangle\,, \quad H|1\rangle = |-\rangle\,, \qquad H = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{4.10}$$

These four gates are fixed operations, but a gate can also turn the qubit by any angle. The rotation gates form a continuous family, one for each axis of the Bloch sphere:

$$R_x(\theta) = \begin{bmatrix} \cos\frac{\theta}{2} & -i\sin\frac{\theta}{2} \\ -i\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix}, \quad R_y(\theta) = \begin{bmatrix} \cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \\ \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix}, \quad R_z(\theta) = \begin{bmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{bmatrix} \tag{4.11}$$

Each rotation turns the Bloch vector about its axis by the angle θ, and any single-qubit gate can be written as a short sequence of such rotations. The angle is a continuous parameter set by the classical program that builds the circuit, and the single-qubit gates a processor provides natively are rotations of this kind, as Section 4.4 describes.

A short single-qubit circuit makes the interference of Section 4.1 concrete. Passing a qubit through H, then Z, then H gives:

$$|0\rangle \xrightarrow{H} \frac{1}{\sqrt{2}}\big(|0\rangle + |1\rangle\big) \xrightarrow{Z} \frac{1}{\sqrt{2}}\big(|0\rangle - |1\rangle\big) \xrightarrow{H} |1\rangle \tag{4.12}$$

The first Hadamard splits the qubit into two paths. The Z gate flips the relative phase between them without changing any probability. The second Hadamard recombines the paths, and the $|0\rangle$ amplitudes cancel while the $|1\rangle$ amplitudes reinforce, leaving $|1\rangle$ with certainty. Without the Z gate the two Hadamards cancel and the qubit returns to $|0\rangle$, so a phase that was invisible to measurement decided the outcome. This is the pattern quantum algorithms exploit. Superposition creates paths, gates write information into relative phases, and a final interference step turns those phases into a measurable answer.

A circuit acts not on one qubit but on a register. A quantum register is a row of n qubits, labelled q_0 through q_{n-1}, all initialised to $|0\rangle$. Gates act on one or more of these qubits in sequence, and every qubit is measured at the end. Two numbers describe the circuit. Its width is the number of qubits, bounded by what the processor provides. Its depth is the number of gate layers applied in sequence, bounded by the coherence time, because each added layer gives decoherence more chance to corrupt the state.

A single-qubit gate must be placed within the register before it can act, and the same tensor product that combined qubit states in Section 4.1 combines gate matrices here. Applying H to the first qubit of a two-qubit register while leaving the second

alone is the operator $H \otimes I$, a four-by-four matrix:

$$H \otimes I = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \tag{4.13}$$

When two gates act on different qubits in the same step, the combined operation is their tensor product, and a qubit that no gate touches receives the identity.

Some gates act on two qubits at once and cannot be split into a tensor product of single-qubit gates. The most important follows the controlled pattern, in which one qubit, the control, decides whether an operation applies to another, the target. The controlled-NOT gate, written CNOT, applies an X to the target when the control is $|1\rangle$ and does nothing when the control is $|0\rangle$ (LaPierre, 2021):

$$\mathrm{CNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{4.14}$$

On definite inputs it computes the exclusive-or, $\mathrm{CNOT}\,|c, t\rangle = |c,\ c \oplus t\rangle$, so the target becomes the parity of the two bits while the control is left unchanged (Table 4.1).

Table 4.1: *The CNOT gate on the computational basis. The control is unchanged; the target is flipped when the control is* $|1\rangle$.

Input $\lvert c\,t\rangle$	**Output** $\lvert c\,t'\rangle$
$\lvert 00\rangle$	$\lvert 00\rangle$
$\lvert 01\rangle$	$\lvert 01\rangle$
$\lvert 10\rangle$	$\lvert 11\rangle$
$\lvert 11\rangle$	$\lvert 10\rangle$

A related two-qubit gate, the SWAP, exchanges the states of two qubits and decomposes into three CNOTs. This decomposition matters for hardware, because a processor can apply a two-qubit gate only to qubits that are physically connected.

The CNOT is what makes entanglement possible, because on a control already in superposition it correlates the two qubits. The smallest example is the two-qubit Bell circuit, a Hadamard on the first qubit followed by a CNOT across both. As a composition of operators on the register it is:

$$|\Phi^{+}\rangle = \mathrm{CNOT}\,(H \otimes I)\,|00\rangle \tag{4.15}$$

The operators apply right to left, so the Hadamard layer acts first. Acting on the register

written as a column vector, the $H \otimes I$ matrix gives:

$$(H \otimes I)\,|00\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2}} \big(|00\rangle + |10\rangle \big) \tag{4.16}$$

The CNOT then acts on this superposition. It leaves the $|00\rangle$ term untouched and flips the target of the $|10\rangle$ term to give $|11\rangle$:

$$\text{CNOT}\, \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} = |\Phi^+\rangle \tag{4.17}$$

The output is the Bell state $|\Phi^+\rangle$ of Section 4.1. Measuring the register gives `00` or `11` with equal probability and never `01` or `10`, and those forbidden outcomes are the signature of entanglement.

A small set of gates is enough to build any circuit. Arbitrary single-qubit rotations together with CNOT form a universal set, in the same sense that the NAND gate is universal for classical logic (LaPierre, 2021). A real processor supports only the native gates fixed in its hardware, so a circuit written in abstract gates must be translated to that native set before it can run. That translation is where hardware constraints first enter the software stack.

Gates combine in two ways. Gates in the same time step combine by the tensor product, and gates applied one after another combine by matrix multiplication, with the first gate written closest to the state. A circuit of any size is therefore a single unitary applied to the initial register:

$$|\psi_{\text{out}}\rangle = U_k \cdots U_2\, U_1\, |0\rangle^{\otimes n} \tag{4.18}$$

Two structural rules follow. A unitary is invertible, so every gate can be undone and a circuit has as many output wires as input wires; no gate can discard a qubit. The no-cloning theorem forbids copying an unknown qubit (LaPierre, 2021), so the wires never fork.

Many quantum algorithms are organised around an oracle, a subroutine that encodes a problem-specific function f as a unitary without prescribing how it is built. The standard construction keeps the operation reversible by writing the result into a separate register rather than overwriting the input:

$$U_f\ |x\rangle\,|y\rangle = |x\rangle\,|y \oplus f(x)\rangle \tag{4.19}$$

Here the first register holds the input $|x\rangle$ and the second the target $|y\rangle$. The oracle writes

into the target, yet an algorithm must read its answer from the input register. Phase kickback connects the two. Preparing the target in $|-\rangle$ rather than $|0\rangle$ turns the recorded bit into a phase, because applying the oracle to $|x\rangle|-\rangle$ leaves the target unchanged and attaches a sign to the input that depends on $f(x)$:

$$U_f\ |x\rangle|-\rangle = (-1)^{f(x)}\ |x\rangle|-\rangle \tag{4.20}$$

When $f(x) = 0$ the target is untouched, and when $f(x) = 1$ its two amplitudes swap and the state picks up an overall minus sign, so the factor $(-1)^{f(x)}$ lands on $|x\rangle$. A single application of the oracle therefore evaluates f on all 2^n inputs at once, recording each value in a phase, the effect Deutsch named quantum parallelism (Deutsch, 1985):

$$U_f\ \frac{1}{\sqrt{2^n}} \sum_x |x\rangle|-\rangle = \frac{1}{\sqrt{2^n}} \sum_x (-1)^{f(x)}\,|x\rangle|-\rangle \tag{4.21}$$

This parallel evaluation cannot be read out wholesale. A measurement returns one $|x\rangle$ at random, so the 2^n values are never all recovered; the phases become useful only when interference concentrates them into a global property of f. A final layer of Hadamard gates does exactly that, just as the H, Z, H sequence of Equation 4.12 turned an invisible phase into a definite outcome.

Deutsch's algorithm is the smallest circuit that uses this (Deutsch, 1985). A one-bit function f is constant when $f(0) = f(1)$ and balanced when $f(0) \neq f(1)$; deciding which classically needs both values, and so two queries, while one quantum query suffices. The input qubit starts in $|0\rangle$ and the target in $|1\rangle$, a Hadamard on each prepares $|+\rangle|-\rangle$, the oracle is applied once, and a final Hadamard returns to the input qubit before it is measured (Figure 4.2):

$$|\psi_{\text{out}}\rangle = (H \otimes I)\, U_f\, (H \otimes H)\ |0\rangle|1\rangle \tag{4.22}$$

Kickback places $(-1)^{f(0)}$ on $|0\rangle$ and $(-1)^{f(1)}$ on $|1\rangle$ in the input register, leaving a relative phase of $(-1)^{f(0)\oplus f(1)}$ between them. The final Hadamard sends the input qubit to $|0\rangle$ when that phase is positive and to $|1\rangle$ when it is negative, so the measured bit equals $f(0) \oplus f(1)$, reading `0` for a constant function and `1` for a balanced one. One measurement decides a global property of f that no single classical evaluation can reach. Shor's algorithm extends the same idea, hiding the period of a modular-exponentiation function in the oracle and using interference to expose it. The oracle is still an ordinary unitary and obeys the same wire rules as any other gate, and the transpiler later synthesises it into the processor's native gate set.

A circuit that ends in superposition or entanglement gives a probabilistic result, so it is run many times, and each run is called a shot. The outcomes across all shots form a histogram of bit-string counts (Javadi-Abhari et al., 2024). A classical computer submits the circuit, collects the histogram, and interprets the distribution, and some algorithms feed that result into a new circuit and run again. The submit, measure, and

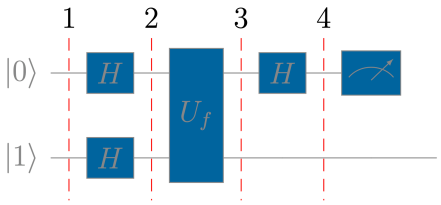


Figure 4.2: *Deutsch's circuit.*

update steps form a loop between a classical computer and a quantum processor.

The circuit model defines what a quantum computer must do, namely initialise a register, apply a gate sequence, and measure the result. The machine that carries out those steps is a physical system unlike any classical processor.

4.3 The Quantum Computer

A quantum processor is not a standalone computer but an accelerator driven by a classical host, in much the same way a GPU is (Shehata et al., 2025; National Academies of Sciences, Engineering, and Medicine, 2019). The host prepares the circuit, submits it, and reads the results back, while the processor runs only the quantum step. Every quantum computation is therefore hybrid by construction, with a classical computer deciding what to compute and interpreting what comes back.

Quantum processors are built on several physical technologies. Superconducting circuits, trapped ions, photons, and neutral atoms each realise a qubit in a different way, with their own control mechanisms and operating conditions, so there is no single physical machine. What they share is a functional architecture, and it is this shared structure, rather than any one technology, that the software above has to address. A quantum computer has no operating system of its own to schedule work or manage resources. Its structure is described instead by four functional planes, set out in the National Academies report and independent of the underlying qubit technology (National Academies of Sciences, Engineering, and Medicine, 2019).

The first is the quantum data plane, the qubits together with the couplings that let them interact. It is where gates act and where measurement reads out a result, and it is unlike any classical data plane because it holds no persistent state. The no-cloning theorem forbids copying a qubit, idle qubits decohere, and measurement destroys what it reads, so a computation must finish inside the coherence window or be lost.

The second and third planes drive the first. The control and measurement plane is the analogue layer that turns abstract gates into physical action. It generates the signals that implement each gate, delivers them to the qubits, and amplifies the faint return signals into a readable 0 or 1. Because qubit properties drift, this mapping is not fixed and has to be recalibrated regularly, often daily (Javadi-Abhari et al., 2024). Above it, the control processor plane is a classical real-time controller that issues the precisely timed sequence of operations the control and measurement plane carries out.

The fourth plane is the host processor, an ordinary classical computer. It runs the software stack, the SDK, the transpiler, the scheduler, and the result handling, and it holds all of the system's persistent data because no other plane can. Only the host is a general-purpose programmable machine, and only the host keeps anything that outlives a single circuit. Taken together, the first two planes are the quantum hardware, the qubits and the analogue electronics that drive and read them, commonly called the quantum processing unit (QPU); the control processor and host are conventional classical computers. The line between them, where digital control meets the analogue quantum device, is the classical–quantum interface this thesis addresses (Döbler et al., 2025). Table 4.2 summarises the four planes and the property that sets each apart.

Table 4.2: *The four functional planes of a quantum computing system, as set out in the National Academies report.*

Plane	Role and key property
Quantum data	Holds the qubits and runs the gates. Keeps no persistent state and is bounded by the coherence time.
Control & measurement	Generates control signals, delivers them, and reads the qubits out. Drifts over time, so needs recalibration.
Control processor	Sequences operations in real time, issuing the timed instructions the hardware carries out.
Host processor	Runs the SDK, transpiler, scheduler, and result handling. Holds all persistent state.

A superconducting machine, the kind this thesis uses, makes the planes concrete. The chip is about the size of a thumbnail and sits at the bottom of a dilution refrigerator that cools it to roughly 15 millikelvin, colder than interstellar space. Heavily attenuated cryogenic wiring carries the control signals down without swamping the qubits in thermal noise (Krantz et al., 2019). Wiring, filtering, amplifiers, and shielding fill most of the volume, so most of the machine is classical hardware wrapped around a few fragile qubits. The planes follow the temperature, with the host and control processor at room temperature, the control and measurement plane spanning the cryogenic wiring, and the quantum data plane at the coldest stage.

A single gate shows the planes working together. It starts as a line of code on the host, which compiles and schedules it and passes it to the control processor for timing. The control electronics generate the corresponding signal and send it to the chip, the qubits interact, and the readout signal travels back up the same path to become a classical bit the host records. This round trip, from a method call to a physical interaction and back to a bit, is the classical–quantum interface, and every stage along it adds latency and a chance of error.

For all this resemblance to a classical computer, one difference has no precedent. The quantum data plane is stateless, with no cache, no memory, no disk, and nothing that survives measurement. A classical processor can run its own operating system and keep track of its own state, but a quantum computer cannot. It cannot schedule its

own work, manage its own calibration, or keep any record of what it did. Every such function lives on the classical side and reaches the processor only across the interface. A quantum computer in this sense is never used alone, but always as one half of a classical–quantum pair.

A typical processor of today, such as IBM's Heron generation, holds more than a hundred transmon qubits, with coherence times of a few hundred microseconds and gates that remain far from perfect. This is the noisy intermediate-scale quantum (NISQ) regime (Preskill, 2018). Each processor also has its own native gate set and its own qubit connectivity, so a circuit cannot simply be written and run (Javadi-Abhari et al., 2024). It must first be compiled for the specific machine and the calibration that machine currently reports.

4.4 Compilation and Runtime

A quantum processor implements only a small set of native gates and couples only qubits that are physically adjacent, so a circuit expressed in a high-level SDK cannot run as written. Turning that abstract circuit into instructions a specific device can execute follows a compilation pipeline, and that pipeline parallels the classical software stack of Chapter 3.

Seen end to end, the toolchain carries a problem to a solution in four phases (Figure 4.3). A problem is first mapped to a quantum circuit; the circuit is transpiled into a form the target device can execute; the device executes it; and the raw measurements are post-processed into a result. The first two phases are compilation and the last two are runtime, the two halves this section examines in turn.

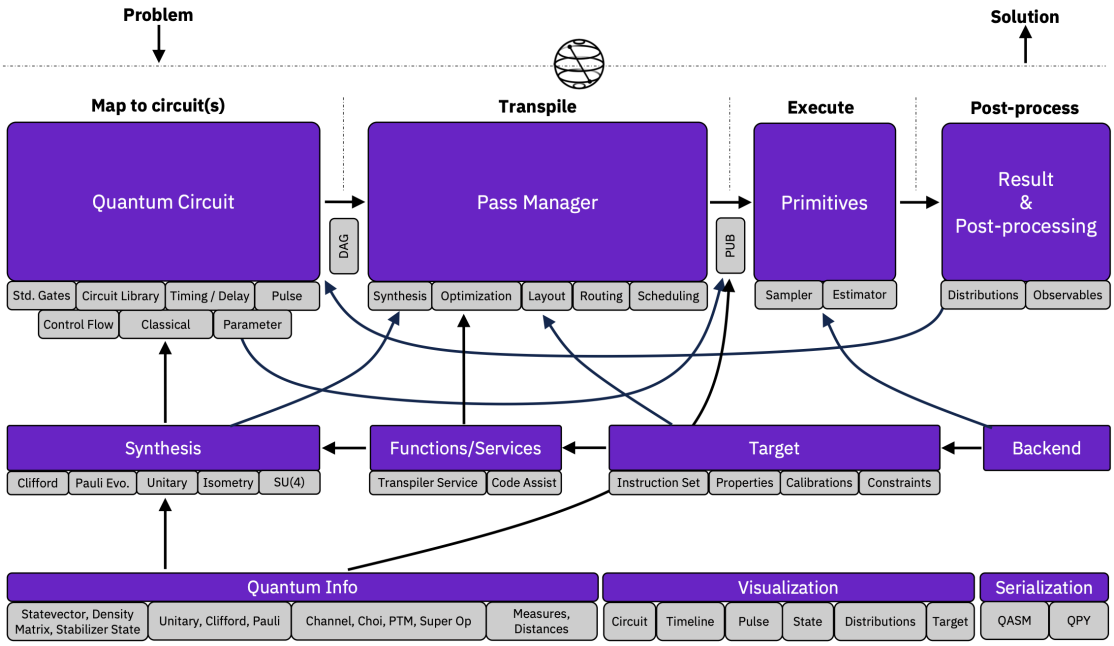


Figure 4.3: *The Qiskit software architecture. Reproduced from Javadi-Abhari et al. (2024) Fig. 2.*

In classical computing, source code is lowered through successive layers, from a high-level language down through an intermediate representation and a compiler backend to the instruction set architecture and finally to hardware control signals. The quantum stack follows the same pattern, with a direct counterpart at each layer (Javadi-Abhari et al., 2024):

1. *Quantum SDK* (Qiskit, Cirq, PennyLane) is the user-facing programming framework, analogous to a high-level language. The developer builds circuits from abstract gates (H, CNOT, $R_z(\theta)$) with no reference to any specific processor.
2. *Quantum Intermediate Representation* is a hardware-agnostic description of the circuit, analogous to LLVM IR or JVM bytecode, a common middle format that many languages compile to and many machines run from. The most widely adopted form is OpenQASM 3 (Cross et al., 2022), which adds classical control flow and explicit timing to a plain gate list, with Microsoft's QIR (QIR Alliance, 2021) and Rigetti's Quil (Smith et al., 2016) as alternatives.
3. *Transpiler* is the compiler backend. It rewrites the abstract circuit into a form a chosen processor can execute, through the six stages detailed below, and is where hardware constraints first enter the pipeline.
4. *Instruction Set Architecture* (ISA) is the contract between software and hardware, fixing which operations the processor natively supports. IBM calls a circuit that already conforms to a backend's native gate set, connectivity, and timing an ISA circuit. The Qiskit `Target` object carries that specification, the basis gates such as $\{CZ, R_z, \sqrt{X}, X\}$, the coupling map, and the per-gate error rates and durations (Javadi-Abhari et al., 2024).
5. *Pulse-Level Control* translates each native instruction into the physical pulses that drive the qubits. It is analogous to microcode, the low-level steps a processor runs for each machine instruction (McKay et al., 2018).
6. *Quantum Processing Unit* executes the native operations on the physical qubits, the role the CPU plays in the classical stack.

Table 4.3 sets the two stacks side by side.

The parallel is a deliberate engineering pattern rather than a loose analogy. Both stacks face the same problem, bridging a high-level hardware-agnostic programming model and the constraints of a specific physical device. The quantum stack differs in one respect that matters later. Its abstraction boundaries leak, exposing physical details such as gate error rates, connectivity, and coherence times to the higher layers, so the layers are co-designed rather than cleanly isolated (Bandic et al., 2022; Shi et al., 2020).

The transpiler, the third layer above, is itself a pipeline. It accepts a circuit expressed in abstract gates and produces an equivalent ISA circuit that uses only the target processor's native gates and respects its connectivity. Qiskit's `StagedPassManager` runs it as six sequential stages, each the quantum counterpart of a classical compiler pass (Javadi-

Table 4.3: *The quantum compilation pipeline mirrors its classical counterpart at every layer of abstraction.*

Layer	Classical	Quantum
Source / SDK	High-level language (C, Python, Go)	Quantum SDK (Qiskit, Cirq)
Intermediate repr.	LLVM IR, bytecode	Quantum IR (OpenQASM 3, QIR, Quil)
Compiler backend	Code generation	Transpiler (decomposition, routing)
ISA	x86, ARM, RISC-V	Native gate set and connectivity
Control signals	Microcode	Pulse sequences
Hardware	CPU	QPU

Abhari et al., 2024):

1. *Initialisation* unrolls high-level constructs and lowers multi-qubit operations into one- and two-qubit primitives, putting the circuit into the standard form the later stages expect. It is analogous to *macro expansion* and inlining.
2. *Layout* maps the logical qubits of the circuit onto physical qubits on the processor, weighing qubit quality and connectivity. It is analogous to *register allocation*, the step that assigns a program's values to the CPU's small set of registers. The placement matters for fidelity, because a low-error region of the chip yields measurably different results from a high-error one.
3. *Routing* inserts SWAP gates wherever a two-qubit operation falls on non-adjacent qubits. It is analogous to *spill code insertion*, the extra moves to and from memory a compiler makes when it runs out of registers, though the no-cloning theorem means a SWAP must physically exchange two qubit states rather than copy data. Each SWAP decomposes into three entangling gates, which makes routing the dominant fidelity cost of compilation, and finding the optimal routing is NP-hard (Siraichi et al., 2018).
4. *Translation* rewrites every gate into the backend's native gate set through an equivalence library. It is analogous to *instruction selection*, choosing the machine instructions that implement each operation. On a device with native gate set $\{CZ, R_z, \sqrt{X}, X\}$ a Hadamard becomes $R_z(\frac{\pi}{2})\,\sqrt{X}\,R_z(\frac{\pi}{2})$, and because R_z is a virtual gate with no physical duration (McKay et al., 2017), only the single $\sqrt{X}$ carries a cost. A CNOT becomes a CZ flanked by single-qubit rotations.
5. *Optimisation* reduces depth and gate count through gate cancellation, commutation-based reordering, and resynthesis of two-qubit blocks. It is analogous to *peephole optimisation*, small local rewrites that simplify a few neighbouring instructions at a time. Qiskit exposes four levels, from level 0, which applies only what correct-

ness requires, to level 3, which can substantially reduce the entangling-gate count at the cost of longer compilation.

6. *Scheduling* assigns a start time to each operation and inserts delays for idle qubits. It is analogous to *instruction scheduling*, though the aim is to limit decoherence rather than to maximise pipeline use. When explicit timing is needed, Qiskit offers two strategies, as soon as possible (ASAP) and as late as possible (ALAP), the latter keeping qubits in the ground state where possible. Dynamical-decoupling pulses can fill idle windows to suppress decoherence.

Once transpiled, the circuit runs on the backend through one of Qiskit's execution primitives. The sampler returns the distribution of measured bit strings; the estimator returns expectation values, the average value of a measured quantity (Javadi-Abhari et al., 2024). Because the circuit runs for many shots, each returns a statistic rather than a single value, as Section 4.2 described. Some algorithms are iterative, with a classical optimiser reading the result and submitting a new circuit, so compilation and execution repeat in a loop between the host and the processor.

A classical ISA is fixed when the chip is manufactured, but a quantum processor's is not. The `Target` carries calibration data, the gate error rates and qubit frequencies, and this data drifts over time, so the device is recalibrated regularly to keep it current (Shi et al., 2020). A circuit transpiled against one day's calibration is tuned to numbers that may not hold the next, so in practice the instruction set is temporally dynamic, changing from one day to the next. The best result comes from transpiling against the calibration the device reports at the moment it runs.

5 Literature Review

The previous chapters established the theoretical framework of computation and introduced the foundations of classical and quantum computing, while beginning to examine the intersection between these paradigms. This chapter surveys the broader literature on quantum–classical interfacing through four software-interface layers: applications, programming frameworks, compilation and runtime, and infrastructure and orchestration. Observability is treated as a cross-cutting concern, spanning all layers.

5.1 Review Methodology

The layer model used in this chapter is a thesis-specific synthesis rather than a taxonomy. It is derived by aligning the QPU plane model discussed in Section 4.3, the QCSC reference architecture of Seelam et al. (2026), the HPC–QC interfacing survey of Rallis et al. (2025), and the software-interface concerns identified across the surveyed literature. Each layer section opens with the problem that the literature at that layer attempts to address, synthesises across sources to evaluate the proposed approaches, and closes by identifying the residual gaps that inform the requirements in Chapter 6. The consolidated gap analysis in Section 5.8 builds on those findings rather than introducing new ones.

Literature was drawn from IEEE Xplore, the ACM Digital Library, arXiv, Scopus, and Google Scholar, supplemented by venue-specific searches on *Nature*, *Physical Review*, *npj Quantum Information*, and the *Proceedings of the IEEE*. Searches combined terms related to quantum–classical interfacing, hybrid quantum workflows, quantum cloud execution, orchestration, observability, K8s, OpenTelemetry, and variational workloads.

The temporal scope extends from 2016 to 2026. The lower bound is anchored at the emergence of publicly accessible gate-model quantum cloud execution, marked by the launch of the IBM Quantum Experience in May 2016. The upper bound is the publication of the QCSC reference architecture in early 2026, which frames the present thesis. Foundational work predating 2016 is included only where it remains the canonical reference.

Authoritative standards and reference documentation are cited where they are the primary description of an interface or operational model. This applies to OpenTelemetry semantic conventions, the Kubernetes API specification, Qiskit Runtime documentation, the CNCF definition of cloud-native (Cloud Native Computing Foundation, 2024), and the Site Reliability Engineering book (Beyer et al., 2016). For these technologies, standards and reference documentation provide the most direct source for terminology, interfaces, and operational semantics.

The surveyed literature is weighted toward the IBM quantum ecosystem for three reasons. First, several systems-level references central to this thesis originate from the IBM Quantum ecosystem, including Javadi-Abhari et al. (2024), Bandic et al. (2022), Kanazawa et al. (2025), and Seelam et al. (2026). Second, Qiskit and OpenQASM 3 provide mature, open-source interfaces that expose enough compilation and runtime detail to support the analysis carried out in this thesis. Third, IBM Quantum was the cloud platform available during the MSc coursework and was therefore the natural execution target for the proof-of-concept implementation developed in Chapter 7. Work from Google, Rigetti, IonQ, Quantinuum, NVIDIA, PASQAL, and other ecosystems is cited where it provides distinctive perspectives on compilation, accelerator integration, orchestration, or cross-platform portability.

5.2 Related Surveys and Positioning

Nguyen et al. (2024) survey quantum cloud computing, defining the service-model taxonomy (QCaaS, QAaaS, Quantum Serverless, Hybrid, and Distributed) that the infrastructure-layer section below adopts as its organising vocabulary. Their survey catalogues open problems in quantum cloud platforms, none of which is observability. Elsharkawy et al. (2025) survey the integration of quantum accelerators with HPC systems, focusing on programming models and compilation toolchains. Caleffi et al. (2024) survey distributed quantum computing, with attention concentrated on quantum-network protocols. Finally, Döbler et al. (2025) provide the most recent structured synthesis of HPC-quantum integration, mapping 107 publications across the period 2016–2025.

5.3 Applications and Hybrid Execution

A workload's operational demands first appear at the application layer, mapping the problem for quantum computation. A classical program runs as a process under an operating system that schedules it, isolates it, and accounts for it. A quantum circuit, today, has no such process environment. Instead, it is submitted to a QPU as a job from a classical computer. Building the circuit, submitting it, reading out the measurements, and interpreting the results all fall to that classical system to arrange. Every quantum workload is therefore hybrid, and the classical–quantum interface belongs to all of them, not to one style of algorithm.

The NISQ-era literature is dominated by the variational quantum algorithm, introduced by Peruzzo et al. (2014) as the variational quantum eigensolver (VQE) for chemistry and extended to combinatorial problems as the Quantum Approximate Optimisation Algorithm (QAOA) (Blekos et al., 2024). Both run a short parameterised circuit inside a loop with a classical optimiser. That loop makes the classical–quantum coupling easy to see, but it is not the source of it: the same coupling surrounds a single deep circuit such as Shor's algorithm, where one quantum step sits between classical pre- and post-processing. Variational work is the most visible case of a structure every workload shares.

Not every workload justifies that machinery. Hoefler et al. (2023) shows that quadratic speedups, such as Grover-style search and many quantum machine-learning proposals, cannot overcome the constant-factor cost of quantum execution, and so deliver no practical advantage. The workloads that remain worthwhile are those with superquadratic or exponential speedups: quantum chemistry, materials science, and factoring via Shor's algorithm. These are also the deepest and most demanding circuits to run.

For exactly those workloads, a usable result is not a property of the algorithm but of the whole stack on the day it runs. The utility demonstration of Y. Kim et al. (2023), which estimated expectation values on IBM's 127-qubit Eagle processor beyond the reach of exact classical simulation, succeeded only because transpilation, calibration, and error mitigation were correct together; had any been wrong, the result might have looked plausible but been false, with nothing in the live output to flag it. The hardware also changes between runs, so the same program submitted twice will not return the same result. To tell whether a different outcome came from the code or from the machine, each run must be recorded together with the calibration, queue, and transpilation state it ran under.

The application literature establishes all of this, but does little to manage it systematically. Classical software achieves reproducibility through hermeticity, pinning every input and isolating the environment so that a build behaves the same anywhere (Beyer et al., 2016). Only part of that transfers here. Qiskit's `seed_transpiler` can pin the software side, fixing the otherwise-random layout and routing so that a circuit compiles the same way each time; the hardware cannot be pinned, because calibration and noise are uncontrolled, time-varying inputs. Reproducibility therefore has to be recovered by recording the backend state behind each run rather than by controlling it. In the application-layer literature this recording is done by hand, one experiment at a time, with no repeatable way to schedule runs, observe them, or correlate them with the backend that produced them.

5.4 Programming Frameworks and SDKs

A quantum workload is expressed through a software development kit, and the SDK layer is where vendor coupling concentrates. What a framework exposes at the classical–

quantum boundary bounds what the orchestration and observability layers above it can do, so the choice of SDK is also a choice about how much of the execution the rest of the stack can reach. The major frameworks differ along two axes: the abstraction each presents to the developer, and how far each reaches across more than one vendor's hardware.

