Quantum Computing with Qiskit

IBM Quantum

What is Qiskit?

Qiskit is an open-source SDK for working with quantum computers at the level of extended quantum circuits, operators, and primitives.

https://github.com/Qiskit/qiskit
https://docs.quantum-computing.ibm.com/

The **Qiskit Ecosystem** is a collection of software and tutorials that build on or extend Qiskit.

http://qiskit.github.io/ecosystem

Getting started with Qiskit

```
pip install 'qiskit[visualization]'
pip install qiskit-ibm-runtime
```

To access IBM hardware, set up your credentials through one of:

- IBM Quantum Platform
- IBM Cloud

https://docs.quantum-computing.ibm.com/start

Qiskit workflow

Build

Design and develop quantum circuits

https://docs.quantum.ibm.com/build

Transpile

Optimize circuits to efficiently run on hardware

https://docs.quantum.ibm.com/transpile

Verify

Validate and evaluate your quantum circuits

https://docs.quantum.ibm.com/verify

Run

Execute programs on quantum hardware via Qiskit Runtime

https://docs.quantum.ibm.com/run

Build

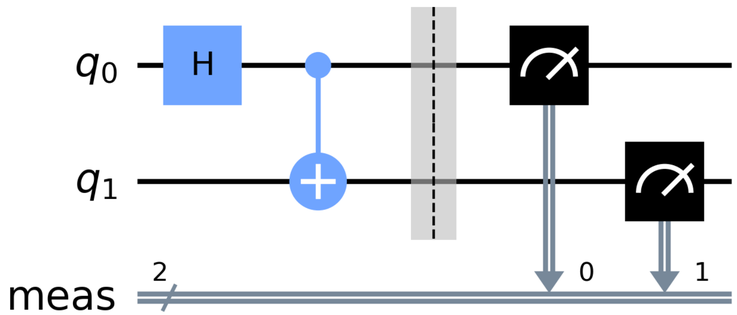
In the build phase, you create quantum programs that represent the problem you are solving.

Build a circuit with Qiskit

The foundation of quantum programs are quantum circuits, which consist of operations - including gates, measurement, and reset - that manipulate qubits in the quantum computer.

To build a circuit:

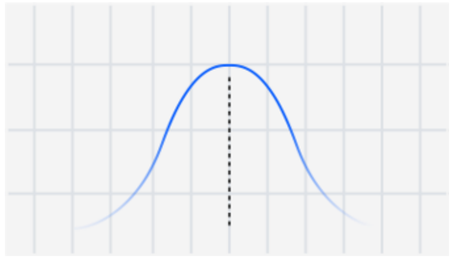
- Initialize a register of qubits
- Add the qubits to a circuit
- Perform operations on those qubits




```
from qiskit import QuantumCircuit, QuantumRegister

qubits = QuantumRegister(2, name="q")
circuit = QuantumCircuit(qubits)

q0, q1 = qubits
circuit.h(q0)
circuit.cx(q0, q1)
circuit.measure_all()

circuit.draw("mpl")
```

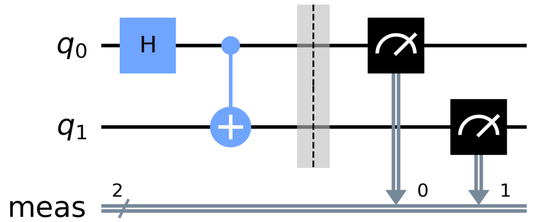
The output of a quantum circuit is encapsulated by **primitives**

Sampler primitive

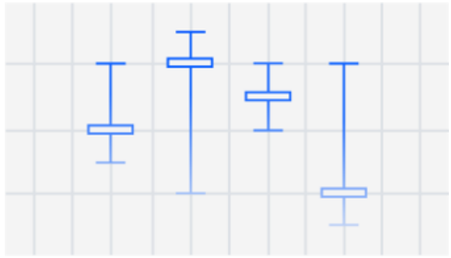


Output is a (quasi-)probability distribution over possible measurement outcomes

Circuit should include measurements

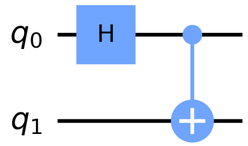


Estimator primitive



Output is the expectation value of an observable (e.g. cost function of an optimization problem)

Circuit should not include measurements



Qiskit includes a library of standard gates and common circuits

Standard gates

Standard quantum gates such as Hadamard, Pauli rotation gates, CNOT, Quantum Fourier Transform...

