Developer Guide

Amazon Braket

Amazon Braket: Developer Guide

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Table of Contents

What is Amazon Braket? ... 1
How it works ... 3
Amazon Braket quantum task flow ... 4
Third-party data processing ... 5
Amazon Braket terms and concepts ... 5
AWS terminology and tips for Amazon Braket ... 9
Cost tracking and saving ... 10
Near real-time cost tracking ... 11
Best practices for cost savings ... 13
API references and repos ... 14
Core repositories ... 15
Plugins ... 15
Supported regions and devices ... 16
Regions and endpoints ... 19
Getting started ... 22
Enable Amazon Braket ... 22
Prerequisites ... 23
Steps to enable Amazon Braket ... 23
Create an Amazon Braket notebook instance ... 24
(Advanced) Create a Braket notebook using CloudFormation ... 26
Step 1: Create a SageMaker AI lifecycle configuration script ... 27
Step 2: Create the IAM role assumed by Amazon SageMaker AI ... 27
Step 3: Create a SageMaker AI notebook instance with the prefix `amazon-braket-` ... 29
Build ... 30
Building your first circuit ... 30
Building your first quantum algorithms ... 35
Constructing circuits in the SDK ... 36
Inspecting the circuit ... 51
List of result types ... 54
Getting Expert advice ... 59
(Advanced) Getting started with Amazon Braket Hybrid Jobs ... 60
What is a Hybrid Job? ... 60
When to use Amazon Braket Hybrid Jobs ... 61
Inputs, outputs, environmental variables, and helper functions ... 62

- Define the environment for your algorithm script 65
- Using hyperparameters 68
- (Advanced) PennyLane with Amazon Braket 70
  - Amazon Braket with PennyLane 71
  - Hybrid algorithms in Amazon Braket example notebooks 73
  - Hybrid algorithms with embedded PennyLane simulators 73
  - Adjoint gradient on PennyLane with Amazon Braket simulators 74
  - Using Hybrid Jobs and PennyLane to run a QAOA algorithm 75
  - Run hybrid workloads with PennyLane embedded simulators 78
- (Advanced) CUDA-Q with Amazon Braket 85
  - Running your workloads on quantum computers 88
- (Advanced) Run your circuits with OpenQASM 3.0 88
  - What is OpenQASM 3.0? 89
  - When to use OpenQASM 3.0 90
  - How OpenQASM 3.0 works 90
  - Prerequisites 90
  - What OpenQASM features does Braket support? 90
  - Create and submit an example OpenQASM 3.0 quantum task 96
  - Support for OpenQASM on different Braket devices 99
  - Simulate noise 109
  - Qubit rewiring 111
  - Verbatim compilation 111
  - The Braket console 112
  - Additional resources 112
  - Computing gradients 112
  - Measuring specific qubits 113
- (Advanced) Explore Experimental Capabilities 114
  - Access to local detuning on QuEra Aquila 115
  - Access to tall geometries on QuEra Aquila 117
  - Access to tight geometries on QuEra Aquila 118
  - Dynamic circuits on IQM devices 119
- (Advanced) Pulse control on Amazon Braket 122
  - Frames 122
  - Ports 123
  - Waveforms 123
  - Roles of frames and ports 124

- Working with Hello Pulse ... 126
- Accessing native gates using pulses ... 129
- (Advanced) Analog Hamiltonian Simulation ... 131
  - Hello AHS: Run your first Analog Hamiltonian Simulation ... 131
  - Submit an analog program using QuEra Aquila ... 145
- (Advanced) Working with AWS Boto3 ... 164
  - Turn on the Amazon Braket Boto3 client ... 165
  - Configure AWS CLI profiles for Boto3 and the Braket SDK ... 168

Test ... 171

- Submitting quantum tasks to simulators ... 171
  - Local state vector simulator (`braket_sv`) ... 172
  - Local density matrix simulator (`braket_dm`) ... 173
  - Local AHS simulator (`braket_ahs`) ... 174
  - State vector simulator (SV1) ... 174
  - Density matrix simulator (DM1) ... 175
  - Tensor network simulator (TN1) ... 176
  - About embedded simulators ... 177
  - Compare simulators ... 178
  - Example quantum tasks on Amazon Braket ... 182
  - Testing a quantum task with the local simulator ... 187
  - Quantum task batching ... 189
- Local quantum device emulator ... 191
  - Benefits of local emulation ... 191
  - Create a local emulator ... 192
- (Advanced) Working with Amazon Braket Hybrid Jobs ... 193
  - Running your local code as a hybrid job ... 193
  - Running a hybrid job with Amazon Braket Hybrid Jobs ... 201
  - Create your first Hybrid Job ... 203
  - Saving your job results ... 213
  - Saving and restarting hybrid jobs using checkpoints ... 214
  - Building and debugging a hybrid job with local mode ... 216

Run ... 218

- Submitting quantum tasks to QPUs ... 219
  - IonQ ... 220
  - IQM ... 221
  - Rigetti ... 221

- QuEra ... 222
- Example: Submitting a quantum task to a QPU ... 222
- Inspecting compiled circuits ... 226
- When will my quantum task run? ... 226
- QPU availability windows and status ... 227
- Queue visibility ... 227
- Set up email or SMS notifications ... 229
- (Advanced) Managing your Amazon Braket Hybrid Job ... 229
- Configure the hybrid job instance to run your script ... 230
- How to cancel a Hybrid Job ... 233
- Using parametric compilation to speed up Hybrid Jobs ... 235
- Bring your own container (BYOC) ... 236
- Interact with hybrid jobs directly using the API ... 244
- (Advanced) Working with reservations ... 247
- How to create a reservation ... 248
- Running quantum tasks during a reservation ... 249
- Running hybrid jobs during a reservation ... 253
- What happens at the end of your reservation ... 254
- Cancel or reschedule an existing reservation ... 255
- (Advanced) Error mitigation techniques ... 255
- Error mitigation techniques on IonQ devices ... 255

Troubleshooting ... 258

- AccessDeniedException ... 258
- An error occurred (ValidationException) when calling the CreateQuantumTask operation ... 258
- An SDK feature does not work ... 259
- Hybrid job fails due to ServiceQuotaExceededException ... 259
- Components stopped working in notebook instance ... 260
- Troubleshooting OpenQASM ... 260
- Include statement error ... 261
- Non-contiguous qubits error ... 261
- Mixing physical qubits with virtual qubits error ... 262
- Requesting result types and measuring qubits in the same program error ... 262
- Classical and qubit register limits exceeded error ... 262
- Box not preceded by a verbatim pragma error ... 263
- Verbatim boxes missing native gates error ... 263
- Verbatim boxes missing physical qubits error ... 263

The verbatim pragma is missing "braket" error 264
Single qubits cannot be indexed error 264
The physical qubits in a two qubit gate are not connected error 264
Local simulator support warning 265
Security 266
Shared responsibility for security 267
Data protection 267
Data retention 268
Managing access to Amazon Braket 268
Amazon Braket resources 269
Notebooks and roles 269
AWS managed policies 269
Restrict user access to certain devices 273
Restrict user access to certain notebook instances 275
Restrict user access to certain S3 buckets 275
Service-linked role 276
Compliance validation 277
Infrastructure Security 278
Third Party Security 279
VPC endpoints (PrivateLink) 279
Considerations for Amazon Braket VPC endpoints 280
Set up Braket and PrivateLink 280
Additional information about creating an endpoint 282
Control access with Amazon VPC endpoint policies 282
Logging and monitoring 284
Tracking quantum tasks from the Amazon Braket SDK 285
Monitoring quantum tasks through the Amazon Braket console 287
Tagging resources 289
Using tags 290
Supported resources for tagging in Amazon Braket 290
Tagging with the Amazon Braket API 291
Tagging restrictions 291
Managing tags in Amazon Braket 291
Example of AWS CLI tagging in Amazon Braket 293
Monitoring your quantum tasks with EventBridge 294
Monitor quantum task status with EventBridge 294

- Example Amazon Braket EventBridge event 295
- Monitoring your metrics with CloudWatch 297
  - Amazon Braket metrics and dimensions 297
- Logging your quantum tasks with CloudTrail 298
  - Amazon Braket information in CloudTrail 298
  - Understanding Amazon Braket log file entries 299
- (Advanced) Logging 301

Quotas 305

- Additional quotas and limits 350

Document history 351

What is Amazon Braket?

Tip

Learn the foundations of quantum computing with AWS! Enroll in the Amazon Braket Digital Learning Plan and earn your own Digital badge after completing a series of learning courses and a digital assessment.

Amazon Braket is a fully managed AWS service that helps researchers, scientists, and developers get started with quantum computing. Quantum computing has the potential to solve computational problems that are beyond the reach of classical computers because it harnesses the laws of quantum mechanics to process information in new ways.

Gaining access to quantum computing hardware can be expensive and inconvenient. Limited access makes it difficult to run algorithms, optimize designs, evaluate the current state of the technology, and plan for when to invest your resources for maximum benefit. Braket helps you overcome these challenges.

Braket offers a single point of access to a variety of quantum computing technologies. With Braket, you can:

- Explore and design quantum and hybrid algorithms.
- Test algorithms on different quantum circuit simulators.
- Run algorithms on different types of quantum computers.
- Create proof of concept applications.

Defining quantum problems and programming quantum computers to solve them requires a new set of skills. To help you gain these skills, Braket offers different environments to simulate and run your quantum algorithms. You can find the approach that best suits your requirements and get started quickly with a set of example environments called *notebooks*.

Braket development has three stages :

- Build - Braket provides fully managed Jupyter notebook environments that make it straightforward to get started. Braket notebooks are pre-installed with sample algorithms, resources, and developer tools, including the Amazon Braket SDK. With the Amazon Braket SDK, you can build quantum algorithms and then test and run them on different quantum computers and simulators by changing a single line of code.

- Test - Braket provides access to fully managed, high-performance quantum circuit simulators. You can test and validate your circuits. Braket handles all the underlying software components and Amazon Elastic Compute Cloud (Amazon EC2) clusters to take away the burden of simulating quantum circuits on classical high performance computing (HPC) infrastructure.
- Run - Braket provides secure, on-demand access to different types of quantum computers. You have access to gate-based quantum computers from IonQ, IQM, and Rigetti, as well as an Analog Hamiltonian Simulator from QuEra. You also have no upfront commitment, and no need to procure access through individual providers.

About quantum computing and Braket

Quantum computing is in its early developmental stage. It's important to understand that no universal, fault-tolerant quantum computer exists at present. Therefore, certain types of quantum hardware are better suited for each use case and it's crucial to have access to a variety of computing hardware. Braket offers a variety of hardware through third-party providers.

Existing quantum hardware is limited due to noise, which introduces errors. The industry is in the Noisy Intermediate Scale Quantum (NISQ) era. In the NISQ era, quantum computing devices are too noisy to sustain pure quantum algorithms, such as *Shor's algorithm* or *Grover's algorithm*. Until better quantum error correction is available, the most practical quantum computing requires the combination of classical (traditional) computing resources with quantum computers to create hybrid algorithms. Braket helps you work with *hybrid quantum algorithms.*

In hybrid quantum algorithms, quantum processing units (QPUs) are used as co-processors for CPUs, thus speeding up specific calculations in a classical algorithm. These algorithms utilize iterative processing, in which computation moves between classical and quantum computers. For example, current applications of quantum computing in chemistry, optimization, and machine learning are based on *variational quantum algorithms*, which are a type of *hybrid quantum algorithm*. In variational quantum algorithms, classical optimization routines adjust the parameters of a parameterized quantum circuit iteratively, much in the same way the weights of a neural network are adjusted iteratively based on the error in a machine learning training set. Braket offers access to the PennyLane open source software library, which assists you with *variational quantum algorithms.*

Quantum computing is gaining traction for computations in four main areas:

- **Number theory** — including factoring and cryptography (for example, *Shor's algorithm* is a primary quantum method for number theory computations)
- **Optimization** — including constraint satisfaction, solving linear systems, and machine learning
- **Oracular computing** — including search, hidden subgroups, and order finding (for example, *Grover's algorithm* is a primary quantum method for oracular computations)
- **Simulation** — including direct simulation, knot invariants, and quantum approximate optimization algorithm (QAOA) applications

Applications for these categories of computations can be found in financial services, biotechnology, manufacturing, and pharmaceuticals, to name a few. Braket offers capabilities and example notebooks that can already be applied to many proof of concept problems in addition to certain practical problems.