Qiskit, IBM's open-source Python framework, is the most widely used in the open-source quantum ecosystem. A program is built as a circuit and then run through execution primitives that hide the mechanics of submitting a job and collecting results. Sampler returns measurement distributions, and Estimator returns expectation values. Qiskit is integrated most tightly with IBM Quantum, but it is not confined to IBM hardware: community provider packages route circuits to Amazon Braket,[1] IonQ,[2] and Rigetti[3] backends through the same programming model. Those providers widen the frontend, not the backend. Native gate sets, calibration data, runtime primitives, and error-mitigation features remain provider- and device-specific. Qiskit therefore exposes a rich view of execution, but a vendor-specific one.

Cirq (Cirq Developers, 2018; A. Ho et al., 2018), from Google Quantum AI, is a Python framework built around explicit, low-level circuit construction, with fine-grained control over gate scheduling and timing suited to research on Google's superconducting hardware. Unlike Qiskit, it keeps circuit construction and execution separate and embeds no execution primitives of its own. TensorFlow Quantum (Broughton et al., 2020) builds on Cirq to add differentiable-programming primitives integrated with the TensorFlow machine-learning stack: a quantum circuit becomes a layer in a model, and gradients flow through both the classical and quantum parts, so hybrid models can be trained end to end with familiar tooling. The pairing is vendor-native, oriented to Google's hardware and ecosystem.

PennyLane (Bergholm et al., 2018), from Xanadu, is the most explicitly hardware-agnostic of the major frameworks. Its model is differentiable quantum programming: a quantum function is differentiable like any other, and gradients pass into and out of quantum subroutines, so it interoperates with automatic-differentiation libraries such as PyTorch, TensorFlow, and JAX. Its portability comes from a plugin architecture, in which the same circuit description runs on Qiskit-reachable QPUs, Cirq-reachable QPUs, Xanadu's photonic hardware, and simulators through one uniform interface. PennyLane occupies the vendor-neutral pole of the landscape and shows that frontend portability is achievable when a framework is designed for it from the start.

CUDA Quantum (J.-S. Kim et al., 2023), since renamed CUDA-Q, is NVIDIA's quantum–classical platform, and it sits differently on the vendor axis than the two frameworks above. It is QPU-agnostic: one kernel runs across IonQ, Quantinuum, IQM, OQC, and Amazon Braket by selecting a target at compile time, so it is bound to no single quantum vendor. Its coupling is on the classical side instead. It treats

[1] https://github.com/qiskit-community/qiskit-braket-provider
[2] https://github.com/qiskit-community/qiskit-ionq
[3] https://github.com/rigetti/qiskit-rigetti

the QPU as one accelerator alongside the GPU and expresses quantum work as kernels, procedures whose body runs on the QPU, marked `@cudaq.kernel` and callable directly from classical accelerator code.

T|ket⟩ (Sivarajah et al., 2020), from Quantinuum, takes the compiler route to neutrality. It is a language-agnostic optimising compiler built around an intermediate representation, a directed acyclic graph with structural annotations, transformed by a sequence of optimisation passes, rather than an execution model of its own. Its modular frontend and backend architecture emits executable circuits for IBM, Google, Rigetti, Quantinuum, AQT, and other targets from a single representation, so one compiled description can be retargeted across heterogeneous hardware. It is the clearest demonstration that retargetable compilation across vendors is an engineering reality rather than an aspiration.

OpenQASM 3 (Cross et al., 2022) is not an SDK but the closest the field has to a vendor-neutral interchange language. It is a text-based quantum assembly. A circuit's qubits, gates, and measurements are written down independently of the framework that produced them, and version 3 adds classical control flow, parameter expressions, and timing. Its place in the stack is the software interchange layer, between the frontends that build circuits and the toolchains that compile them, so a program exported by one tool can be read by another. That reach is broad: version 2 was supported across Cirq, PennyLane, and XACC among others, and version 3 is now implemented by Qiskit, Amazon Braket, and Zurich Instruments' LabOne Q[4], which makes it the de facto way to move a circuit between tools. What it standardises, though, is that software representation, not the hardware beneath it: a backend must still support the required instruction set and timing semantics, and the mapping to a vendor's native gates is not itself standardised. OpenQASM is therefore a software-layer agreement, not a hardware one. It is the field's proof that a vendor-neutral standard can be reached at the representation layer, and a reminder that reaching it there leaves the execution layer untouched.

Table 5.1: *Major quantum SDKs, by abstraction and multi-vendor posture.*

SDK	Developer	Abstraction	Multi-vendor posture
Qiskit	IBM	Circuit + Primitives	IBM-native, provider-extensible
Cirq + TFQ	Google	Circuit / Differentiable	Vendor-native (Google)
CUDA-Q	NVIDIA	Kernel-based	QPU-agnostic, GPU-coupled
PennyLane	Xanadu	Differentiable	Plugin-based
t|ket⟩	Quantinuum	Compiler IR	Compiler-frontend

Taken together, these frameworks show that portability is genuinely achievable, not aspirational, at the SDK level (PennyLane, CUDA-Q), the compiler level (t|ket⟩), and the interchange level (OpenQASM). Nation et al. (2024), an IBM-authored open-

[4] https://github.com/openqasm/openqasm/blob/main/implementations.md

source benchmark covering seven SDKs at their default settings across more than 1,000 tests, found Qiskit producing 24 % fewer two-qubit gates than t|ket⟩ and transpiling 13× faster on geometric mean. The point is not which compiler leads but that retargeting across heterogeneous backends works. Error-mitigation primitives, pulse-level access (McKay et al., 2018), and calibration-aware transpilation differ per device and are either hidden behind the common abstraction or reached through backend-specific extensions. And whatever the frontend, execution still delegates to a vendor-specific compiler and runtime, so the portable layer is always thinner than the native interface beneath it, and can offer no more neutrality than the runtime it delegates to.

This is the layer's defining limitation. It concentrates vendor coupling more than any layer above the hardware, and the neutrality it does achieve does not propagate upward: because every portable frontend delegates down to vendor-bound execution, the orchestration and observability layers cannot inherit vendor-neutrality from the SDK. Therefore, they have to build it themselves, on standards of their own. Classical systems reached neutrality the same way, through standards that sit outside any single framework, such as POSIX, container runtimes, and OTel, rather than from the frameworks themselves.

5.5 Compilation and Runtime Toolchains

Compilation turns an abstract circuit into one specific machine's program: logical qubits are placed onto physical ones, two-qubit gates are routed onto the device's connectivity, and the circuit is rewritten into the backend's native gate set. A computation only happens when the circuit's abstract dynamics match the physical dynamics of the device; where the two diverge, the execution returns noise rather than a result (Section 2.3).

Every choice a transpiler makes turns on the device's current error profile: some qubits and couplings carry far lower error than others, and good compilation steers the circuit onto the better ones. Qiskit's transpiler (pipeline introduced in Section 4.3) is built like a classical optimising compiler, in which a pass manager runs an ordered sequence of passes over an in-memory representation, the `DAGCircuit`. Several of those passes are calibration-aware: they read the device's measured gate and readout errors and place the circuit accordingly (Javadi-Abhari et al., 2024).

Routing is the difficult stage, since the qubit mapping problem has been proved to be NP-Complete (Siraichi et al., 2018). Thus, transpilers fall back on heuristics and the open-source baseline is the SWAP-based bidirectional heuristic search algorithm (SABRE), which fixes an initial layout and inserts SWAP gates by alternating forward and reverse passes over the circuit (Li et al., 2019). SABRE is stochastic, since its output depends on a random seed and it is reproducible only when that seed is fixed.

So a transpiled Qiskit circuit is compiled against real, device-specific calibration data (Javadi-Abhari et al., 2024). The catch is that the data is the last *published* calibration snapshot, not a measurement taken now. The numbers are real; they are not

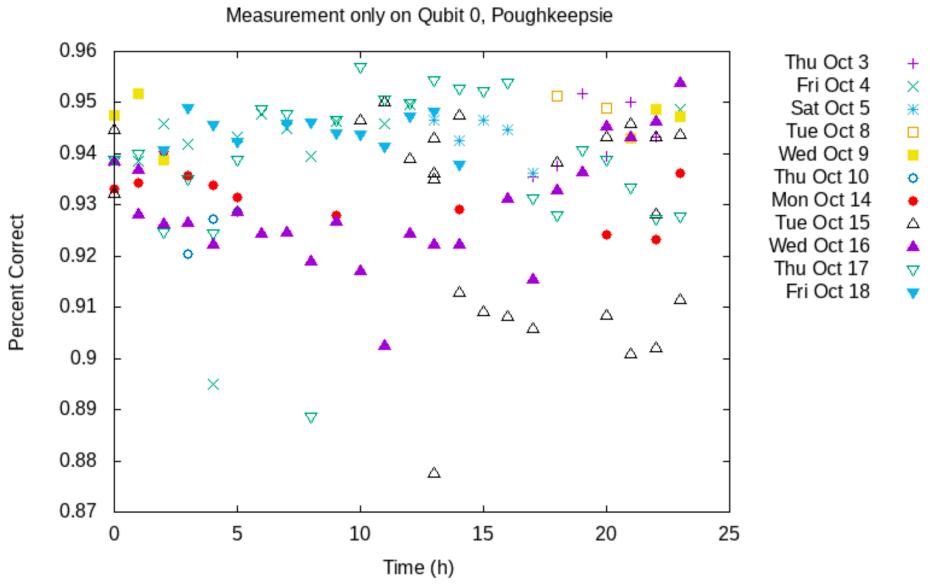


Figure 5.1: *Non-predictable qubit fidelity between IBM's daily calibrations. Reproduced from Wilson et al. (2020) Fig. 1.*

current. Calibration-awareness here means snapshot-awareness.

That distinction would not matter if the device held still between calibrations. It does not. Wilson et al. (2020) measured qubit fidelity hourly on IBM Poughkeepsie for two weeks and found that it varies chaotically and unpredictably between IBM's once-daily calibrations (Figure 5.1): the best-quality qubit layout changes at least hourly, sometimes within minutes. The drift is unreliable rather than a predictable trend, so the snapshot cannot track the live state, and it can only be re-measured. And its shelf life is often shorter than the queue wait (Ravi et al., 2021): a circuit compiled against the published snapshot waits in a queue and, by the time it runs, is mapped to a device state that no longer holds.

Compiling against fresh data closes much of that gap. Wilson's just-in-time transpilation re-measures readout and two-qubit errors immediately before submission and feeds them to the level-3 transpiler, improving result accuracy by 3–304 % across representative workloads, and up to 400 %, over the default snapshot-based mappings (Wilson et al., 2020). The full-stack picture is the same: Murali et al. (2019)'s noise-adaptive TriQ toolflow, driving mapping, routing, and scheduling from current calibration, improved program success rate by up to 28× (geomean 3×) on IBM hardware, gains on the order of the differences between whole hardware platforms. Across both, the more closely compilation tracked the device's present state, the better the result. On this evidence, compilation is a runtime concern.

This is the move classical compilers made long ago. An LLVM-based toolchain can keep a program in its intermediate representation and lower it to machine code only once the target is known. This practice is called just-in-time compilation (Lattner et al., 2004). A classical JIT fits a target that is unknown beforehand but fixed once found, whereas calibration keeps drifting, so the same mapping can quietly fall out of date.

Tracking the device that closely only pays off if re-transpiling is cheap, and the reference SDK transpilers are the bottleneck: re-mapping before every submission repeats routing, the NP-hard and most expensive stage, and Qiskit's Python transpiler can exceed an hour on the large, dense circuits of chemistry and optimisation (Hua et al., 2023). A separate line of work attacks exactly this cost. QASMTrans is a self-contained C++ transpiler that parses OpenQASM, maps and routes with SABRE for IBM, Rigetti, IonQ, and Quantinuum, and runs 10–369× faster than the Qiskit transpiler, finishing in seconds circuits that Qiskit cannot transpile within an hour (Hua et al., 2023). The authors note that this speed matters because hybrid workloads transpile constantly. A hybrid algorithm is not a single fixed circuit but a loop. Every iteration produces a structurally new circuit that must be transpiled before it can run, so transpilation is paid on each pass through the loop rather than once, and a slow transpiler's cost multiplies with the iteration count. A circuit may also be re-transpiled mid-run to track the device, re-mapping against fresher calibration as the hardware drifts.

Table 5.2: *Compilation toolchains compared by intermediate representation, backend-neutrality, and calibration data.*

Toolchain	IR	Backend-neutral	Calibration data
Qiskit transpiler	DAGCircuit	No	Snapshot
QASMTrans	Gate IR (C++)	Yes	Snapshot
t|ket⟩	Internal DAG	Yes	Snapshot
XACC	XACC IR	Yes	None
CUDA-Q	MLIR / QIR	Yes	None
Cirq compiler	Cirq IR	No	Snapshot (Google)

This is not a Qiskit quirk. Across the compilation landscape, no toolchain reads the device live; the most current any of them uses is a periodic snapshot. T|ket⟩ retargets one compiled representation across IBM, Google, Rigetti, and Quantinuum, and its noise-aware placement uses reported error data, but that data is the same periodic characterisation. QASMTrans reads device topology and fidelity from a static description loaded from file, so its speed buys faster re-transpilation, not fresher data. XACC (McCaskey et al., 2018) and CUDA-Q map against no calibration at all: XACC treats the quantum kernel as a callable component in a classical workflow, and CUDA-Q compiles QPU-agnostically across vendors through an MLIR-and-QIR toolchain, but neither maps against the device's current errors. Cirq incorporates calibration only inside Google's own toolflow (Table 5.2).

What the field *has* standardised at this layer is the part that can be agreed on paper: the representation. OpenQASM 3 (Section 5.4) gives a vendor-neutral interchange for circuits. QIR offers a common compiler-level target: a set of rules for expressing quantum programs inside LLVM IR, the classical "one IR, many back ends" recipe applied to quantum. It is backed by a Linux Foundation alliance, though the largest vendors,

IBM and Google, stay on their own representations (QIR Alliance, 2021). The effort is maturing along a telling trajectory. Earlier IR-unification frameworks did not bring qubit mapping and routing to real devices, and coupled only weakly to classical HPC (Zhu et al., 2026); QLLVM closes both gaps, lowering an MLIR quantum dialect to QIR, running SABRE mapping at the LLVM IR level against real coupling graphs, and linking quantum kernels with CUDA, MPI, and C++ into a single executable. But a representation describes the program, not the running machine. The one gap QLLVM leaves open is the one this section turns on: it maps against the static coupling graph, not live calibration, and lists just-in-time and adaptive compilation as future work.

Whether to re-map a circuit against the device's current state is a decision that depends on live information, and in classical operations such decisions are run as a control loop: observe the current state through telemetry, decide, act, and repeat. For instance, in cloud computing, autoscaling provisions resources from current load and events, and load balancers route traffic based on runtime conditions. Vendor runtimes already act at execution time, but only to mitigate errors on a circuit that has already been mapped against the snapshot.

5.6 Infrastructure and Orchestration

Above the compiler, the question is no longer how a circuit is rewritten for one machine but how a job reaches hardware at all: which cloud model exposes the QPU, which scheduler places the job on a backend, and which substrate coordinates the surrounding hybrid quantum–classical loop. Two axes order the survey: how vendor-neutral each mechanism is, and how far it has matured from sketch to deployed system.

Quantum cloud computing has a stabilised vocabulary. Nguyen et al. (2024) organise the field into five aspects: computing models, software use cases, providers and platforms, resource management, and security. Three computing models matter here:

- *QCaaS* (Quantum Computing as a Service): remote access to a QPU as a raw resource, with the *Quantum-Algorithms-as-a-Service* refinement hiding the circuit behind a higher-level API.
- *Quantum Serverless*: a function-as-a-service abstraction that removes infrastructure management from the user.
- *Hybrid quantum–classical computing*: classical pre- and post-processing wrapped around quantum execution.

A fourth strand, distributed quantum computing across networked QPUs (Cuomo et al., 2020; Caleffi et al., 2024), is networking-bound and outside the scope set in Section 1.3.2. Production workloads combine several of these, with QCaaS providing access and the hybrid model the orchestration around it.

Operational access follows one of two arrangements, distinguished by Nguyen et al. (2024): hardware vendors that run their own cloud over their own devices (IBM Quantum, Google, Rigetti, IonQ), and aggregator platforms that expose third-party

hardware behind a uniform job-submission API (Amazon Braket, Azure Quantum). The distinction matters to this thesis because it bounds what can be observed: a first-party cloud can in principle instrument deep into the backend but ties the user to one vendor, whereas an aggregator is portable across vendors yet cannot see inside the remote backend it reaches.

How quantum workloads actually behave on these platforms has been characterised empirically by Ravi et al. (2021), who analysed more than six thousand jobs and six hundred thousand circuit executions on over twenty IBM machines across a two-year period to April 2021. The study documents the realities a scheduler must accommodate: jobs enter a per-machine queue ordered by a fair-share policy rather than first-come-first-served. The median wait is around an hour, yet more than a tenth of jobs queue for a day or longer, and waits vary by orders of magnitude across machines. Giortamis et al. (2025b) corroborate the heterogeneity on a more recent IBM fleet (six 27-qubit Falcon-r5 systems, November 2023), reporting up to a 38 % fidelity difference between the best and worst same-vendor same-day QPU and up to ~100× queue-size differences across QPUs at the same calibration cycle.

These measurements define what scheduling means in this setting. Nguyen et al. (2024) decompose the resource-management problem into two coupled sub-problems: *resource estimation* (predicting a circuit's runtime and fidelity on a candidate backend) and *resource allocation, or job scheduling* (placing and ordering the job). They frame the surrounding orchestration as the joint task of allocating, scheduling, and *monitoring* quantum and classical resources. Ravi et al. (2021) draw the line from classical practice directly: as in classical HPC a vendor wants to maximise throughput, but *unlike* classical HPC a quantum backend's usefulness is governed by fidelity constraints. Its qubit topology is static and its error profile is dynamic and calibration-dependent, so a backend cannot be packed to capacity the way a homogeneous classical node can.

Scheduling a quantum job is therefore a two-part decision over a fleet that is both heterogeneous and time-varying: *which* backend the job runs on (backends differ in qubit count, connectivity, and a fidelity profile that drifts with each calibration cycle) and *when* it runs, against per-machine queues that today order jobs by fair share (the policy Ravi et al. (2021) observe across the fleet and Nguyen et al. (2024) confirm for IBM's Open plan) and can hold a job for a day. The two objectives pull against each other, since the highest-fidelity backend is also the most contended; this is the fidelity-versus-wait tradeoff. The decision sits between the two layers the rest of the section examines: the infrastructure substrate that exposes a backend as a schedulable resource, and the orchestration that coordinates the surrounding hybrid loop. The systems below differ chiefly in how much quantum-specific awareness they bring to it, from delegating placement to a stock scheduler to casting it as explicit multi-objective optimisation. The third process in Nguyen's triad, monitoring, is the one each system handles only on its own terms.

Integrating quantum accelerators with classical HPC infrastructure has produced a body of work that maps the design space without converging on a shared engineering

practice: early software-systems requirements (Humble et al. 2016; Britt et al. 2017), algorithmic implications (Möller et al. 2017; Harrow et al. 2017), and the practical integration challenges of heterogeneous scheduling, data-movement cost, and absent interface standards (Gropp et al., 2020). The most pointed framing of integration as a software-engineering problem rather than a hardware one comes from M. Schulz et al. (2022), which Shehata et al. (2025) concretise into a comprehensive, hardware-agnostic software-stack architecture for HPC–QPU integration. Having manually evaluated 107 publications across 2016–2025, Döbler et al. (2025) identify standardisation of interfaces and tools as the principal next step for the field.

A common abstraction has begun to emerge from this body of work. Bacher et al. (2025) consolidate the integration problem around a vendor-neutral middleware interface, the Quantum Resource Management Interface (QRMI), that treats quantum hardware as a first-class schedulable resource in classical workload managers (Figure 5.2). Vendor-specific complexity is encapsulated behind modular adapters, so scheduler-level code remains agnostic to vendor details. The interface decomposes the job into three lifecycle phases: *acquire* (the manager locks a quantum resource before execution), *execute* (tasks run through `task_start`, `task_status`, and `task_result` primitives), and *release* (the resource is freed at completion). It groups its endpoints into three API surfaces: resource acquisition, task running, and metrics.

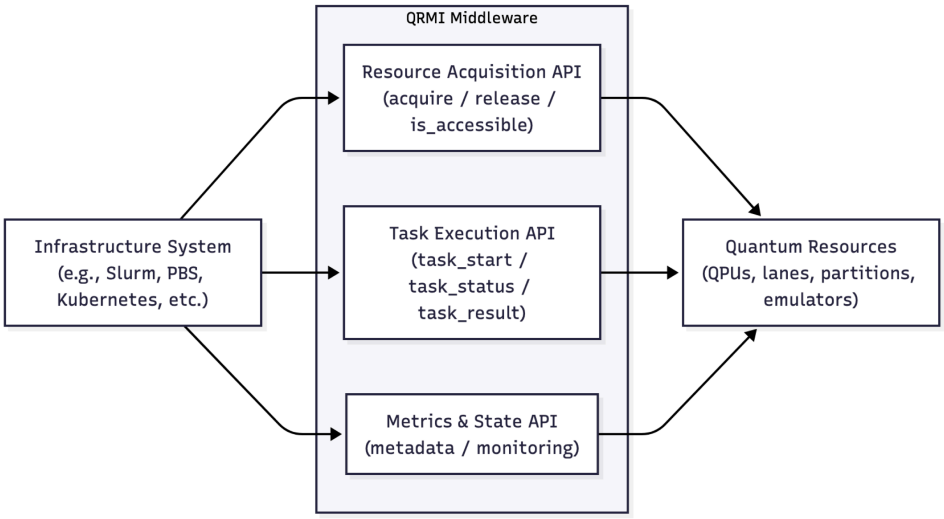


Figure 5.2: *QRMI as thin middleware between infrastructure systems and quantum resources. Reproduced from Bacher et al. (2025) Fig. 3.*

The same middleware supports two host integrations and two access models. The reference implementation is a SPANK plugin that registers each QPU as a schedulable Slurm resource; a Kubernetes device-plugin variant, reusing the same adapters, is sketched in an appendix. The two access models are *on-premises direct access*, where a co-located QPU is held for the duration of the classical job, and *quantum cloud bursting*, where remote QPUs are pulled in at runtime; both have been deployed at HPC centres. Cloud bursting is the case that most needs a vendor-neutral interface, since the local

and remote devices may come from different vendors. QRMI ships adapters for IBM and PASQAL hardware today. As the work of a broad industry and HPC-centre collaboration, QRMI sits alongside the QCSC reference architecture of Seelam et al. (2026) at the centre of current cross-vendor effort.

The QRMI design also names a capability it does not yet implement, which it calls *resource selection through attributes*: a submission requests a quantum resource by qubit count and minimum quality metrics rather than by named backend. Its hardware-topology and monitoring-metadata endpoints expose what such selection would consume, but the semantic conventions that would make those metrics comparable across vendors are not defined, a gap returned to in Section 5.7.

Pre-QCSC full-stack models (Bertels et al., 2020; Bandic et al., 2022) contribute the QPU-as-accelerator framing that Seelam et al. (2026) later integrate into a system-scale reference architecture. The move from those models to operational orchestration runs through two routes. The first adapts the pilot-job abstraction from HPC grid computing: Mantha et al. (2025) wrap quantum workloads in a placeholder job that pulls tasks from a queue, hiding backend-selection complexity but adding neither observability nor vendor-neutral orchestration.

The second route is Kubernetes-native, and it has followed two architecturally distinct patterns. Stirbu et al. (2024) demonstrate the lower-level pattern through Qubernetes, a research prototype that maps quantum execution onto existing Kubernetes abstractions. In their model, a quantum-capable backend is represented as an ordinary Kubernetes node annotated with a label such as `accelerator: qpu` and an extended resource such as `vendor.example.com/qpu`. A quantum computation is packaged as a container image and submitted as a standard Kubernetes `Job`; the resulting Pod uses a node selector and an extended-resource request to ensure placement on a node that advertises QPU capacity. Scheduling is therefore delegated to the default `kube-scheduler`, which performs conventional label and resource matching rather than quantum-aware backend selection.

In the authors' experimental setup, the QPU-capable node acts as a proxy to the HELMI quantum computer: the container entrypoint connects to HELMI over SSH, submits the workload, waits for completion, and exposes the result through the normal Kubernetes Job and logging model. The same Job-based pattern is also used for CPU- and GPU-based quantum simulation, with the execution target selected through Kubernetes resource requirements. The prototype therefore shows that Kubernetes can provide a common low-level execution substrate for simulated and hardware-backed quantum tasks. However, calibration awareness, fidelity-aware placement, queue-state awareness, and backend selection across a heterogeneous QPU fleet remain outside its scope. The evaluation is also limited to a single physical QPU, HELMI, which the authors identify as an internal threat to validity. They mention CRDs as a possible path beyond the built-in `Job` primitive, but do not implement such an extension.

Bacher et al. (2025) also outline a Kubernetes-oriented path for QRMI (Figure 5.2). Rather than defining quantum execution as a new Kubernetes workload abstraction,

this path follows the existing accelerator integration model. Kubernetes Device Plugins are already used to expose vendor-specific hardware such as GPUs, FPGAs, and high-performance network devices as schedulable resources. In the same spirit, a quantum Device Plugin could expose QPU capacity as a Kubernetes extended resource, while QRMI would remain responsible for the provider-facing acquisition, execution, and release logic. This model is most direct when the QPU is node-bound, since the backend can then be treated similarly to a specialised accelerator associated with a worker node. The paper also points to a broader case in which quantum capacity is shared rather than node-local. This is where Kubernetes Dynamic Resource Allocation (DRA) becomes relevant. In current Kubernetes practice, DRA is motivated by accelerator scenarios that are difficult to express as simple integer resources, such as partitioned GPUs, device slices, or resources with specific placement and configuration requirements. The analogous quantum case is a backend that is remote, facility-level, cloud-hosted, partitioned, or exposed through execution lanes. Such a backend is less like a physical device attached to one worker node and more like an allocatable access right that must be claimed and coordinated across Pods. This suggests a useful separation of concerns: Device Plugins are a natural fit for node-local QPU exposure, DRA is better aligned with shared or non-local quantum capacity, and QRMI can provide the backend-facing resource-management interface behind both.

Kaya et al. (2024) situate this within the Munich Quantum Software Stack for disaggregated accelerators; the cloud-scheduling layer is taken up by Qonductor (Giortamis et al., 2025b), whose companion system QOS (Giortamis et al., 2025a) handles the separate problem of co-locating several circuits on one QPU. Qonductor is a Kubernetes-based scheduler that places whole circuits across a fleet of backends. For each job it predicts runtime and fidelity with a regression model (reported R^2 of 0.998 and 0.976), then picks a backend that balances fidelity, job-completion time, and utilisation through a multi-objective search (NSGA-II). When a schedule would span a calibration cycle, it re-plans the affected jobs. Its evaluation, though, is run in simulation rather than on live hardware: Qiskit noise-model backends drawn from November 2023 IBM calibration data, over more than 70,000 circuits. In that setting, against a first-come-first-served baseline, it reports up to 54 % lower job-completion time and 66 % higher utilisation at roughly a 3 % fidelity cost on average (Giortamis et al., 2025b). Like Qubernetes it is a single-vendor study, scoped to one circuit on one QPU at a time, with multi-programming left to future work.

The QCSC reference architecture of Seelam et al. (2026) (Figure 5.3) synthesises the orchestration concerns above into a unified system model. Four horizontal layers (Hardware Infrastructure, System Orchestration, Application Middleware, and Applications) are crossed by three vertical concerns: Cloud Software, System Management and Monitoring, and Security. QRMI sits inside System Orchestration as a lightweight library, not as a layer of its own. The Cloud Software cross-cut acknowledges Kubernetes and OpenShift adoption in HPC environments as a foundation for integrating quantum systems, but does not commit QCSC to Kubernetes as its deployment sub-

strate, which remains Slurm-extended HPC.

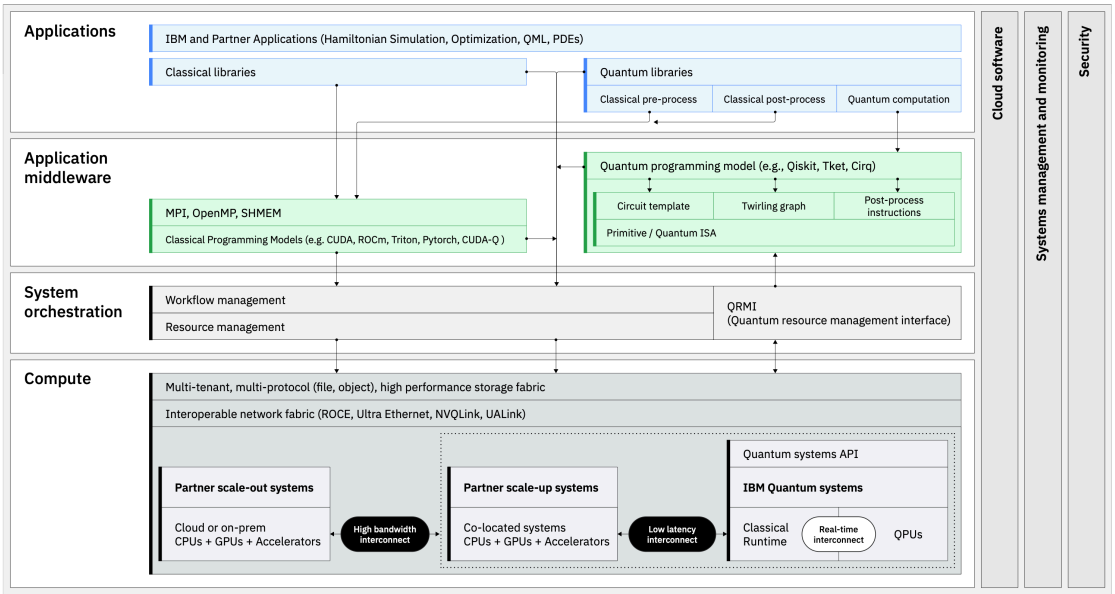


Figure 5.3: *The Quantum-Centric Supercomputing (QCSC) reference architecture with QRMI. Reproduced from Seelam et al. (2026) Fig. 4.*

Table 5.3 consolidates the orchestration landscape. Two lineages are visible: a Kubernetes-native route and a Slurm-native route. They converge on one design pattern, vendor-agnostic middleware with modular adapters exposing quantum hardware as a schedulable resource, but diverge on substrate and on how far each has been demonstrated.