Variational ansatzes

Parameterized quantum circuits for chemistry and combinatorial optimization, include hardware efficient ansatzes

Benchmarking circuits

Circuits used for measuring quantum volume and other benchmarks of quantum devices

And more!

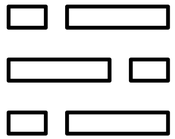
Transpile

In the transpile phase, you rewrite your circuit into a form that can be efficiently executed on a quantum device

Real quantum devices are subject to constraints

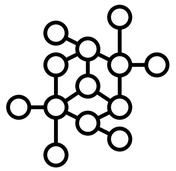
Basis gate set

Only a limited set of gates can be executed directly on the hardware. Other gates must be rewritten in terms of these basis gates.



Qubit connectivity

Only certain pairs of qubits can be directly interacted with each other.



Errors

Each operation has a chance of error, so circuit optimizations can greatly affect performance.

Transpiler definitions

Transpilation occurs in a series of passes, with the output of one pass becoming the input to the next.

A **transpiler pass** defines a standalone circuit transformation.

A **pass manager** is a list of transpiler passes grouped into a logical unit.

A **staged pass manager** is a list of pass managers, with each one representing a discrete stage of a transpilation pipeline.

Transpiler stages

1 Initialization

The circuit is prepared for transpilation.

2 Layout

The abstract qubits of the circuit are mapped to physical qubits on the device.

3 Routing

Swap gates are inserted to enable interactions between qubits that are not physically connected.

4 Translation

The gates of the circuit are translated to the basis gate set of the device.

5 Optimization

The circuit is rewritten to minimize its depth or number of operations in order to decrease the effect of errors.

6 Scheduling

The precise timing of pulse sequences is determined based on hardware information.

Transpile a circuit

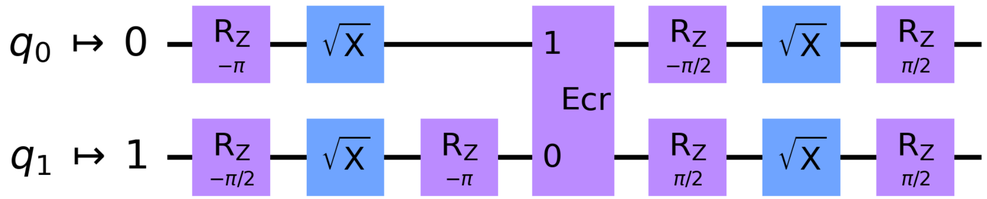
Qiskit can transpile your circuit for you with reasonable default settings.

To transpile a circuit using Qiskit's presets:

- Choose which device backend you want to target
- Create a preset staged pass manager with your desired optimization level
- Run the pass manager on the circuit

```python
from qiskit import QuantumCircuit, QuantumRegister
from qiskit.transpiler.preset_passmanagers import generate_preset_pass_manager
from qiskit_ibm_runtime import QiskitRuntimeService

qubits = QuantumRegister(2, name="q")
circuit = QuantumCircuit(qubits)

q0, q1 = qubits
circuit.h(q0)
circuit.cx(q0, q1)

service = QiskitRuntimeService()
backend = service.backend("ibm_brisbane")
pass_manager = generate_preset_pass_manager(1, backend=backend)
transpiled = pass_manager.run(circuit)

transpiled.draw("mpl", idle_wires=False)
```

Global Phase: 7π/4



You don't need to transpile your circuits to submit them to Qiskit Runtime!

But for the best performance, you should design your own transpilation pipeline.

If you transpile your circuits locally, remember to set `skip_transpilation=True` (more on this later).

Verify

In the verify phase, you test your quantum programs by running them on simulated devices and exploring their performance under realistic device noise models.

The cost of simulating quantum circuits scales **exponentially** with the number of qubits.

Roughly speaking, circuits larger than 50 qubits can't be simulated.