In this section:

- How Amazon Braket works
- Amazon Braket terms and concepts
- Cost tracking and saving
- API references and repos for Amazon Braket
- Amazon Braket supported regions and devices

How Amazon Braket works

> **Tip**
>
> **Learn the foundations of quantum computing with AWS!** Enroll in the Amazon Braket Digital Learning Plan and earn your own Digital badge after completing a series of learning courses and a digital assessment.

Amazon Braket provides on-demand access to quantum computing devices, including on-demand circuit simulators and different types of QPUs. In Amazon Braket, the atomic request to a device is a quantum task. For gate-based QC devices, this request includes the quantum circuit (including the measurement instructions and number of shots) and other request metadata. For Analog Hamiltonian Simulators, the quantum task contains the physical layout of the quantum register and the time- and space-dependence of the manipulating fields.

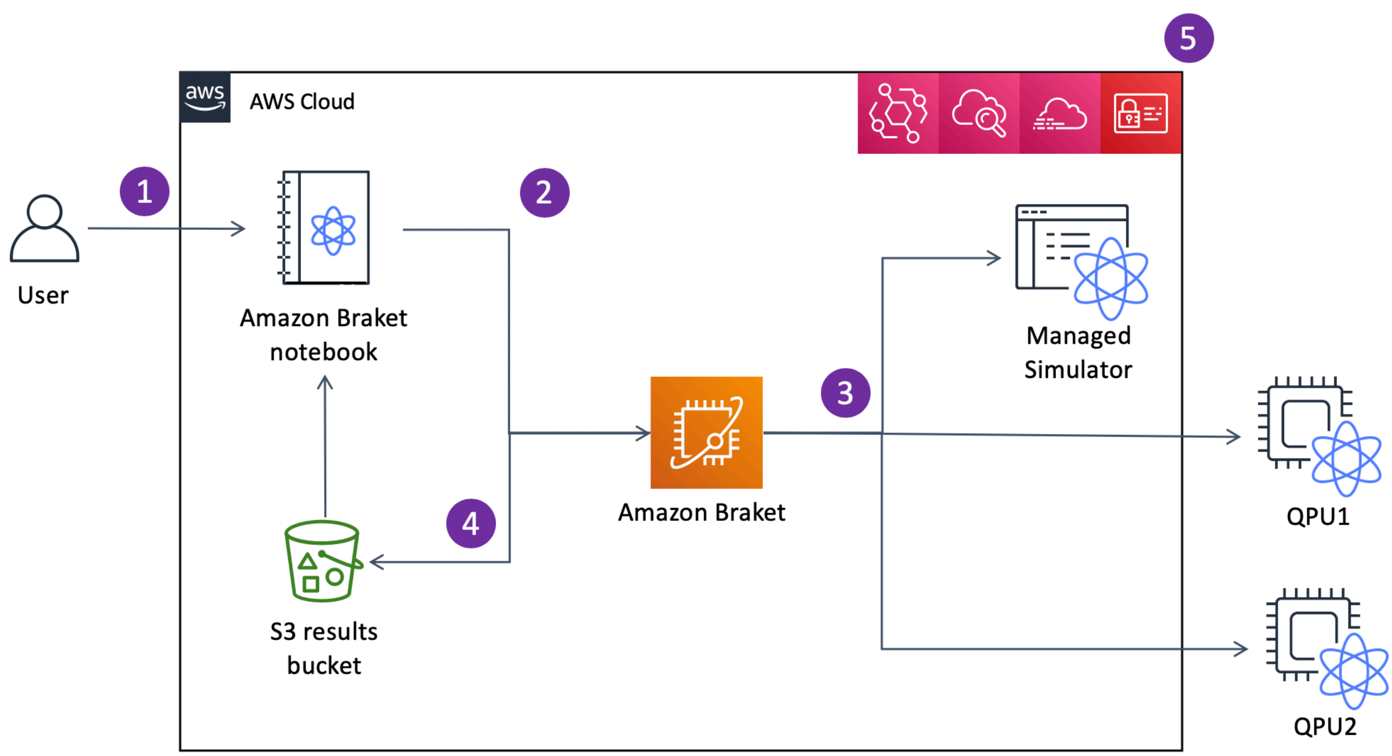
Braket Direct is a program expanding how you can explore quantum computing on AWS, accelerating research and innovation. You can reserve dedicated capacity on various quantum devices, engage directly with quantum computing specialists, and have early access to next-generation capabilities, including the latest trapped-ion device from IonQ, Forte.

In this section, we are going to learn about the high-level flow of running quantum tasks on Amazon Braket.

In this section:

- Amazon Braket quantum task flow
- Third-party data processing

Amazon Braket quantum task flow



With Jupyter notebooks, you can conveniently define, submit, and monitor your quantum tasks from the Amazon Braket Console or using the Amazon Braket SDK. You can build your quantum circuits directly in the SDK. However, for Analog Hamiltonian Simulators, you define the register layout and the controlling fields. After your quantum task is defined, you can choose a device to run it on and submit it to the Amazon Braket API (2). Depending on the device you chose, the quantum task is queued until the device becomes available and the task is sent to the QPU or simulator for implementation (3). Amazon Braket gives you access to different types of QPUs (IonQ, IQM, QuEra, Rigetti), three on-demand simulators (SV1, DM1, TN1), two local simulators, and one embedded simulator. To learn more, see Amazon Braket supported devices.

After processing your quantum task, Amazon Braket returns the results to an Amazon S3 bucket, where the data is stored in your AWS account (4). At the same time, the SDK polls for the results in the background and loads them into the Jupyter notebook at quantum task completion. You can also view and manage your quantum tasks on the **Quantum Tasks** page in the Amazon Braket console or by using the `GetQuantumTask` operation of the Amazon Braket API.

Amazon Braket is integrated with AWS Identity and Access Management (IAM), Amazon CloudWatch, AWS CloudTrail and Amazon EventBridge for user access management, monitoring and logging as well as for event based processing (5).

Third-party data processing

Quantum tasks that are submitted to a QPU device are processed on quantum computers located in facilities operated by third party providers. To learn more about security and third-party processing in Amazon Braket, see Security of Amazon Braket Hardware Providers.

Amazon Braket terms and concepts

> **Tip**
>
> **Learn the foundations of quantum computing with AWS!** Enroll in the Amazon Braket Digital Learning Plan and earn your own Digital badge after completing a series of learning courses and a digital assessment.

The following terms and concepts are used in Braket:

Analog Hamiltonian Simulation

Analog Hamiltonian Simulation (AHS) is a distinct quantum computing paradigm for direct simulation of time-dependent quantum dynamics of many-body systems. In AHS, users directly specify a time-dependent Hamiltonian and the quantum computer is tuned in such a way that it directly emulates the continuous time evolution under this Hamiltonian. AHS devices are typically special-purpose devices and not universal quantum computers like gate-based devices. They are limited to a class of Hamiltonians they can simulate. However, since these Hamiltonians are naturally implemented on the device, AHS does not suffer from the overhead required to formulate algorithms as circuits and implement gate operations.

Braket

We named the Braket service after the bra-ket notation, a standard notation in quantum mechanics. It was introduced by Paul Dirac in 1939 to describe the state of quantum systems, and it is also known as the Dirac notation.

Braket Direct

With Braket Direct, you can reserve dedicated access to different quantum devices of your choice, connect with quantum computing specialists to receive guidance for your workload, and gain early access to next-generation capabilities, such as new quantum devices with limited availability.

Braket hybrid job

Amazon Braket has a feature called Amazon Braket Hybrid Jobs that provides fully managed executions of hybrid algorithms. A Braket hybrid job consists of three components:

1. The definition of your algorithm, which can be provided as a script, Python module, or Docker container.
2. The *hybrid job instance*, based on Amazon EC2, on which to run your algorithm. The default is an ml.m5.xlarge instance.
3. The *quantum device* on which to run the *quantum tasks* that are part of your algorithm. A single hybrid job typically contains a collection of many quantum tasks.

Device

In Amazon Braket, a device is a backend that can run *quantum tasks*. A device can be a *QPU* or a *quantum circuit simulator*. To learn more, see Amazon Braket supported devices.

Error mitigation

Error mitigation involves running multiple physical circuits and combining their measurements to give an improved result. For more information, see Error mitigation techniques.

Gate-based quantum computing

In gate-based quantum computing (QC), also called circuit-based QC, computations are broken down into elementary operations (gates). Certain sets of gates are universal, meaning that every computation can be expressed as a finite sequence of those gates. Gates are the building blocks of *quantum circuits* and are analogous to the logic gates of classical digital circuits.

Gateshot limit

A gateshot limit refers to the total gate count per shot (the sum of all gate types) and shot count per task. Mathematically, the gateshot limit can be expressed as:

```
Gateshot limit = (Gate count per shot) * (Shot count per task)
```

Hamiltonian

The quantum dynamics of a physical system are determined by its Hamiltonian, which encodes all information about the interactions between constituents of the system and the effects of exogenous driving forces. The Hamiltonian of an N-qubit system is commonly represented as a 2^N by 2^N matrix of complex numbers on classical machines. By running an Analog Hamiltonian Simulation on a quantum device, you can avoid these exponential resource requirements.

Pulse

A pulse is a transient physical signal transmitted to the qubits. It is described by a waveform played in a frame that serves as a support for the carrier signal and is bound to the hardware channel or port. Customers can design their own pulses by providing the analog envelope that modulates the high-frequency sinusoidal carrier signal. The frame is uniquely described by a frequency and a phase that are often chosen to be on resonance with the energy separation between the energy levels for $|0\rangle$ and $|1\rangle$ of the qubit. Gates are thus enacted as pulses with a predetermined shape and calibrated parameters such as its amplitude, frequency and duration. Use cases that are not covered by template waveforms will be enabled through custom waveforms which will be specified at the single sample resolution by providing a list of values separated by a fixed, physical cycle-time.

Quantum circuit

A quantum circuit is the instruction set that defines a computation on a gate-based quantum computer. A quantum circuit is a sequence of quantum gates, which are reversible transformations on a qubit register, together with measurement instructions.

Quantum circuit simulator

A quantum circuit simulator is a computer program that runs on classical computers and calculates the measurement outcomes of a *quantum circuit*. For general circuits, the resource requirements of a quantum simulation grow exponentially with the number of qubits to simulate. Braket provides access to both managed (accessed through the Braket API) and local (part of the Amazon Braket SDK) quantum circuit simulators.

Quantum computer

A quantum computer is a physical device that uses quantum-mechanical phenomena, such as superposition and entanglement, to perform computations. There are different paradigms to quantum computing (QC), such as *gate-based* QC.

Quantum processing unit (QPU)

A QPU is a physical quantum computing device that can run on a quantum task. QPUs can be based on different QC paradigms, such as gate-based QC. To learn more, see [Amazon Braket supported devices].

QPU native gates

QPU native gates can be directly mapped to control pulses by the QPU control system. Native gates can be run on the QPU device without further compilation. Subset of *QPU supported gates*. You can find the native gates of a device on the **Devices** page in the Amazon Braket console and through the Braket SDK.

QPU supported gates

QPU supported gates are the gates accepted by the QPU device. These gates might not run directly on the QPU, meaning that they might need to be decomposed into native gates. You can find the supported gates of a device on the **Devices** page in the Amazon Braket console and through the Amazon Braket SDK.

Quantum task

In Braket, a quantum task is the atomic request to a *device*. For *gate-based QC* devices, this includes the quantum circuit (including the measurement instructions and number of shots) and other request metadata. You can create quantum tasks through the Amazon Braket SDK or by using the CreateQuantumTask API operation directly. After you create a quantum task, it will be queued until the requested device becomes available. You can view your quantum tasks on the **Quantum Tasks** page of the Amazon Braket console or by using the GetQuantumTask or SearchQuantumTasks API operations.