Table 5.3: *Orchestration systems surveyed, by substrate and maturity.*

System	Substrate / scheduler	Vendor reach & maturity
Qubernetes	Kubernetes Job, default kube-scheduler	Single-vendor research prototype (HELMI)
QRMI + Slurm	Slurm, SPANK plugin over Generic Resources	IBM and PASQAL adapters; deployed at two HPC centres
QRMI device plugin	Kubernetes, DRA device-plugin	Reuses QRMI adapters; appendix sketch, not evaluated
Qonductor	Kubernetes scheduler, NSGA-II	Single-vendor (IBM); simulation evaluation
QCSC	Slurm-extended HPC (Kubernetes acknowledged)	Cross-vendor reference architecture (specification)

Two findings close the layer. The first is that orchestration of quantum workloads is feasible but fragmented: the Kubernetes-native and Slurm-native lineages agree on the adapter pattern yet share no common operational semantics, and no single system has been demonstrated across the breadth of the vendor landscape. The second concerns observability, and it is not that the literature ignores it. Seelam et al. (2026) place system management and monitoring among the cross-cutting concerns of the reference architecture, and QRMI exposes a metrics surface alongside its execution endpoints.

5.7 Observability

Observability cross-cuts the four layers reviewed above. In classical systems, it has matured into a standardised practice rather than a single tool (Section 3.5). Software is instrumented to emit metrics, logs, traces, and increasingly profiles in agreed formats, allowing collectors, exporters, and dashboards from different vendors to consume the same signals. This standardisation spans the stack. At the operating-system layer, the Linux kernel exposes runtime state through stable interfaces such as `/proc`; at the application and platform layers, Prometheus and OpenTelemetry define common data models, protocols, and semantic conventions. These conventions are what make telemetry portable: a signal is collected, named, typed, labelled, and correlated in a way that other systems can understand.

Quantum execution has no equivalent layer. Existing cloud and runtime interfaces expose provider-specific job information, but no telemetry for signals such as calibration state, shot counts, queue position, backend selection, and transpilation timing, together with a schema for correlating them across the classical–quantum boundary. No semantic convention currently exists for quantum or the QPU in the OpenTelemetry registry,[5] which defines conventions for domains from HTTP and databases to generative-AI inference; the Prometheus exposition format and W3C Trace Context are the same. Döbler et al. (2025) report the same from the literature side, finding little published detail on communication protocols across the 107 publications they surveyed, and Nguyen et al. (2024) do not list observability among the open problems of quantum cloud computing (Section 5.2).

What the literature names, at this layer, is classical tooling; the quantum semantics above it stay open. The QCSC reference architecture places monitoring among its cross-cutting concerns and names the stack directly: Prometheus, Grafana, node exporters, and OpenTelemetry. It calls for exporters specialised to QPU metrics, and as a reference architecture it names the requirement and leaves the vendor-neutral implementation open (Seelam et al., 2026). Qubernetes lists monitoring as an explicit objective and points to Prometheus and Grafana for the Kubernetes layer, leaving quantum-specific metrics and cross-boundary correlation out of scope (Stirbu et al., 2024). Qonductor keeps an internal system-monitor datastore that refreshes calibration data each cycle to feed its scheduler; the telemetry is consumed for scheduling and is not surfaced through a shared convention (Giortamis et al., 2025b). QRMI exposes a metrics API group alongside its execution endpoints, leaving the schema for those metrics open (Bacher et al., 2025). Each reaches for the classical observability stack; none yet defines the quantum-specific semantics, namely what to name and how to correlate it across the boundary.

Kanazawa et al. (2025) present, to the author's knowledge, the first observability architecture built for quantum-centric supercomputing, and it argues a specific point

[5] https://opentelemetry.io/docs/specs/semconv/

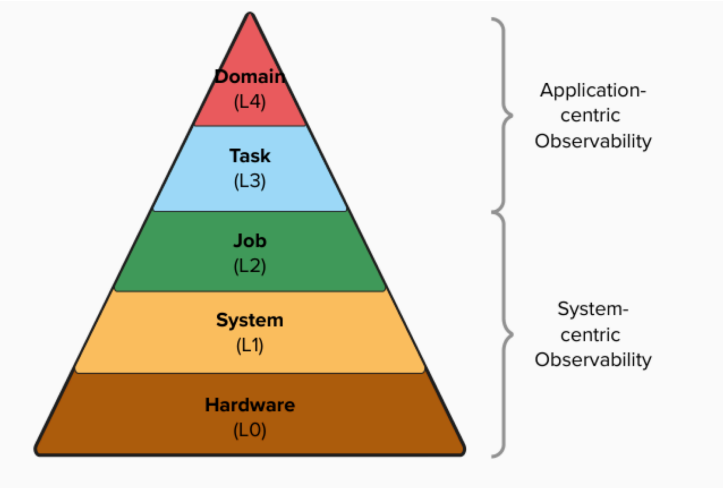


Figure 5.4: *Kanazawa workflow metrics pyramid for QCSC. Adapted from Kanazawa et al. (2025) Fig. 1(a).*

about what to observe. Observability in HPC has been system-centric: CPU, memory, and thermal metrics that administrators read for performance tuning and anomaly detection. These reveal little about whether the algorithm running on top is making progress. A probabilistic hybrid workflow needs telemetry that reaches into the application itself.

The paper formalises this as a five-level metrics pyramid (Figure 5.4), running from hardware (L0: power, thermal) and system (L1: CPU, memory, I/O) through job (L2: scheduler accounting and node usage) to task (L3: per-step artifacts and wall-clock times) and domain (L4: solver convergence and fidelity). The lower three levels are the system-centric view HPC monitoring already provides; the top two are the application-centric view, and the paper notes they cannot be read from job-scheduler APIs. They require integration with the workflow orchestrator itself.

The implementation decouples the observability platform from the compute platform and persists raw telemetry, so new metrics can be derived after a run without repeating it. Repeating a run is costly, since a single execution consumes 19.6 % of the QPU allocation and 7.7 % of the HPC, enough that the QPU side caps the experiment at about five runs. The architecture is realised on the Miyabi supercomputer against IBM Quantum with Prefect and Apache Superset.

Providers today expose per-job statistics, not telemetry in the scrapeable sense. IBM Quantum, AWS Braket, and Azure Quantum return per-job status and usage through their own REST APIs and portals, which a user must query by hand for each provider. The observability layer is therefore missing two things. The first is a standardised, vendor-neutral semantic convention for quantum signals and the instrumentation to emit them as metrics. The second is a telemetry pipeline to collect, export, store, and aggregate those metrics for analysis and visualisation. Section 5.8 sets these alongside the gaps identified at the other layers.

5.8 Gap Summary

The five layers reviewed in this chapter expose the same structural pattern. At each layer, the literature already contains techniques, prototypes, and partial implementations that address the immediate technical problem. What is still missing is the standardisation layer that turns those techniques into repeatable engineering practice across providers. In classical computing, such standards enabled software engineering, cloud operations, and Site Reliability Engineering to operate at velocity and scale: stable abstractions, shared interfaces, portable artefacts, common telemetry, and operational conventions made systems faster to build, safer to operate, and reproducible across environments. The quantum–classical interface has not yet reached that level of operational maturity. Table 5.4 summarises the resulting gaps.

Table 5.4: *The gap identified at each layer of the classical–quantum interface.*

Layer	Gaps
Application	A device's calibration and noise change from one run to the next and cannot be held fixed, so the same circuit can return different non-reproducible results, which could be amplified in variational pattern algorithms. The only way to tell whether a change came from the code or from the hardware is to record the hardware state alongside each run, and the literature does this one experiment at a time, not systematically.
Framework	The code that describes a circuit is portable across vendors, but running it is not, because each vendor still compiles and executes through its own toolchain. Since the layer that talks to the hardware stays vendor-specific, the orchestration and observability layers above cannot inherit vendor-neutrality and have to add it themselves.
Compilation	A circuit is compiled for a specific device from that device's calibration data, but the data is the most recent published snapshot, not a live reading. The device keeps changing, and the snapshot often goes out of date while the job waits in the queue, so the circuit can run against a device that no longer matches it and the result suffers. Re-compiling against fresh data fixes most of this, but no current toolchain reads the device live for JIT transpilation.
Orchestration	Several systems can already run hybrid workloads (HPC SLURM and Kubernetes jobs, as middleware that treats a QPU as a schedulable resource, or as schedulers that pick a backend by predicted fidelity), but they do not share a common approach. The most vendor-neutral of them is a library rather than an agreed standard, the most thoroughly tested was tried on only one vendor, and running across several vendors is still future work.
Observability	The classical monitoring tools (Kubernetes events, Prometheus, OpenTelemetry) already run beneath the quantum stack, so the collection machinery is in place. What is missing is agreement on what a quantum signal is; with no naming convention for signals such as calibration state or queue position, the tools have nothing to collect, and providers expose only per-job numbers through their own APIs. The two gaps are a metrics specification for quantum signals and their telemetry pipeline.

6

Architecture for Cloud-Native Quantum--Classical Interfacing

The five layers of the quantum–classical stack reviewed in Chapter 5 share the same structural pattern. The techniques and partial implementations exist, but the cross-vendor conventions that would make them operable across providers do not (Section 5.8). Each gap maps onto a principle already matured in classical computing: hermeticity and reproducibility at the application layer, operational standards and portable delivery at the framework layer, performance optimisation at the compilation layer, stability and scaling at the orchestration layer, and observability across the quantum–classical boundary. This thesis applies the discipline of SRE to hybrid quantum–classical execution, treating circuits and the QPUs that execute them as observable and schedulable resources, and instrumenting them with operational standards already established in classical systems.

This chapter contributes the Quantum Circuit Controller (QCC) as the architectural prototype. QCC is a Kubernetes operator built on four primary components: a pair of custom resources, a controller, an executor, and a CLI. Quantum circuits and execution backends are modelled as the custom resources, defined by CRDs that extend the Kubernetes API. A Go-based reconciler drives each `Circuit` through a lifecycle phase machine. A separately-deployed executor performs the quantum-provider-facing work (source conversion, transpilation, submission, and result retrieval) behind a typed gRPC contract. Telemetry leaves the system through OpenTelemetry into standard cloud-native observability tooling. The combination delivers independently scalable components, declarative submission, and standards-based observability.

The load-bearing architectural contribution is a cross-boundary identifier convention that links each Kubernetes `Circuit` to its vendor-side execution handle in both directions, threading through the lifecycle and the observability surface. Empirical validation of QCC is the subject of Chapter 7; limitations and the contribution position are revisited in Chapter 8 and Chapter 9.

6.1 Requirements for the Architecture

Five requirements translate the gap synthesis of Chapter 5 into the architecture's design criteria. They are deliberately operational. The first four require a quantum workload to satisfy expectations already established for production systems under SRE: declarative deployment on commodity infrastructure, standards-based observability, modular provider integration, and end-to-end correlation across execution boundaries. The fifth captures the additional demand introduced by the quantum substrate itself: backend quality cannot be treated as a static property, but must be measured, exposed, and available to the execution process. Table 6.1 states these criteria within the limits bounded in Section 1.3.

Table 6.1: *The five requirements the architecture must satisfy.*

Req.	Name	Requirement
R1	Declarative infrastructure	Stand up declaratively on commodity hardware (a laptop, a cloud VM, a managed Kubernetes cluster) so a single developer can operate backends and run circuits without specialized infrastructure; HPC stays a valid target, not a prerequisite.
R2	Observable through industry standards	Surface every circuit's and backend's operational state over industry-standard telemetry, so it aggregates in any existing observability stack; instrument platform behaviour, leave algorithm internals to the workflow.
R3	Portable and modular across providers	Keep each provider behind an adapter against a stable API contract (out-of-tree, the way Kubernetes moved storage drivers out of its core) so a new provider is additive and provider-maintained.
R4	Correlated across the boundary in real time	Follow a circuit's execution across the boundary and see its internals in real time (transpiled shape, queue position, on-QPU versus orchestration time, resource-to-job mapping) via one identifier that resolves both directions from the telemetry.
R5	Monitored backend quality	Treat backend quality as a measured signal. Expose each backend's measured characteristics and each run's outcome as first-class telemetry, so backend choice and result-trust rest on execution evidence rather than on nominal backend claims.

R1: Declarative infrastructure. QCC must stand up declaratively on commodity hardware (a single development machine, a cloud VM, or a managed Kubernetes cluster) so that a single developer can describe the backends and circuits they want and have the platform operate them, without specialized infrastructure or an operations team. Supporting commodity hardware is what brings a cloud-native architecture to quantum execution and closes the gap that otherwise confines it to specialized infrastructure. HPC remains a valid target, and the right direction for scale and critical infrastructure, but it must not be a prerequisite for getting started.

R2: Observable through industry standards. Every circuit and every backend must surface its operational state through industry-standard telemetry. The signals that matter are operational: how a circuit moves through its lifecycle and where it spends time, what each backend's current calibration and load look like, and what a run produced. Emitted over open standards, the signals aggregate in whatever observability stack an operator already runs, with nothing bespoke between the system and the data; the quantities an algorithm computes internally stay with the workflow that owns them.

R3: Portable and modular across providers. Provider-specific behaviour must sit behind a single adapter against a stable API contract, so that supporting a new provider is an additive, self-contained change. The model is Kubernetes' own move from in-tree vendor drivers, once built into the core and released on its cadence, to out-of-tree interfaces such as the Container Storage Interface, which a provider maintains independently against a published contract. The controller imports no vendor code. A provider owns its adapter, and adding one touches neither the controller, the schema, nor the telemetry surface. Portability becomes a property of the contract.

R4: Correlated across the boundary in real time. An operator must be able to follow a circuit's execution across the quantum–classical boundary and see into its internals while it runs: the shape it took after transpilation, where it sits in the vendor's queue, how its wall-clock time splits between the QPU and orchestration, and how the cluster-side resource maps to the vendor-side job.

R5: Monitored backend quality. Backend quality must be treated as a measured signal rather than an assumed property. Each backend's measured characteristics (coherence times, gate and readout error rates, gate timing, and calibration vintage), together with each run's outcome distribution, are exposed as first-class telemetry, so that the choice of backend and the trust placed in a result rest on evidence collected at execution time. The same evidence remains available after the run through the resource status, lifecycle events, and telemetry emitted during execution, rather than resting on a nominal backend specification.

6.2 Design goals and architectural position

QCC is shaped like a production control plane but evaluated as a research prototype. Its design turns on one architectural separation: the component that orchestrates work and the component that communicates with quantum providers are kept apart, joined only by a stable contract. The Kubernetes side (declaring work, tracking lifecycle state, recording execution progress, and exposing telemetry) belongs to the controller. The vendor-facing side (source conversion, transpilation, submission, and result retrieval) belongs to the executor. The user states intent; the platform carries that intent across the quantum–classical boundary and reports back through Kubernetes-native state and standard telemetry.

The resulting contribution is a cross-substrate architectural surface rather than a hardened production platform. Production hardening (high availability, security, and multi-tenant isolation) is left as a set of seams the architecture exposes but does not implement, and is revisited in Chapter 8.

Set against the QCSC reference architecture of Seelam et al. (2026) (Figure 5.3), QCC sits primarily in the System Orchestration layer through its Kubernetes-native controller. It also absorbs a deliberate slice of the Application Middleware layer: each circuit is transpiled and laid out against its selected backend, producing the backend-specific circuit shape required for backend-quality evaluation. Its main weight, however, falls on the management and monitoring concern that cuts across the stack. A declarative per-instance lifecycle provides the audit surface, while an open-standards telemetry vocabulary aggregates circuit and backend state across runs.

Section 5.6 surveyed the closest systems in detail; the relevant point here is how QCC's choices differ. Qubernetes (Stirbu et al., 2024) also uses Kubernetes, but runs quantum workloads through built-in batch primitives rather than a purpose-built resource model. The QRMI device-plugin sketch of Bacher et al. (2025, App. B) works lower in the stack, exposing QPUs as allocatable resources beneath the scheduler. Qonductor (Giortamis et al., 2025b) addresses a different optimisation problem: multi-objective scheduling over candidate backends. The remaining comparators sit outside the Kubernetes substrate. The QCSC reference architecture is framed around Slurm-extended HPC, while managed platforms such as IBM Quantum, AWS Braket, and Azure Quantum are end-to-end services that QCC consumes through adapters rather than peers of the control plane.

QCC's distinctive choice is therefore not to improve global scheduling or to allocate quantum hardware directly. It models circuits and backends as first-class declarative resources, then makes the crossing of the quantum–classical boundary observable through lifecycle state, events, and telemetry. In this sense, its architectural claim is legibility: a hybrid run should be inspectable from declared intent to provider-side execution and back to recorded outcome. A fuller prior-art comparison follows in Chapter 8, after the implementation and empirical evaluation have established the evidence base.

6.3 System overview

QCC comprises two services in one namespace, a pair of custom resources, and a CLI, with every quantum-specific concern behind a single internal contract. Figure 6.1 shows how the pieces fit together, from the user down to the quantum backend, and Table 6.2 summarises the key components.

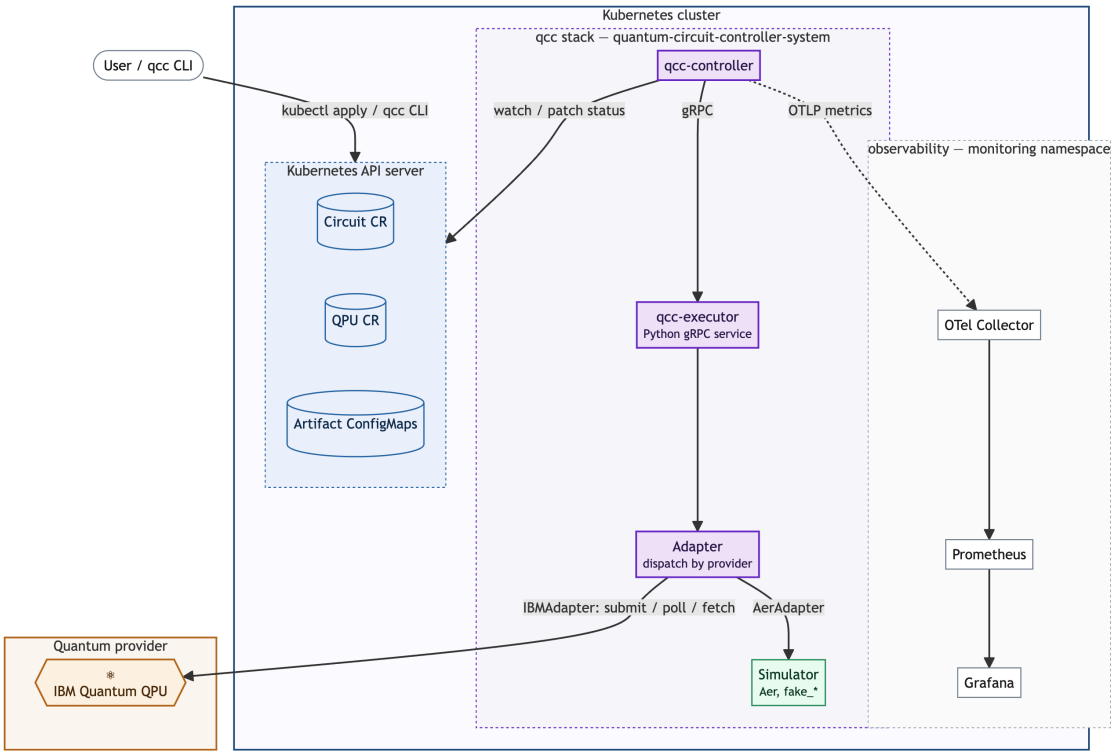


Figure 6.1: *The QCC stack, from user to backend. A request enters as a* `Circuit` *resource on the Kubernetes API server; the controller reconciles it and delegates the quantum-facing work to the executor over gRPC; the executor's adapter dispatches each circuit to an in-cluster simulator or to an external provider such as IBM Quantum.*

A user submits work through the `qcc` CLI or any standard Kubernetes client. Either way, the request becomes a `Circuit` resource on the API server, which is the system's only entry point and its durable boundary. The controller watches that resource, selects one of the backends registered as `QPU` resources, and drives the circuit through its lifecycle, all without loading a quantum SDK of its own. The quantum-facing work is delegated to the executor over gRPC: source conversion, transpilation, submission, and result retrieval. Inside the executor, an adapter dispatches each circuit by its declared provider: to a local simulator, either Aer or a calibration-faithful `fake_*` snapshot, or to a real provider such as IBM Quantum. Results return along the same path to the controller.

All telemetry originates in the controller. The executor computes the per-run quantities (timings, usage, and outcome quality) but reports them to the controller over gRPC rather than exporting anything itself. The controller then emits them, with its own metrics, over open standards for any observability stack to collect (Section 6.7).

Table 6.2: *QCC components at a glance.*

Component	Kind	Role
`qcc-controller`	Go binary, Deployment	Reconciles `Circuit` and `QPU` CRs; drives the lifecycle phase machine; patches status; emits telemetry.
`qcc-executor`	Python gRPC service, Deployment	Performs source conversion, ASCII rendering, transpilation, submission, and result retrieval behind an internal `Adapter` contract.
`qcc` CLI	Go binary, user-side	Constructs `Circuit` resources from local source files and submits them through the Kubernetes API.
`Circuit`	CR, namespace-scoped	Declarative submission unit: circuit body, execution intent, selection constraints, observed status.
`QPU`	CR, cluster-scoped	Registered execution backend: provider, kind, capability metadata, calibration metadata, availability.
Executor gRPC API	`qcc.executor.v1` (proto)	Wire contract between `qcc-controller` and `qcc-executor`: eight RPCs across three families (synchronous, asynchronous, utilities).
`Adapter`	Python Module	Vendor abstraction inside the executor: one class per quantum provider, registered by the `provider` string on `QPU.spec`.

6.4 Control plane components

6.4.1 qcc-controller

The qcc-controller is a Kubernetes operator, written in Go. It owns the two custom resources, `Circuit` and `QPU`, and reconciles each toward its declared state. It links no quantum library: every operation that touches a vendor SDK is delegated to the executor over gRPC.

The operator runs two reconcilers. The CircuitReconciler drives each `Circuit` through its execution phases, patching the resource status and its conditions on every transition. Because each transition is written back to the resource, the circuit's history stays visible in its own status, and a plain `kubectl describe` renders that history without any QCC-specific tooling. The same reconciler stamps the controller-owned labels onto a `Circuit` when it is first admitted. The QPUReconciler watches `QPU` resources: when one is registered, it calls the executor's `ProbeBackend` to fill in the backend's qubit count, basis gates, coupling map, calibration vintage, and operation-error medians, and to mark the backend available. Selection then draws only from the backends that are available.

The operator takes its input from Kubernetes API watches, and its output is of three kinds: status patches on the resources, metrics emitted over OTLP, and gRPC calls into the executor. Large artefacts, such as Circuit Drawings, and converted OpenQASM code are written to `ConfigMaps` that `Circuit` owns.

6.4.2 qcc-executor

The qcc-executor is the operator's Python counterpart, and the only component that links a quantum SDK. It runs as its own Kubernetes `Deployment`, separate from the controller, so the two scale independently. The controller reaches it over a gRPC call with a typed protobuf contract that carries the calls. Keeping every vendor call on the executor side leaves the controller free of quantum library SDK requirements, and extends the Adapter contract to support new providers.

The executor's calls group into three scopes. The first is discovery: a single call, `ProbeBackend`, made when a `QPU` is registered, which returns the backend's QPU characteristics (qubit number, processor name, gate and readout errors, and its coherence times T_1 and T_2). The second is source handling, with three calls: `ConvertSource` translates Qiskit Python into OpenQASM 3, `DrawCircuit` returns an ASCII rendering, and `ScheduleCircuit` returns a per-instruction timeline. The third, and the largest, is execution. A simulator job runs in-process and returns from a single `RunCircuit` call, while a hardware job is threaded through three calls: `SubmitTask` submits it, `WatchTask` polls its status, and `FetchTaskResult` retrieves its result.

Every call stands on its own. The request carries the circuit source, the chosen backend, and the correlation identifier, and the response carries either the result or a handle to a running job.

6.4.3 qcc CLI

The qcc CLI is the user-facing entry point. It reads circuit source from local files, builds Circuit resources, and submits them through the Kubernetes API. It links no quantum SDK, since the Python-to-OpenQASM conversion runs server-side in the executor, so it ships as a single binary whose only trust boundary is the Kubernetes API.

Its commands fall into two groups. The first submits work from a local source file: run executes a circuit, while draw and schedule produce an ASCII diagram or a per-instruction timeline. The second inspects what is already in the cluster: get reads back Circuit and QPU resources by kind and name, in the same style as kubectl, so qcc get qpus lists the registered backends and qcc get circuit <name> shows one run's status and artefacts. Because circuits and QPUs are ordinary Kubernetes resources, everything the CLI does is equally reachable through kubectl or any other Kubernetes client.

By default a run blocks until the circuit completes, which fits a simulator that returns within seconds. For a hardware backend, whose queue wait can run to minutes, the --detach flag changes that contract: the CLI submits the circuit, waits only until the controller has accepted it and a provider job is queued, and then exits. The controller keeps polling in the background, and the user checks back later with qcc get circuit <name>. This is the Kubernetes-native habit of submitting work and walking away, applied to circuit execution. A second mode serves comparison: --performance-test submits the same circuit to every available simulator, and to hardware as well when --include-hardware is set, under one shared experiment label so the runs can be read together.

Listing 1: *Representative qcc CLI invocations.*

```
qcc run bell.qasm                                    # submit and wait
qcc run shor.py --shots 4096
qcc run bell.qasm --backend ibm_kingston --detach    # submit and exit; check later
qcc run shor.py --performance-test --algorithm shor
qcc get qpus                                         # list registered backends
qcc get circuit shor-7p4jp                           # one run's status and artefacts
```

6.4.4 Executor gRPC API

The wire contract between the controller and the executor is the qcc.executor.v1 protocol, which defines eight calls in three families.

The synchronous family is a single call. RunCircuit transpiles, submits, and waits for the result in one round trip, which suits in-process simulators and short hardware jobs.

The asynchronous family threads a longer hardware job through three calls: SubmitTask submits the job, WatchTask polls its status, and FetchTaskResult retrieves the result once it is ready. These three follow the same lifecycle as a Qiskit JobV1, with its run,

status, and result methods, and as the QRMI task interface (Bacher et al., 2025), so the contract stays compatible with both substrates.

The utility family is four calls with no provider-side effect: `ConvertSource`, `DrawCircuit`, `ScheduleCircuit`, and `ProbeBackend`. Every execution call returns a provider job ID, which the controller records on the `Circuit`'s status. Both the simulator and the IBM adapter supply one, so a `Circuit` is linked to its execution on the backend by that recorded ID.

6.4.5 Internal Adapter contract

Inside the executor, every operation that touches a vendor SDK goes through an internal `Adapter` base class. A registry maps each `QPU.spec.provider` string to a concrete `Adapter`, which the executor selects at request time. Adding a vendor therefore means writing one `Adapter` subclass and registering its provider string; the controller, the CRD schema, and the telemetry surface all stay untouched. Table 6.3 lists the two adapters shipped today.

Table 6.3: *Adapters shipped with the executor.*

Provider	Adapter	Reach
`local`	`AerAdapter`	Qiskit Aer in-process plus `fake_*` snapshots via `FakeProviderForBackendV2`. No credentials.
`ibm`	`IBMAdapter`	IBM Quantum hardware and cloud simulators through `qiskit-ibm-runtime` (`QiskitRuntimeService` + `SamplerV2`). Credentials via `QISKIT_IBM_TOKEN` as Kubernetes `Secret`.

An adapter implements six methods: `transpile`, `submit`, `poll`, `fetch_result`, `inspect`, and `schedule`. Each returns a structured type that already carries the fields the controller persists on the `Circuit` status, so no second round trip is needed. Because the shape is fixed, swapping one backend for another is a matter of configuration rather than code. A new provider, whether AWS Braket, Azure Quantum, or the QRMI standard (Bacher et al., 2025), can be abstracted as one more adapter against the same contract.

6.5 API objects

QCC's API surface is two custom resources, the `Circuit` and the `QPU`, both managed through the Kubernetes API. The schema captures only what the controller and executor act upon. Algorithm semantics, iteration policy, and vendor-specific construction live either in user-supplied workflow code or behind the executor's `Adapter`.

6.5.1 Two-tier schema

Vendor SDKs change faster than a CRD schema can. If QCC modelled every possible parameter, the CRD would need a new version every time IBM shipped a new tran-

spiler flag. If it modelled too few, users could not reach the SDK's full surface. QCC resolves this with a two-tier spec.

Tier 1 is the typed `camelCase` vocabulary QCC owns and evolves: `mode`, `source`, `shots`, `optimizationLevel`, `backendSelector`, and `timeoutSeconds` on `Circuit`; `provider`, `backendName`, `kind`, `qubits`, `access`, `capabilities`, and `region` on `QPU`. Each resource has under a dozen of these fields; they change slowly and are exposed as CLI flags. The controller looks them up by name, and they are the stable part of the API.

Tier 2 is a pair of open blocks on `Circuit`, `transpile` and `execute`, whose contents the controller does not interpret but passes straight through to Qiskit. The keys of `transpile` become keyword arguments to Qiskit's `transpile()` function; the keys of `execute` become keyword arguments to the backend's run call, `AerSimulator.run()` for the local provider and `SamplerV2.run()` for IBM. The shot count is set once, through the Tier-1 `shots` field, and is stripped from `execute` so it cannot be given in two places. For example, a `transpile` block of `{scheduling_method: alap}` reaches Qiskit as `transpile(circuit, backend, scheduling_method="alap")`, requesting as-late-as-possible instruction scheduling.

Because the schema accepts arbitrary keys, the API server does not reject an option the schema does not name. When a vendor SDK gains a new parameter, a user can set it as soon as the new SDK version ships in the executor pod, with no CRD version bump and no controller rebuild.

6.5.2 Circuit

The `Circuit` is the declarative unit of submission. It lives at `qcc.io/v1alpha1` and is namespace-scoped. Each `Circuit` represents one execution intent: a body, a mode, selection constraints, and the status the controller writes as it drives the circuit through its lifecycle.