For such large circuits, you can:

Test smaller versions of your circuit

Modify the circuit so that it becomes classically simulable (e.g. a stabilizer circuit)

Several simulation tools are available for verifying your quantum programs

Reference primitives included with Qiskit

For exact simulation of small quantum circuits

Qiskit Aer primitives

For higher-performance simulation that can handle larger circuits, or to incorporate noise models

https://qiskit.org/ecosystem/aer/

Qiskit Aer stabilizer simulator

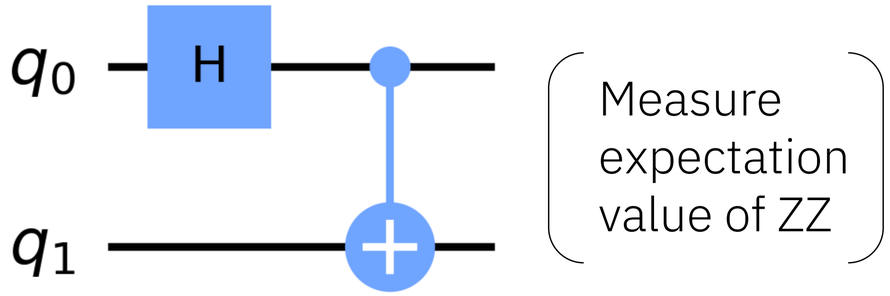
For efficient simulation of stabilizer circuits

Use the Estimator primitive from Qiskit Aer

The Estimator primitive is used to estimate the expectation value of an observable.

To use the Estimator primitive, pass it:

- A circuit without measurements
- An observable



Measure expectation value of ZZ

```python
from qiskit import QuantumCircuit, QuantumRegister
from qiskit.quantum_info import SparsePauliOp
from qiskit_aer.primitives import Estimator

qubits = QuantumRegister(2, name="q")
circuit = QuantumCircuit(qubits)

q0, q1 = qubits
circuit.h(q0)
circuit.cx(q0, q1)
# no measurements because we're using the Estimator primitive

observable = SparsePauliOp("ZZ")

estimator = Estimator()
job = estimator.run(circuit, observable)
exact_value = job.result().values[0]
print(exact_value)  # prints 1.0
```