Qubit

The basic unit of information in a quantum computer is called a qubit (quantum bit), much like a bit in classical computing. A qubit is a two-level quantum system that can be realized by different physical implementations, such as superconducting circuits or individual ions and atoms. Other qubit types are based on photons, electronic or nuclear spins, or more exotic quantum systems.

Queue depth

Queue depth refers to the number of quantum tasks and hybrid jobs queued for a particular device. A device's quantum task and hybrid job queue count are accessible through the Braket Software Development Kit (SDK) or Amazon Braket Management Console.

1. *Task queue depth* refers to the total number of quantum tasks waiting to run in normal priority.
2. *Priority task queue depth* refers to the total number of submitted quantum tasks waiting to run through Amazon Braket Hybrid Jobs. These tasks get priority over standalone tasks once a hybrid job starts.
3. *Hybrid jobs queue depth* refers to the total number of hybrid jobs currently queued on a device. Quantum tasks submitted as part of a hybrid job have priority, and are aggregated in the Priority Task Queue.

Queue position

Queue position refers to the current position of your quantum task or hybrid job within a respective device queue. It can be obtained for quantum tasks or hybrid jobs through the Braket Software Development Kit (SDK) or Amazon Braket Management Console.

Shots

Since quantum computing is inherently probabilistic, any circuit needs to be evaluated multiple times to get an accurate result. A single circuit execution and measurement is called a shot. The number of shots (repeated executions) for a circuit is chosen based on the desired accuracy for the result.

AWS terminology and tips for Amazon Braket

IAM policies

An IAM policy is a document that allows or denies permissions to AWS services and resources. IAM policies enable you to customize users' levels of access to resources. For example, you can allow users access to all of the Amazon S3 buckets within your AWS account, or only a specific bucket.

- **Best practice:** Follow the security principle of *least privilege* when granting permissions. By following this principle, you help to prevent users or roles from having more permissions than needed to perform their quantum tasks. For example, if an employee needs access to only a specific bucket, specify the bucket in the IAM policy instead of granting the employee access to all of the buckets in your AWS account.

IAM roles

An IAM role is an identity that you can assume to gain temporary access to permissions. Before a user, application, or service can assume an IAM role, they must be granted permissions to switch to the role. When someone assumes an IAM role, they abandon all previous permissions that they had under a previous role and assume the permissions of the new role.

- **Best practice:** IAM roles are ideal for situations in which access to services or resources needs to be granted temporarily, instead of long-term.

Amazon S3 bucket

Amazon Simple Storage Service (Amazon S3) is an AWS service that lets you store data as *objects* in *buckets*. Amazon S3 buckets offer unlimited storage space. The maximum size for an object in an Amazon S3 bucket is 5 TB. You can upload any type of file data to an Amazon S3 bucket, such as images, videos, text files, backup files, media files for a website, archived documents, and your Braket quantum task results.

- **Best practice:** You can set permissions to control access to your S3 bucket. For more information, see Bucket policies in the Amazon S3 documentation.

Cost tracking and saving

Tip

Learn the foundations of quantum computing with AWS! Enroll in the Amazon Braket Digital Learning Plan and earn your own Digital badge after completing a series of learning courses and a digital assessment.

With Amazon Braket, you have access to quantum computing resources on demand without upfront commitment. You pay only for what you use. To learn more about pricing, visit our pricing page.

In this section:

- Near real-time cost tracking

- Best practices for cost savings

Near real-time cost tracking

The Braket SDK offers you the option to add a near real-time cost tracking to your quantum workloads. Each of our example notebooks includes cost tracking code to provide you with a maximum cost estimate on Braket's quantum processing units (QPUs) and on-demand simulators. Maximum cost estimates will be shown in USD and are not inclusive of any credits or discounts.

Note

Charges shown are estimates based on your Amazon Braket simulator and quantum processing unit (QPU) task usage. Estimated charges shown may differ from your actual charges. Estimated charges do not factor in any discounts or credits and you may experience additional charges based on your use of other services such as Amazon Elastic Compute Cloud (Amazon EC2).

Cost tracking for SV1

In order to demonstrate how the cost tracking function can be used, we will be constructing a Bell State circuit and running it on our SV1 simulator. Begin by importing the Braket SDK modules, defining a Bell State and adding the `Tracker()` function to our circuit:

```
#import any required modules
from braket.aws import AwsDevice
from braket.circuits import Circuit
from braket.tracking import Tracker

#create our bell circuit
circ = Circuit().h(0).cnot(0,1)
device = AwsDevice("arn:aws:braket:::device/quantum-simulator/amazon/sv1")
with Tracker() as tracker:
    task = device.run(circ, shots=1000).result()

#Your results
print(task.measurement_counts)
```

```
Counter({'00': 500, '11': 500})
```

When you run your Notebook, you can expect the following output for your Bell State simulation. The tracker function will show you the number of shots sent, quantum tasks completed, the execution duration, the billed execution duration, and your maximum cost in USD. Your execution time may vary for each simulation.

```
import datetime

tracker.quantum_tasks_statistics()
{'arn:aws:braket:::device/quantum-simulator/amazon/sv1':
 {'shots': 1000,
  'tasks': {'COMPLETED': 1},
  'execution_duration': datetime.timedelta(microseconds=4000),
  'billed_execution_duration': datetime.timedelta(seconds=3)}}

tracker.simulator_tasks_cost()
```

```
Decimal('0.0037500000')
```

Using the cost tracker to set maximum costs

You can use the cost tracker to set maximum costs on a program. You may have a maximum threshold for how much you want to spend on a given program. In this way, you can use the cost tracker to build out cost control logic in your execution code. The following example takes the same circuit on a Rigetti QPU and limits the cost to 1 USD. The cost to run one iteration of the circuit in our code is 0.30 USD. We have set the logic to repeat the iterations until the total cost exceeds 1 USD; hence, the code snippet will run three times until the next iteration exceeds 1 USD. Generally, a program would continue to iterate until it reaches your desired maximum cost, in this case - three iterations.

```
device = AwsDevice("arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3")
with Tracker() as tracker:
    while tracker.qpu_tasks_cost() < 1:
        result = device.run(circ, shots=200).result()
print(tracker.quantum_tasks_statistics())
print(tracker.qpu_tasks_cost(), "USD")
```

```
{'arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3': {'shots': 600, 'tasks':
 {'COMPLETED': 3}}}
1.4400000000 USD
```

> **Note**
>
> The cost tracker will not track duration for failed TN1 quantum tasks. During a TN1 simulation, if your rehearsal completes, but the contraction step fails, your rehearsal charge will not be shown in the cost tracker.

Best practices for cost savings

Consider the following best practices for using Amazon Braket. Save time, minimize costs, and avoid common errors.

Verify with simulators

- Verify your circuits using a simulator before you run it on a QPU, so you can fine-tune your circuit without incurring charges for QPU usage.
- Although the results from running the circuit on a simulator may not be identical to the results from running the circuit on a QPU, you can identify coding errors or configuration issues using a simulator.

Restrict user access to certain devices

- You can set up restrictions that keep unauthorized users from submitting quantum tasks on certain devices. The recommended method for restricting access is with AWS IAM. For more information about how to do that, see Restrict access.
- We recommend that you do **not** use your **admin** account as a way to give or restrict user access to Amazon Braket devices.

Set billing alarms

- You can set a billing alarm to notify you when your bill reaches a preset limit. The recommended way to set up an alarm is through AWS Budgets. You can set custom budgets and receive alerts when your costs or usage may exceed your budgeted amount. Information is available at AWS Budgets.

Test TN1 quantum tasks with low shot counts

- Simulators cost less than QPUs, but certain simulators can be expensive if quantum tasks are run with high shot counts. We recommend that you test your TN1 tasks with a low shot count. Shot count does not affect the cost for SV1 and local simulator tasks.

Check all Regions for quantum tasks

- The console displays quantum tasks only for your current AWS Region. When looking for billable quantum tasks that have been submitted, be sure to check all Regions.
- You can view a list of devices and their associated Regions on the Supported Devices documentation page.

API references and repos for Amazon Braket

> **Tip**
>
> **Learn the foundations of quantum computing with AWS!** Enroll in the Amazon Braket Digital Learning Plan and earn your own Digital badge after completing a series of learning courses and a digital assessment.

Amazon Braket provides APIs, SDKs, and a command line interface that you can use to create and manage notebook instances and train and deploy models.

- Amazon Braket Python SDK (Recommended)
- Amazon Braket API Reference
- AWS Command Line Interface
- AWS SDK for .NET
- AWS SDK for C++
- AWS SDK for GoAPI Reference
- AWS SDK for Java
- AWS SDK for JavaScript
- AWS SDK for PHP
- AWS SDK for Python (Boto)
- AWS SDK for Ruby

You can also get code examples from the Amazon Braket Tutorials GitHub repository.

- Braket Tutorials GitHub

Core repositories

The following displays a list of core repositories that contain key packages that are used for Braket:

- Braket Python SDK - Use the Braket Python SDK to set up your code on Jupyter notebooks in the Python programming language. After your Jupyter notebooks are set up, you can run your code on Braket devices and simulators
- Braket Schemas - The contract between the Braket SDK and the Braket service.
- Braket Default Simulator - All our local quantum simulators for Braket (state vector and density matrix).

Plugins

Then there are the various plugins that are used along with various devices and programming tools. These include Braket supported plugins as well as plugins that are supported by third parties as shown below.

Amazon Braket supported:

- Amazon Braket algorithm library - A catalog of pre-built quantum algorithms written in Python. Run them as they are or use them as a starting point to build more complex algorithms.
- Braket-PennyLane plugin - Use PennyLane as the QML framework on Braket.

Third-party (Braket team monitors and contributes):

- Qiskit-Braket provider - Use the Qiskit SDK to access Braket resources.
- Braket-Julia SDK - (EXPERIMENTAL) A Julia native version of the Braket SDK

Amazon Braket supported regions and devices

Tip

Learn the foundations of quantum computing with AWS! Enroll in the Amazon Braket Digital Learning Plan and earn your own Digital badge after completing a series of learning courses and a digital assessment.

In Amazon Braket, a device represents a QPU or simulator that you can call to run quantum tasks. Amazon Braket provides access to QPU devices from IonQ, IQM, QuEra, and Rigetti, three on-demand simulators, three local simulators, and one embedded simulator.

For information on supported quantum hardware providers, see Submitting quantum tasks to QPUs. For information on available simulators, see Submitting quantum tasks to simulators. The following table displays the list of available devices and simulators.

Provider	Device Name	Paradigm	Type	Device ARN	Region
IonQ	Aria-1	gate-based	QPU	arn:aws:braket:us-east-1::device/qpu/ionq/Aria-1	us-east-1
IonQ	Aria-2	gate-based	QPU	arn:aws:braket:us-east-1::device/qpu/ionq/Aria-2	us-east-1
IonQ	Forte-1	gate-based	QPU	arn:aws:braket:us-east-1::device/qpu/ionq/Forte-1	us-east-1
IonQ	Forte-Enterprise-1	gate-based	QPU	arn:aws:braket:us-east-1::device/qpu/ionq/Forte-Enterprise-1	us-east-1

Provider	Device Name	Paradigm	Type	Device ARN	Region
IQM	Garnet	gate-based	QPU	arn:aws:braket:eu-north-1::device/qpu/iqm/Garnet	eu-north-1
IQM	Emerald	gate-based	QPU	arn:aws:braket:eu-north-1::device/qpu/iqm/Emerald	eu-north-1
QuEra	Aquila	Analog Hamiltonian Simulation	QPU	arn:aws:braket:us-east-1::device/qpu/quera/Aquila	us-east-1
Rigetti	Ankaa-3	gate-based	QPU	arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3	us-west-1
AWS	braket_sv	gate-based	Local simulator	N/A (local simulator in Braket SDK)	N/A
AWS	braket_dm	gate-based	Local simulator	N/A (local simulator in Braket SDK)	N/A
AWS	braket_ahs	Analog Hamiltonian Simulation	Local simulator	N/A (local simulator in Braket SDK)	N/A

Provider	Device Name	Paradigm	Type	Device ARN	Region
AWS	SV1	gate-based	On-demand simulator	arn:aws:braket:::device/quantum-simulator/amazon/sv1	us-east-1, us-west-1, us-west-2, eu-west-2
AWS	DM1	gate-based	On-demand simulator	arn:aws:braket:::device/quantum-simulator/amazon/dm1	us-east-1, us-west-1, us-west-2, eu-west-2
AWS	TN1	gate-based	On-demand simulator	arn:aws:braket:::device/quantum-simulator/amazon/tn1	us-east-1, us-west-2, and eu-west-2

To view additional details about the QPUs you can use with Amazon Braket, see Amazon Braket Quantum Computers.