The spec captures the user's submission intent in six Tier-1 fields. `mode` is the lifecycle verb, one of `run`, `select`, `draw`, or `schedule`. `source` is the circuit body, given as a `{format, body}` pair; the format is either OpenQASM 3 (Cross et al., 2022) or Qiskit, and a Qiskit body is converted to OpenQASM 3 server-side by the executor. `shots` is the shot count. `optimizationLevel` is the transpilation policy, an integer from 0 to 3. `backendSelector` carries hard constraints on selection: provider, named backend, hardware-or-simulator kind, and minimum qubit count. Its `allowedQPURefs` and `region` fields are reserved in the schema. `timeoutSeconds` bounds execution.

The `Circuit` also carries the two Tier-2 blocks, `transpile` and `execute`. Listing 2 shows a minimal `Circuit`.

Listing 2: *A minimal `Circuit` manifest. The `spec.source.body` carries an OpenQASM 3 Bell-state program; `backendSelector.kind` requests any registered simulator.*

```yaml
apiVersion: qcc.io/v1alpha1
kind: Circuit
metadata:
  name: bell-state
```

```
  labels:
    qcc.io/algorithm: bell-state
    qcc.io/algorithm-version: v1
spec:
  mode: run
  shots: 1024
  source:
    format: openqasm3
    body: |
      OPENQASM 3.0;
      include "stdgates.inc";
      qubit[2] q;
      bit[2] c;
      h q[0];
      cx q[0], q[1];
      c[0] = measure q[0];
      c[1] = measure q[1];
  backendSelector:
    kind: simulator
```

The controller writes status as the `Circuit` progresses. The status block carries the lifecycle `phase` and an array of `conditions`, the `selectedQPU` and `providerJobId`, the `results` map of measurement counts, `usageSeconds` for backend-reported usage time, and a `transpile.{depth, twoQubitGates, totalGates}` block of post-transpilation circuit metrics.

Eight condition types track the lifecycle's decision points: `Accepted`, `Validated`, `Selected`, `Rendered`, `Scheduled`, `Submitted`, `Completed`, and `Failed`. Each carries a `reason` code, a `lastTransitionTime` stamp, and an operator-facing message.

Three further status fields point to the larger artefacts a run produces: `status.drawingRef` to an ASCII drawing, `status.scheduleRef` to a per-instruction timeline, and `status.convertedRef` to a converted OpenQASM 3 body.

6.5.3 QPU

The `QPU` is a registered execution backend. Its schema separates what the operator declares from what the controller discovers: the operator states which backend this is and how to reach it, and the controller discovers the backend's characteristics and live state by probing it through the executor. The resource is cluster-scoped at `qcc.io/v1alpha1`.

The declared block has six fields. `provider` selects the executor's adapter (`local` for Aer and `fake_*` snapshots, `ibm` for IBM Quantum hardware). `backendName` is the provider-native identifier, such as `fake_brisbane` or `ibm_kingston`; if it is omitted, the controller derives it from `metadata.name` by replacing dashes with underscores, so the Kubernetes-friendly `fake-brisbane` maps to the Qiskit-native `fake_brisbane`. `kind` is required and is either `hardware` or `simulator`. `access.credentialSecretRef` names the Kubernetes `Secret` that holds a provider's credentials. `capabilities` holds operator-declared caps, most prominently `maxShots`, which the controller enforces at

submission. `qubits` and `region` are optional: `qubits` is a user-asserted hint that the probe overwrites with the observed value, and `region` is reserved in the schema.

Listing 3 shows a simulator entry and a hardware entry side by side.

Listing 3: *`QPU` manifests for a simulator entry (`fake-brisbane`, IBM Eagle r3 calibration snapshot) and a real-hardware entry (`ibm-kingston`, IBM Heron r2). The simulator entry needs no credentials and lets the probe derive most fields; the hardware entry references a credential `Secret` and declares the maximum shot count.*

```yaml
# Simulator backend: declared block minimal; status populated by probe.
apiVersion: qcc.io/v1alpha1
kind: QPU
metadata:
  name: fake-brisbane
  labels:
    qcc.io/provider: local
    qcc.io/family: eagle-r3
spec:
  provider: local
  kind: simulator
---
# Real-hardware backend: adds credential reference and operator-declared caps.
apiVersion: qcc.io/v1alpha1
kind: QPU
metadata:
  name: ibm-kingston
  labels:
    qcc.io/provider: ibm
    qcc.io/family: heron-r2
spec:
  provider: ibm
  backendName: ibm_kingston
  kind: hardware
  access:
    credentialSecretRef:
      name: ibm-quantum-token
      namespace: quantum-circuit-controller-system
  capabilities:
    maxShots: 100000
```

The operator writes none of the status. The controller probes each registered backend at registration and re-probes it on a configurable interval, so the recorded calibration follows the backend's live state. For an IBM backend the executor connects to the IBM Quantum platform through the Qiskit runtime (`qiskit-ibm-runtime`) and reads the backend's published calibration, returning the live values to the controller; for a local snapshot it reads the bundled `fake_*` data instead. Listing 4 shows the result for `ibm-kingston`: the short spec the operator applied expands into a full description of the backend, none of it written by hand.

Listing 4: *Discovered status for `ibm-kingston` after the probe (abridged; calibration values rounded). The operator applied only the spec in Listing 3; every field shown here was filled in by the controller's probe of the backend.*

```yaml
# kubectl get qpu ibm-kingston -o yaml   (status only)
status:
  availability: Available
  qubits: 156
  processor: {family: Heron, revision: "2"}
  couplingEdges: 352
  basisGates: [cz, delay, id, if_else, measure, measure_2, reset, rz, sx, x]
  lastCalibrationTime: "2026-05-16T10:35:26Z"
  coherenceMedians: {t1Micros: 174.6, t2Micros: 117.0}
  errorMedians: {singleQubit: 0.000274, twoQubit: 0.00203, readout: 0.0165}
  instructionDurationMedians: {singleQubitSeconds: 3.2e-08, twoQubitSeconds: 6.8e-08}
  dtSeconds: 4e-09
  conditions:
  - type: Ready
    status: "True"
    reason: ProviderProbeOK
    message: ibm provider is Available; live calibration via probe
```

The status falls into three groups. *Capability metadata* (`qubits`, `basisGates`, `couplingEdges`, `processor.family`, `processor.revision`) describes what the backend is. *Calibration metadata* (`lastCalibrationTime`, `errorMedians.{singleQubit, twoQubit, readout}`, `coherenceMedians.{t1Micros, t2Micros}`, `dtSeconds`, `instructionDurationMedians`) describes the backend's current characteristics in the units the schema publishes: microseconds for coherence, seconds for sample periods. *Operational signals* (`availability`, `queueDepth`, `lastError`) describe live state. `availability` takes one of `Available`, `Unavailable`, or `Unknown`, and is the field the controller reads when selecting a backend.

Two conditions report health. `Ready` signals that the adapter resolved, carries the reason `ProviderProbeOK`, and is always present. `MetadataFresh` signals calibration freshness, and appears only where freshness is meaningful. Local-provider QPUs carry it permanently `True`, since a frozen snapshot cannot become stale. Real IBM hardware does not carry it, since IBM calibration changes over hours and a static condition would misrepresent it; freshness there is read from `status.lastCalibrationTime`, which each re-probe refreshes.

6.5.4 Label conventions

Five reserved labels on `Circuit.metadata.labels` carry algorithm-grouping metadata, which propagates to the metric labels the controller emits. Each has explicit ownership, summarised in Table 6.4.

The two controller-stamped labels enable query patterns the user cannot get from metadata alone. `run-index` gives a stable ordinal within an experiment series. `source-sha256` answers whether the source body actually changed between versions: a label-only edit produces the same hash, so a real source change is distinguished from a re-stamped version label.

Table 6.4: *Reserved labels for algorithm grouping.*

Label	Owner	Stamped at	Notes
`qcc.io/algorithm`	User	Submission	Optional. Names the algorithm family (`bell-state`, `shor`). Without it, Circuits are one-off runs.
`qcc.io/algorithm-version`	User	Submission	Optional. Iteration of the algorithm. Requires `algorithm`.
`qcc.io/experiment`	User	Submission	Optional. Campaign identifier grouping runs across multiple algorithms. Requires `algorithm`.
`qcc.io/run-index`	Controller	First reconcile	Ordinal among Circuits sharing `algorithm` (and optional `experiment`); computed as $\max(\text{existing}) + 1$.
`qcc.io/source-sha256`	Controller	First reconcile	First 16 hexadecimal characters of the SHA-256 of `spec.source.body`. Always stamped.

The resource model described here is the static contract between the controller, the executor, and the cluster's API server.

6.6 Execution lifecycle

The lifecycle is the dynamic counterpart of the resource model: how the controller drives a `Circuit` from admission to a terminal phase.

6.6.1 State machine

Each `Circuit` follows one of four paths through the lifecycle, chosen by `spec.mode`. All four begin at `Pending` and end at `Succeeded` or `Failed`; any non-terminal phase moves to `Failed` when its step fails (Figure 6.2).

The modes share their early phases and diverge by how far they go. `run` executes end to end. `select` stops after transpilation and uses no QPU time. `draw` renders the circuit as ASCII, referenced by `status.drawingRef`. `schedule` produces a per-instruction timeline in control-electronics sample-period cycles, written to a `ConfigMap` the `Circuit` owns and referenced by `status.scheduleRef`.

The `Submitting` phase is the lifecycle's one external side effect: committing a job to the backend. Before issuing that call, the controller patches `phase=Submitting` to the cluster store, so the intent to submit is durable even if the controller restarts before the call returns.

The link between a `Circuit` and its job on the backend is established on QCC's side, not by reaching into the provider. With each submission the controller sends a

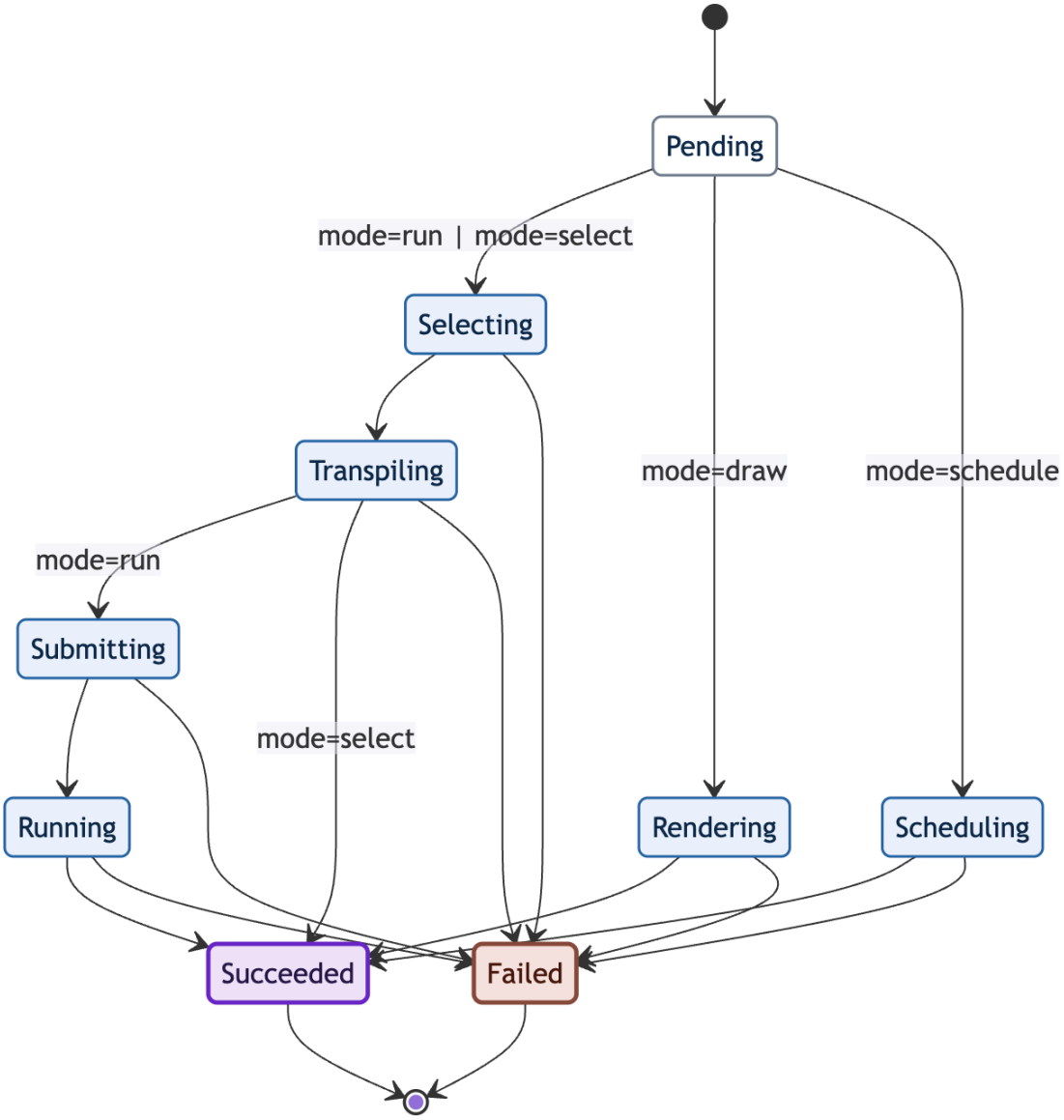


Figure 6.2: *The* `Circuit` *lifecycle. Each* `Circuit` *progresses through one of four mode-conditioned paths from* `Pending` *to a terminal* `Succeeded` *or* `Failed` *phase.*

correlation identifier, the gRPC `idempotency_key`, formed from the `Circuit`'s and its observed generation; the executor reads the `Circuit`'s back from it. The submission returns a provider job id, which the controller fetches and records on `status.providerJobId` in the same patch that sets `phase=Running`. That recorded id is the cross-boundary identifier: it ties the Kubernetes `Circuit` to its job on the backend, it lives on the `Circuit` itself, and it survives a controller restart, since the next reconcile reads it back from the resource. Figure 6.3 shows the sequence.

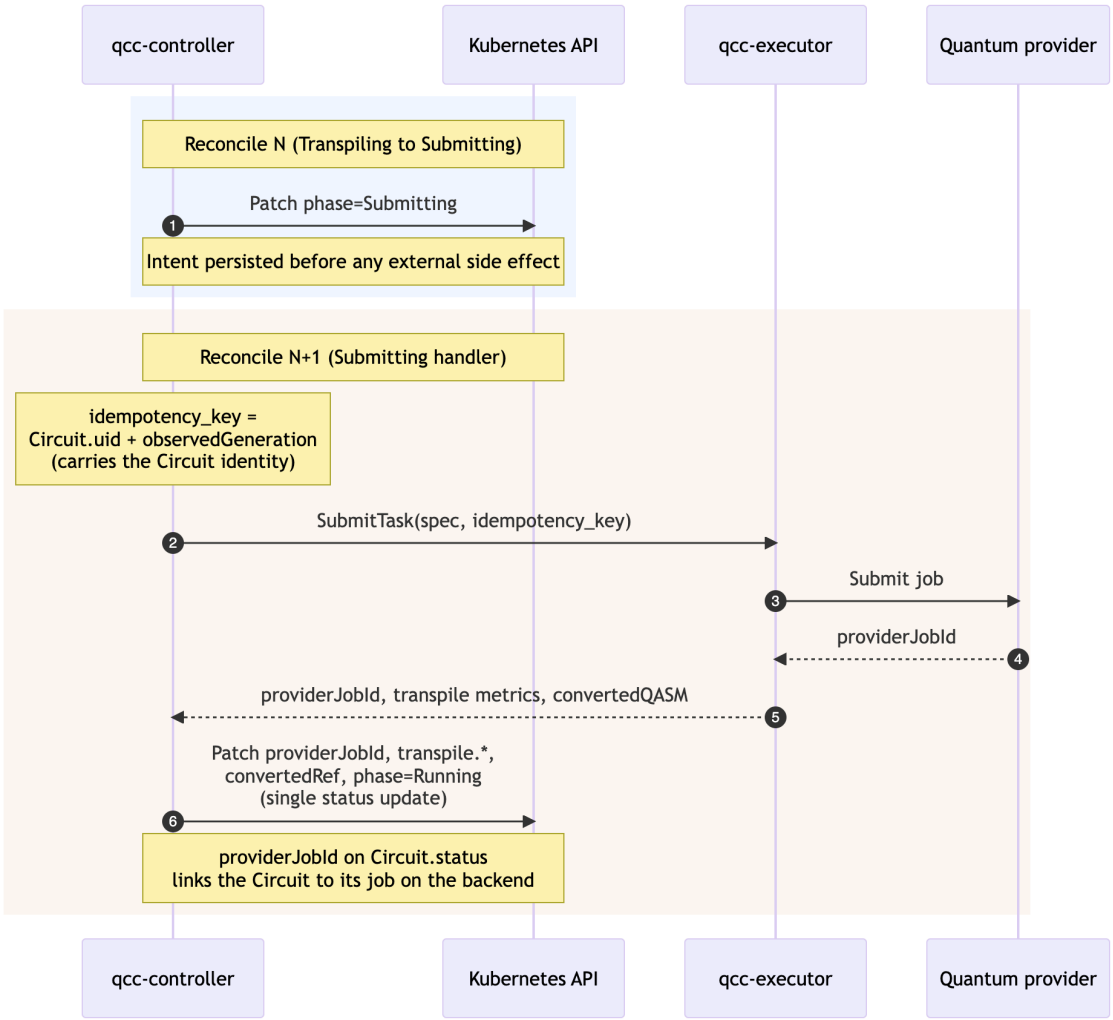


Figure 6.3: *Submission sequence and the cross-boundary identifier. The controller commits `phase=Submitting` before issuing the submit call, sends an idempotency key derived from the `Circuit`'s and observed generation, and records the provider job id returned by the backend on `status.providerJobId` in a single status patch. That recorded id links the `Circuit` to its job on the backend.*

After submission, a hardware job is polled by `WatchTask` against the recorded provider job id, while a simulator job returns its result inline on the same call.

6.6.2 Backend selection

The `Selecting` phase chooses which registered `QPU` the rest of the lifecycle uses. The shipped selector applies hard constraints only. A `QPU` is eligible when its `status.availability` is `Available`, its declared `capabilities.maxShots` is at least the requested `spec.shots`, and it satisfies the `Circuit`'s `backendSelector`: provider, named backend, hardware-or-simulator kind, and minimum qubit count. The controller takes the first eligible candidate and patches it onto `status.selectedQPU`; if none qualifies, the `Circuit` ends in a terminal `NoEligibleBackend` failure.

Calibration does not enter the selection step; it enters after a run completes, at the evaluation surface. When a `Circuit` finishes, the `qcc get` command turns the run's recorded numbers into a readable verdict. Listing 5 shows it for a completed Shor circuit on `ibm-kingston`.

Listing 5: *Post-completion summary from `qcc get circuit` (abridged; result histogram omitted). Every number is read back from the `Circuit`'s status and the chosen `QPU`'s status.*

```
shor-2vv42 · run · on ibm-kingston · degraded signal (error exposure  1.5)

  backend
    name          ibm-kingston  (156q · 352 edges · 10 basis gates)
    gate errors   1Q 2.74e-04 · 2Q 2.03e-03 · readout 1.65e-02
    coherence     T1 175 µs · T2 117 µs
    cycle time    dt = 4 ns

  circuit
    transpiled       depth 2048 · 2441 gates · 474 2Q
    exec time        ~139.26 µs (critical-path estimate)
    error exposure    1.5 events/shot
    fidelity bound   P(no gate error)   0.22  (excludes readout & coherence)
```

The headline is an *error-exposure* indicator: the expected number of gate-error events per shot. It is a first-order sum of the transpiled gate counts weighted by the backend's per-operation error medians,

$$\text{exposure} \approx n_{1Q}\,\epsilon_{1Q} + n_{2Q}\,\epsilon_{2Q},$$

where the single-qubit count is taken as the total gates minus the two-qubit gates. For this run the transpiled circuit has 2441 gates, of which 474 are two-qubit, leaving 1967 single-qubit; `ibm-kingston` reports a single-qubit error median of 2.74×10^{-4} and a two-qubit median of 2.03×10^{-3}. The exposure is therefore $1967\times2.74\times10^{-4} + 474\times2.03\times10^{-3} \approx 0.54 + 0.96 \approx 1.5$ events per shot. A value near 1 places the run in the *degraded signal* band, one of four: *signal preserved* below 0.1, *noisy signal* from 0.1, *degraded signal* from 1, and *signal expected lost* at 5 or above. The paired bound $P(\text{no gate error}) \approx \exp(-\text{exposure}) \approx 0.22$ says fewer than a quarter of shots are expected to be free of any gate error.

This is the SRE error-budget framing (Beyer et al., 2016) carried to a quantum

backend, and it is a regime signal, not a fidelity model. Only gate errors enter the sum; readout and coherence do not, which the command marks explicitly. The same summary makes the omission visible: the critical-path execution estimate for this run, $\text{depth} \times \max(t_{1Q}, t_{2Q}) \approx 139\,\mu\text{s}$, is already comparable to the backend's T_2 of $117\,\mu\text{s}$, so coherence decay is a real error source here that the gate-only exposure does not capture. The indicator places a run in a regime; it does not predict its fidelity.

Cross-substrate comparison is reachable through `qcc run --performance-test`, which submits the same circuit to every available simulator, and to hardware as well when `--include-hardware` is set, under one shared `qcc.io/experiment` label. It then prints a comparison table and a deep link to the Grafana dashboard for those runs.

The shipped selector ranks nothing; it filters and picks. Richer strategies, such as scoring candidates against current calibration or transpiling each to compare gate counts, would extend the same surface and consume the metrics the lifecycle already exposes.

The lifecycle is the dynamic counterpart of the static resource model. Every phase transition, selection decision, and terminal outcome is recorded on the `Circuit`'s status and exported as telemetry, which gives the run a machine-readable trace.

6.7 Observability

QCC's principal contribution sits at the System Management and Monitoring cross-cut of the QCSC reference architecture. Earlier sections covered the per-run view: each `Circuit` records its progress in `status.conditions`, which a plain `kubectl describe` renders. This section covers the aggregate view, the signals that describe many `Circuits` and QPUs at once. There are two: a metrics specification built entirely on cloud-native open standards, and a query convention that lets ordinary PromQL answer cross-substrate questions without distributed tracing.

6.7.1 Telemetry pipeline

The observability surface pairs Kubernetes-native primitives with a standard Open-Telemetry pipeline. Two data paths feed a single Prometheus instance, so Grafana sees one data source.

The `qcc.*` domain metrics flow through the OpenTelemetry SDK, over OTLP/gRPC to an OpenTelemetry Collector running beside the controller, and on to the cluster's `kube-prometheus-stack`, which scrapes the Collector's Prometheus exporter. Emitting OTLP rather than a backend-specific format keeps the application code independent of the metrics store: switching stores is a change to the Collector's configuration, not to QCC.

The controller-runtime built-ins (`controller_runtime_reconcile_*`, `go_*`, `process_*`) take their own Prometheus path through a separate `ServiceMonitor`, leaving the upstream library untouched. The Kubernetes-native side needs no separate pipeline at

all: a `Circuit`'s `status` fields and its `ConfigMap`-backed artefact pointers are already part of the cluster API, readable with `kubectl` or any Kubernetes client.

6.7.2 Metric specification

QCC publishes fourteen domain metrics in the two patterns standard for a Kubernetes operator. *Resource-state* metrics mirror the current state of a CR: they are observable gauges, read from the controller-runtime cache once per Prometheus scrape, the `kube-state-metrics` idiom. *Operational-event* metrics record what happens over time: they are synchronous counters and histograms emitted at the moment of a lifecycle transition, the `controller-runtime` idiom. Keeping the two apart bounds the active-series count and keeps the reconcile hot path cheap.

The two families answer two questions. The six QPU metrics (Table 6.5) describe the *substrate*: how good each registered backend currently is. The eight `Circuit` metrics (Table 6.6) describe the *work*: what happened to each circuit as it moved through the lifecycle. Each metric is a single number; its *dimensions*, the key-value labels it carries, are what let one metric name stand for several related quantities.

Table 6.5: *QPU-side metrics. All six are observable gauges populated from `QPU.status` via the informer cache on each Prometheus scrape. All series carry `qpu` as an identity label; only the additional dimensions are listed below.*

Metric	Additional dimensions
`qcc_qpu_info`	`uid`, `provider`, `kind`, `processor_family`, `processor_revision`
`qcc_qpu_operation_error_median`	`operation` ∈ {`gate_1q`, `gate_2q`, `readout`}
`qcc_qpu_operation_duration_median_seconds`	`operation` ∈ {`gate_1q`, `gate_2q`}
`qcc_qpu_coherence_seconds`	`type` ∈ {`t1`, `t2`}
`qcc_qpu_last_calibration_timestamp_seconds`	(none)
`qcc_qpu_condition`	`condition`, `status` ∈ {`true`, `false`, `unknown`}

QPU dimensions. These name the physical quantities a quantum engineer reasons about. The executor reads them from the backend's Qiskit `Target` at probe time, and each is a *median* across the device's qubits or qubit pairs, one representative figure per backend. The error and duration metrics share an `operation` dimension:

`gate_1q` a single-qubit gate. On IBM hardware these are the native `sx`, `x`, and `rz`; `rz` is virtual and error-free, so the figure reflects the physical `sx`/`x` quality.

`gate_2q` the two-qubit entangling gate, the operation that builds entanglement and dominates a deep circuit's error budget. It is `cz` on IBM Heron and `ecr` on IBM Eagle, the quantity IBM reports as its median 2Q error.

`readout` the measurement (assignment) error: the chance that reading a qubit returns the wrong classical bit. It appears on the error metric only, because the IBM `Target` reports no measurement duration, so `readout` is absent from the duration metric rather than zero.

Coherence carries a `type` dimension instead:

`t1` energy relaxation: how long a qubit holds its excited state.
`t2` dephasing: how long it holds its phase information.

Both bound how much computation fits before noise takes over. Qiskit reports them in seconds; the CR keeps microseconds (the unit IBM publishes) and the metric converts back to seconds to match its `_seconds` suffix.

QPU identity and conditions. `qcc_qpu_info` is the info-metric anchor: its value is always 1, and its labels hold the static identity, provider, kind, processor family and revision. Every other QPU metric joins to it in PromQL with `group_left` to borrow those labels rather than repeat them on each series, the standard `kube-state-metrics` pattern. `qcc_qpu_condition` follows the same library's Conditions idiom: for each `(qpu, condition)` it emits one row per `status` value (`true`, `false`, `unknown`), exactly one of which is 1, so a dashboard selects on a backend's readiness the way it would for a Pod. The `QPU` resource is cluster-scoped, so no `namespace` label appears.

`Circuit` dimensions. These come in three kinds: identity, decomposition, and timing. Identity labels say which run a series belongs to: every `Circuit` series carries `circuit`, `namespace`, and `uid`, and `qcc_circuit_info` adds the descriptive labels (`mode`, `source_format`, `shots`, `qpu`, `provider_job_id`) together with the algorithm-grouping labels promoted from the resource. A *decomposition* label splits one metric across related parts:

`kind` on `qcc_circuit_transpile_gates`, one of `single_qubit`, `two_qubit`, or `total`: the gate counts after the circuit is transpiled to the backend's native set. `two_qubit` is the count that drives fidelity; `single_qubit` is derived as `total` minus `two_qubit`.
`phase` on the phase-duration metrics, one of `Pending`, `Selecting`, `Submitting`, or `Running`: the wall-clock time spent in each lifecycle stage, taken from the `status.conditions` timestamps. `Submitting` spans transpilation and submission together, which the Conditions vocabulary does not separate.
`bitstring` on `qcc_circuit_result_count`: a measured outcome such as `0000`, valued by how many shots produced it. This is the raw measurement histogram and the input to any fidelity analysis; it adds 2^q series for a q-qubit circuit.

The timing metrics report duration. The phase-duration pair gives the same per-phase numbers in two forms, kept on purpose: `qcc_circuit_phase_duration_seconds` is a

Table 6.6: *`Circuit`-side metrics. All series carry `circuit`, `namespace`, and `uid` as identity labels; only the additional dimensions are listed below.*

Metric	Type	Additional dimensions
qcc_circuit_info	gauge	mode, source_format, shots, qpu, provider_job_id, algorithm, algorithm_version, experiment, run_index, source_sha256
qcc_circuits_total	counter	mode, qpu, phase, reason, provider_job_id
qcc_circuit_phase_duration_seconds	histogram	qpu, phase, provider_job_id
qcc_circuit_phase_duration_seconds_observed	gauge	qpu, phase, provider_job_id
qcc_circuit_usage_seconds	gauge	qpu, provider_job_id
qcc_circuit_transpile_depth	gauge	qpu
qcc_circuit_transpile_gates	gauge	qpu, kind ∈ {single_qubit, two_qubit, total}
qcc_circuit_result_count	gauge	qpu, bitstring

histogram for fleet-wide percentiles (via `histogram_quantile`), while its `_observed` gauge is recomputed from the Condition timestamps on every scrape, so per-Circuit panels survive a controller restart and Prometheus's staleness window. `qcc_circuit_usage_seconds` is the substrate's own billable on-QPU time, taken from the Qiskit Runtime `Job.usage()` handle; it is emitted only when the substrate reports it, so any value in Prometheus marks a real-hardware run, and dividing it by the `Running` duration gives the orchestration overhead.

Cross-boundary handle. `qcc_circuit_info` also carries the cross-boundary identifier that satisfies R4. The provider job id the submission returns is recorded on the `Circuit`'s status and promoted onto `qcc_circuit_info` as the `provider_job_id` label, so a `Circuit` and its job on the backend resolve to each other with ordinary tooling. Given a job id, taken for example from a provider's console or billing export, a PromQL lookup on `provider_job_id` returns the `Circuit`'s full identity record; given a `Circuit`, the same id already sits on its status. The link reads from either side, with no separate trace-context channel. Distributed tracing over OpenTelemetry would extend this but is not a prerequisite.

6.7.3 Algorithm-aware queries

Researchers ask questions like "how does Shor v2 compare to v1 in execution time?" or "show me every Bell-state run in the last hour." These are algorithm-level questions, but Prometheus sees only the metrics QCC emits: there is no algorithm catalogue inside it. QCC closes the gap by carrying the algorithm-grouping labels from the Kubernetes resource into the metric label-space.