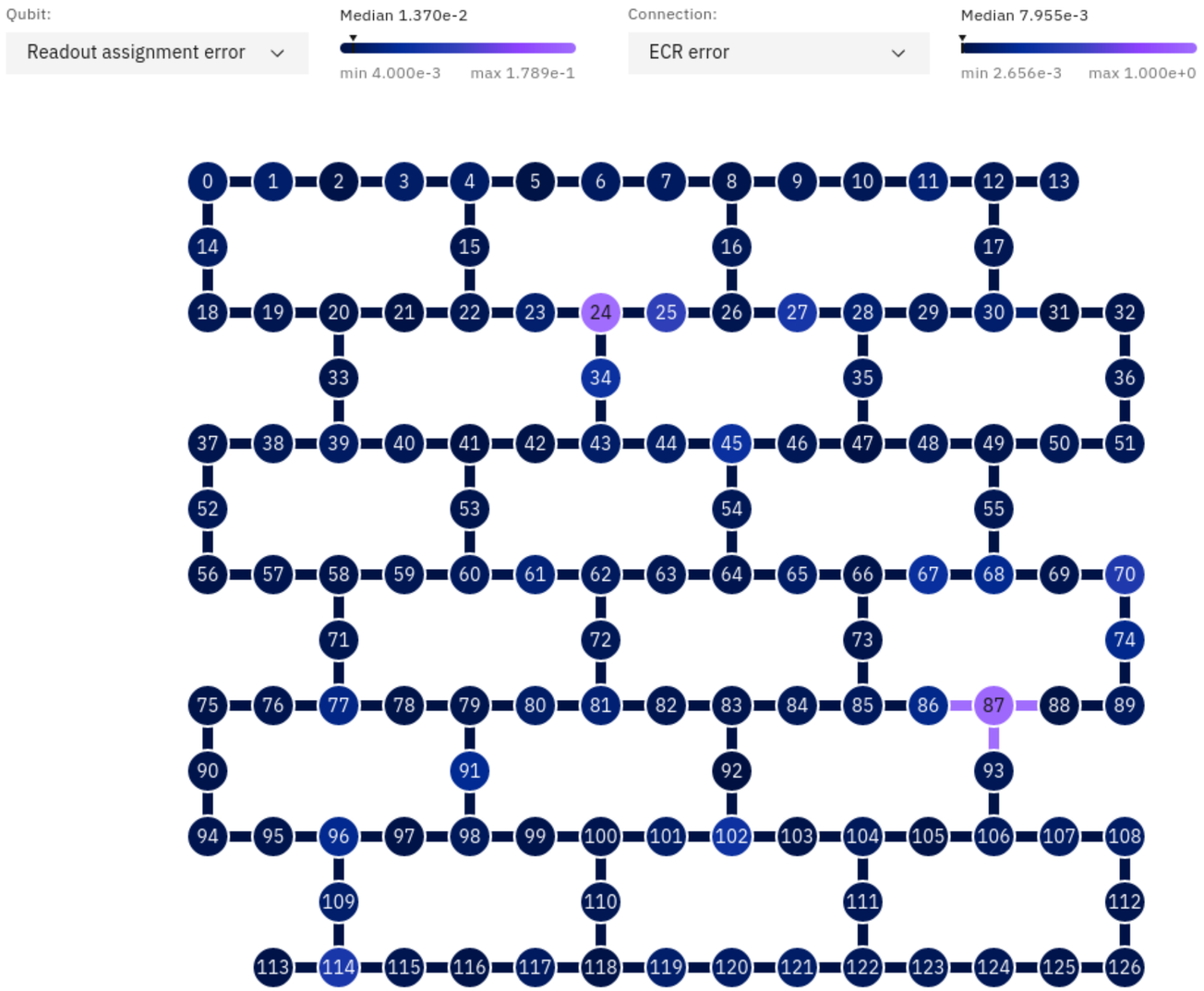
Simulate noise

Real quantum devices are subject to errors. These errors vary across different devices, different qubits and connections on the same device, and time.

In the near term, a variety of *error-mitigation* techniques help us counter the effects of noise.

In the long term, we will use *error-correction* to build fault-tolerant quantum computers.

Calibration data for ibm_brisbane
9am ET, Nov 16, 2023
https://quantum-computing.ibm.com/



Build noise models with Qiskit Aer

You can use Qiskit Aer to build and simulate noise models.

You can initialize a noise model with parameters set from the calibration data of a real backend.

You can also build your own custom noise models from scratch.

Depolarizing error:

$$E(\rho) = (1-p)\rho + p\frac{I}{2^n}$$

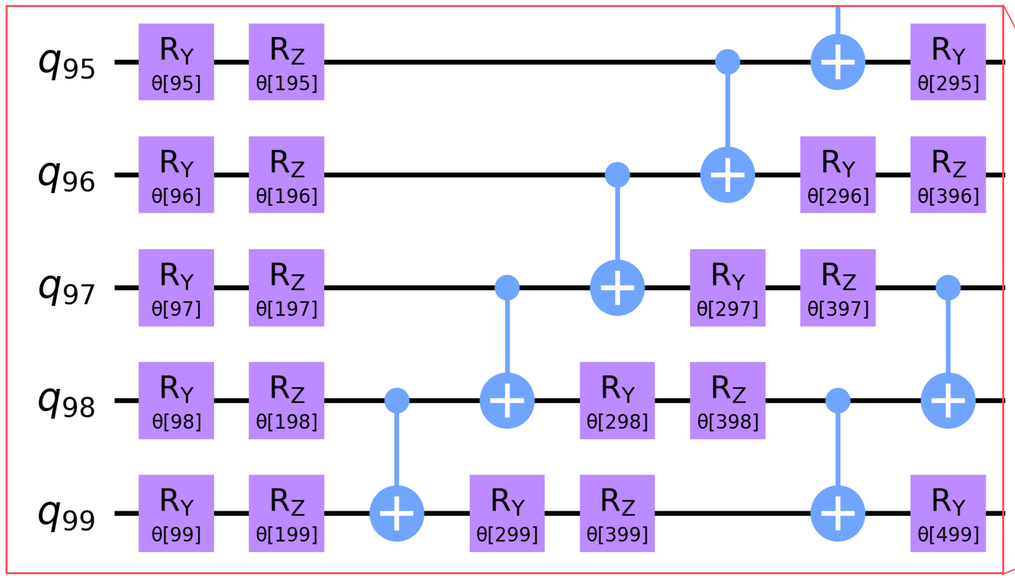
```python
from qiskit_aer.noise import NoiseModel
from qiskit_aer.primitives import Estimator
from qiskit_ibm_runtime import QiskitRuntimeService

service = QiskitRuntimeService()
backend = service.backend("ibm_brisbane")
noise_model = NoiseModel.from_backend(backend)

estimator = Estimator(backend_options=dict(noise_model=noise_model))
```

```python
from qiskit_aer.noise import ReadoutError, depolarizing_error

noise_model = NoiseModel()
cx_depolarizing_prob = 0.02
bit_flip_prob = 0.05
noise_model.add_all_qubit_quantum_error(
    depolarizing_error(cx_depolarizing_prob, 2), ["cx"]
)
noise_model.add_all_qubit_readout_error(
    ReadoutError(
        [
            [1 - bit_flip_prob, bit_flip_prob],
            [bit_flip_prob, 1 - bit_flip_prob],
        ]
    )
)
```