Device properties

For all devices, you can find further device properties, such as device topology, calibration data, and native gate sets, on the **Devices** tab of the Amazon Braket console or by the `GetDevice` API. When constructing a circuit with the simulators, Amazon Braket requires that you use contiguous qubits or indices. When working with the SDK, the following code example shows how to get access to the device properties for each available device and simulator.

```
from braket.aws import AwsDevice
from braket.devices import LocalSimulator
```

```
device = AwsDevice('arn:aws:braket:::device/quantum-simulator/amazon/sv1')
 #SV1
# device = LocalSimulator()
 #Local State Vector Simulator
# device = LocalSimulator("default")
 #Local State Vector Simulator
# device = LocalSimulator(backend="default")
 #Local State Vector Simulator
# device = LocalSimulator(backend="braket_sv")
 #Local State Vector Simulator
# device = LocalSimulator(backend="braket_dm")
 #Local Density Matrix Simulator
# device = LocalSimulator(backend="braket_ahs")
 #Local Analog Hamiltonian Simulation
# device = AwsDevice('arn:aws:braket:::device/quantum-simulator/amazon/tn1')
 #TN1
# device = AwsDevice('arn:aws:braket:::device/quantum-simulator/amazon/dm1')
 #DM1
# device = AwsDevice('arn:aws:braket:us-east-1::device/qpu/ionq/Aria-1')
 #IonQ Aria-1
# device = AwsDevice('arn:aws:braket:us-east-1::device/qpu/ionq/Aria-2')
 #IonQ Aria-2
# device = AwsDevice('arn:aws:braket:us-east-1::device/qpu/ionq/Forte-1')
 #IonQ Forte-1
# device = AwsDevice('arn:aws:braket:us-east-1::device/qpu/ionq/Forte-Enterprise-1')
 #IonQ Forte-Enterprise-1
# device = AwsDevice('arn:aws:braket:eu-north-1::device/qpu/iqm/Garnet')
 #IQM Garnet
# device = AwsDevice('arn:aws:braket:eu-north-1::device/qpu/iqm/Emerald')
 #IQM Emerald
# device = AwsDevice('arn:aws:braket:us-east-1::device/qpu/quera/Aquila')
 #QuEra Aquila
# device = AwsDevice('arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3')
 #Rigetti Ankaa-3

# get device properties
device.properties
```

Regions and endpoints for Amazon Braket

For a full list of regions and endpoints, see the AWS General Reference.

Quantum tasks that run on a QPU device can be viewed in the Amazon Braket console in the Region of that device. When using the Amazon Braket SDK, you can submit quantum tasks to any QPU device, regardless of the Region in which you are working. The SDK automatically creates a session to the Region for the QPU specified.

Amazon Braket is available in the following AWS Regions:

Region name	Region	Braket endpoints
US East (N. Virginia)	us-east-1	braket.us-east-1.amazonaws.com (IPv4 only) braket.us-east-1.api.aws (Dual stack)
US West (N. California)	us-west-1	braket.us-west-1.amazonaws.com (IPv4 only) braket.us-west-1.api.aws (Dual stack)
US West 2 (Oregon)	us-west-2	braket.us-west-2.amazonaws.com (IPv4 only) braket.us-west-2.api.aws (Dual stack)
EU North 1 (Stockholm)	eu-north-1	braket.eu-north-1.amazonaws.com (IPv4 only) braket.eu-north-1.api.aws (Dual stack)
EU West 2 (London)	eu-west-2	braket.eu-west-2.amazonaws.com (IPv4 only) braket.eu-west-2.api.aws (Dual stack)

> **Note**
>
> The Amazon Braket SDK does not support IPv6-only networks.

Getting started with Amazon Braket

> **Tip**
>
> **Learn the foundations of quantum computing with AWS!** Enroll in the Amazon Braket Digital Learning Plan and earn your own Digital badge after completing a series of learning courses and a digital assessment.

After you have followed the instructions in Enable Amazon Braket , you can get started with Amazon Braket.

The steps to get started include:

- Enable Amazon Braket
- Create an Amazon Braket notebook instance
- Create a Braket notebook instance using AWS CloudFormation

Enable Amazon Braket

> **Tip**
>
> **Learn the foundations of quantum computing with AWS!** Enroll in the Amazon Braket Digital Learning Plan and earn your own Digital badge after completing a series of learning courses and a digital assessment.

You can enable Amazon Braket in your account through the AWS console.

In this section:

- Prerequisites
- Steps to enable Amazon Braket

Prerequisites

To enable and run Amazon Braket, you must have a user or role with permission to initiate Amazon Braket actions. These permissions are included in the **AmazonBraketFullAccess** IAM policy (arn:aws:iam::aws:policy/AmazonBraketFullAccess).

> **Note**
>
> *If you are an administrator:*
> To give other users access to Amazon Braket, grant users permissions by attaching the *AmazonBraketFullAccess* policy or by attaching a custom policy that you create. To learn more about the permissions necessary to use Amazon Braket, see Managing access to Amazon Braket .

Steps to enable Amazon Braket

1. Sign in to the Amazon Braket console with your AWS account.
2. Open the Amazon Braket console.
3. From the Braket landing page, click Get Started to be taken to the **Service Dashboard** page. The alert at the top of your service dashboard will walk you through the following three steps:
   a. Creating service-linked roles (SLR)
   b. Enabling access to third-party quantum computers
   c. Creating a new Jupyter notebook instance

In order to use third-party quantum devices, you need to agree to certain conditions regarding data transfer between yourself, AWS, and those devices. The terms and conditions of this agreement are provided on the **General** tab of the **Permissions and settings** page in the Amazon Braket console.

> **Note**
>
> Quantum devices that don't involve any third-parties, such as the Braket local simulators or on-demand simulators, can be used without agreeing to the **Enable third-party devices** agreement.
> Accepting these terms to enable use of third-party devices only needs to be done **once per account** if you are accessing third-party hardware.

Create an Amazon Braket notebook instance

> **Tip**
>
> **Learn the foundations of quantum computing with AWS!** Enroll in the Amazon Braket Digital Learning Plan and earn your own Digital badge after completing a series of learning courses and a digital assessment.

Amazon Braket provides fully managed Jupyter notebooks to get you started. The Amazon Braket notebook instances are based on Amazon SageMaker AI notebook instances. The following steps outline how to create a new notebook instance for new and existing customers.

New Amazon Braket customers:

1. Open the Amazon Braket console and navigate to the **Dashboard** page in the left pane.
2. Click **Get Started** on the **Welcome to Amazon Braket** modal in the center of your dashboard page. Provide a notebook name to create a default Jupyter notebook.
   a. It may take several minutes to create your notebook.
   b. Your notebook will be listed on the **Notebooks** page with a status of **Pending**.
   c. When your notebook instance is ready to use, the status changes to **InService**.
   d. Refresh the page to display the updated status for the notebook.

Existing Amazon Braket customers:

1. Open the Amazon Braket console and select **Notebooks** in the left pane.
2. Select **Create notebook instance**.
   a. If you have zero notebooks, select the **Standard setup** to create a default Jupyter notebook.
3. Enter a **Notebook instance name**, using only alphanumeric and hyphen characters, and select your preferred **Visual Mode**.
4. Enable or disable the **Notebook inactivity manager** for your notebook.
   a. If enabled, select the desired idle duration time before the notebook is reset. When a notebook is reset, the compute charges stop incurring, but the storage charges will continue.
   b. To check how much idle time remains for your notebook instance, navigate to the command bar, select the **Braket** tab, and then the **Inactivity Manager** tab.

> **Note**
>
> To save your work, integrate your SageMaker AI notebook instance with a Git repository. Alternately, move your work outside of the `/Braket Algorithms` and `/Braket Examples` folders so they are not overwritten by the notebook instance restarting.

5. (Optional) With **Advanced setup**, you can create a notebook with access permissions, additional configurations, and network access settings:
   a. In **Notebook configuration** choose your instance type.
      i. The standard, cost-effective instance type, **ml.t3.medium** is chosen by default. To learn more about instance pricing, see Amazon SageMaker AI pricing.
   b. To associate a public Github repository with your notebook instance, click on the **Git repository** dropdown and select **Clone a public git repository from url** from the **Repository** dropdown menu. Enter the URL of the repo in the **Git repository URL** text bar.
   c. In **Access permissions**, configure any optional IAM roles, root access, and encryption keys.
   d. In **Network access**, configure custom network and access settings for your Jupyter Notebook instance.
6. Review your settings, and set any tags to identify your notebook instance. Click **Launch**.

> **Note**
>
> View and manage your Amazon Braket notebook instances in the Amazon Braket and Amazon SageMaker AI consoles. Additional Amazon Braket notebook settings are available through the SageMaker console.

If you are working in the Amazon Braket console within AWS the Amazon Braket SDK and plugins are preloaded in the notebooks you created. To run on your own machine, install the SDK and plugins when you run the command `pip install amazon-braket-sdk` or when you run the command `pip install amazon-braket-pennylane-plugin` for PennyLane plugins.

Create a Braket notebook instance using AWS CloudFormation

> **Tip**
>
> **Learn the foundations of quantum computing with AWS!** Enroll in the Amazon Braket Digital Learning Plan and earn your own Digital badge after completing a series of learning courses and a digital assessment.

You can use AWS CloudFormation to manage your Amazon Braket notebook instances. Braket notebook instances are built on Amazon SageMaker AI. With CloudFormation, you can provision a notebook instance with a template file that describes the intended configuration. The template file is written in JSON or YAML format. You can create, update, and delete instances in an orderly and repeatable fashion. You may find this useful when you manage multiple Braket notebook instances in your AWS account.

After you create a CloudFormation template for a Braket notebook, you use AWS CloudFormation to deploy the resource. For more information, see Creating a stack on the AWS CloudFormation console in the *AWS CloudFormation user guide*.

To create a Braket notebook instance using CloudFormation, you perform these three steps:

1. Create a SageMaker AI lifecycle configuration script.
2. Create an AWS Identity and Access Management (IAM) role to be assumed by SageMaker AI.
3. Create a SageMaker AI notebook instance with the prefix **`amazon-braket-`**

You can reuse the lifecycle configuration for all of the Braket notebooks that you create. You can also reuse the IAM role for the Braket notebooks that you assign the same execution permissions.

In this section:

- Step 1: Create a SageMaker AI lifecycle configuration script
- Step 2: Create the IAM role assumed by Amazon SageMaker AI
- Step 3: Create a SageMaker AI notebook instance with the prefix amazon-braket-

Step 1: Create a SageMaker AI lifecycle configuration script

Use the following template to create a SageMaker AI lifecycle configuration script. The script customizes a SageMaker AI notebook instance for Braket. For configuration options for the lifecycle CloudFormation resource, see AWS::SageMaker::NotebookInstanceLifecycleConfig in the *AWS CloudFormation user guide*.

```
BraketNotebookInstanceLifecycleConfig:
    Type: "AWS::SageMaker::NotebookInstanceLifecycleConfig"
    Properties:
      NotebookInstanceLifecycleConfigName: BraketLifecycleConfig-${AWS::StackName}
      OnStart:
      - Content:
          Fn::Base64: |
            #!/usr/bin/env bash
            sudo -u ec2-user -i #EOS
            curl -o braket-notebook-lcc.zip https://d3ded4lzb1lnme.cloudfront.net/
notebook/braket-notebook-lcc.zip
            unzip braket-notebook-lcc.zip
            ./install.sh
            EOS

            exit 0
```

Step 2: Create the IAM role assumed by Amazon SageMaker AI

When you use a Braket notebook instance, SageMaker AI performs operations on your behalf. For example, suppose you run a Braket notebook using a circuit on a supported device. Within the notebook instance, SageMaker AI runs the operation on Braket for you. The notebook execution role defines the exact operations that SageMaker AI is permitted to execute on your behalf. For more information, see SageMaker AI roles in the *Amazon SageMaker AI developer guide*.