When a `Circuit` is submitted with reserved labels such as `qcc.io/algorithm=shor` and `qcc.io/algorithm-version=v2`, the controller promotes them onto the metrics it emits. The identity-bearing labels (algorithm, algorithm version, experiment, run index, source hash) land on the single info-metric `qcc_circuit_info`; the operational metrics (phase durations, transpile shape, result counts) stay low-cardinality and carry only what they need to join back: `uid`, `qpu`, `phase`. A query that needs algorithm context joins the operational series to `qcc_circuit_info` on `uid` with `group_left`, again the `kube-state-metrics` pattern.

Algorithm-level questions therefore resolve through plain PromQL, with no QCC-specific dashboard tooling and no out-of-band metadata lookup.

The architecture developed in this chapter answers R1–R5 through three surfaces. The resource model and the controller–executor split address R1 and R3. The lifecycle phases and the evaluation surface that follows a completed run address R5. The observability primitives, the `qcc.*` metrics specification, the cross-boundary identifier it carries on `qcc_circuit_info`, and the algorithm-aware query convention, address R2 and R4.

7

Implementation and Demonstration

This chapter shows the QCC architecture of Chapter 6 as a working system by walking through a single workload: Shor's algorithm factoring $N = 15$ with $a = 4$, the same circuit whose `ibm_sherbrooke` run opened Chapter 1. The algorithm and parameters are unchanged, but the execution is now carried through a Kubernetes-native control plane whose state, provider boundary, backend quality, and outcome distribution are observable. Following the path an operator takes on a fresh cluster, the chapter brings the platform up, registers backends, draws and runs the circuit, compares it across simulators, executes it on real hardware, reads the results through the shipped dashboards, and finally tunes one transpiler setting to show the two-tier schema absorbing a vendor feature without changing the platform.

7.1 The QCC reference implementation

QCC v1.0.0 is the open-source release that realises the Chapter 6 design: the *controller*, the *executor*, and the `qcc` CLI. The release ships a declarative Kubernetes cluster for local development, along with the required platform stack and the two Grafana dashboards for `QPU` and `Circuit` views. This chapter exercises this release end to end.

7.2 Bringing up the platform

The requirements are modest and entirely standard. QCC needs a Kubernetes cluster to host its controller and executor, and an OpenTelemetry-compatible observability stack to receive their telemetry: a Prometheus-compatible metrics backend, an OpenTelemetry Collector, and Grafana for visualisation. None of these is QCC-specific or modified. The contribution is the `qcc.*` schema that flows through the stack (Section 6.7.2), not the stack itself, so any conformant cluster, local or managed, satisfies them.

The proof-of-concept was evaluated on a local `kind`[1] cluster (kind v0.30.0), backed by Docker through Colima[2] 0.9.1 on Apple Silicon. The observability stack was installed with Helm using the latest stable chart versions available at the time of writing: `kube-prometheus-stack`[3] 85.0.3, including Prometheus Operator v0.90.1, and the OpenTelemetry Collector[4] chart 0.155.0, including collector v0.151.0. QCC itself was deployed from its Helm package.

> **Note**
>
> The provisioning of the Kubernetes cluster and the installation of the supporting observability stack are outside the scope of this work. The evaluation assumes an existing Kubernetes environment.

7.3 Registering a backend

A QPU enters the platform through a Kubernetes manifest, while discoverable backend details are filled in during reconciliation. The operator provides only the fields needed to identify the backend, and the platform records the remaining metadata: qubit count, basis gates, coupling-map size, calibration vintage, and per-operation error medians (Section 6.5.3). A representative catalogue ships under `config/samples/qpu/`. In the following step, we will register a `fake-fez` QPU. The relevant manifest `fake-fez.yaml` is shown in (Figure 7.1).

```
apiVersion: qcc.io/v1alpha1
kind: QPU
metadata:
  name: fake-fez
  labels:
    app.kubernetes.io/name: quantum-circuit-controller
    app.kubernetes.io/managed-by: kustomize
    qcc.io/provider: local
    qcc.io/family: heron-r2
spec:
  provider: local
  kind: simulator
```

Figure 7.1: *The `fake-fez` QPU manifest. The operator declares only the provider and the kind and everything else is left to the QCC probe.*

Applying the manifest with `kubectl apply -f fake-fez.yaml` creates a QPU resource. After the QCC operator runs the periodic probes, each QPU's discovered status is being populated to its CRD `status` field.

[1] https://kind.sigs.k8s.io
[2] https://github.com/abiosoft/colima
[3] https://github.com/prometheus-community/helm-charts
[4] https://github.com/open-telemetry/opentelemetry-helm-charts

```
$ kubectl get qpu fake-fez -o yaml
```

```
apiVersion: qcc.io/v1alpha1
kind: QPU
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"qcc.io/v1alpha1","kind":"QPU","metadata":{"annotations":{},"labels":
  creationTimestamp: "2026-05-24T09:30:25Z"
  generation: 1
  labels:
    app.kubernetes.io/managed-by: kustomize
    app.kubernetes.io/name: quantum-circuit-controller
    qcc.io/family: heron-r2
    qcc.io/provider: local
  name: fake-fez
  resourceVersion: "792160"
  uid: a30b5e9d-ea80-45a6-9297-82a1e8b79fd5
spec:
  kind: simulator
  provider: local
status:
  availability: Available
  basisGates:
  - cz
  - delay
  - for_loop
  - id
  - if_else
  - measure
  - reset
  - rz
  - switch_case
  - sx
  - x
  coherenceMedians:
    t1Micros: 144.85526609546145
    t2Micros: 87.95068681159161
  conditions:
  - lastTransitionTime: "2026-05-24T09:30:25Z"
    message: local provider "fake_fez" is Available without probe
    observedGeneration: 1
    reason: ProviderProbeOK
    status: "True"
    type: Ready
  - lastTransitionTime: "2026-05-24T09:30:25Z"
    message: local providers have no remote calibration; freshness is degenerate
    observedGeneration: 1
    reason: CalibrationRefreshed
    status: "True"
    type: MetadataFresh
  couplingEdges: 352
  dtSeconds: 4e-09
  errorMedians:
    readout: 0.007568359375
    singleQubit: 0.000198265882920662
    twoQubit: 0.0039033828523107883
  instructionDurationMedians:
    singleQubitSeconds: 2.4e-08
    twoQubitSeconds: 8.4e-08
  lastCalibrationTime: "2025-02-26T20:16:25Z"
  observedGeneration: 1
  processor:
    family: Heron
    revision: "2"
  qubits: 156
```

Figure 7.2: *The probe-populated* `fake-fez` *QPU status.*

Listing all registered QPUs:

```
$ kubectl get qpu
```

```
NAME                 PROVIDER   KIND        QUBITS   AVAILABLE   AGE
aer-statevector      local      simulator   28       Available   7d22h
fake-belem           local      simulator   5        Available   9d
fake-brisbane        local      simulator   127      Available   9d
fake-brisbane-test   local      simulator   127      Available   6d20h
fake-fez             local      simulator   156      Available   2m1s
fake-kyoto           local      simulator   127      Available   9d
fake-marrakesh       local      simulator   156      Available   9d
fake-osaka           local      simulator   127      Available   9d
fake-sherbrooke      local      simulator   127      Available   9d
fake-torino          local      simulator   133      Available   9d
ibm-fez              ibm        hardware    156      Available   7d22h
ibm-kingston         ibm        hardware    156      Available   7d22h
ibm-marrakesh        ibm        hardware    156      Available   7d22h
```

Figure 7.3: *The deployed QPU registry of three live IBM backends alongside simulators.*

7.4 The workload: Shor's algorithm

The `mode=draw` lifecycle renders the circuit without consuming any QPU time, and during Quantum algorithm development, is one of the first steps an engineer does, to have an early feedback of the circuit's structure and behavior. Instead of using a Qiskit function, we are using the QCC CLI to draw the circuit:

```
$ qcc draw examples/thesis/algorithms/shor.py
```

```
Q•C•C  1.0.0

▸ loading shor.py · qiskit · 0 ms
▯ rendering▯ rendering✓ rendered · 259ms
```

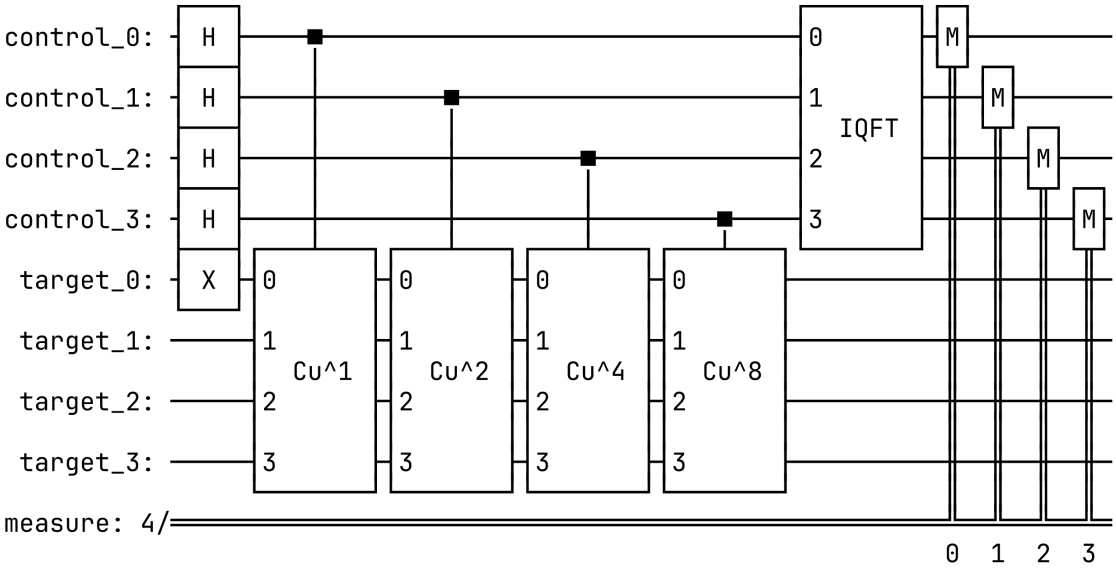


Figure 7.4: *Shor's $N = 15$ circuit rendered through mode=draw.*

7.5 Executing a circuit

Drawing renders the circuit and the next step is to run it. The simplest execution targets the ideal Aer simulator to produce the noise-free QPU behaviour. The `mode=run` names an explicit backend and an optional set of reserved `experiment` and `version` labels for quick result comparison, among other params.

```
$ qcc run examples/thesis/algorithms/shor.py \
    --algorithm shor --experiment thesis \
    --version v1 --backend aer-statevector
```

```
Q•C•C  1.0.0

▸ loading shor.py · qiskit · 0 ms
▸ submitted default/shor-lpdb7
⠋ waiting for selection⠙ waiting for selection⠹ waiting for selection⠸
✓ queued · job aer-eccea297-b1f6-49e9-a044-f50a5225474e
✓ completed · 510ms

results

  ✓ shor-lpdb7 · run · on aer-statevector succeeded

  setup
    phase    Succeeded
    source   qiskit
    shots    1024
    job      aer-eccea297-b1f6-49e9-a044-f50a5225474e

  backend
    name  aer-statevector  (28q · 87 basis gates)

  circuit
    transpiled  depth 506 · 665 gates · 288 2Q

  results

    0000  ████████████████████████  517
    1000  ███████████████████████▒  507

▸ qasm qcc get circuit shor-lpdb7 --qasm
```

Figure 7.5: *Executing the circuit on the ideal Aer simulator, which produces Circuit `shor-lpdb7`.*

The `qcc` output (Figure 7.5) shows that the run completed in 510 ms after transpilation to a circuit of depth 506. On the ideal simulator, all 1024 shots fall on the two expected period bitstrings, `0000` and `1000`, with counts of 517 and 507. The small difference between them is sampling variation.

The rest of the chapter compares circuit runs using the *correct-period mass* metric. For this fixed Shor instance, the period information is carried by the two expected bitstrings, `0000` and `1000`. The metric adds the counts for those two outcomes and divides

by the total number of shots:

$$\text{correct-period mass} = \frac{\text{count}[0000] + \text{count}[1000]}{\text{shots}} \tag{7.1}$$

For the ideal simulator this gives $(517 + 507)/1024 = 1.0$, so the period signal accounts for all observed shots. On noisy simulators and real hardware, probability mass can spread into the other fourteen four-bit outcomes. A fully uniform distribution would put only two of the sixteen possible outcomes in the period set, giving $2/16 = 12.5\%$.

Every run is a Custom Resource defined by QCC's `Circuit` CRD and inspected like any other Kubernetes object. The full resource shows the shape the controller maintains. During `qcc run`, the CLI parses `shor.py` and writes the declared execution intent into `spec`, including the reserved `qcc.io/*` labels and the selected `aer-statevector` backend (Figure 7.6). As the lifecycle progresses, the controller assembles `status` with the `conditions` array, the selected backend, the measured counts, and the post-compilation transpile shape (Figure 7.7).

```
$ kubectl get circuits shor-lpdb7 -o yaml
```

```
apiVersion: qcc.io/v1alpha1
kind: Circuit
metadata:
  creationTimestamp: "2026-05-24T09:44:25Z"
  generation: 1
  labels:
    qcc.io/algorithm: shor
    qcc.io/algorithm-version: v1
    qcc.io/experiment: thesis
    qcc.io/run-index: "1"
    qcc.io/source-sha256: a721bd0a0a24dad6
  name: shor-lpdb7
  namespace: default
  resourceVersion: "793723"
  uid: 0a5d99cc-9486-4678-a333-54e63a845475
spec:
  backendSelector:
    backendName: aer-statevector
  mode: run
  shots: 1024
  source:
    body: |
      """
      Shor's algorithm — quantum period-finding for N=15, a=4.

section suppressed

      the same one whose ibm_brisbane / ibm_sherbrooke runs open Chapter 1, kept
```

Figure 7.6: *The Circuit resource, top: the declared* `spec`.

```
status:
  conditions:
  - lastTransitionTime: "2026-05-24T09:44:25Z"
    message: Circuit accepted by controller
    observedGeneration: 1
    reason: CircuitAccepted
    status: "True"
    type: Accepted
  - lastTransitionTime: "2026-05-24T09:44:25Z"
    message: Static validation succeeded
    observedGeneration: 1
    reason: CircuitValidated
    status: "True"
    type: Validated
  - lastTransitionTime: "2026-05-24T09:44:25Z"
    message: Selected QPU "aer-statevector" (provider=local, backend=aer_statevector)
    observedGeneration: 1
    reason: BackendSelected
    status: "True"
    type: Selected
  - lastTransitionTime: "2026-05-24T09:44:25Z"
    message: Submitted to aer_statevector (taskID=aer-eccea297-b1f6-49e9-a044-f50a5225474e)
    observedGeneration: 1
    reason: ProviderSubmitted
    status: "True"
    type: Submitted
  - lastTransitionTime: "2026-05-24T09:44:25Z"
    message: Execution completed with 2 outcome(s)
    observedGeneration: 1
    reason: ExecutionCompleted
    status: "True"
    type: Completed
  convertedRef:
    name: shor-lpdb7-converted
  observedGeneration: 1
  phase: Succeeded
  providerJobId: aer-eccea297-b1f6-49e9-a044-f50a5225474e
  results:
    "0000": 517
    "1000": 507
  selectedQPU: aer-statevector
  selectionSummary:
    candidates: 1
    score: 'n/a (M1: first-match policy; M2 adds calibration-aware scoring)'
    selected: aer-statevector
  transpile:
    depth: 506
    totalGates: 665
    twoQubitGates: 288
```

Figure 7.7: *The same resource, bottom: the assembled* `status`*.*

This single run establishes both halves of the operator's basic loop, submit a circuit, then inspect its result and its full resource through standard tooling, and leaves a noise-free baseline on record. The next section runs the same circuit across every available simulator at once, to see how that baseline degrades by substrate before any hardware time is spent.

7.6 Comparing the simulators

Before spending hardware time, the operator can run the same circuit across the registered simulator backends with `--performance-test` flag. The command fans out to all Aer simulators and to the `fake_*` snapshots in the QPU registry, giving a first comparison of how the same workload behaves across available substrates.

```
$ qcc run examples/thesis/algorithms/shor.py \
    --performance-test \
    --algorithm shor \
    --version v1 \
    --experiment thesis-perf-test
```

The flag creates one `Circuit` per available QPU simulator. All generated Circuits share the same source body and the same `qcc.io/experiment` label, which can be later used for performance analysis. This turns the simulator sweep into a single labelled experiment rather than a set of unrelated runs.

```
Q·C·C  1.0.0

▸ loading shor.py · qiskit · 0 ms
▸ performance test · 10 candidates algorithm=shor · experiment=thesis-perf-test
▸ submitted default/shor-zcztt → aer-statevector
▸ submitted default/shor-n7mbm → fake-belem
▸ submitted default/shor-kg7nr → fake-brisbane
▸ submitted default/shor-xnrck → fake-brisbane-test
▸ submitted default/shor-m4n5v → fake-fez
▸ submitted default/shor-s8626 → fake-kyoto
▸ submitted default/shor-qpp8j → fake-marrakesh
▸ submitted default/shor-ggvf5 → fake-osaka
▸ submitted default/shor-4x992 → fake-sherbrooke
▸ submitted default/shor-rcjm9 → fake-torino
▸ completed shor-zcztt on aer-statevector · 1.395s
× failed · shor-n7mbm on fake-belem TranspilationFailed
▸ completed shor-kg7nr on fake-brisbane · 3.186s
▸ completed shor-xnrck on fake-brisbane-test · 3.783s
▸ completed shor-m4n5v on fake-fez · 9.181s
▸ completed shor-s8626 on fake-kyoto · 11.779s
▸ completed shor-qpp8j on fake-marrakesh · 16.776s
▸ completed shor-ggvf5 on fake-osaka · 18.375s
▸ completed shor-4x992 on fake-sherbrooke · 19.579s
▸ completed shor-rcjm9 on fake-torino · 23.305s

results · thesis-perf-test

  BACKEND            PHASE                        DEPTH  1Q    2Q   TOTAL  TOP OUTCOMES                TIME     JOB
  aer-statevector    Succeeded                    506    377   288  665    0000:520 1000:504           1.395s   aer-1a7f54b2-
  fake-belem         FAILED · TranspilationFailed –      –     –    –      –                           1.79s    –
  fake-brisbane      Succeeded                    1785   2358  477  2835   1000:124 0000:119 0001:69   3.186s   aer-ed9b7cef-
  fake-brisbane-test Succeeded                    1811   2348  474  2822   1000:155 0000:145 1111:86   3.783s   aer-87f1dbfe-
  fake-fez           Succeeded                    2062   1983  483  2466   0000:236 1000:217 1001:77   9.181s   aer-ad1e59db-
  fake-kyoto         Succeeded                    1774   2315  474  2789   0000:81 1100:73 1111:72     11.779s  aer-f35a7344-
  fake-marrakesh     Succeeded                    2060   1995  483  2478   0000:289 1000:239 1001:66   16.776s  aer-d5b476a9-
  fake-osaka         Succeeded                    1811   2348  474  2822   0000:125 1000:115 0001:92   18.375s  aer-7251fceb-
  fake-sherbrooke    Succeeded                    1789   2348  474  2822   0000:103 1001:83 1000:75    19.579s  aer-e2faf5c6-
  fake-torino        Succeeded                    2048   1967  474  2441   0000:203 1000:185 1001:69   23.305s  aer-194514a9-

▸ dashboard /d/qcc-circuit?var-algorithm=shor&var-experiment=thesis-perf-test
```

Figure 7.8: *The `--performance-test` fan-out: one Circuit per available simulator, collected into a single cross-substrate table of transpiled depth, gate counts, and top measured bitstrings. The trailing line is a Grafana deep-link pre-loaded with the experiment filter.*

The fan-out also surfaces capability mismatches as first-class results: the 5-qubit `fake-belem` candidate returns `TranspilationFailed`, because the circuit does not fit a device that small, recorded as a terminal phase rather than a silent omission.

Read against the noise-free Aer reference of Section 7.5, the fan-out (Figure 7.8) separates the processor families more clearly than the individual backends. The Heron r2 snapshots preserve the period-finding structure, while the Eagle r3 snapshots wash it into noise. In the table, `fake-fez` (Heron r2) keeps 0000 and 1000 as the two dominant outcomes. By contrast, `fake-kyoto` (Eagle r3) spreads probability mass far enough that 1000 drops out of the top three and non-period bitstrings take its place.

7.7 Running on real hardware

The survey of Section 7.6 directed the hardware budget at the Heron r2 family; which Heron r2 device performs best is the question only live hardware can answer. This section answers it. The simulator perf-test showed that substrate-comparative evaluation is observable through telemetry on idealised substrates. The open question is whether the same surface holds against the operational realities of real hardware: vendor queues, calibration drift, and same-vendor same-day fidelity variance. The same Shor source body is submitted to all three registered Heron r2 backends.

The vendor queue is the most prominent operational difference. A submitted job sits in IBM Quantum's queue for minutes to hours before executing, and waiting synchronously is operationally unworkable. QCC's `--detach` flag exits the CLI as soon as the provider job is queued; the controller continues polling in the background, and the engineer returns to inspect the result asynchronously.

Qiskit's transpiler provides four optimisation presets, numbered 0 to 3. Level 3 applies the most aggressive optimisation, while level 1 is the default. Every real-hardware run in this section uses that default setting, with no explicit `optimizationLevel` in the `Circuit` spec. This section submits the same Shor workload to the three registered real Heron r2 backends at IBM Quantum. Each run uses 1024 shots and the default Qiskit transpilation preset, with no explicit `optimizationLevel` set.

```
$ qcc run examples/thesis/algorithms/shor.py \
    --algorithm shor --experiment thesis --version v1 \
    --backend ibm-fez --detach

$ qcc run examples/thesis/algorithms/shor.py \
    --algorithm shor --experiment thesis --version v1 \
    --backend ibm-kingston --detach

$ qcc run examples/thesis/algorithms/shor.py \
    --algorithm shor --experiment thesis --version v1 \
    --backend ibm-marrakesh --detach
```

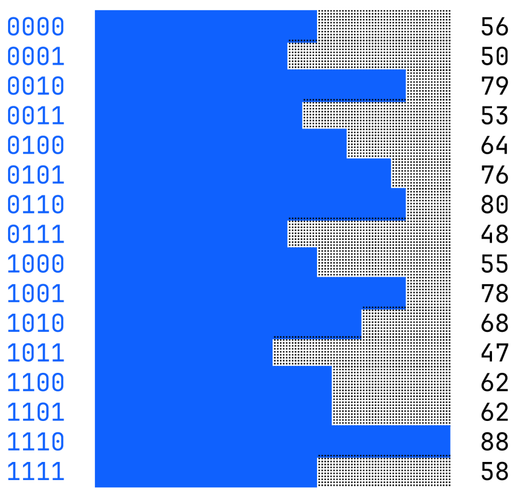
With `--detach`, the CLI exits as soon as the provider job is queued. It prints the Circuit name, UID, and provider job ID, so the run can be followed later through QCC or the vendor console. Figure 7.9 shows the detached submission for `ibm-kingston`.

```
Q•C•C  1.0.0

▸ loading shor.py · qiskit · 0 ms
▸ submitted default/shor-2vv42
✓ targeting ibm-kingston
✓ queued · job d89dinp789is73935r0g

▸ detached check progress with: qcc get circuit shor-2vv42
```

Figure 7.9: *Terminal output of a `--detach` submission against `ibm-kingston`. The CLI prints the Circuit `shor-2vv42` and the provider job `d89dinp789is73935r0g`.*

Each run is inspected through the result card rendered by `qcc get circuit <the returned Circuit ID from the CLI e.g. shor-wffxg>`.

```
Q•C•C  1.0.0

shor-wffxg

  ✓ shor-wffxg · run · on ibm-fez · degraded signal (error exposure ≈ 2.0)

  setup
    phase    Succeeded
    source   qiskit
    shots    1024
    job      d89dgpqs46sc73fafp3g

  backend
    name          ibm-fez  (156q · 352 edges · 9 basis gates)
    processor     Heron r2
    calibrated    2026-05-16 (8d ago)
    gate errors   1Q 3.00e-04 · 2Q 2.81e-03 · readout 2.15e-02
    coherence     T1 102 μs · T2 90 μs
    cycle time    dt = 4 ns

  circuit
    transpiled      depth 2062 · 2466 gates · 483 2Q
    exec time       ~140.22 μs (critical-path estimate)
    error exposure  ≈ 2.0 events/shot (approaching gate-error budget)
    fidelity bound  P(no gate error) ≈ 0.14  (excludes readout & coherence)

  results
```



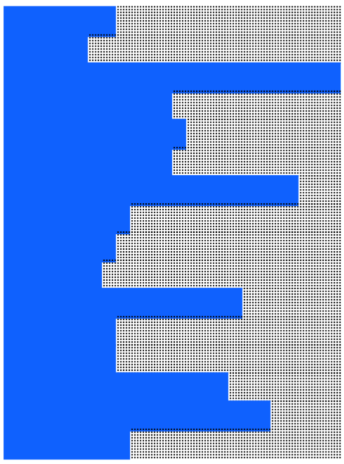

```
▸ qasm qcc get circuit shor-wffxg --qasm
```

Figure 7.10: *The terminal result card for the* `ibm-fez` *submission (Circuit* `shor-wffxg`*).*

```
Q•C•C  1.0.0

shor-2vv42

  ▸ shor-2vv42 · run · on ibm-kingston · phase=Running

  setup
    phase    Running
    source   qiskit
    shots    1024
    job      d89dinp789is73935r0g

  backend
    name          ibm-kingston  (156q · 352 edges · 10 basis gates)
    processor     Heron r2
    calibrated    2026-05-16 (8d ago)
    gate errors   1Q 2.74e-04 · 2Q 2.03e-03 · readout 1.65e-02
    coherence     T1 175 µs · T2 117 µs
    cycle time    dt = 4 ns

  circuit
    transpiled       depth 2048 · 2441 gates · 474 2Q
    exec time        ~139.26 µs (critical-path estimate)
    error exposure   ≈ 1.5 events/shot (approaching gate-error budget)
    fidelity bound   P(no gate error) ≈ 0.22  (excludes readout & coherence)

▸ qasm qcc get circuit shor-2vv42 --qasm
```

Figure 7.11: *The `ibm-kingston` card (Circuit `shor-2vv42`), captured mid-execution.*

```
Q•C•C  1.0.0

shor-2vv42

  ✓ shor-2vv42 · run · on ibm-kingston · degraded signal (error exposure ≈ 1.5)

  setup
    phase    Succeeded
    source   qiskit
    job      d89dinp789is73935r0g

  backend
    name         ibm-kingston  (156q · 352 edges · 10 basis gates)
    processor    Heron r2
    calibrated   2026-05-16 (15d ago)
    gate errors  1Q 2.74e-04 · 2Q 2.03e-03 · readout 1.65e-02
    coherence    T1 175 µs · T2 117 µs
    cycle time   dt = 4 ns

  circuit
    transpiled      depth 2048 · 2441 gates · 474 2Q
    exec time       ~139.26 µs (critical-path estimate)
    error exposure  ≈ 1.5 events/shot (approaching gate-error budget)
    fidelity bound  P(no gate error) ≈ 0.22  (excludes readout & coherence)

  results
```

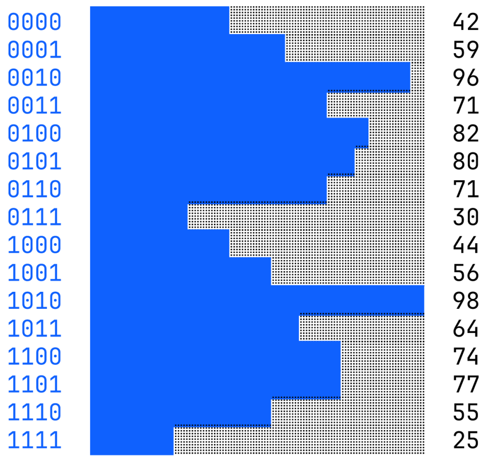


0000	41
0001	34
0010	120
0011	61
0100	66
0101	64
0110	108
0111	46
1000	41
1001	39
1010	88
1011	44
1100	44
1101	84
1110	97
1111	47

```
▸ qasm qcc get circuit shor-2vv42 --qasm
```

Figure 7.12: *The terminal card for the same* `ibm-kingston` *submission (Circuit* `shor-2vv42`*). The period states* `0000` *and* `1000` *hold 41 counts each, giving* $(41 + 41)/1024 \approx 8.0\%$ *correct-period mass.*

```
Q•C•C  1.0.0

shor-wr9ds

  ✓ shor-wr9ds · run · on ibm-marrakesh · degraded signal (error exposure ≈ 1.9)

  setup
    phase    Succeeded
    source   qiskit
    shots    1024
    job      d89dir1789is73935r4g

  backend
    name          ibm-marrakesh  (156q · 352 edges · 9 basis gates)
    processor     Heron r2
    calibrated    2026-05-16 (8d ago)
    gate errors   1Q 2.92e-04 · 2Q 2.76e-03 · readout 1.48e-02
    coherence     T1 196 μs · T2 99 μs
    cycle time    dt = 4 ns

  circuit
    transpiled      depth 2048 · 2441 gates · 474 2Q
    exec time       ~139.26 μs (critical-path estimate)
    error exposure  ≈ 1.9 events/shot (approaching gate-error budget)
    fidelity bound  P(no gate error) ≈ 0.15  (excludes readout & coherence)

  results
```




```
▸ qasm qcc get circuit shor-wr9ds --qasm
```

Figure 7.13: *The terminal result card for the* `ibm-marrakesh` *submission (Circuit* `shor-wr9ds`*).*

On `ibm-fez`, the circuit transpiles to depth 2062 and returns $(56+55)/1024 \approx 10.8\%$ correct-period mass (Figure 7.10). On `ibm-marrakesh`, the depth is 2048 and the correct-period mass is $(42+44)/1024 \approx 8.4\%$ (Figure 7.13). The `ibm-kingston` card is shown twice. The first capture shows the run while it is still in `Running` phase in IBM Quantum space, demonstrating live lifecycle visibility (Figure 7.11). The terminal card then shows `Succeeded`, with $(41+41)/1024 \approx 8.0\%$ correct-period mass (Figure 7.12).

After the real-hardware sweep, the full set of Circuits in the cluster makes the experiment visible at a glance. The variance between `ibm-fez` and `ibm-marrakesh`, observed at the same shot count and optimisation level, is an operational property of the workload on the available hardware. QCC surfaces that variance through one telemetry sur-

face for every backend rather than through vendor-specific dashboards. An engineer asking *which Heron r2 backend is least degraded for this circuit* can resolve the question through standard `kubectl get` output and PromQL queries against `qcc_circuit_*`, without using separate tooling per substrate.