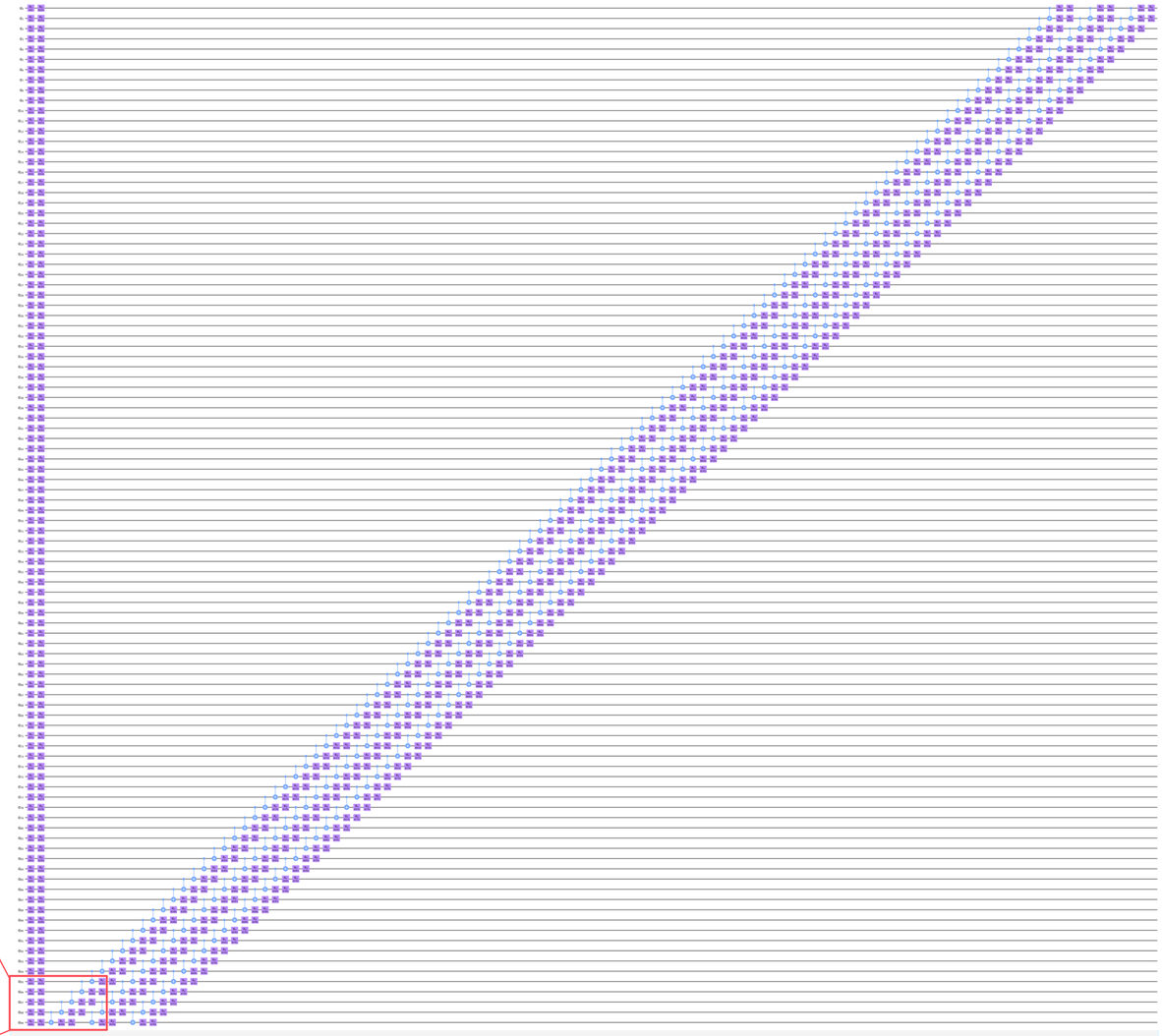
Stabilizer circuits can be simulated efficiently