Use the following example to create a Braket notebook execution role with the required permissions. You can modify the policies according to your needs.

> **Note**
>
> Make sure that the role has permission for the `s3:ListBucket` and `s3:GetObject`operations on Amazon S3 buckets prefixed with `braketnotebookcdk-"`.

The lifecycle configuration script requires these permissions to copy the Braket notebook installation script.

```
ExecutionRole:
    Type: "AWS::IAM::Role"
    Properties:
      RoleName: !Sub AmazonBraketNotebookRole-${AWS::StackName}
      AssumeRolePolicyDocument:
        Version: "2012-10-17"
        Statement:
        -
          Effect: "Allow"
          Principal:
            Service:
              - "sagemaker.amazonaws.com"
          Action:
          - "sts:AssumeRole"
      Path: "/service-role/"
      ManagedPolicyArns:
        - arn:aws:iam::aws:policy/AmazonBraketFullAccess
      Policies:
        -
          PolicyName: "AmazonBraketNotebookPolicy"
          PolicyDocument:
            Version: "2012-10-17"
            Statement:
              - Effect: Allow
                Action:
                  - s3:GetObject
                  - s3:PutObject
                  - s3:ListBucket
                Resource:
                  - arn:aws:s3:::amazon-braket-*
                  - arn:aws:s3:::braketnotebookcdk-*
              - Effect: "Allow"
                Action:
                  - "logs:CreateLogStream"
                  - "logs:PutLogEvents"
                  - "logs:CreateLogGroup"
                  - "logs:DescribeLogStreams"
                Resource:
                  - !Sub "arn:aws:logs:*:${AWS::AccountId}:log-group:/aws/sagemaker/*"
```

```
              - Effect: "Allow"
                Action:
                  - braket:*
                Resource: "*"
```

Step 3: Create a SageMaker AI notebook instance with the prefix `amazon-braket-`

Use the SageMaker AI lifecycle script and the IAM role created in step 1 and step 2 to create a SageMaker AI notebook instance. The notebook instance is customized for Braket and can be accessed with the Amazon Braket console. For more information about configuration options for this CloudFormation resource, see [AWS::SageMaker::NotebookInstance]() in the *AWS CloudFormation user guide*.

```
BraketNotebook:
    Type: AWS::SageMaker::NotebookInstance
    Properties:
      InstanceType: ml.t3.medium
      NotebookInstanceName: !Sub amazon-braket-notebook-${AWS::StackName}
      RoleArn: !GetAtt ExecutionRole.Arn
      VolumeSizeInGB: 30
      LifecycleConfigName: !GetAtt
 BraketNotebookInstanceLifecycleConfig.NotebookInstanceLifecycleConfigName
```

Building your quantum tasks with Amazon Braket

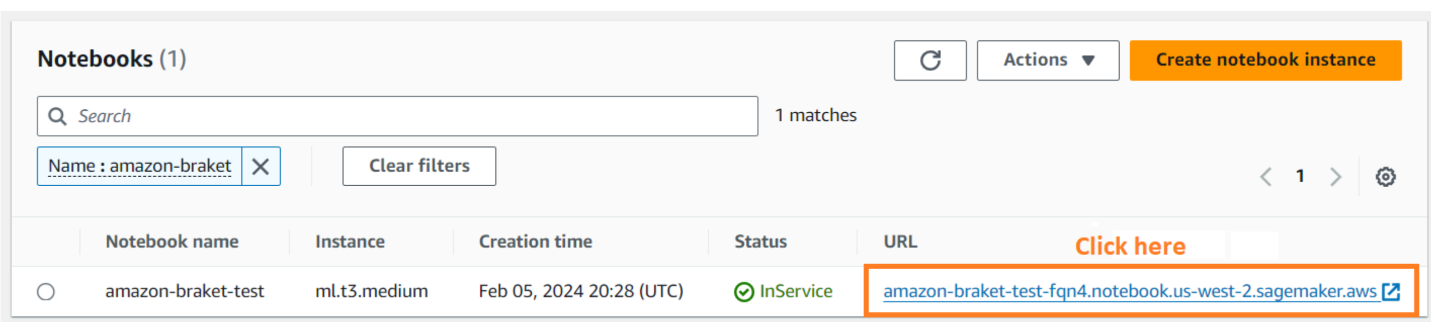
Braket provides fully managed Jupyter notebook environments that make it straightforward to get started. Braket notebooks are pre-installed with sample algorithms, resources, and developer tools, including the Amazon Braket SDK. With the Amazon Braket SDK, you can build quantum algorithms and then test and run them on different quantum computers and simulators by changing a single line of code.

In this section:

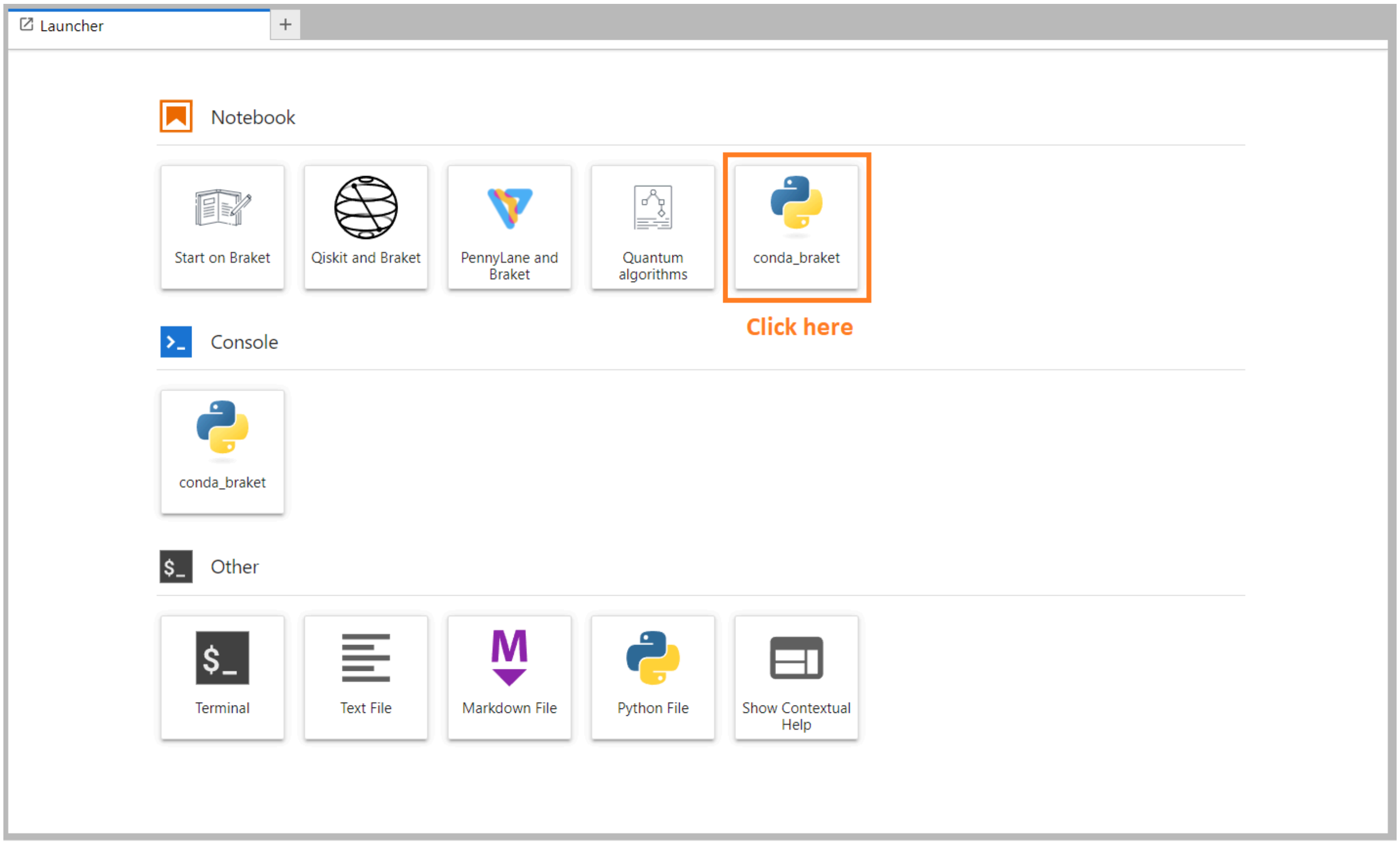
- Building your first circuit
- Getting Expert advice
- Getting started with Amazon Braket Hybrid Jobs
- Using PennyLane with Amazon Braket
- Using CUDA-Q with Amazon Braket
- Run your circuits with OpenQASM 3.0
- Explore Experimental Capabilities
- Pulse control on Amazon Braket
- Analog Hamiltonian Simulation
- Working with AWS Boto3

Building your first circuit

After your notebook instance has launched, open the instance with a standard Jupyter interface by choosing the notebook you just created.



Amazon Braket notebook instances are pre-installed with the Amazon Braket SDK and all its dependencies. Start by creating a new notebook with `conda_braket` kernel.



You can start with a simple “Hello, world!” example. First, construct a circuit that prepares a Bell state, and then run that circuit on different devices to obtain the results.

Begin by importing the Begin by importing the Amazon Braket SDK modules and defining a simpleBRAKETlong; SDK modules and defining a basic Bell State circuit.

```
import boto3
from braket.aws import AwsDevice
from braket.devices import LocalSimulator
from braket.circuits import Circuit

# Create the circuit
bell = Circuit().h(0).cnot(0, 1)
```

You can visualize the circuit with this command:

```
print(bell)
```

```
T  : #  0  #  1  #
```

```
      #####
q0 : ## H ########
      #####   #
            #####
q1 : ######## X ##
            #####
T  : #  0  #  1  #
```

Run your circuit on the local simulator

Next, choose the quantum device on which to run the circuit. The Amazon Braket SDK comes with a local simulator for rapid prototyping and testing. We recommend using the local simulator for smaller circuits, which can be up to 25 qubits (depending on your local hardware).

To instantiate the local simulator:

```
# Instantiate the local simulator
local_sim = LocalSimulator()
```

and run the circuit:

```
# Run the circuit
result = local_sim.run(bell, shots=1000).result()
counts = result.measurement_counts
print(counts)
```

You should see a result something like this:

```
Counter({'11': 503, '00': 497})
```

The specific Bell state you have prepared is an equal superposition of $|00\rangle$ and $|11\rangle$, and an about equal (up to shot noise) distribution of 00 and 11 as measurement outcomes, as expected.

Run your circuit on an on-demand simulator

Amazon Braket also provides access to an on-demand, high-performance simulator, SV1, for running larger circuits. SV1 is an on-demand state-vector simulator that allows for simulation of quantum circuits of up to 34 qubits. You can find more information on SV1 in the Supported Devices section and in the AWS console. When running quantum tasks on SV1 (and on TN1 or any QPU), the results of your quantum task are stored in an S3 bucket in your account. If you do not specify a bucket, the Braket SDK creates a default bucket `amazon-braket-{region}-{accountID}` for you. To learn more, see [Managing access to Amazon Braket](#).

> **Note**
>
> Fill in your actual, existing bucket name where the following example shows `amazon-braket-s3-demo-bucket` as your bucket name. Bucket names for Amazon Braket always begin with `amazon-braket-` followed by other identifying characters you add. If you need information on how to set up an S3 bucket, see [Getting started with Amazon S3](#).

```
# Get the account ID
aws_account_id = boto3.client("sts").get_caller_identity()["Account"]

# The name of the bucket
my_bucket = "amazon-braket-s3-demo-bucket"

# The name of the folder in the bucket
my_prefix = "simulation-output"
s3_folder = (my_bucket, my_prefix)
```

To run a circuit on SV1, you must provide the location of the S3 bucket you previously selected as a positional argument in the `.run()` call.

```
# Choose the cloud-based on-demand simulator to run your circuit
device = AwsDevice("arn:aws:braket:::device/quantum-simulator/amazon/sv1")

# Run the circuit
task = device.run(bell, s3_folder, shots=100)

# Display the results
print(task.result().measurement_counts)
```

The Amazon Braket console provides further information about your quantum task. Navigate to the **Quantum Tasks** tab in the console and your quantum task should be on the top of the list. Alternatively, you can search for your quantum task using the unique quantum task ID or other criteria.