The cross-substrate variance illustrates why backend quality cannot be reduced to a single calibration number. If coherence time alone were a reliable predictor for this workload, the longest-T_1 backend would produce the strongest period signal. The observed runs show the opposite. At 1024 shots, the Heron r2 backends rank by T_1 as `ibm-marrakesh` (~196 µs), `ibm-kingston` (~175 µs), and `ibm-fez` (~102 µs). The correct-period mass does not follow that order. `ibm-fez`, the shortest-T_1 device, returns the highest mass at 10.8 %, while `ibm-marrakesh` returns 8.4 % and `ibm-kingston` returns 8.0 %.

This does not mean that `ibm-fez` produced a successful run. All three values remain at or below the 12.5 % uniform noise floor defined in Section 7.4. Under the default level-1 transpilation, the circuit is therefore degraded on all three backends. The useful finding is narrower: for this circuit and transpilation setting, the least-degraded backend is not the one that a simple T_1 ranking would select. The signal is recovered later through the transpiler tuning of Section 7.9.

Listing every Circuit after the sweep in a single view (Figure 7.14):

```
$ qcc get circuits
```

```
NAME        PHASE      MODE  QPU                ALGORITHM  VER  RUN  JOB                                    AGE
shor-2vv42  Running    run   ibm-kingston       shor       v1   5    d89dinp789is73935r0g                   2m
shor-4x992  Succeeded  run   fake-sherbrooke    shor       v1   9    aer-e2faf5c6-4265-4556-8613-fa31e193dd0e  38m
shor-ggvf5  Succeeded  run   fake-osaka         shor       v1   8    aer-7251fceb-149f-429d-af2a-ebecd17efb50  38m
shor-kg7nr  Succeeded  run   fake-brisbane      shor       v1   3    aer-ed9b7cef-2688-4611-9028-e02e7696e191  38m
shor-lpdb7  Succeeded  run   aer-statevector    shor       v1   1    aer-eccea297-b1f6-49e9-a044-f50a5225474e  1h
shor-m4n5v  Succeeded  run   fake-fez           shor       v1   5    aer-ad1e59db-e714-4c34-9b8a-d02794f7a590  38m
shor-n7mbm  Failed     run   fake-belem         shor       v1   2    –                                      38m
shor-qpp8j  Succeeded  run   fake-marrakesh     shor       v1   7    aer-d5b476a9-9ea7-461e-b3d6-14c1d74bd6cc  38m
shor-rcjm9  Succeeded  run   fake-torino        shor       v1   10   aer-194514a9-8e48-4c89-b380-1407b7775a55  38m
shor-s8626  Succeeded  run   fake-kyoto         shor       v1   6    aer-f35a7344-3f6f-4c2d-8b04-5c582a32beff  38m
shor-wffxg  Succeeded  run   ibm-fez            shor       v1   4    d89dgpqs46sc73fafp3g                   7m
shor-wr9ds  Succeeded  run   ibm-marrakesh      shor       v1   6    d89dir1789is73935r4g                   2m
shor-xbbkc  Succeeded  run   ibm-fez            shor       v1   3    d89dg5gp0eas73dofmm0                   8m
shor-xd76l  Succeeded  run   ibm-fez            shor       v1   2    d89dfn8p0eas73dofm30                   9m
shor-xnrck  Succeeded  run   fake-brisbane-test shor       v1   4    aer-87f1dbfe-59ac-4c6d-827e-43d18313299d  38m
shor-zcztt  Succeeded  run   aer-statevector    shor       v1   1    aer-1a7f54b2-f7e9-422a-8768-89a2228bca22  38m
```

Figure 7.14: *The list of Circuits in the cluster after the real-hardware sweep. Every Circuit carries its algorithm label, experiment label, selected QPU, `providerJobId`, and terminal phase, so the state of the experiment reads from a single `kubectl get`.*

7.8 Observing the results

The `qcc.*` metric specification (Section 6.7.2) defines the schema used for cross-boundary observability. After the QPU registration and Circuit executions, the telemetry pipeline exposes these signals as standard Prometheus metrics Figure 7.15, Figure 7.16.

The two prototype QCC Grafana dashboards render the same schema for operators and engineers. The QPU dashboard focuses on substrate health and calibration trends across the registered backend fleet, while the `Circuit` dashboard focuses on individual circuit executions, lifecycle state, transpilation shape, and observed outcomes.

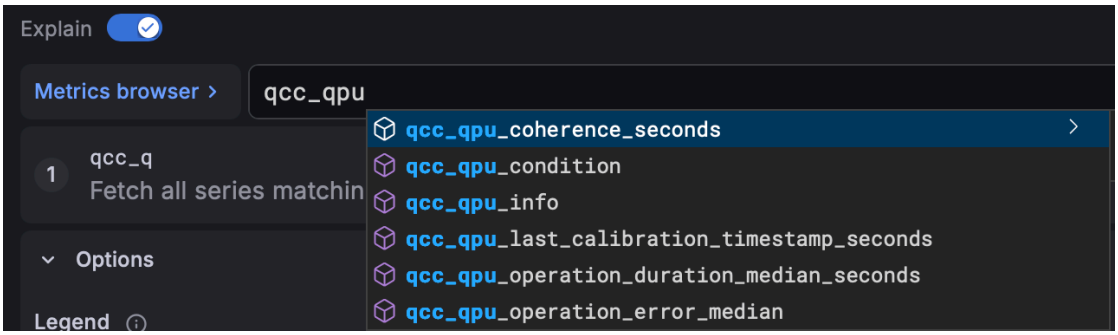


Figure 7.15: *The `QPU` metrics.*

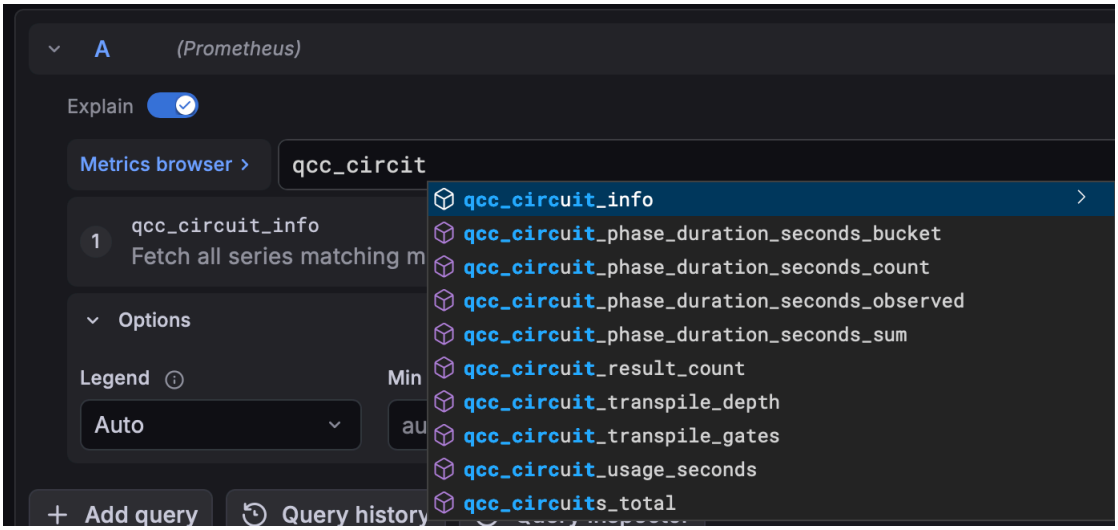


Figure 7.16: *The `Circuit` metrics.*

7.8.1 QPU dashboard: an experimental substrate-monitoring view

The QPU dashboard is an experimental prototype for monitoring quantum backends. It draws on established resource-monitoring methods from classical systems practice: the USE method (Utilisation, Saturation, Errors) (Gregg, 2014), the request-oriented RED method (Rate, Errors, Duration), and the SRE golden signals (Beyer et al., 2016).The grouping should be read as a design choice for this prototype, not as a settled methodology.

Saturation asks whether the backend can currently accept work. The panel group shows the availability state of each QPU, a Ready-over-Registered snapshot for the fleet, and availability over time. In classical systems, saturation is often measured through queue length, throughput, or pressure against capacity. In this prototype, the corresponding quantum question is simpler and more operational: whether each registered backend is available for submission.

Utilisation asks whether the backend is fresh enough to be useful. This requires a quantum-specific adaptation. In classical systems, utilisation usually measures how much of a resource's capacity is being consumed at a point in time (time-based and capacity-based). For a quantum backend, the useful operational signal is not only cur-

rent demand, but also calibration freshness, because backend quality can drift between calibrations. The dashboard therefore uses calibration age as a substrate-freshness view, shown together with availability in Figure 7.17.

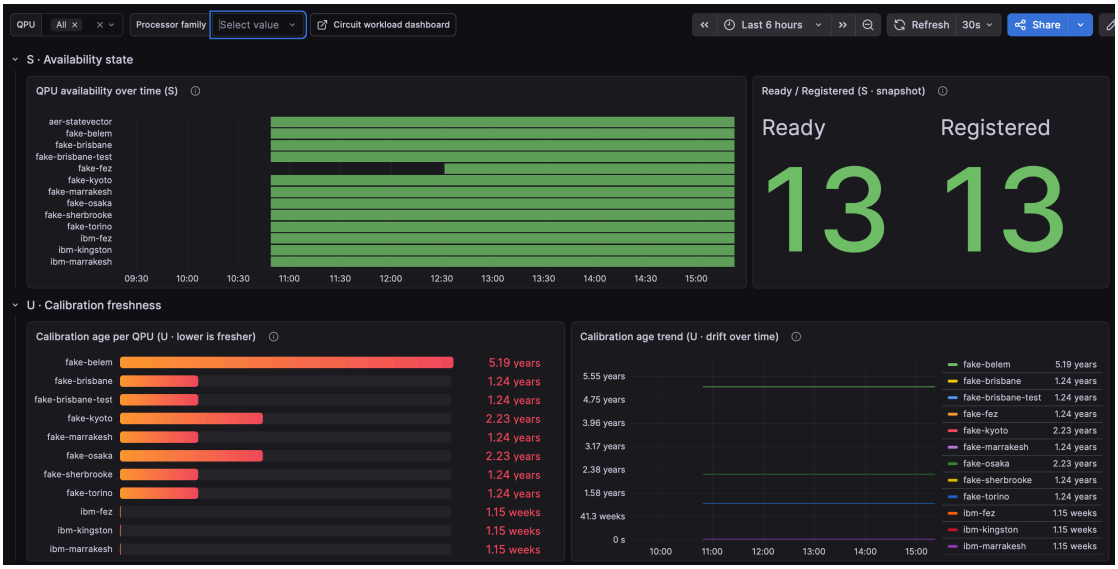


Figure 7.17: *The QPU dashboard's Saturation and Utilisation sections: per-QPU availability state, ready-over-registered snapshot, and calibration freshness.*

The Errors axis shows the backend's noise profile. The panels report per-operation gate and readout error medians, together with coherence times for T_1 relaxation and T_2 dephasing. A family-comparison row groups QPUs by processor family. In Figure 7.18, the registered Heron r2 QPUs show a median two-qubit error of 0.32 %, while the registered Eagle-family snapshots show 20.60 %. This is the same generation-level contrast seen in the simulator sweep, now visible from the fleet dashboard.

Figure 7.18: *The QPU dashboard's Errors section: per-operation gate and readout error medians, T_1 and T_2 coherence, and a family-comparison row grouping QPUs by processor family.*

7.8.2 Circuit dashboard: per-instance drilldown

The Circuit dashboard pivots follows the primitives of RED monitoring method as a characterized workload. Its panels are organised around circuit performance dimensions, trying to explain the circuit results and execution internals Figure 7.19.

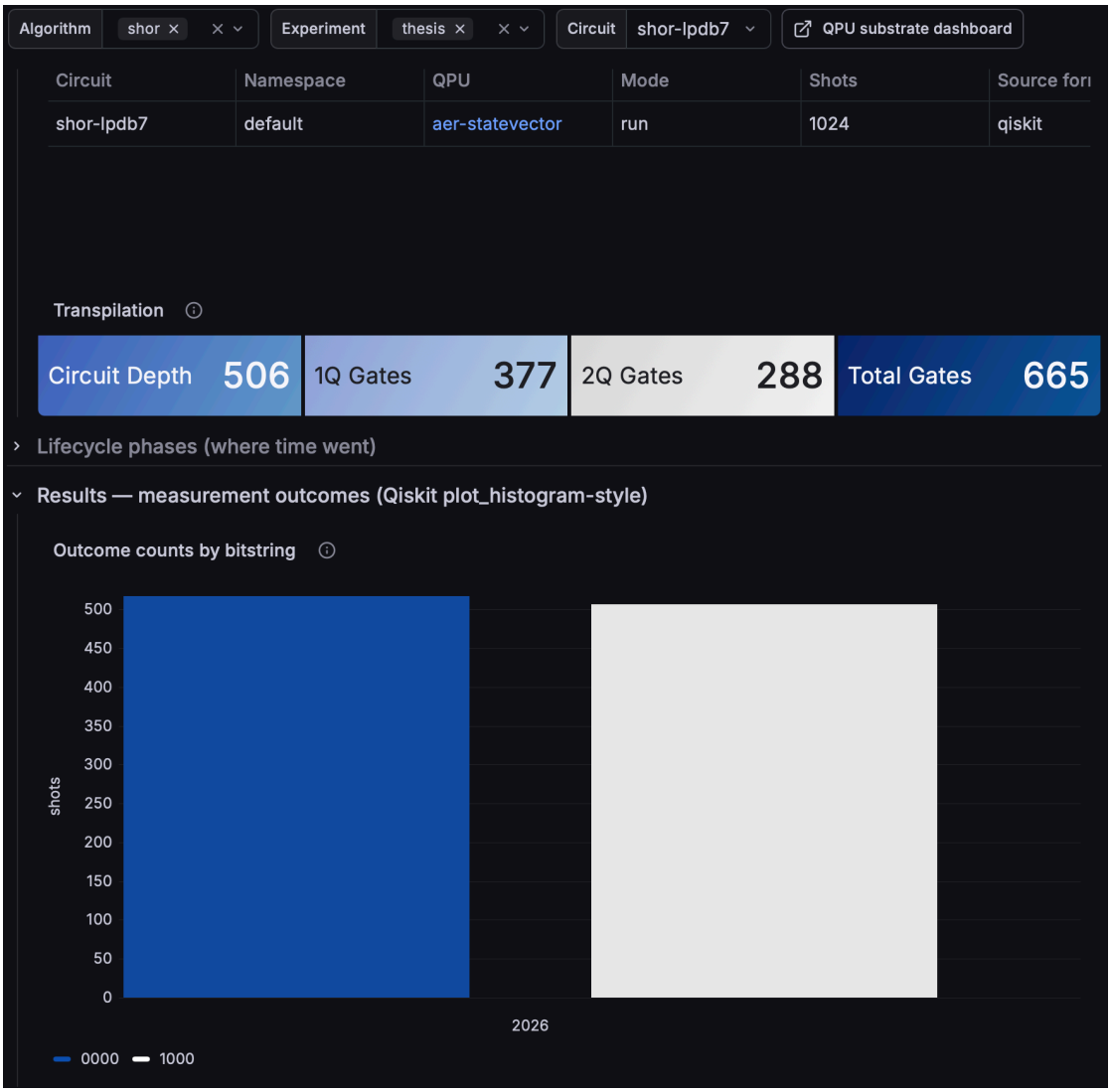


Figure 7.19: *The Circuit dashboard for one of the executed circuits, with the label* `experiment=thesis`.

The dashboard pivot shows the value of the label model. The `$experiment` template variable filters every panel to the Circuits that share the same experiment label. This lets an operator or engineer to compare substrate behaviour side by side without writing custom PromQL queries. (Figure 7.20, Figure 7.21, Figure 7.22).

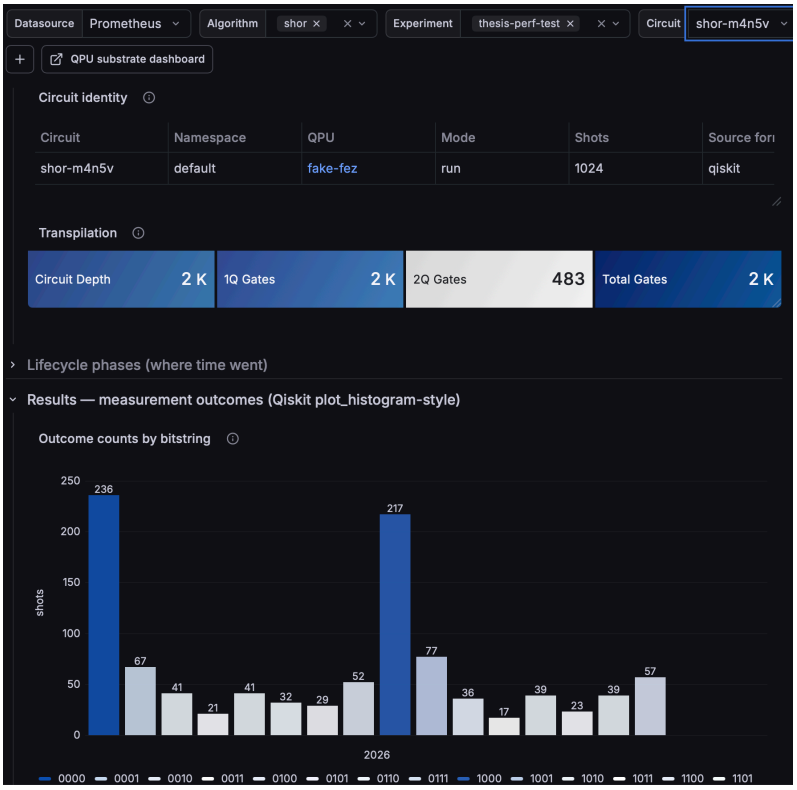


Figure 7.20: *The Circuit dashboard for the perf-test execution on* `fake-fez`.

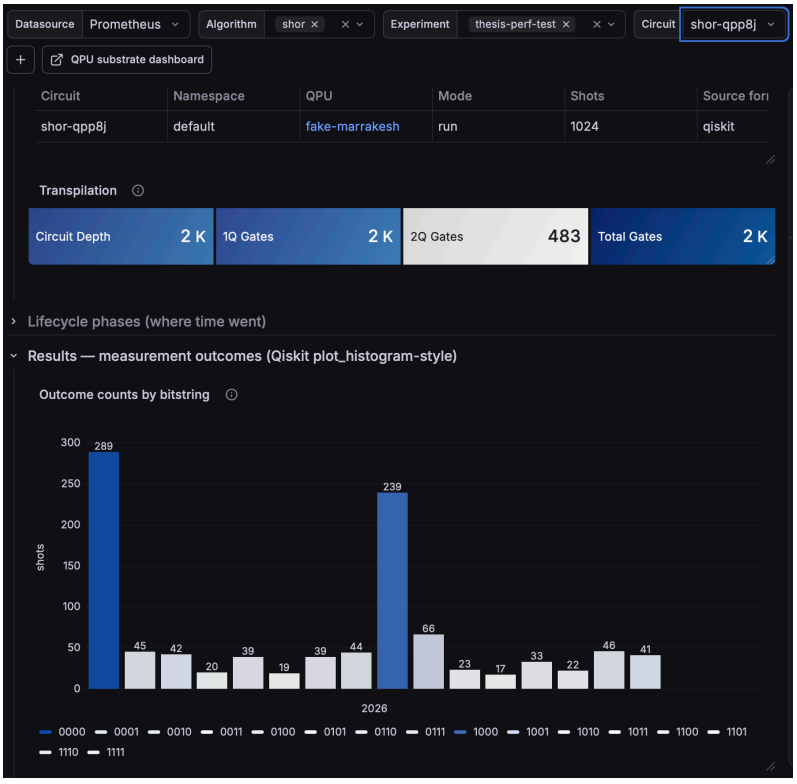


Figure 7.21: *The same dashboard pivoted on the* `fake-marrakesh` *variant. Substrate-comparative reading reduces to changing one template-variable selector.*

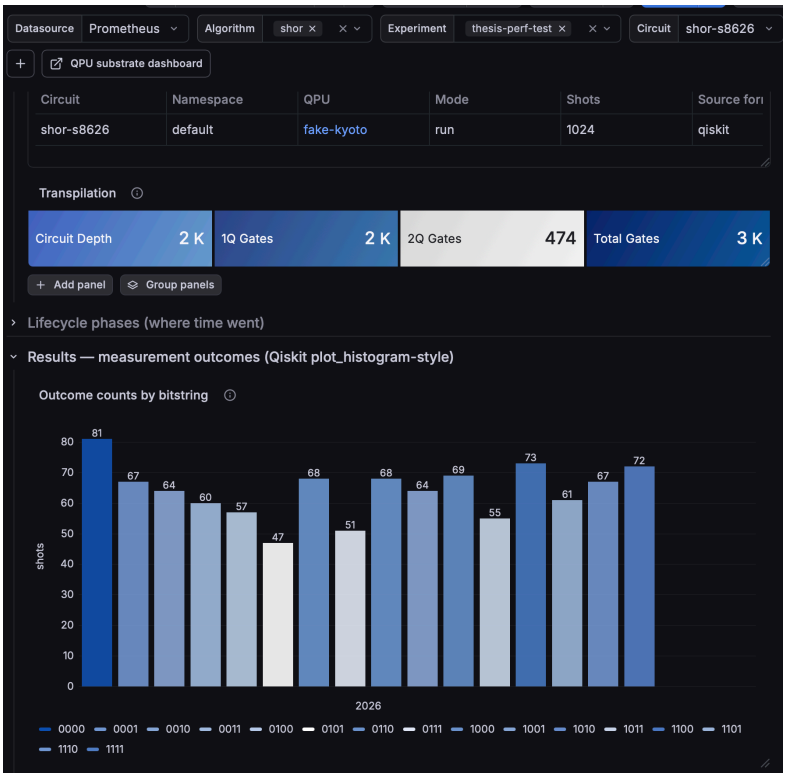


Figure 7.22: *The same dashboard pivoted on* `fake-kyoto` *with Eagle r3.*

7.8.3 Cross-boundary correlation: linking to IBM Quantum

The Circuit dashboard's identity panel links a QCC run to the vendor-side job. The `provider_job_id` value returned at submission is stored on `status.providerJobId` and exposed as a label on `qcc_circuit_info`. Together, `metadata.uid` and `status.providerJobId` connect the Kubernetes `Circuit` to the corresponding IBM Quantum job. This is shown for Circuit `shor-2vv42` on `ibm-kingston`, whose provider job is `d89dinp789is73935r0g`.

The same identifier is used as a Grafana deep-link. In Figure 7.23, the `provider_job_id` cell provides an *Open on IBM Quantum* action. The dashboard resolves the link from the `qcc_circuit_info` series and opens the IBM Quantum Console result page for the same job (Figure 7.24).

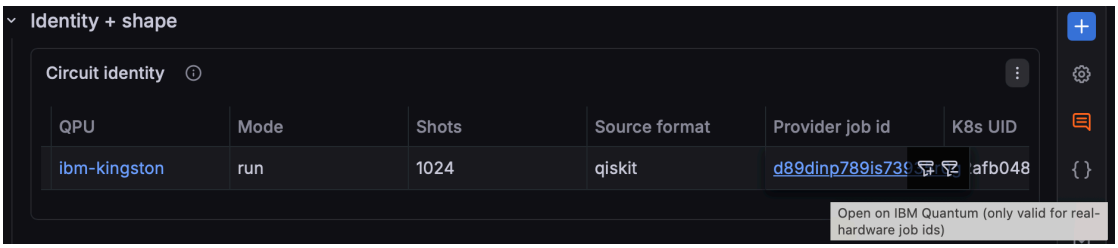


Figure 7.23: *The Circuit dashboard's identity panel for* `shor-2vv42`*, showing the* `provider_job_id` *cell with the* Open on IBM Quantum *data link.*

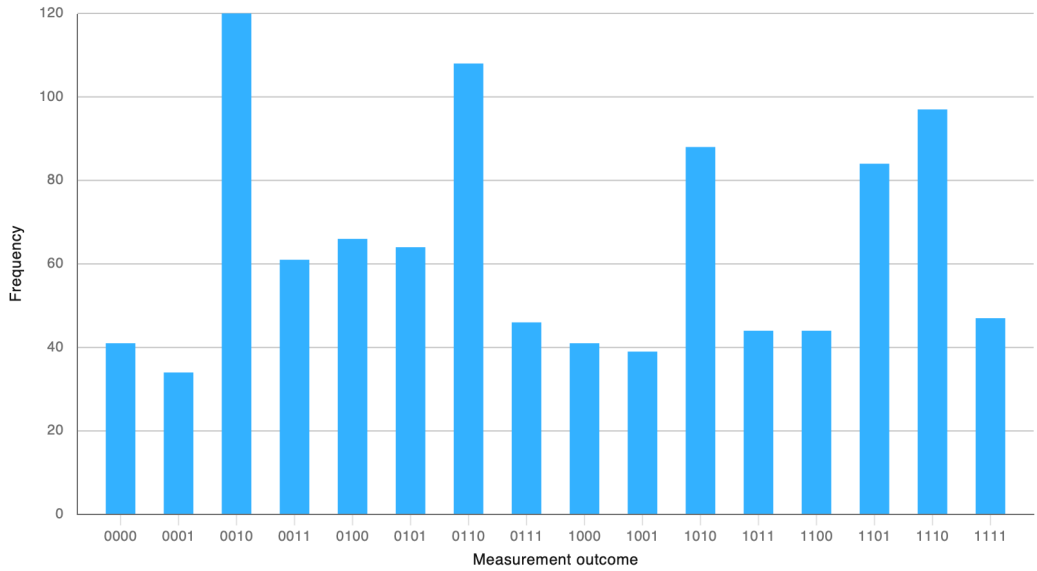


Figure 7.24: *The IBM Quantum Console results page for job* `d89dinp789is73935r0g`.

Starting from the vendor console, the correlation works in the opposite direction as well. At submission time, the executor stamps the `Circuit`'s Kubernetes UID onto the IBM Quantum job as a `qcc.circuit.uid` tag. The console can therefore show which Kubernetes `Circuit` owns the vendor-side job (Figure 7.25).

This vendor-side tag complements the recorded `providerJobId`. Together, the two identifiers make the run traceable from either side: from QCC to IBM Quantum through the provider job id, and from IBM Quantum back to QCC through the Circuit UID. The correlation is available while the job is running and after it terminates, using standard QCC and vendor tooling rather than a separate execution archive.

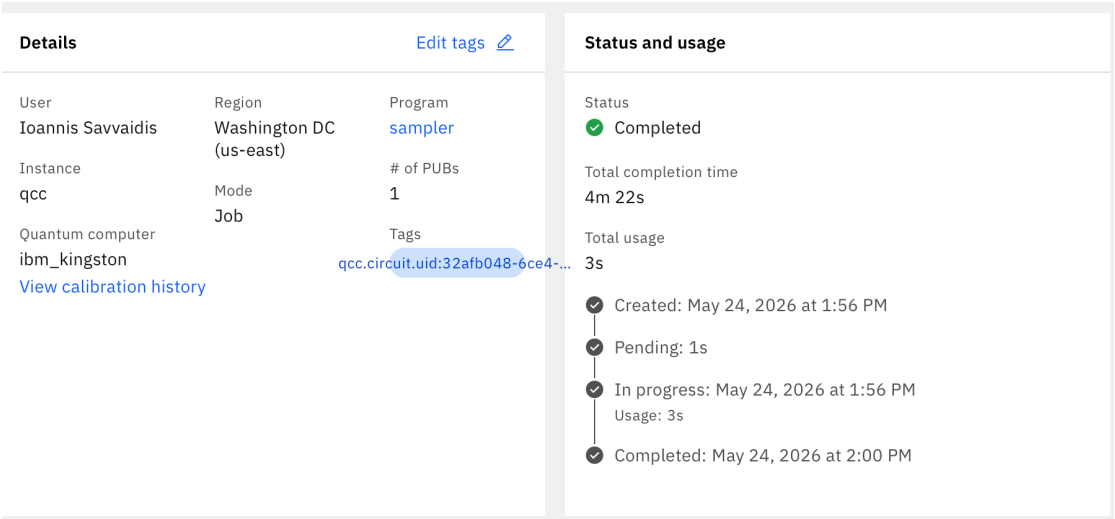


Figure 7.25: *The IBM Quantum Console Details panel for the* `ibm_kingston` *job behind Circuit* `shor-2vv42`*. The* `qcc.circuit.uid:32afb048-…` *tag is visible in the panel's Tags field, stamped by the executor at circuit submission. The vendor-side annotation lets the console point back to QCC stack's* `Circuit`.

7.9 Tuning the transpiler

QCC uses a two-tier schema to absorb vendor-SDK parameters without turning every provider option into a first-class API field. Tier 1 contains the stable orchestration params supported directly for QCC CLI, for easy command-line use during experiments — fields such as backend selection, shot count, and optimisation level. Tier 2 contains passthrough blocks for more advanced use cases, as an experimental feature, of transpilation and execution stages — provider or stage-specific options that QCC forwards to the executor without interpreting them.

The evaluation uses the substrate `ibm-kingston` and uses 4096 shots, so the comparison isolates transpilation rather than backend variance. The default v1 run from Section 7.7 is the baseline. The next two runs apply changes in sequence: v2 raises the Tier 1 optimisation level, and v3 adds a Tier 2 scheduling pass.

The first manifest, `shor-v2.yaml`, changes only the Tier 1 field. The v1 runs left `optimizationLevel` unset, which used Qiskit's default level 1 preset. v2 sets it to 3, Qiskit's most aggressive optimisation preset, and makes no other transpilation change:

```yaml
apiVersion: qcc.io/v1alpha1
kind: Circuit
metadata:
  generateName: shor-tuned-kingston-
  labels:
    qcc.io/algorithm: shor
    qcc.io/algorithm-version: v2
    qcc.io/experiment: thesis
spec:
  mode: run
  shots: 4096
  optimizationLevel: 3
```

`shor-v3.yaml` adds a single field, a `transpile` block carrying `scheduling_method: alap`, the *as-late-as-possible* scheduling pass that inserts explicit delay instructions to push gate operations toward the end of the schedule:

```yaml
apiVersion: qcc.io/v1alpha1
kind: Circuit
metadata:
  generateName: shor-tuned-kingston-
  labels:
    qcc.io/algorithm: shor
    qcc.io/algorithm-version: v3
    qcc.io/experiment: thesis
spec:
  mode: run
  shots: 4096
  optimizationLevel: 3
  transpile:
    scheduling_method: alap
```

The `scheduling_method: alap`, passes the scheduling method to Qiskit's `transpile()`[5] function signature. QCC accepts the value in the Tier 2 block and passes it to the executor's transpilation call unchanged, so the option can be used without a CRD change or controller rebuild.