Stabilizer circuits are a special restricted class of circuits that can be efficiently simulated classically.

To simulate a stabilizer circuit, use a Qiskit Aer primitive with the "stabilizer" simulation method.



Stabilizer circuit if every θ is an integer multiple of $\pi/2$

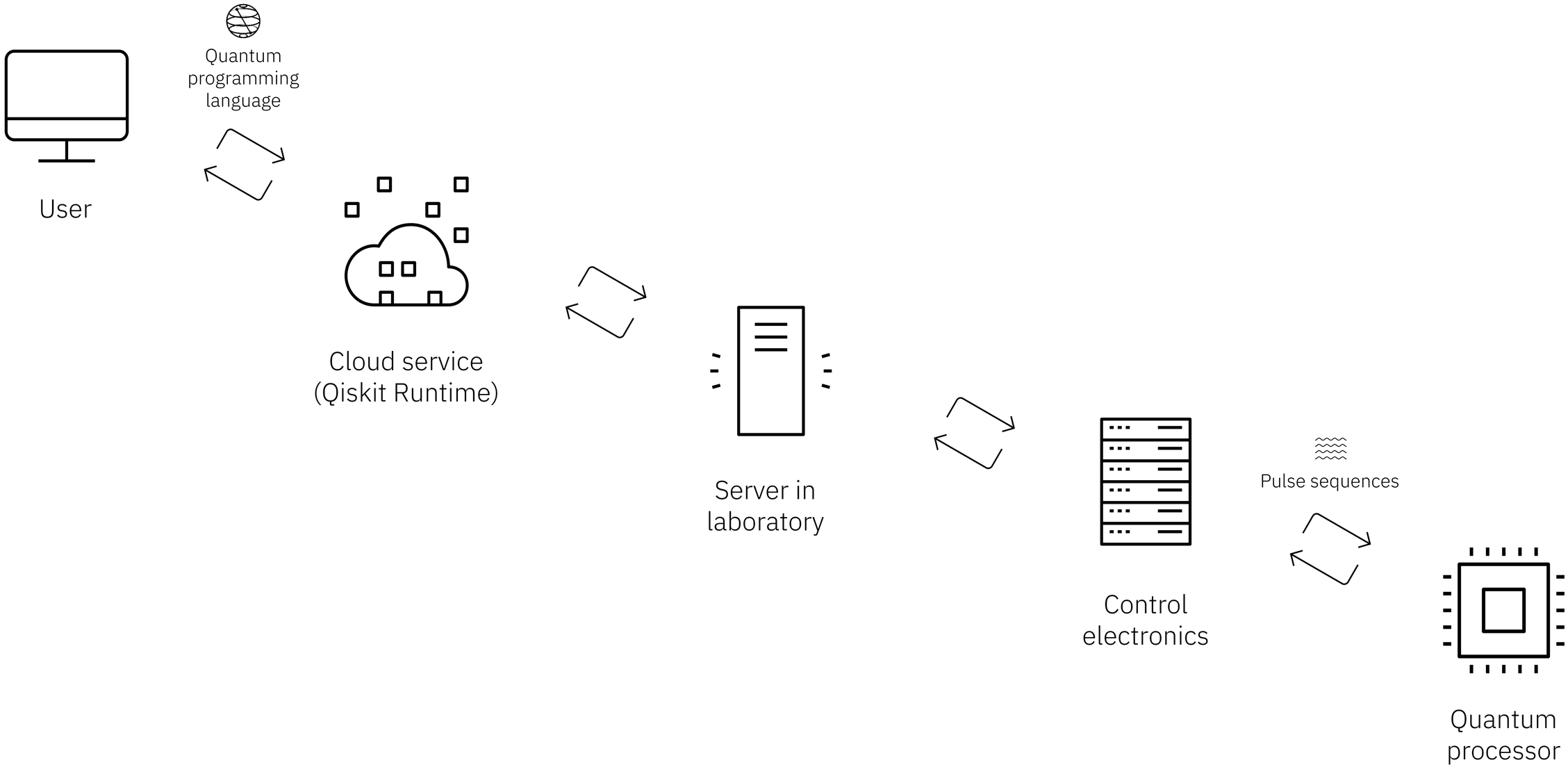


```
estimator = Estimator(backend_options=dict(method="stabilizer"))
```