Note

After 90 days, Amazon Braket automatically removes all quantum task IDs and other metadata associated with your quantum tasks. For more information, see Data retention.

Running on a QPU

With Amazon Braket, you can run the previous quantum circuit example on a physical quantum computer by just changing a single line of code. Amazon Braket provides access to QPU devices from IonQ, IQM, QuEra, and Rigetti. You can find information about the different devices and availability windows in the Supported Devices section, and in the AWS console under the **Devices** tab. The following example shows how to instantiate an IQM device.

```
# Choose the IQM hardware to run your circuit
device = AwsDevice("arn:aws:braket:eu-north-1::device/qpu/iqm/Garnet")
```

Or choose an IonQ device with this code:

```
# Choose the Ionq device to run your circuit
device = AwsDevice("arn:aws:braket:us-east-1::device/qpu/ionq/Aria-1")
```

After selecting a device and before running your workload, you can query device queue depth with the following code to determine the number of quantum tasks or hybrid jobs. Additionally, customers can view device specific queue depths on the Devices page of the Amazon Braket Management Console.

```
# Print your queue depth
print(device.queue_depth().quantum_tasks)
# Returns the number of quantum tasks queued on the device
# {<QueueType.NORMAL: 'Normal'>: '0', <QueueType.PRIORITY: 'Priority'>: '0'}

print(device.queue_depth().jobs)
# Returns the number of hybrid jobs queued on the device
# '2'
```

When you run your task, the Amazon Braket SDK polls for a result (with a default timeout of 5 days). You can change this default by modifying the `poll_timeout_seconds` parameter in the the `.run()` command as shown in the example that follows. Keep in mind that if your polling timeout is too short, results may not be returned within the polling time, such as when a QPU is unavailable and a local timeout error is returned. You can restart the polling by calling the `task.result()` function.

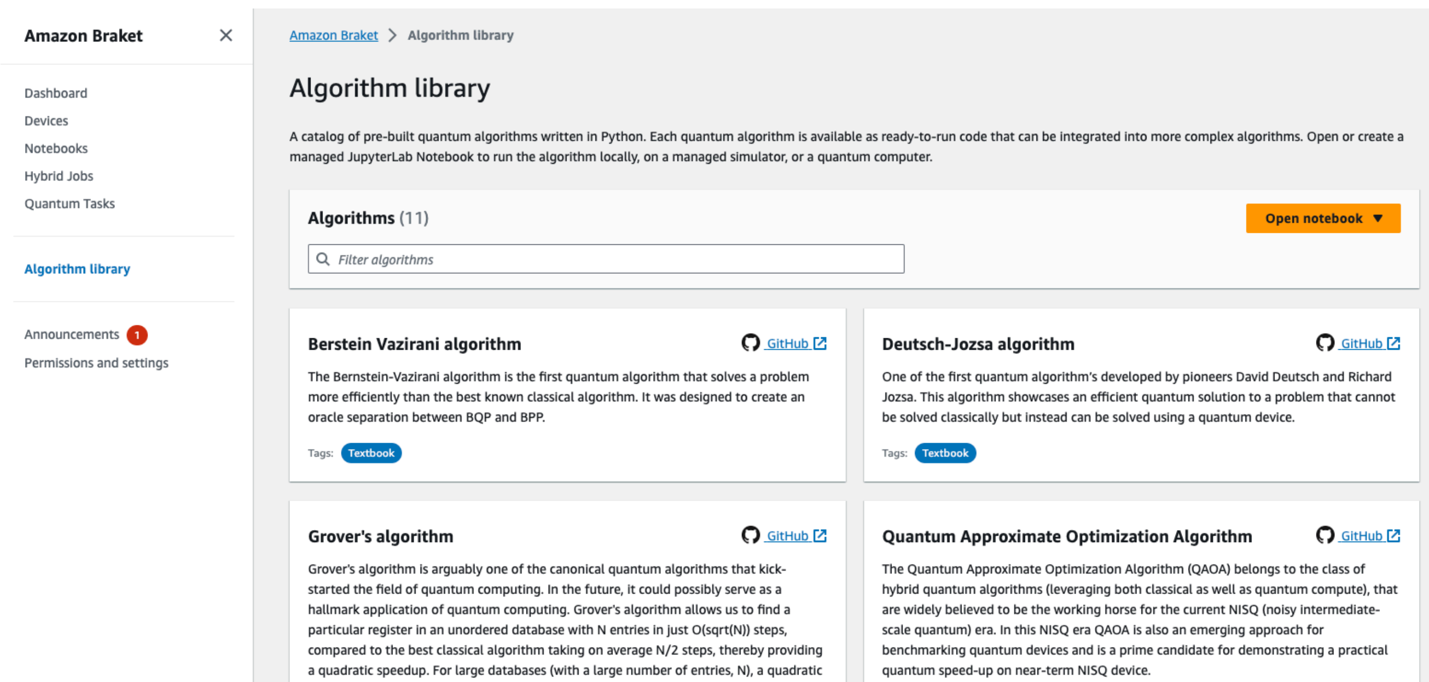
```
# Define quantum task with 1 day polling timeout
task = device.run(bell, s3_folder, poll_timeout_seconds=24*60*60)
print(task.result().measurement_counts)
```

Additionally, after submitting your quantum task or hybrid job, you can call the `queue_position()` function to check your queue position.

```
print(task.queue_position().queue_position)
# Return the number of quantum tasks queued ahead of you
# '2'
```

Building your first quantum algorithms

The Amazon Braket algorithm library is a catalog of pre-built quantum algorithms written in Python. Run these algorithms as they are, or use them as starting points for building more complex algorithms. You can access the algorithm library from the Braket console. For more information, see the [Braket Github algorithm library](https://github.com/aws-samples/amazon-braket-algorithm-library).



The Braket console provides a description of each available algorithm in the algorithm library. Choose a GitHub link to see the details of each algorithm, or choose **Open notebook** to open or create a notebook that contains all of the available algorithms. If you choose the notebook option, you can then find the Braket algorithm library in the root folder of your notebook.

Constructing circuits in the SDK

This section provides examples of defining a circuit, viewing available gates, extending a circuit, and viewing gates that each device supports. It also contains instructions on how to manually allocate qubits, instruct the compiler to run your circuits exactly as defined, and build noisy circuits with a noise simulator.

You can also work at the pulse level in Braket for various gates with certain QPUs. For more information, see Pulse Control on Amazon Braket.

In this section:

- Gates and circuits
- Program sets
- Partial measurement
- Manual qubit allocation
- Verbatim compilation
- Noise simulation

Gates and circuits

Quantum gates and circuits are defined in the `braket.circuits` class of the Amazon Braket Python SDK. From the SDK, you can instantiate a new circuit object by calling `Circuit()`.

Example: Define a circuit

The example starts by defining a sample circuit of four qubits (labelled `q0`, `q1`, `q2`, and `q3`) consisting of standard, single-qubit Hadamard gates and two-qubit CNOT gates. You can visualize this circuit by calling the `print` function as the following example shows.

```
# Import the circuit module
from braket.circuits import Circuit

# Define circuit with 4 qubits
my_circuit = Circuit().h(range(4)).cnot(control=0, target=2).cnot(control=1, target=3)
print(my_circuit)
```

```
T  : #  0  #     1     #
      #####
q0 : ## H ##############
      #####   #
      #####   #
q1 : ## H ##############
      #####   #     #
      ##### #####   #
q2 : ## H ### X ########
      ##### #####   #
      #####       #####
q3 : ## H ######### X ##
      #####       #####
T  : #  0  #     1     #
```

Example: Define a parameterized circuit

In this example, we define a circuit with gates that depend on free parameters. We can specify the values of these parameters to create a new circuit, or, when submitting the circuit, to run as a quantum task on certain devices.

```
from braket.circuits import Circuit, FreeParameter

# Define a FreeParameter to represent the angle of a gate
alpha = FreeParameter("alpha")

# Define a circuit with three qubits
my_circuit = Circuit().h(range(3)).cnot(control=0, target=2).rx(0, alpha).rx(1, alpha)
print(my_circuit)
```

You can create a new, non-parameterized circuit from a parametrized one by supplying either a single `float` (which is the value all free parameters will take) or keyword arguments specifying each parameter's value to the circuit as follows.

```
my_fixed_circuit = my_circuit(1.2)
my_fixed_circuit = my_circuit(alpha=1.2)
print(my_fixed_circuit)
```

Note that `my_circuit` is unmodified, so you can use it to instantiate many new circuits with fixed parameter values.

Example: Modify gates in a circuit

The following example defines a circuit with gates that use control and power modifiers. You can use these modifications to create new gates, such as the controlled Ry gate.

```
from braket.circuits import Circuit

# Create a bell circuit with a controlled x gate
my_circuit = Circuit().h(0).x(control=0, target=1)

# Add a multi-controlled Ry gate of angle .13
my_circuit.ry(angle=.13, target=2, control=(0, 1))

# Add a 1/5 root of X gate
my_circuit.x(0, power=1/5)

print(my_circuit)
```

Gate modifiers are supported only on the local simulator.

Example: See all available gates

The following example shows how to look at all the available gates in Amazon Braket.

```
from braket.circuits import Gate
# Print all available gates in Amazon Braket
gate_set = [attr for attr in dir(Gate) if attr[0].isupper()]
print(gate_set)
```

The output from this code lists all of the gates.

```
['CCNot', 'CNot', 'CPhaseShift', 'CPhaseShift00', 'CPhaseShift01', 'CPhaseShift10',
 'CSwap', 'CV', 'CY', 'CZ', 'ECR', 'GPhase', 'GPi', 'GPi2', 'H', 'I', 'ISwap', 'MS',
 'PRx', 'PSwap', 'PhaseShift', 'PulseGate', 'Rx', 'Ry', 'Rz', 'S', 'Si', 'Swap', 'T',
 'Ti', 'U', 'Unitary', 'V', 'Vi', 'X', 'XX', 'XY', 'Y', 'YY', 'Z', 'ZZ']
```

Any of these gates can be appended to a circuit by calling the method for that type of circuit. For example, call `circ.h(0)`, to add a Hadamard gate to the first qubit.