The runs are submitted with `kubectl create -f` rather than through the CLI. This is intentional since Tier 2 block is not supported in QCC CLI and is not typed in `Circuit` CRD.

```
$ kubectl create -f examples/thesis/circuits/shor-v2.yaml
$ kubectl create -f examples/thesis/circuits/shor-v3.yaml
```

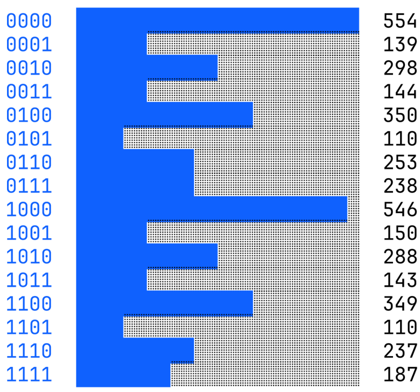
The result cards show how the transpiled circuit shape and outcome distribution change on the same backend, `ibm-kingston`, Figure 7.26 and Figure 7.27.

```
Q•C•C  1.0.0

shor-tuned-kingston-nb7q9

  ✓ shor-tuned-kingston-nb7q9 · run · on ibm-kingston · degraded signal (error exposure ≈ 1.3)

  setup
    phase    Succeeded
    source   qiskit
    shots    4096
    job      d89elfqs46sc73fah88g

  backend
    name          ibm-kingston  (156q · 352 edges · 10 basis gates)
    processor     Heron r2
    calibrated    2026-05-16 (8d ago)
    gate errors   1Q 2.74e-04 · 2Q 2.03e-03 · readout 1.65e-02
    coherence     T1 175 µs · T2 117 µs
    cycle time    dt = 4 ns

  circuit
    transpiled       depth 1523 · 2054 gates · 440 2Q
    exec time        ~103.56 µs (critical-path estimate)
    error exposure   ≈ 1.3 events/shot (approaching gate-error budget)
    fidelity bound   P(no gate error) ≈ 0.26  (excludes readout & coherence)

  results

    0000 ▏                                                  540
    0001 ▏                                                  149
    0010 ▏                                                  315
    0011 ▏                                                  173
    0100 ▏                                                  394
    0101 ▏                                                  125
    0110 ▏                                                  214
    0111 ▏                                                  176
    1000 ▏                                                  530
    1001 ▏                                                  129
    1010 ▏                                                  314
    1011 ▏                                                  167
    1100 ▏                                                  362
    1101 ▏                                                  132
    1110 ▏                                                  218
    1111 ▏                                                  158

▸ qasm qcc get circuit shor-tuned-kingston-nb7q9 --qasm
```

Figure 7.26: *v2, the level-3 transpile: `optimizationLevel=3`, a Tier-1 field, with no Tier-2 block.*

[5] https://tinyurl.com/bdht6bzk

```
Q•C•C  1.0.0

shor-tuned-kingston-62qzb

  ✓ shor-tuned-kingston-62qzb · run · on ibm-kingston · degraded signal (error exposure ≈ 1.5)

  setup
    phase    Succeeded
    source   qiskit
    shots    4096
    job      d89el10p0eas73doh6u0

  backend
    name          ibm-kingston  (156q · 352 edges · 10 basis gates)
    processor     Heron r2
    calibrated    2026-05-16 (8d ago)
    gate errors   1Q 2.74e-04 · 2Q 2.03e-03 · readout 1.65e-02
    coherence     T1 175 µs · T2 117 µs
    cycle time    dt = 4 ns

  circuit
    transpiled       depth 1524 · 2548 gates · 442 2Q
    exec time        ~103.63 µs (critical-path estimate)
    error exposure   ≈ 1.5 events/shot (approaching gate-error budget)
    fidelity bound   P(no gate error) ≈ 0.23  (excludes readout & coherence)

  results
```




```
▸ qasm qcc get circuit shor-tuned-kingston-62qzb --qasm
```

Figure 7.27: *v3: the same level-3 transpile plus* `transpile.scheduling_method=alap`*, a Tier-2 passthrough.*

Each result card reports two related quantities. The first is the empirical *correct-period mass*, defined in Equation 7.1. For the v2 baseline (Figure 7.26), 540 shots land on 0000 and 530 on 1000, gave $(540 + 530)/4096 = 26.1\%$.

The second is the modelled *fidelity bound*, $P(\text{no gate error}) \approx 0.26$, computed from the transpiled gate count and the backend's per-gate error rates. This bound excludes readout error and coherence loss, so it is not a prediction of the final bitstring distribution. It is a compact estimate of gate-error exposure.

The purpose of the comparison is not to claim an optimal transpilation strategy. Raising the optimisation level is a routine Qiskit operation. The point is that QCC makes both the empirical outcome and the modelled exposure visible for each run. A larger shot count sharpens the estimate of correct-period mass, but it does not itself improve the result. The result changes when the transpiled circuit changes.

The v2→v3 step isolates the effect of the Tier 2 scheduling option. The same teleme-

try surface shows the change in circuit shape: the transpiled gate count rises from 2054 in v2 (Figure 7.26) to 2548 in v3 (Figure 7.27), a +494 increase caused by inserted delay instructions. The modelled fidelity bound falls from 0.26 to 0.23. The measured outcome, however, remains close to v2:

$$(554 + 546)/4096 = 26.9\%.$$

Read end to end, the experiment forms a single visible arc (Table 7.1). The ideal simulator returns the expected period states with full mass,

$$(517 + 507)/1024 = 100\%.$$

The default level-1 real-hardware runs do not clear the 12.5% noise floor. `ibm-fez` gives $(56+55)/1024 \approx 10.8\%$, `ibm-marrakesh` gives $(42+44)/1024 \approx 8.4\%$, and `ibm-kingston` gives $(41 + 41)/1024 \approx 8.0\%$.

The v2 run changes one Tier 1 field by setting `optimizationLevel` to 3. This reduces the transpiled depth from 2048 to 1523 and lifts the correct-period mass to

$$(540 + 530)/4096 \approx 26.1\%,$$

clearing the noise floor. The v3 run then adds the Tier 2 `alap` scheduling option. This changes the schedule and increases the gate count, but leaves the measured outcome within sampling noise of v2. The point of Table 7.1 is therefore operational: a stable Tier 1 field recovers the signal, while a vendor-specific Tier 2 option changes the transpiled shape without materially changing the observed period mass.

Table 7.1: *The full run arc on the Shor $N = 15$ circuit, from the ideal simulator through the default level-1 v1 sweep to the tuned v2 / v3 runs on `ibm-kingston`.*

Run	Transpilation	Shots	`0000`+`1000`	Success
Aer (ideal)	default	1024	517 + 507	$\approx 100\%$
`ibm-fez` (v1)	default, L1	1024	56 + 55	10.8%
`ibm-marrakesh` (v1)	default, L1	1024	42 + 44	8.4%
`ibm-kingston` (v1)	default, L1	1024	41 + 41	8.0%
`ibm-kingston` (v2)	level 3 (Tier-1)	4096	540 + 530	26.1%
`ibm-kingston` (v3)	level 3 + `alap` (Tier-2)	4096	554 + 546	26.9%

The same v2→v3 comparison can be achieved through the Circuit dashboard of Section 7.8.2. All statistics are published as `qcc_circuit_*` metrics and can be inspected through the dashboard or queried directly with PromQL. This lets an engineer examine an experiment from the metric surface itself Figure 7.28, Figure 7.29.

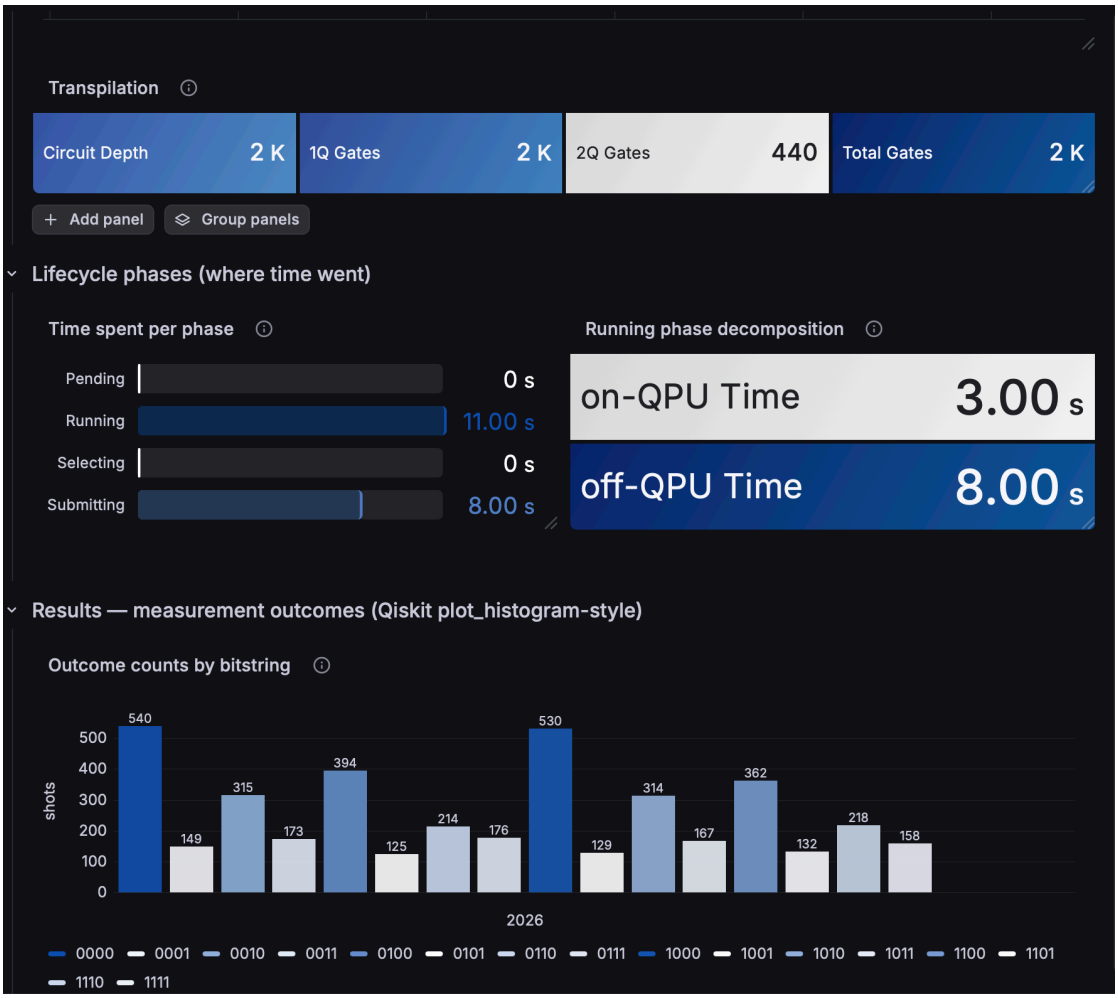


Figure 7.28: *v2 rendered through the Circuit dashboard.*

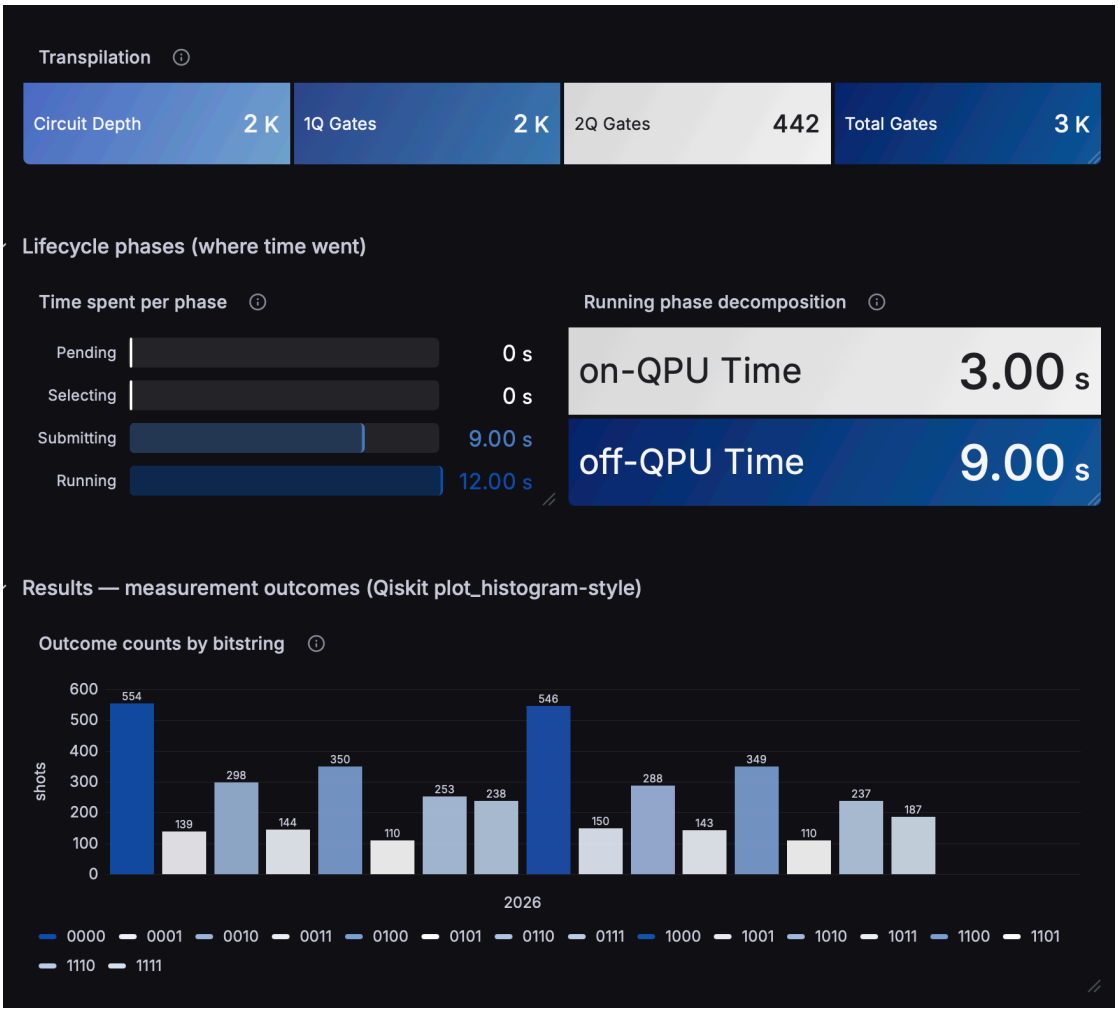


Figure 7.29: *v3 rendered through the same dashboard: the gate-count panel shows the* `alap` *inflation as a panel-level delta.*

8 Discussion

Chapter 7 demonstrated QCC end to end on a single workload. This chapter reads that demonstration back against the five requirements of Section 6.1, reflects on which architectural commitments held under implementation, sets out the limitations of the system and its evaluation, and closes on the direction the work points.

8.1 Analysis

Each of the five requirements in Section 6.1 set an acceptance criterion. Chapter 7 supplies the evidence. Table 8.1 gives the verdict for each one, with the demonstration that supports it.

Table 8.1: *R1–R5 acceptance status under the demonstrated prototype (QCC v1.0.0).*

Req.	Requirement	Verdict	Evidence
R1	Declarative infrastructure	Satisfied	Section 7.2; Figure 7.3
R2	Observable through industry standards	Satisfied	Section 7.8; Figure 7.15, Figure 7.16
R3	Portable and modular across providers	Satisfied	Section 7.1; Section 7.9
R4	Correlated across the boundary in real time	Satisfied	Section 7.7; Section 7.8.3
R5	Monitored backend quality	Satisfied	Section 7.6, Section 7.7, Section 7.8

R1 called for the platform to deploy declaratively on commodity hardware, by a single developer and with no dedicated operations team. The demonstration met this directly. One declarative package brought up the controller, the executor, the CRDs, and the observability wiring on a local `kind` cluster, on a single machine (Section 7.2). The registry then listed three IBM Heron QPUs, `ibm-fez`, `ibm-kingston`, and `ibm-marrakesh`, alongside ten simulators (Figure 7.3). `kind` is a conformant cluster, and the platform uses only standard Kubernetes primitives.

R2 called for operational state to be observable through industry-standard telemetry. Each circuit and backend emits its state through the `qcc.*` specification, over OTel and Prometheus. A stock `kube-prometheus-stack` collects the signals, and two Grafana dashboards display them (Section 7.8). Nothing custom sits between the platform and the data. An operator already running this stack sees quantum workloads the same way as everything else on the cluster. The dashboards show change as well as state. The transpiler experiments of Section 7.9 appear there as shifts in the same metrics, with no separate measurement.

R3 called for portability across providers behind a stable interface. Two adapters, Aer and IBM, were built against one gRPC contract, and the controller imports no vendor code (Section 7.1). The boundary was then tested with a provider option the platform had never seen, the `alap` scheduling pass. It was carried through as a Tier-2 passthrough. The gate count moved from 2054 to 2548, while the controller, the schema, and the telemetry stayed the same (Section 7.9). The vendor option reached the executor with no change above the adapter.

R4 called for a workload to be followed across the boundary in real time. The recorded `providerJobId` linked a `Circuit` to its vendor job in both directions. It was queryable on `qcc_circuit_info` and appeared on the result card as a console deep-link. While a job was still running, the card showed its transpiled shape, its queue position, and how wall-clock time split between the QPU and orchestration (Section 7.8.3, Section 7.7). The boundary stayed visible during the run, not only after it.

R5 called for backend quality to be treated as a measured signal, read from telemetry rather than assumed from a published specification. Each backend exposed its coherence, its gate and readout error rates, its gate timing, and its calibration date, alongside every run's outcome distribution (Section 7.6). The real-hardware sweep showed why this matters. At the default transpilation, all three Heron r2 backends sat at or below the $12.5\,\%$ noise floor, the level a fully decohered run produces. The period signal did not survive on any of them (Section 7.7). No backend cleared the floor until the transpiler tuning of Section 7.9 improved the results dramatically.

8.2 Retrospective

Three design decisions shaped the build. Each proved its worth in a different way as the work progressed.

Separating the controller from the executor added a second moving part where one might have been enough. The cost was clear from the start. The value is that the two sides change on their own terms. The executor holds the quantum SDK and can be updated by itself. The controller handles orchestration and stays put. A single combined process would tie the two together, so a change to one would disturb the other.

Keeping provider options in a passthrough tier, instead of modelling each one in the schema, let the schema stay still as the evaluation grew. Each option that mattered

during testing was reachable by editing a manifest, with no new fields and no new code. The work could try different transpiler settings without the schema chasing every knob.

Treating backend selection as something to measure, not predict, kept the system modest about what it claimed. Rather than commit to a scoring function and defend it, the platform runs a workload across candidates and reads the result. A richer policy can be added later, in the same place. The hardware findings rewarded this caution. As the analysis showed, the obvious single-number ranking would have been wrong.

8.3 Limitations

The work is bounded in two ways: in scope, and in what a single evaluation can establish. The exclusions of Section 1.3 are out of scope by design: hardware-level control, algorithm design, error correction, and quantum networking. So are the concerns of a hardened production deployment, such as high availability and multi-tenant isolation. Within scope, the system has the following bounds.

- *Vendor coverage.* Vendor neutrality rests on the gRPC contract. Two adapters, Aer and IBM, exercise it here. Further providers, such as Rigetti or AWS Braket, are additional adapters against the same contract.
- *Input format.* The platform accepts Qiskit and OpenQASM. Qiskit source runs in the executor's Python runtime, so it depends on the libraries in that image. OpenQASM is self-contained, which makes it the portable submission format.
- *Workload and selection.* The evaluation uses one algorithm, Shor's factoring of $N = 15$, submitted once. Iterative variational workloads such as VQE use a tightly-looped model, different from the asynchronous submission the platform targets.
- *Calibration freshness.* A backend's calibration is read once, at registration. For live hardware it ages between probes. The dashboard shows this, and a periodic refresh would keep it current.
- *Submission boundary.* A short window sits between a submission and the recording of its `providerJobId`. A controller restart in that window could leave a job unreconciled. None did across the evaluation.

The evaluation itself is a set of single runs at fixed shot counts. It shows that the telemetry tells substrates and transpiler settings apart; it does not benchmark them. Calibration refresh and the submission boundary return as future work in Chapter 9.

8.4 Outlook

Two consequences of expressing quantum execution in Kubernetes-native terms reach beyond what the prototype demonstrates. Both are stated here as direction, not specification.

The first is ecosystem alignment, and the literature already traces the path. The orchestration prototypes surveyed in Chapter 5 move quantum execution steadily deeper

into the Kubernetes resource model. Qubernetes runs a circuit as a built-in Kubernetes `Job` (Stirbu et al., 2024). Qonductor adds a custom scheduler over CRDs (Giortamis et al., 2025b). QCC raises that to a full operator, with typed `Circuit` and QPU resources. The endpoint of the path is for a QPU to become a first-class resource the scheduler allocates by profile, a request for a given fidelity, connectivity, or qubit count. Cloud-native systems are not the only ones pointing there. Bacher et al. (2025), from the HPC side, devote an appendix to the same idea: a Kubernetes Device Plugin that exposes quantum computers as cluster resources, the way GPUs and FPGAs already are. It names Dynamic Resource Allocation (DRA, generally available since Kubernetes v1.34[1]) as the mechanism for sharing them across Pods. QCC is a current, operator-level instance of this model, and its native form is that scheduler-allocated backend. Table 8.2 sets this progression against the work that traces it.

Table 8.2: *Kubernetes-native quantum execution across the literature and this work, ordered by depth of integration.*

Work	Year	Kubernetes mechanism	Status
Qubernetes (Stirbu et al., 2024)	2024	Built-in `Job`	Implemented prototype
Qonductor (Giortamis et al., 2025b)	2025	Custom scheduler over CRDs	Implemented prototype
QCC (this work)	2026	Operator with typed `Circuit` and QPU resources	Implemented
QRMI appendix (Bacher et al., 2025)	2025	Device Plugin with DRA	Proposed

The second is software-engineering practice. Once a circuit is an ordinary Kubernetes resource, the practices the rest of the software stack relies on carry over to quantum work with little change. Circuits can be version-controlled, reviewed, and GitOps-deployed like any other manifest. This gives quantum experiments a managed lifecycle and a shared platform, where many people work against common infrastructure and dashboards. Release engineering follows from the same property. An executor image, its adapters, and the circuits it runs are versioned artefacts that can be promoted across environments and rolled back. Performance testing becomes routine rather than bespoke: the performance-test that fanned one workload across substrates here extends to a standing suite on a schedule. The same mechanism suggests a form of synthetic monitoring for QPUs. A small, known circuit submitted on a regular cadence turns each run into a probe of a backend's current behaviour. Drift in calibration or outcome quality then shows up as a monitored signal, not a surprise at run time. These are directions the resource model opens, not features the prototype ships.

[1] https://kubernetes.io/blog/2025/09/01/kubernetes-v1-34-dra-updates/

9 Conclusions and Future Work

This thesis set out to close an observability gap at the quantum–classical interface. Chapter 1 established the gap through a single diagnostic episode on `ibm_sherbrooke`. Chapter 5 developed it through a survey of existing orchestration and observability work, and Section 6.1 named it as five requirements. Chapter 6 answered with the QCC architecture, Chapter 7 showed it working, and Chapter 8 reflected on what that revealed. This chapter consolidates the contribution and sets out where it leads.

9.1 Summary of contributions

The thesis contributes at the interface layer, not in quantum algorithms or execution speed. Five things stand out.

The first is the QCC architecture: a cloud-native form of the orchestration layer that Seelam et al. (2026) identified but left unspecified. It applies the Kubernetes operator pattern: custom resources reconciled by a controller, with workload-specific work delegated to an executor behind a typed gRPC contract. A Go controller handles the Kubernetes side, a Python executor the Qiskit side. Chapter 6 describes the design, and Section 8.2 reflects on the choice.

The second is a set of three observability primitives. A per-instance audit trail is recorded in each resource's `status.conditions`. A `qcc.*` Prometheus metrics specification, fourteen in all, aggregates behaviour across many Circuits. A cross-boundary identifier links a Circuit's Kubernetes UID to its vendor `provider_job_id` in both directions: the UID travels forward as an IBM job tag, and the `provider_job_id` returns as a Prometheus metric label.

The third is the cross-substrate evaluation primitive, `qcc run --performance-test`. One command fans the same workload across every available substrate, and the Circuit dashboard's experiment-label pivot renders the comparison. This is the R5 contribution (Table 8.1). It ships measurement, not prediction.

The fourth is the operator-facing dashboards. They recast two classical SRE methods for quantum substrates. The USE method drives the backend dashboard, with util-

isation read as calibration freshness, saturation as availability, and errors as the gate-error and coherence envelope. The RED method drives the per-Circuit view. These are working dashboards, not a new methodology (Section 7.8.1).

The fifth is the open-source release of QCC v1.0.0: the controller, the executor, the CLI, the deployment manifests, the two dashboards, and example workloads as reproduction targets. The standardisation path below depends on a reference implementation that others can run and extend.

Together these satisfy all five requirements. R1 to R4 are met by artefacts the prototype ships and Chapter 7 demonstrated. R5, calibration-aware *evaluation*, is met by the `--performance-test` primitive. The predictive scoring that would automate backend choice is future work, not a missing part of R5.

9.2 Standardisation trajectory

Two of the architectural decisions are candidates to outlive the thesis as community standards.

The first is a Quantum Execution Interface (QEI): a vendor-neutral wire specification for backend access, modelled on the Container Runtime Interface (Kubernetes Authors, 2025) and the Container Network Interface (CNI Maintainers, 2025). The Kubernetes precedent shows what such a standard produces: the kubelet ships no container runtime; it consumes CRI, and the ecosystem supplies the plugins. A QEI would specify the executor's RPCs, their conformance rules, and a discovery mechanism. Vendors could then ship QEI plugins from their own runtimes, and any conformant orchestrator would consume them, QCC being the first. QCC's executor is one reference implementation; a standard would let several orchestrators and vendor plugins coexist.

The second is the `qcc.*` metric schema. The OpenTelemetry semantic-conventions registry has no quantum namespace, a gap confirmed during the Chapter 5 survey. The schema developed in Section 6.7 is structured to be submitted as a quantum-workload proposal under a `quantum.*` namespace. The thesis ships the working vocabulary; the community would refine and ratify it.

The two compose. A QEI plugin would emit `quantum.*` telemetry, and any OpenTelemetry backend would render it, with or without an orchestrator in between.

9.3 Future work

Several extensions are within reach of the prototype. Each fits a slot the architecture already exposes.

- *Predictive selection.* Only the first move of the selection chain ships, a hard-constraint first-match (Section 6.6.2). The later moves add calibration-aware scoring over transpilation metrics and layout fidelity. The selection slot is a strategy interface,

so a Qonductor-style scheduler (Giortamis et al., 2025b) or a `mapomatic`-style layout evaluator is a single swap.

- *Calibration refresh.* Calibration is read once, at registration (Section 8.2). A TTL-driven re-probe in the `QPUReconciler`, with a deadline policy that fails Circuits whose substrate is too stale, would close the gap.
- *Distributed tracing.* The OpenTelemetry tracer is wired but emits no spans in v1.0.0. Emitting spans at phase boundaries would render each Circuit's lifecycle as a trace, with the trace ID joining the forward-correlation set.
- *Vendor reach.* Three adapters are near-term: a generic Qiskit-provider adapter (IonQ, Rigetti, IQM, AQT), a QRMI adapter (Bacher et al., 2025), and a CUDA-Q adapter. Each is a Python class on the executor's resolver.
- *Iterative workloads.* The `Circuit` CRD captures one submission. Hybrid algorithms such as VQE (Peruzzo et al., 2014) and QAOA (Farhi et al., 2014) need a higher-level `Workflow` CRD that composes Circuits and inherits their observability surface.
- *Production hardening.* Mutual TLS on the controller–executor channel, multi-tenant isolation, durable result storage, and a reconciliation primitive for the submission-boundary window (Figure 6.6.1) remain outside the MSc scope.

9.4 Closing remarks

The episode that motivated the thesis was a small circuit execution on `ibm_sherbrooke` with a noisy output and with no operational way to ask why it looked that way. The thesis answered it by building the layer that was missing.

The same design choice that makes the answer work makes it extensible. The `Circuit` and QPU resources are not only how QCC expresses quantum work; they are a foundation other systems can build on without touching the control plane.

The machinery is borrowed from Site Reliability Engineering (Beyer et al., 2016), Systems Performance Methodology (Gregg, 2014), OpenTelemetry, the Prometheus operator, and the Kubernetes ecosystem. The contribution is not that the machinery is new, but that the wiring did not exist before. The thesis wired it once, showed that it works, and proposed the schemas under which others can extend it. The code is an artefact, the schemas are an invitation, and the wiring is the contribution.

This page intentionally left blank.

Bibliography

ADAC Institute QC Working Group (2025). *The Role of Quantum Computing in Advancing Scientific High-Performance Computing*. arXiv: 2508.11765.

Armbrust, Michael, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia (2010). "A View of Cloud Computing". In: *Communications of the ACM* 53.4, pp. 50–58. DOI: 10.1145/1721654.1721672.

Bacher, Utz, Mark Birmingham, Christopher D. Carothers, Andrew Damin, Carlos D. Gonzalez Calaza, Ashwin Kumar Karnad, Stefano Mensa, Matthieu Moreau, Aurelien Nober, Munetaka Ohtani, Max Rossmannek, Philippa Rubin, M. Emre Sahin, Oscar Wallis, Amir Shehata, Iskandar Sitdikov, and Aleksander Wennersteen (2025). *Quantum resources in resource management systems*. arXiv: 2506.10052 [quant-ph]. URL: https://arxiv.org/abs/2506.10052.

Bandic, Medina, Sebastian Feld, and Carmen G. Almudever (2022). "Full-Stack Quantum Computing Systems in the NISQ Era: Algorithm-Driven and Hardware-Aware Compilation Techniques". In: *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*. DOI: 10.23919/DATE54114.2022.9774643.