Run

In the run phase, you send your quantum program to be executed on a quantum system.

Anatomy of a quantum computing service



Run a circuit on quantum hardware using Qiskit Runtime

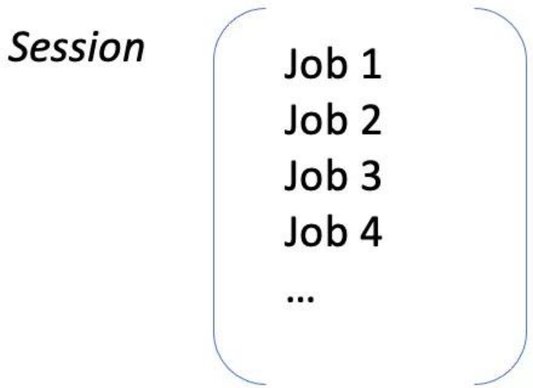
Running a circuit on a real quantum device is similar to running it on a simulator. The main difference is that you use a Qiskit Runtime primitive and choose the hardware backend you want to use.

To run a circuit on quantum hardware:

- Initialize the Qiskit Runtime service (requires setting up account credentials beforehand)
- Initialize a Qiskit Runtime primitive
- Invoke the primitive with your circuit

```python
import numpy as np
from qiskit.circuit.library import IQP
from qiskit.quantum_info import SparsePauliOp, random_hermitian
from qiskit_ibm_runtime import Estimator, QiskitRuntimeService

# Initialize Qiskit Runtime service
service = QiskitRuntimeService()
backend = service.backend("ibm_brisbane")

# Create a circuit and observable
n_qubits = 127
mat = np.real(random_hermitian(n_qubits, seed=1234))
circuit = IQP(mat)
observable = SparsePauliOp("Z" * n_qubits)

# Initialize the Estimator primitive
estimator = Estimator(backend=backend)

# Invoke the Estimator and get results
job = estimator.run(circuit, observable)
result = job.result()
```

A **session** allows a collection of jobs to be grouped and jointly scheduled by the Qiskit Runtime service, facilitating iterative use of quantum computers without incurring queuing delays on each iteration.



Jobs within an active session take priority over other queued jobs.