> **Note**
>
> Gates are appended in place, and the example that follows adds all of the gates listed in the previous example to the same circuit.

```
circ = Circuit()
# toffoli gate with q0, q1 the control qubits and q2 the target.
circ.ccnot(0, 1, 2)
# cnot gate
circ.cnot(0, 1)
# controlled-phase gate that phases the |11> state, cphaseshift(phi) =
 diag((1,1,1,exp(1j*phi))), where phi=0.15 in the examples below
circ.cphaseshift(0, 1, 0.15)
# controlled-phase gate that phases the |00> state, cphaseshift00(phi) =
 diag([exp(1j*phi),1,1,1])
circ.cphaseshift00(0, 1, 0.15)
# controlled-phase gate that phases the |01> state, cphaseshift01(phi) =
 diag([1,exp(1j*phi),1,1])
circ.cphaseshift01(0, 1, 0.15)
# controlled-phase gate that phases the |10> state, cphaseshift10(phi) =
 diag([1,1,exp(1j*phi),1])
circ.cphaseshift10(0, 1, 0.15)
# controlled swap gate
circ.cswap(0, 1, 2)
# swap gate
circ.swap(0,1)
# phaseshift(phi)= diag([1,exp(1j*phi)])
circ.phaseshift(0,0.15)
# controlled Y gate
circ.cy(0, 1)
# controlled phase gate
circ.cz(0, 1)
# Echoed cross-resonance gate applied to q0, q1
circ = Circuit().ecr(0,1)
# X rotation with angle 0.15
circ.rx(0, 0.15)
# Y rotation with angle 0.15
circ.ry(0, 0.15)
# Z rotation with angle 0.15
circ.rz(0, 0.15)
# Hadamard gates applied to q0, q1, q2
```

```
circ.h(range(3))
# identity gates applied to q0, q1, q2
circ.i([0, 1, 2])
# iswap gate, iswap = [[1,0,0,0],[0,0,1j,0],[0,1j,0,0],[0,0,0,1]]
circ.iswap(0, 1)
# pswap gate, PSWAP(phi) = [[1,0,0,0],[0,0,exp(1j*phi),0],[0,exp(1j*phi),0,0],
[0,0,0,1]]
circ.pswap(0, 1, 0.15)
# X gate applied to q1, q2
circ.x([1, 2])
# Y gate applied to q1, q2
circ.y([1, 2])
# Z gate applied to q1, q2
circ.z([1, 2])
# S gate applied to q0, q1, q2
circ.s([0, 1, 2])
# conjugate transpose of S gate applied to q0, q1
circ.si([0, 1])
# T gate applied to q0, q1
circ.t([0, 1])
# conjugate transpose of T gate applied to q0, q1
circ.ti([0, 1])
# square root of not gate applied to q0, q1, q2
circ.v([0, 1, 2])
# conjugate transpose of square root of not gate applied to q0, q1, q2
circ.vi([0, 1, 2])
# exp(-iXX theta/2)
circ.xx(0, 1, 0.15)
# exp(i(XX+YY) theta/4), where theta=0.15 in the examples below
circ.xy(0, 1, 0.15)
# exp(-iYY theta/2)
circ.yy(0, 1, 0.15)
# exp(-iZZ theta/2)
circ.zz(0, 1, 0.15)
# IonQ native gate GPi with angle 0.15 applied to q0
circ.gpi(0, 0.15)
# IonQ native gate GPi2 with angle 0.15 applied to q0
circ.gpi2(0, 0.15)
# IonQ native gate MS with angles 0.15, 0.15, 0.15 applied to q0, q1
circ.ms(0, 1, 0.15, 0.15, 0.15)
```

Apart from the pre-defined gate set, you also can apply self-defined unitary gates to the circuit. These can be single-qubit gates (as shown in the following source code) or multi-qubit gates applied to the qubits defined by the `targets` parameter.

```
import numpy as np

# Apply a general unitary
my_unitary = np.array([[0, 1],[1, 0]])
circ.unitary(matrix=my_unitary, targets=[0])
```

Example: Extend existing circuits

You can extend existing circuits by adding instructions. An `Instruction` is a quantum directive that describes the quantum task to perform on a quantum device. `Instruction` operators include objects of type `Gate` only.

```
# Import the Gate and Instruction modules
from braket.circuits import Gate, Instruction

# Add instructions directly.
circ = Circuit([Instruction(Gate.H(), 4), Instruction(Gate.CNot(), [4, 5])])

# Or with add_instruction/add functions
instr = Instruction(Gate.CNot(), [0, 1])
circ.add_instruction(instr)
circ.add(instr)

# Specify where the circuit is appended
circ.add_instruction(instr, target=[3, 4])
circ.add_instruction(instr, target_mapping={0: 3, 1: 4})

# Print the instructions
print(circ.instructions)
# If there are multiple instructions, you can print them in a for loop
for instr in circ.instructions:
     print(instr)

# Instructions can be copied
new_instr = instr.copy()
# Appoint the instruction to target
new_instr = instr.copy(target=[5, 6])
new_instr = instr.copy(target_mapping={0: 5, 1: 6})
```

Example: View the gates that each device supports

Simulators support all gates in the Braket SDK, but QPU devices support a smaller subset. You can find the supported gates of a device in the device properties. The following shows an example with an IonQ device:

```
# Import the device module
from braket.aws import AwsDevice

device = AwsDevice("arn:aws:braket:us-east-1::device/qpu/ionq/Aria-1")

# Get device name
device_name = device.name
# Show supportedQuantumOperations (supported gates for a device)
device_operations = device.properties.dict()['action']['braket.ir.openqasm.program']
['supportedOperations']
print('Quantum Gates supported by {}:\n {}'.format(device_name, device_operations))
```

```
Quantum Gates supported by Aria 1:
 ['x', 'y', 'z', 'h', 's', 'si', 't', 'ti', 'v', 'vi', 'rx', 'ry', 'rz', 'cnot',
 'swap', 'xx', 'yy', 'zz']
```

Supported gates may need to be compiled into native gates before they can run on quantum hardware. When you submit a circuit, Amazon Braket performs this compilation automatically.

Example: Programmatically retrieve the fidelity of native gates supported by a device

You can view the fidelity information on the **Devices** page of the Braket console. Sometimes it is helpful to access the same information programmatically. The following code shows how to extract the two qubit gate fidelity between two gates of a QPU.

```
# Import the device module
from braket.aws import AwsDevice

device = AwsDevice("arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3")

# Specify the qubits
a=10
b=11
edge_properties_entry =
 device.properties.standardized.twoQubitProperties['10-11'].twoQubitGateFidelity
```

```
gate_name = edge_properties_entry[0].gateName
fidelity = edge_properties_entry[0].fidelity
print(f"Fidelity of the {gate_name} gate between qubits {a} and {b}: {fidelity}")
```

Program sets

Program sets efficiently run multiple quantum circuits in a single quantum task. In that one task, you can submit up to 100 quantum circuits or a single parametric circuit with up to 100 different parameter sets. This operation minimizes the time between subsequent circuit executions and reduces quantum task processing overhead. Currently, program sets are supported on the Amazon Braket Local Simulator and on IQM and Rigetti devices.

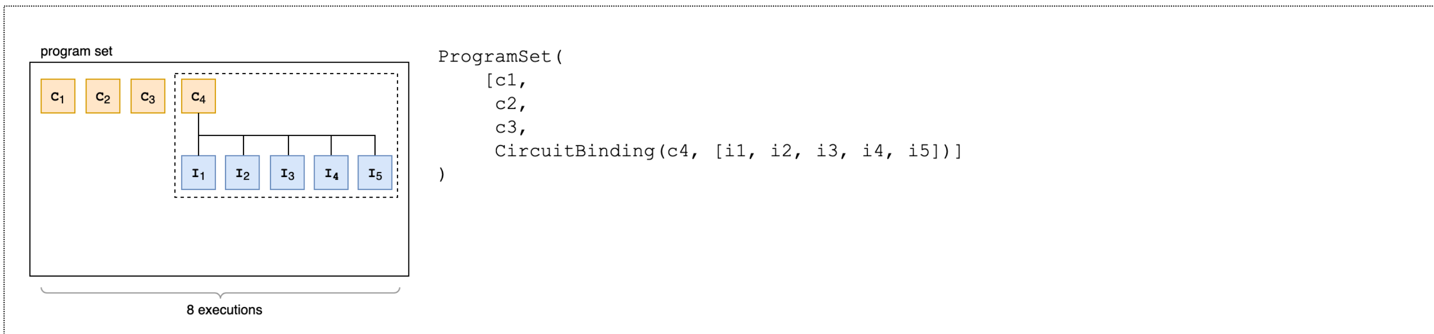
Defining a ProgramSet

The following first code example demonstrates how to build a `ProgramSet` using both parameterized circuits and circuits without parameters.

```
from braket.aws import AwsDevice
from braket.circuits import Circuit, FreeParameter
from braket.program_sets.circuit_binding import CircuitBinding
from braket.program_sets import ProgramSet

# Initialize the quantum device
device = AwsDevice("arn:aws:braket:eu-north-1::device/qpu/iqm/Garnet")

# Define circuits
circ1 = Circuit().h(0).cnot(0, 1)
circ2 = Circuit().rx(0, 0.785).ry(1, 0.393).cnot(1, 0)
circ3 = Circuit().t(0).t(1).cz(0, 1).s(0).cz(1, 2).s(1).s(2)
parameterize_circuit = Circuit().rx(0, FreeParameter("alpha")).cnot(0, 1).ry(1,
 FreeParameter("beta"))

# Create circuit bindings with different parameters
circuit_binding = CircuitBinding(
    circuit=parameterize_circuit,
    input_sets={
            'alpha': (0.10, 0.11, 0.22, 0.34, 0.45),
            'beta': (1.01, 1.01, 1.03, 1.04, 1.04),
    })

# Creating the program set
program_set_1 = ProgramSet([
```

```
    circ1,
    circ2,
    circ3,
    circuit_binding,
])
```

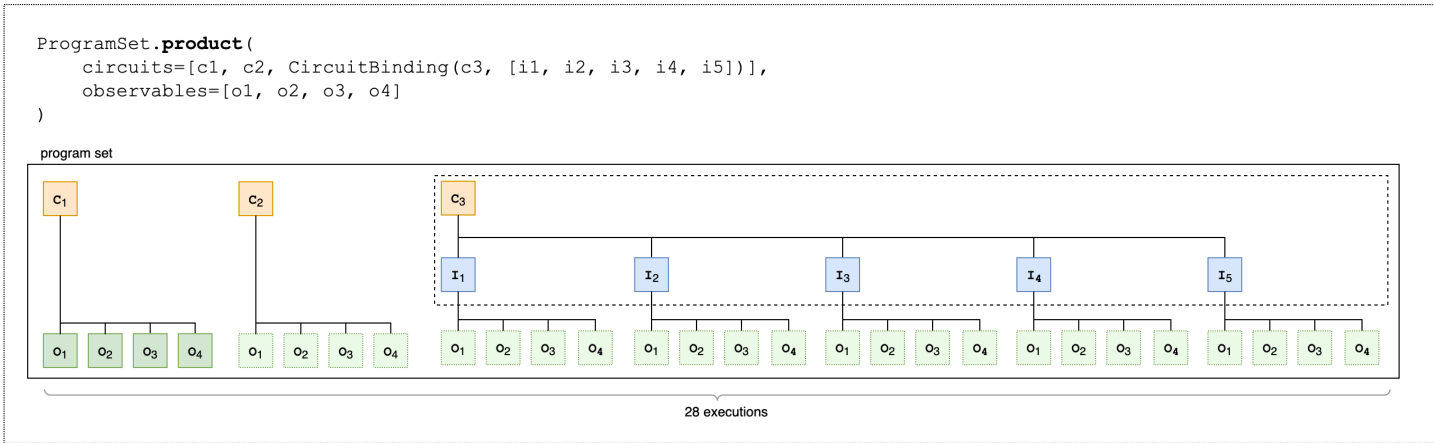
This program set contains four unique programs: `circ1`, `circ2`, `circ3`, and `circuit_binding`. The `circuit_binding` program runs with five different parameter bindings, creating five executables. The other three parameter-free programs create one executable each. This results in eight total executables, as shown in the following image.



The following second code example demonstrates how to use the `product()` method to attach the same set of observables to each executable of the program set.

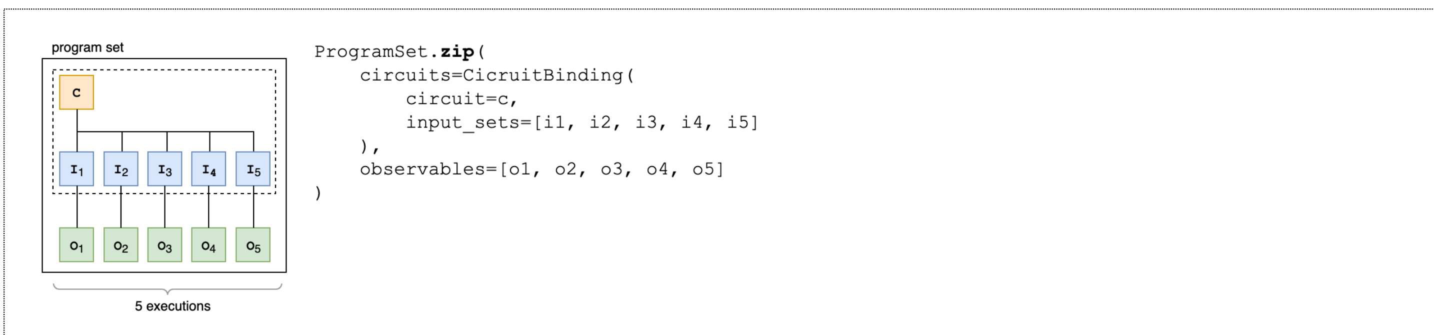
```
from braket.circuits.observables import I, X, Y, Z

observables = [Z(0) @ Z(1), X(0) @ X(1), Z(0) @ X(1), X(0) @ Z(1)]

program_set_2 = ProgramSet.product(
    circuits=[circ1, circ2, circuit_binding],
    observables=observables
)
```

For parameter-free programs, each observable is measured for each circuit. For parametric programs, each observable is measured for each input set, as shown in the following image.