Beck, Thomas, Alessandro Baroni, Ryan Bennink, Gilles Buchs, Eduardo Antonio Coello Pérez, Markus Eisenbach, Rafael Ferreira da Silva, et al. (2024). "Integrating Quantum Computing Resources into Scientific HPC Ecosystems". In: *Future Generation Computer Systems* 161, pp. 11–25. DOI: 10.1016/j.future.2024.06.058.

Bennett, Charles H. (1982). "The Thermodynamics of Computation—a Review". In: *International Journal of Theoretical Physics* 21.12, pp. 905–940. DOI: 10.1007/BF02084158.

Bergholm, Ville, Josh Izaac, Maria Schuld, Christian Gogolin, Shahnawaz Ahmed, Vishnu Ajith, M. Sohaib Alam, Guillermo Alonso-Linaje, B. AkashNarayanan, Ali Asadi, Juan Miguel Arrazola, Nathan Killoran, et al. (2018). "PennyLane: Automatic differentiation of hybrid quantum-classical computations". In: *arXiv preprint arXiv:1811.04968*. Latest revision 2022. arXiv: 1811.04968 [quant-ph].

Bertels, K., A. Sarkar, T. Hubregtsen, A. A. Mouedenne, A. Yadav, A. Krol, and I. Ashraf (2020). "Quantum Computer Architecture: Towards Full-Stack Quantum Ac-

celerators". In: *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, pp. 1–6. DOI: 10.23919/DATE48585.2020.9116502. arXiv: 1903.09575 [quant-ph].

Beyer, Betsy, Chris Jones, Jennifer Petoff, and Niall Richard Murphy (2016). *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media. ISBN: 978-1-491-92912-4.

Blekos, Kostas, Dean Brand, Andrea Ceschini, Chiao-Hui Chou, Rui Li, Kriti S. Pandya, and Alejandro Summer (2024). "A Review on Quantum Approximate Optimization Algorithm and its Variants". In: *Physics Reports* 1068, pp. 1–66. DOI: 10.1016/j.physrep.2024.03.002.

Boltzmann, Ludwig (1877). "Über die Beziehung zwischen dem zweiten Hauptsatze der mechanischen Wärmetheorie und der Wahrscheinlichkeitsrechnung respektive den Sätzen über das Wärmegleichgewicht". In: *Wiener Berichte* 76, pp. 373–435.

Britt, Keith A., Fahd A. Mohiyaddin, and Travis S. Humble (2017). "Quantum Accelerators for High-Performance Computing Systems". In: *IEEE International Conference on Rebooting Computing (ICRC)*. DOI: 10.1109/icrc.2017.8123664.

Broughton, Michael, Guillaume Verdon, Trevor McCourt, Antonio J. Martinez, Jae Hyeon Yoo, Sergei V. Isakov, Philip Massey, Ramin Halavati, Murphy Yuezhen Niu, Alexander Zlokapa, Evan Peters, Owen Lockwood, Andrea Skolik, Sofiene Jerbi, Vedran Dunjko, Martin Leib, Michael Streif, David Von Dollen, Jarrod R. McClean, Ryan Babbush, Sergio Boixo, Dave Bacon, Alan K. Ho, Hartmut Neven, Masoud Mohseni, et al. (2020). "TensorFlow Quantum: A Software Framework for Quantum Machine Learning". In: *arXiv preprint arXiv:2003.02989*. arXiv: 2003.02989 [quant-ph].

Burns, Brendan (2024). *Designing Distributed Systems*. 2nd. O'Reilly Media. ISBN: 978-1-098-15634-3.

Burns, Brendan, Joe Beda, Kelsey Hightower, and Lachlan Evenson (2022). *Kubernetes: Up and Running*. 3rd. O'Reilly Media. ISBN: 978-1-098-11020-8.

Cai, Zhenyu, Ryan Babbush, Simon C. Benjamin, Suguru Endo, William J. Huggins, Ying Li, Jarrod R. McClean, and Thomas E. O'Brien (2023). "Quantum Error Mitigation". In: *Reviews of Modern Physics* 95.4, p. 045005. DOI: 10.1103/RevModPhys.95.045005.

Caleffi, Marcello, Michele Amoretti, Davide Ferrari, Davide Cuomo, Jessica Illiano, Antonio Manzalini, and Angela Sara Cacciapuoti (2024). "Distributed Quantum Computing: A Survey". In: *Computer Networks* 254, p. 110672. DOI: 10.1016/j.comnet.2024.110672. arXiv: 2212.10609 [cs.ET].

Cirq Developers (2018). *Cirq: A Python framework for creating, editing, and invoking Noisy Intermediate-Scale Quantum (NISQ) circuits*. URL: https://github.com/quantumlib/Cirq.

Cloud Native Computing Foundation (2024). *CNCF Cloud Native Definition v1.0*. URL: https://github.com/cncf/toc/blob/main/DEFINITION.md.

CNI Maintainers (2025). *Container Network Interface (CNI) Specification*. https://github.com/containernetworking/cni/blob/main/SPEC.md.

Cross, Andrew W., Lev S. Bishop, John A. Smolin, and Jay M. Gambetta (2022). "Open-QASM 3: A Broader and Deeper Quantum Assembly Language". In: *ACM Transactions on Quantum Computing* 3.3, pp. 1–50. DOI: 10.1145/3505636.

Cuomo, Davide, Marcello Caleffi, and Angela Sara Cacciapuoti (2020). "Towards a Distributed Quantum Computing Ecosystem". In: *IET Quantum Communication* 1.1, pp. 3–8. DOI: 10.1049/iet-qtc.2020.0002.

Deutsch, David (1985). "Quantum Theory, the Church–Turing Principle and the Universal Quantum Computer". In: *Proceedings of the Royal Society of London A* 400.1818, pp. 97–117. DOI: 10.1098/rspa.1985.0070.

Döbler, Philip and Manpreet Singh Jattana (2025). *A Survey on Integrating Quantum Computers into High Performance Computing Systems*. arXiv: 2507.03540.

Elsharkawy, Amr, Xiaorang Guo, and Martin Schulz (2024). "Integration of Quantum Accelerators into HPC: Toward a Unified Quantum Platform". In: *2024 IEEE International Conference on Quantum Computing and Engineering (QCE)*. IEEE, pp. 774–783. arXiv: 2407.18527 [quant-ph].

Elsharkawy, Amr, Xiao-Ting Michelle To, Philipp Seitz, Yanbin Chen, Yannick Stade, Manuel Geiger, Qunsheng Huang, Xiaorang Guo, Muhammad Arslan Ansari, Christian B. Mendl, Dieter Kranzlmüller, and Martin Schulz (2025). "Integration of Quantum Accelerators with High Performance Computing—A Review of Quantum Programming Tools". In: *ACM Transactions on Quantum Computing* 6.3, pp. 1–46. DOI: 10.1145/3743149.

EuroHPC Joint Undertaking (2025). *HPCQS: High Performance Computer and Quantum Simulator Hybrid*. CORDIS Project ID 101018180. URL: https://www.hpcqs.eu/.

Farhi, Edward, Jeffrey Goldstone, and Sam Gutmann (2014). *A Quantum Approximate Optimization Algorithm*. arXiv:1411.4028. Preprint.

Feynman, Richard P. (1982). "Simulating Physics with Computers". In: *International Journal of Theoretical Physics* 21.6–7, pp. 467–488. DOI: 10.1007/BF02650179.

Giortamis, Emmanouil, Francisco Romão, Nathaniel Tornow, and Pramod Bhatotia (2025a). "QOS: Quantum Operating System". In: *Proceedings of the 19th USENIX Symposium on Operating Systems Design and Implementation (OSDI '25)*, pp. 429–447.

Giortamis, Emmanouil, Francisco Romão, Nathaniel Tornow, Dmitry Lugovoy, and Pramod Bhatotia (2025b). "Qonductor: A Cloud Orchestrator for Quantum Comput-

ing". In: *Proceedings of SC'25: The International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 728–745. DOI: 10.1145/3712285.3759785.

Gregg, Brendan (2014). *The USE Method*. https://www.brendangregg.com/usemethod.html. Methodology for systems performance analysis: Utilization, Saturation, Errors.

Gregg, Brendan (2020). *Systems Performance: Enterprise and the Cloud*. 2nd. Addison-Wesley. ISBN: 978-0-13-682015-4.

Gropp, William, Sujata Banerjee, and Ian Foster (2020). *Infrastructure for Artificial Intelligence, Quantum and High Performance Computing*. DOI: 10.48550/arxiv.2012.09303. arXiv: 2012.09303.

Harrow, Aram W. and Ashley Montanaro (2017). "Quantum Computational Supremacy". In: *Nature* 549, pp. 203–209. DOI: 10.1038/nature23458.

Ho, Alan and Dave Bacon (2018). *Announcing Cirq: An Open Source Framework for NISQ Algorithms*. Google AI Blog. Public alpha announcement, International Workshop on Quantum Software and Quantum Machine Learning, July 2018. URL: https://research.google/blog/announcing-cirq-an-open-source-framework-for-nisq-algorithms/.

Hoefler, Torsten, Thomas Häner, and Matthias Troyer (2023). "Disentangling Hype from Practicality: On Realistically Achieving Quantum Advantage". In: *Communications of the ACM* 66.5, pp. 82–87. DOI: 10.1145/3571725.

Horsman, D., S. Stepney, R. C. Wagner, and V. Kendon (2014). "When Does a Physical System Compute?" In: *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 470.2169. DOI: 10.1098/rspa.2014.0182.

Hua, Fei, Meng Wang, Gushu Li, Bo Peng, Chenxu Liu, Muqing Zheng, Samuel Stein, Yufei Ding, Eddy Z. Zhang, Travis S. Humble, and Ang Li (2023). *QASMTrans: A QASM based Quantum Transpiler Framework for NISQ Devices*. DOI: 10.48550/arXiv.2308.07581. arXiv: 2308.07581 [quant-ph].

Humble, Travis S. and Keith A. Britt (2016). "Software Systems for High-Performance Quantum Computing". In: *IEEE High Performance Extreme Computing Conference (HPEC)*. DOI: 10.1109/hpec.2016.7761628.

Javadi-Abhari, Ali, Matthew Treinish, Kevin Krsulich, Christopher J. Wood, Jake Lishman, Julien Gacon, et al. (2024). *Quantum Computing with Qiskit*. arXiv: 2405.08810.

Kalman, Rudolf E. (1960). "On the General Theory of Control Systems". In: *IFAC Proceedings Volumes* 1, pp. 491–502. DOI: 10.1016/S1474-6670(17)70094-8.

Kanazawa, Naoki, Hitomi Takahashi, Yukio Kawashima, Hiroshi Horii, Yuto Morohoshi, and Kengo Nakajima (2025). "Observability Architecture for Quantum-Centric Supercomputing Workflows". In: *arXiv preprint arXiv:2512.05484*. DOI: 10.48550/arXiv.2512.05484.

Kaya, Ercüment, Jorge Echavarria, Muhammad Nufail Farooqi, Aleksandra Swierkowska, Patrick Hopf, Burak Mete, Lukas Burgholzer, Robert Wille, Laura Schulz, Martin Schulz, et al. (2024). "A software platform to support disaggregated quantum accelerators". In: *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1646–1653. DOI: 10.1109/SCW63240.2024.00205.

Kim, Jin-Sung, Alex McCaskey, Bettina Heim, Manish Modani, Sam Stanwyck, and Timothy Costa (2023). "CUDA Quantum: The Platform for Integrated Quantum-Classical Computing". In: *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, pp. 1–4. DOI: 10.1109/DAC56929.2023.10247886.

Kim, Youngseok, Andrew Eddins, Sajant Anand, Ken Xuan Wei, Ewout van den Berg, Sami Rosenblatt, Hasan Nayfeh, Yantao Wu, Michael Zaletel, Kristan Temme, and Abhinav Kandala (2023). "Evidence for the utility of quantum computing before fault tolerance". In: *Nature* 618.7965, pp. 500–505. DOI: 10.1038/s41586-023-06096-3.

Kleppmann, Martin and Chris Riccomini (2026). *Designing Data-Intensive Applications*. 2nd. O'Reilly Media. ISBN: 978-1-098-11905-8.

Krantz, Philip, Morten Kjaergaard, Fei Yan, Terry P. Orlando, Simon Gustavsson, and William D. Oliver (2019). "A Quantum Engineer's Guide to Superconducting Qubits". In: *Applied Physics Reviews* 6.2, p. 021318. DOI: 10.1063/1.5089550.

Kubernetes Authors (2025). *Container Runtime Interface (CRI)*. https://kubernetes.io/docs/concepts/architecture/cri/.

Landauer, Rolf (1961). "Irreversibility and Heat Generation in the Computing Process". In: *IBM Journal of Research and Development* 5.3, pp. 183–191. DOI: 10.1147/rd.53.0183.

Landauer, Rolf (1996). "The Physical Nature of Information". In: *Physics Letters A* 217.4, pp. 188–193. DOI: 10.1016/0375-9601(96)00453-7.

LaPierre, Ray (2021). *Introduction to Quantum Computing*. The Materials Research Society Series. Springer. ISBN: 978-3-030-69318-3. DOI: 10.1007/978-3-030-69318-3.

Lattner, Chris and Vikram Adve (Mar. 2004). "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. San Jose, CA, USA: IEEE Computer Society, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

Li, Gushu, Yufei Ding, and Yuan Xie (2019). "Tackling the Qubit Mapping Problem for NISQ-Era Quantum Devices". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. New York, NY, USA: ACM, pp. 1001–1014. DOI: 10.1145/3297858.3304023. arXiv: 1809.02573 [quant-ph].

Majors, Charity, Liz Fong-Jones, and George Miranda (2022). *Observability Engineering: Achieving Production Excellence*. O'Reilly Media. ISBN: 978-1-492-07643-8.

Mantha, Pradeep, Florian J. Kiwit, Nishant Saurabh, Shantenu Jha, and Andre Luckow (2025). "Pilot-Quantum: A Middleware for Quantum-HPC Resource, Workload and Task Management". In: *Proceedings of the 25th IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid 2025)*. IEEE, pp. 164–173. DOI: 10.1109/CCGRID64434.2025.00070.

Matthews, Suzanne J., Tia Newhall, and Kevin C. Webb (2022). *Dive into Systems: A Gentle Introduction to Computer Systems*. Free online edition at https://diveintosystems.org. No Starch Press.

Maxwell, James Clerk (1871). *Theory of Heat*. London: Longmans, Green, and Co.

McCaskey, Alexander J., Eugene F. Dumitrescu, Dmitry Liakh, Mengsu Chen, Wu-chun Feng, and Travis S. Humble (2018). "A Language and Hardware Independent Approach to Quantum–Classical Computing". In: *SoftwareX* 7, pp. 245–254. DOI: 10.1016/j.softx.2018.07.007.

McKay, David C., Thomas Alexander, Luciano Bello, Michael J. Biber, Lev S. Bishop, Jiayin Chen, Jerry M. Chow, Antonio D. Córcoles, Daniel Egger, Stefan Filipp, Juan Gomez, Michael Hush, Ali Javadi-Abhari, Diego Moreda, Paul Nation, Brent Paulovicks, Erick Winston, Christopher J. Wood, James Wootton, and Jay M. Gambetta (2018). "Qiskit Backend Specifications for OpenQASM and OpenPulse Experiments". In: *arXiv preprint arXiv:1809.03452*. DOI: 10.48550/arXiv.1809.03452.

McKay, David C., Christopher J. Wood, Sarah Sheldon, Jerry M. Chow, and Jay M. Gambetta (2017). "Efficient Z Gates for Quantum Computing". In: *Physical Review A* 96.2, p. 022330. DOI: 10.1103/PhysRevA.96.022330.

Möller, Matthias and Cornelis Vuik (2017). "On the Impact of Quantum Computing Technology on Future Developments in High-Performance Scientific Computing". In: *Ethics and Information Technology* 19.4, pp. 253–269. DOI: 10.1007/s10676-017-9438-0.

Murali, Prakash, Norbert Matthias Linke, Margaret Martonosi, Ali Javadi-Abhari, Nhung Hong Nguyen, and Cinthia Huerta Alderete (2019). "Full-Stack, Real-System Quantum Computer Studies: Architectural Comparisons and Design Insights". In: *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19)*. New York, NY, USA: ACM, pp. 527–540. DOI: 10.1145/3307650.3322273. arXiv: 1905.11349 [quant-ph].

Nation, Paul D., Abdullah Ash Saki, Sebastian Brandhofer, Luciano Bello, Shelly Garion, Matthew Treinish, and Ali Javadi-Abhari (2024). "Benchmarking the performance of quantum computing software". In: *arXiv preprint arXiv:2409.08844*. arXiv: 2409.08844 [quant-ph].

National Academies of Sciences, Engineering, and Medicine (2019). *Quantum Computing: Progress and Prospects*. Washington, DC: The National Academies Press. DOI: 10.17226/25196.

Nguyen, Hoa T., Prabha Krishnan, Dilip Krishnaswamy, Muhammad Usman, and Rajkumar Buyya (2024). *Quantum Cloud Computing: A Review, Open Problems, and Future Directions*. arXiv: 2404.11420.

Peruzzo, Alberto, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J. Love, Alán Aspuru-Guzik, and Jeremy L. O'Brien (2014). "A variational eigenvalue solver on a photonic quantum processor". In: *Nature Communications* 5, p. 4213. DOI: 10.1038/ncomms5213.

Popek, Gerald J. and Robert P. Goldberg (1974). "Formal Requirements for Virtualizable Third Generation Architectures". In: *Communications of the ACM* 17.7, pp. 412–421. DOI: 10.1145/361011.361073.

Preskill, John (2018). "Quantum Computing in the NISQ Era and Beyond". In: *Quantum* 2, p. 79. DOI: 10.22331/q-2018-08-06-79.

Preskill, John (2025). "Beyond NISQ: The Megaquop Machine". In: *ACM Transactions on Quantum Computing* 6.3, p. 18. DOI: 10.1145/3723153.

QIR Alliance (2021). *QIR: Quantum Intermediate Representation Specification*. https://www.qir-alliance.org. Linux Foundation, Joint Development Foundation. Accessed 2026-05-29.

Quantinuum (2025). *Hybrid Quantum–HPC Computing with Trapped Ions Is Here*. RIKEN Fugaku + Reimei integration. URL: https://www.quantinuum.com/blog/hybrid-quantum-hpc-computing-with-trapped-ions-is-here.

Rallis, Konstantinos, Ioannis Liliopoulos, Georgios D. Varsamis, Evangelos Tsipas, Ioannis G. Karafyllidis, Georgios Ch. Sirakoulis, and Panagiotis Dimitrakis (2025). *Interfacing Quantum Computing Systems with High-Performance Computing Systems: An Overview*. arXiv: 2509.06205.

Ravi, Gokul Subramanian, Kaitlin N. Smith, Pranav Gokhale, and Frederic T. Chong (2021). "Quantum Computing in the Cloud: Analyzing Job and Machine Characteristics". In: *2021 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, pp. 39–50. DOI: 10.1109/IISWC53511.2021.00015. arXiv: 2203.13121 [quant-ph].

Schulz, Martin, Martin Ruefenacht, Dieter Kranzlmüller, and Laura B. Schulz (2022). "Accelerating HPC with Quantum Computing: It Is a Software Challenge Too". In: *Computing in Science & Engineering* 24.4, pp. 60–64. DOI: 10.1109/MCSE.2022.3221847.

Seelam, Seetharami, Jerry M. Chow, Antonio Córcoles, Sarah Sheldon, Tushar Mittal, Abhinav Kandala, Sean Dague, Ian Hincks, Hiroshi Horii, Blake Johnson, Michael Le, Hani Jamjoom, and Jay M. Gambetta (2026). "Reference Architecture of a Quantum-

Centric Supercomputer". In: *arXiv preprint arXiv:2603.10970*. DOI: 10.48550/arXiv.2603.10970.

Seitz, Philipp, Amr Elsharkawy, Xiao-Ting Michelle To, and Martin Schulz (2023). "Toward a Unified Hybrid HPCQC Toolchain". In: *IEEE International Conference on Quantum Computing and Engineering (QCE)*, pp. 96–102. DOI: 10.1109/QCE57702.2023.10191.

Shannon, Claude E. (1948). "A Mathematical Theory of Communication". In: *Bell System Technical Journal* 27.3, pp. 379–423. DOI: 10.1002/j.1538-7305.1948.tb01338.x.

Shehata, Amir, Peter Groszkowski, Thomas Naughton, Muralikrishnan Gopalakrishnan Meena, Elaine Wong, Daniel Claudino, Rafael Ferreira da Silva, and Thomas Beck (2025). "Bridging paradigms: Designing for HPC-Quantum convergence". In: *Future Generation Computer Systems* 174, p. 107980. DOI: 10.1016/j.future.2025.107980.

Shi, Yunong, Pranav Gokhale, Prakash Murali, Jonathan M. Baker, Casey Duckering, Yongshan Ding, Natalie C. Brown, Christopher Chamberland, Ali Javadi-Abhari, Andrew W. Cross, David I. Schuster, Kenneth R. Brown, Margaret Martonosi, and Frederic T. Chong (2020). "Resource-Efficient Quantum Computing by Breaking Abstractions". In: *Proceedings of the IEEE* 108.8, pp. 1353–1370. DOI: 10.1109/JPROC.2020.2994765. arXiv: 2011.00028 [quant-ph].

Shor, Peter W. (1994). "Algorithms for Quantum Computation: Discrete Logarithms and Factoring". In: *Proceedings 35th Annual Symposium on Foundations of Computer Science*. IEEE, pp. 124–134. DOI: 10.1109/SFCS.1994.365700.

Silberschatz, Abraham, Peter B. Galvin, and Greg Gagne (2018). *Operating System Concepts*. 10th. Wiley. ISBN: 978-1-119-32091-3.

Siraichi, Marcos Yukio, Vinícius Fernandes dos Santos, Caroline Collange, and Fernando Magno Quintão Pereira (2018). "Qubit Allocation". In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO)*. New York, NY, USA: ACM, pp. 113–125. DOI: 10.1145/3168822.

Sivarajah, Seyon, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan (2020). "t|ket: a retargetable compiler for NISQ devices". In: *Quantum Science and Technology* 6.1, p. 014003. DOI: 10.1088/2058-9565/ab8e92. arXiv: 2003.10611 [quant-ph].

Smith, Robert S., Michael J. Curtis, and William J. Zeng (2016). "A Practical Quantum Instruction Set Architecture". In: *arXiv preprint arXiv:1608.03355*. DOI: 10.48550/arXiv.1608.03355.

Stepney, Susan and Viv Kendon (2021). "The Representational Entity in Physical Computing". In: *Natural Computing* 20.2, pp. 233–242. DOI: 10.1007/s11047-020-09805-3.

Stirbu, Vlad, Otso Kinanen, Majid Haghparast, and Tommi Mikkonen (2024). "Qubernetes: Towards a Unified Cloud-Native Execution Platform for Hybrid Classic-Quantum Computing". In: *Information and Software Technology* 175, p. 107529. DOI: 10.1016/j.infsof.2024.107529.

Turing, Alan M. (1936). "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* 1, pp. 230–265. DOI: 10.1112/plms/s2-42.1.230.

Uhlig, Rich, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kägi, Felix H. Leung, and Larry Smith (2005). "Intel Virtualization Technology". In: *Computer* 38.5, pp. 48–56. DOI: 10.1109/MC.2005.163.

University of Innsbruck and AQT (2024). *A Hybrid Supercomputer: Researchers Integrate a Quantum Computer into a High-Performance Computing Environment*. HPQC project, LEO5 + IBEX Q1. URL: https://techxplore.com/news/2024-07-hybrid-supercomputer-quantum-high-environment.html.

Verma, Abhishek, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes (2015). "Large-scale Cluster Management at Google with Borg". In: *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*. ACM. DOI: 10.1145/2741948.2741964.

Wilson, Ellis, Sudhakar Singh, and Frank Mueller (2020). "Just-in-time Quantum Circuit Transpilation Reduces Noise". In: *2020 IEEE International Conference on Quantum Computing and Engineering (QCE)*. IEEE, pp. 345–355. DOI: 10.1109/QCE49297.2020.00050. arXiv: 2005.12820 [quant-ph].

Zhu, Yu, Qiming Du, Yuqiong Jin, Woji He, Hang Lian, Xin Zhou, Jianyu Zhang, Yiyang Chen, Jinchen Xu, and Zheng Shan (2026). *QLLVM: A Scalable Quantum-Classical Co-Compilation Framework based on LLVM*. DOI: 10.48550/arXiv.2604.15094. arXiv: 2604.15094 [quant-ph].

This page intentionally left blank.

Appendices

This page intentionally left blank.

A Diagnostic Tools

The following functions were developed during MSc coursework to investigate quantum circuit execution behaviour across different backends. They are referenced in Chapter 1 and informed the experimental methodology described in Chapter 7. The code was developed using Python 3.11.1, Qiskit 1.1.1, Qiskit IBM Runtime 0.27.0, and Qiskit Aer 0.14.2.

A.1 Required Imports

```
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister
from qiskit.providers import BackendV2
from qiskit.visualization import plot_histogram, timeline_drawer
from qiskit.transpiler.preset_passmanagers import (
    generate_preset_pass_manager)
from qiskit_aer import AerSimulator
from qiskit_ibm_runtime.fake_provider import FakeSherbrooke
from qiskit_ibm_runtime import (
    QiskitRuntimeService, SamplerV2 as Sampler)
import matplotlib.pyplot as plt
```

A.2 Circuit Diagnostics

The `metrics()` function accepts a quantum circuit, its transpiled form, a backend reference, and measurement counts. It produces a suite of diagnostic plots: circuit diagrams before and after transpilation, circuit depth comparison, operation counts (total operations, gates, and non-local gates), scheduling timeline via `timeline_drawer()` for real or noisy backends, per-gate duration on a logarithmic scale with estimated total execution time, and measurement histograms.

```
def metrics(circuit=QuantumCircuit,
```

```
            transpiled_circuit=QuantumCircuit,
            backend=BackendV2, counts=None):
    labels = ['Before Transpilation', 'After Transpilation']
    experiment = backend.name

    circuit.draw(output='mpl')
    plt.show()
    transpiled_circuit.draw(output='mpl', idle_wires=False)
    plt.show()

    # Circuit depth comparison
    depths = [circuit.depth(), transpiled_circuit.depth()]
    plt.figure(figsize=(6, 4))
    plt.bar(labels, depths, width=0.7, color=['blue', 'green'])
    plt.ylabel('circuit_depth')
    for index, data in enumerate(depths):
        plt.text(x=index, y=data + 1, s=f"{data}")
    plt.tight_layout()
    plt.show()

    # Operation counts: total ops, gates, non-local gates
    operations = [sum(circuit.count_ops().values()),
                  sum(transpiled_circuit.count_ops().values())]
    gates = [circuit.size(), transpiled_circuit.size()]
    nonlocal_gates = [circuit.num_nonlocal_gates(),
                      transpiled_circuit.num_nonlocal_gates()]
    fig, (ax1, ax2, ax3) = plt.subplots(3, 1)
    ax1.bar(labels, operations, color=['blue', 'green'])
    ax1.set_ylabel('operations')
    for index, data in enumerate(operations):
        ax1.text(x=index, y=data + 1, s=f"{data}")
    ax2.bar(labels, gates, color=['blue', 'green'])
    ax2.set_ylabel('Gates')
    for index, data in enumerate(gates):
        ax2.text(x=index, y=data + 1, s=f"{data}")
    ax3.bar(labels, nonlocal_gates, color=['blue', 'green'])
    ax3.set_ylabel('Non-local Gates')
    for index, data in enumerate(nonlocal_gates):
        ax3.text(x=index, y=data + 1, s=f"{data}")
    fig.tight_layout()
    plt.show()

    # Scheduling timeline (real/noisy backends only)
    if experiment != "aer_simulator":
        timeline_drawer(transpiled_circuit, show_idle=False)
        plt.show()

    # Per-gate duration analysis (real/noisy backends only)
    if experiment != "aer_simulator":
        gate_lengths = {}
        backend_properties = backend.properties()
        for instruction in transpiled_circuit.data:
```

```
            qubits = [qubit._index for qubit in instruction.qubits]
            gate_name = instruction.operation.name
            if gate_name in ['barrier', 'measure', 'delay']:
                continue
            gate_duration = backend_properties.gate_length(
                gate_name, qubits)
            if gate_duration:
                if gate_name in gate_lengths:
                    gate_lengths[gate_name] += gate_duration
                else:
                    gate_lengths[gate_name] = gate_duration
        total_gate_time = sum(gate_lengths.values())
        estimated_time = (transpiled_circuit.depth()
                          * total_gate_time)
        plt.figure(figsize=(6, 4))
        plt.bar(gate_lengths.keys(), gate_lengths.values())
        plt.xlabel('Gate')
        plt.ylabel('duration')
        plt.yscale('log')
        plt.title(
            f"Estimated Duration {estimated_time:.10f} seconds")
        plt.tight_layout()
        plt.show()

    # Measurement histogram
    if counts is not None:
        plot_histogram(counts)
        plt.show()
```

A.3 Backend Comparison

The `aer_mimic_simulation()` function transpiles and executes a circuit on a given backend using `SamplerV2` with 4096 shots (the primitive's default). It applies optimisation level 3 and ASAP scheduling for real or noisy backends, then passes results to `metrics()` for analysis.

```
def aer_mimic_simulation(circuit=QuantumCircuit,
                         backend=BackendV2):
    """Simulation of a real backend characteristics."""
    experiment = backend.name
    if experiment == "aer_simulator":
        pass_manager = generate_preset_pass_manager(
            optimization_level=3,
            backend=backend,
            timing_constraints=(
                backend.target.timing_constraints())
        )
    else:
        pass_manager = generate_preset_pass_manager(
```

```
            optimization_level=3,
            backend=backend,
            timing_constraints=(
                backend.target.timing_constraints()),
            scheduling_method='asap'
        )
    transpiled_circuit = pass_manager.run(circuit, backend)
    sampler = Sampler(mode=backend)
    sampler.options.default_shots = 4096
    job = sampler.run([transpiled_circuit])
    print(f"Job ID is {job.job_id()}")
    pub_result = job.result()[0]
    counts = pub_result.data.c0.get_counts()
    metrics(circuit, transpiled_circuit, backend, counts)
```

This page intentionally left blank.