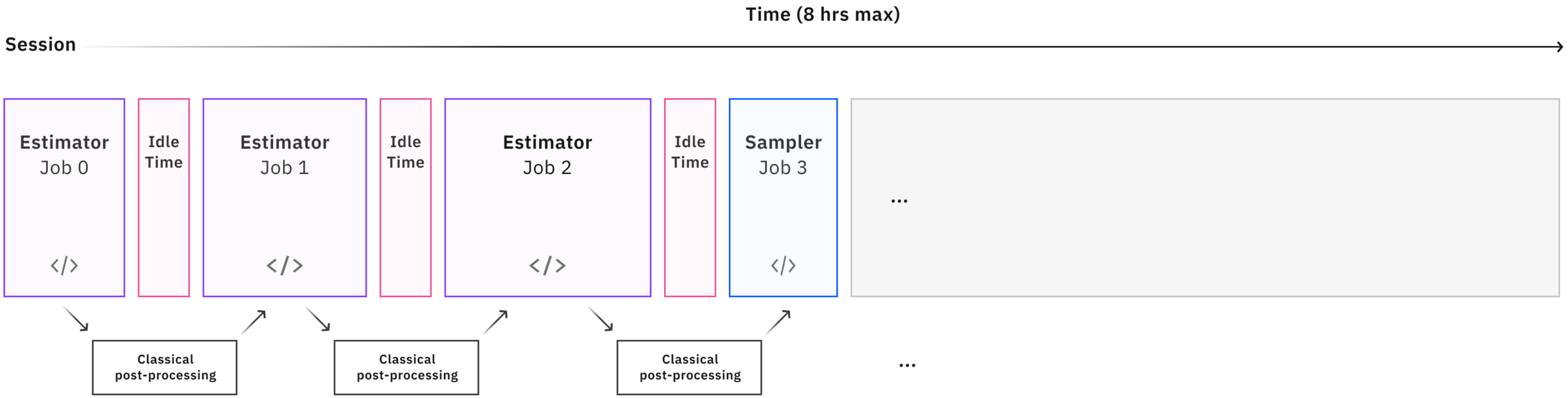
A session becomes active when its first job starts running.

A session stays active until one of the following happens:

- Its maximum timeout value is reached.
- Its interactive timeout value is reached. In this case the session is deactivated but can be resumed if another session job starts running.
- The session is closed or cancelled.

Note: The queuing time does not decrease for the first job submitted within a session.

Sessions for iterative job execution



Run jobs in a session

A session is typically created as a context manager using the `with` statement. The context manager takes care of closing the session when you're done.

To run jobs in a session:

- Initialize the Qiskit Runtime service
- Initialize a session as a context manager, specifying the backend
- Initialize a Qiskit Runtime primitive within the session
- Invoke the primitive as usual

```python
from qiskit_ibm_runtime import Estimator, QiskitRuntimeService, Session

service = QiskitRuntimeService()
backend = service.backend("ibm_brisbane")

with Session(backend=backend):
    estimator = Estimator()
    # invoke the Estimator as usual
```

Several options are available to customize the behavior of Qiskit Runtime primitives

Shots

The number of measurement shots used by the primitives to compute their results. Increasing this value lowers statistical error, but increases running time.

Runtime transpilation

The primitives may perform runtime transpilation to optimize circuits, with the degree of optimization controlled by an optimization level option. The optimization level you choose affects the transpilation strategy, with higher levels invoking more expensive or aggressive optimizations.

Error mitigation

Several error mitigation techniques are available to help you counter the effects of device noise. These techniques incur computational overhead and should be evaluated case-by-case.

Configuring runtime transpilation

The primitives expose preset transpilation configurations via the `optimization_level` option.

You can also elect to disable runtime transpilation using the `skip_transpilation` option.

Optimization level	Effect
0	No optimization: typically used for hardware characterization or debugging • Basis translation • Layout (as specified) • Routing (stochastic swaps)
1 [Default]	Light optimization: • Layout (trivial → vf2 → SabreLayout if routing is required) • Routing (SabreSWAPs if needed) • 1Q gate optimization • Error suppression: dynamical decoupling

Configuring error mitigation

The primitives expose error mitigation via the `resilience_level` option.

Higher resilience levels incur a greater cost.

Resilience level	Definition	Estimator	Sampler
0	No mitigation		
1 [Default]	Minimal mitigation costs: Mitigate error associated with readout errors	Twirled Readout Error eXtinction (TREX)	Matrix-free Measurement Mitigation (M3)
2	Medium mitigation costs. Typically reduces bias in estimators, but is not guaranteed to be zero-bias.	Zero Noise Extrapolation (ZNE)	
3	Heavy mitigation with layer sampling. Theoretically expected to deliver zero-bias estimators.	Probabilistic Error Cancellation (PEC)	

Links

IBM Quantum Documentation

https://docs.quantum-computing.ibm.com/

Access the documentation for Qiskit and IBM Quantum services.

IBM Quantum Learning

https://learning.quantum-computing.ibm.com/

Learn the basics of quantum computing, and how to use IBM Quantum services and systems to solve real-world problems.

Qiskit YouTube

https://www.youtube.com/qiskit

Join us for engaging lectures, tips & tricks, tutorials, community updates and access to exclusive Qiskit content!