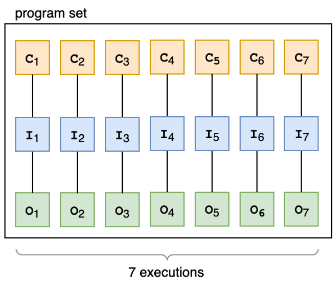
The following third code example demonstrates how to use the `zip()` method to pair individual observables with specific parameter sets in the `ProgramSet`.

```
program_set_3 = ProgramSet.zip(
    circuits=circuit_binding,
    observables=observables + [Y(0) @ Y(1)]
)
```



Instead of `CircuitBinding()`, you can directly zip a list of observables with a list of circuits and input sets.

```
program_set_4 = ProgramSet.zip(
    circuits=[circ1, circ2, circ3],
    input_sets=[{}, {}, {}],
    observables=observables[:3]
)
```




```
ProgramSet.zip(
    circuits=[c1, c2, c3, c4, c5, c6, c7],
    input_sets=[i1, i2, i3, i4, i5, i6, i7],
    observables=[o1, o2, o3, o4, o5, o6, o7]
)
```

For more information and examples on program sets, see the Program set folder in the amazon-braket-examples Github.

Inspect and run a program set on a device

A program set's executable count equals its number of unique parameter-bound circuits. Calculate the total number of circuit executables and shots using the following code example.

```
# Number of shots per executable
shots = 10
num_executables = program_set_1.total_executables

# Calculate total number of shots across all executables
total_num_shots = shots*num_executables
```

> **Note**
>
> With program sets, you pay a single per-task fee and a per-shot fee based on the total number of shots across all circuits in a program set.

To run the program set, use the following code example.

```
# Run the program set
task = device.run(
   program_set_1, shots=total_num_shots,
)
```

When using Rigetti devices, your program set may remain in the RUNNING state while tasks are partially finished and partially queued. For faster results, consider submitting your program set as a Hybrid Job.

Analyzing results

Run the following code to analyze and measure the results of the executables in a `ProgramSet`.

```
# Get the results from a program set
result = task.result()

# Get the first executbable
first_program = result[0]
first_executable = first_program[0]

# Inspect the results of the first executable
measurements_from_first_executable = first_executable.measurements
print(measurements_from_first_executable)
```

Partial measurement

Instead of measuring all qubits in a quantum circuit, use partial measurement to measure individual qubits or a subset of qubits.

> **Note**
>
> Additional features such as mid-circuit measurement and feed-forward operations are available as Experimental Capabilities, see Access to dynamic circuits on IQM devices.

Example: Measure a subset of qubits

The following code example demonstrates partial measurement by measuring only qubit 0 in a Bell state circuit.

```
from braket.devices import LocalSimulator
from braket.circuits import Circuit

# Use the local state vector simulator
device = LocalSimulator()

# Define an example bell circuit and measure qubit 0
circuit = Circuit().h(0).cnot(0, 1).measure(0)

# Run the circuit
```

```
task = device.run(circuit, shots=10)

# Get the results
result = task.result()

# Print the circuit and measured qubits
print(circuit)
print()
print("Measured qubits: ", result.measured_qubits)
```

Manual qubit allocation

When you run a quantum circuit on quantum computers from Rigetti, you can optionally use manual qubit allocation to control which qubits are used for your algorithm. The Amazon Braket Console and the Amazon Braket SDK help you to inspect the most recent calibration data of your selected quantum processing unit (QPU) device, so you can select the best qubits for your experiment.

Manual qubit allocation enables you to run circuits with greater accuracy and to investigate individual qubit properties. Researchers and advanced users optimize their circuit design based on the latest device calibration data and can obtain more accurate results.

The following example demonstrates how to allocate qubits explicitly.

```
# Import the device module
from braket.aws import AwsDevice

device = AwsDevice("arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3")
circ = Circuit().h(0).cnot(0, 7)  # Indices of actual qubits in the QPU

# Set up S3 bucket (where results are stored)
my_bucket = "amazon-braket-s3-demo-bucket" # The name of the bucket
my_prefix = "your-folder-name" # The name of the folder in the bucket
s3_location = (my_bucket, my_prefix)

my_task = device.run(circ, s3_location, shots=100, disable_qubit_rewiring=True)
```

For more information, see the Amazon Braket examples on GitHub, or more specifically, this notebook: Allocating Qubits on QPU Devices.

Verbatim compilation

When you run a quantum circuit on gate-based quantum computers you can direct the compiler to run your circuits exactly as defined without any modifications. Using verbatim compilation, you can specify either that an entire circuit be preserved precisely as specified or that only specific parts of it be preserved (supported by Rigetti only). When developing algorithms for hardware benchmarking or error mitigation protocols, you need have the option to exactly specify the gates and circuit layouts that are running on the hardware. Verbatim compilation gives you direct control over the compilation process by turning off certain optimization steps, thereby ensuring that your circuits run exactly as designed.

Verbatim compilation is supported on the Rigetti, IonQ, and IQM devices and requires the use of native gates. When using verbatim compilation, it is advisable to check the topology of the device to ensure that gates are called on connected qubits and that the circuit uses the native gates supported on the hardware. The following example shows how to programmatically access the list of native gates supported by a device.

```
device.properties.paradigm.nativeGateSet
```

For Rigetti, qubit rewiring must be turned off by setting `disableQubitRewiring=True` for use with verbatim compilation. If `disableQubitRewiring=False` is set when using verbatim boxes in a compilation, the quantum circuit fails validation and does not run.

If verbatim compilation is enabled for a circuit and run on a QPU that does not support it, an error is generated indicating that an unsupported operation has caused the task to fail. As more quantum hardware natively support compiler functions, this feature will be expanded to include these devices. Devices that support verbatim compilation include it as a supported operation when queried with the following code.

```
from braket.aws import AwsDevice
from braket.device_schema.device_action_properties import DeviceActionType
device = AwsDevice("arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3")
device.properties.action[DeviceActionType.OPENQASM].supportedPragmas
```

There is no additional cost associated with using verbatim compilation. You continue to be charged for quantum tasks executed on Braket QPU devices, notebook instances, and on-demand simulators based on current rates as specified on the Amazon Braket Pricing page. For more information, see the Verbatim compilation example notebook.

> **Note**
>
> If you are using OpenQASM to write your circuits for the IonQ device, and you wish to map your circuit directly to the physical qubits, you need to use the `#pragma braket verbatim` as the `disableQubitRewiring` flag is completely ignored by OpenQASM.

Noise simulation

To instantiate the local noise simulator you can change the backend as follows.

```
# Import the device module
from braket.aws import AwsDevice

device = LocalSimulator(backend="braket_dm")
```

You can build noisy circuits in two ways:

1. Build the noisy circuit from the bottom up.
2. Take an existing, noise-free circuit and inject noise throughout.

The following example shows the approaches using a basic circuit with depolarizing noise and a custom Kraus channel.

```
import scipy.stats
import numpy as np

# Bottom up approach
# Apply depolarizing noise to qubit 0 with probability of 0.1
circ = Circuit().x(0).x(1).depolarizing(0, probability=0.1)

# Create an arbitrary 2-qubit Kraus channel
E0 = scipy.stats.unitary_group.rvs(4) * np.sqrt(0.8)
E1 = scipy.stats.unitary_group.rvs(4) * np.sqrt(0.2)
K = [E0, E1]

# Apply a two-qubit Kraus channel to qubits 0 and 2
circ = circ.kraus([0, 2], K)
```

```
from braket.circuits import Noise
```

```
# Inject noise approach
# Define phase damping noise
noise = Noise.PhaseDamping(gamma=0.1)
# The noise channel is applied to all the X gates in the circuit
circ = Circuit().x(0).y(1).cnot(0, 2).x(1).z(2)
circ_noise = circ.copy()
circ_noise.apply_gate_noise(noise, target_gates=Gate.X)
```

Running a circuit is the same user experience as before, as shown in the following two examples.

Example 1

```
task = device.run(circ, shots=100)
```

Or

Example 2

```
task = device.run(circ_noise, shots=100)
```

For more examples, see the Braket introductory noise simulator example

Inspecting the circuit

Quantum circuits in Amazon Braket have a pseudo-time concept called `Moments`. Each qubit can experience a single gate per `Moment`. The purpose of `Moments` is to make circuits and their gates easier to address and to provide a temporal structure.

> **Note**
>
> Moments generally do not correspond to the real time at which gates are executed on a QPU.

The depth of a circuit is given by the total number of Moments in that circuit. You can view the circuit depth calling the method `circuit.depth` as shown in the following example.

```
from braket.circuits import Circuit
```

```
# Define a circuit with parametrized gates
circ = Circuit().rx(0, 0.15).ry(1, 0.2).cnot(0, 2).zz(1, 3, 0.15).x(0)
print(circ)
print('Total circuit depth:', circ.depth)
```

```
T  : #     0      #        1         #  2  #
      ############                    #####
q0 : ## Rx(0.15) ###################### X ##
      ############   #                #####
      ############   #   ############
q1 : ## Ry(0.20) ######### ZZ(0.15) ########
      ############   #   ############
                   #####      #
q2 : ############### X #####################
                   #####      #
                         ############
q3 : ##################### ZZ(0.15) ########
                         ############
T  : #     0      #        1         #  2  #
Total circuit depth: 3
```

The total circuit depth of the circuit above is 3 (shown as moments 0, 1, and 2). You can check the gate operation for each moment.

`Moments` functions as a dictionary of *key-value* pairs.

- The key is `MomentsKey()`, which contains pseudo-time and qubit information.
- The value is assigned in the type of `Instructions()`.

```
moments = circ.moments
for key, value in moments.items():
    print(key)
    print(value, "\n")
```

```
MomentsKey(time=0, qubits=QubitSet([Qubit(0)]), moment_type=<MomentType.GATE: 'gate'>,
 noise_index=0, subindex=0)
Instruction('operator': Rx('angle': 0.15, 'qubit_count': 1), 'target':
 QubitSet([Qubit(0)]), 'control': QubitSet([]), 'control_state': (), 'power': 1)
```

```
MomentsKey(time=0, qubits=QubitSet([Qubit(1)]), moment_type=<MomentType.GATE: 'gate'>,
 noise_index=0, subindex=0)
Instruction('operator': Ry('angle': 0.2, 'qubit_count': 1), 'target':
 QubitSet([Qubit(1)]), 'control': QubitSet([]), 'control_state': (), 'power': 1)

MomentsKey(time=1, qubits=QubitSet([Qubit(0), Qubit(2)]), moment_type=<MomentType.GATE:
 'gate'>, noise_index=0, subindex=0)
Instruction('operator': CNot('qubit_count': 2), 'target': QubitSet([Qubit(0),
 Qubit(2)]), 'control': QubitSet([]), 'control_state': (), 'power': 1)

MomentsKey(time=1, qubits=QubitSet([Qubit(1), Qubit(3)]), moment_type=<MomentType.GATE:
 'gate'>, noise_index=0, subindex=0)
Instruction('operator': ZZ('angle': 0.15, 'qubit_count': 2), 'target':
 QubitSet([Qubit(1), Qubit(3)]), 'control': QubitSet([]), 'control_state': (), 'power':
 1)

MomentsKey(time=2, qubits=QubitSet([Qubit(0)]), moment_type=<MomentType.GATE: 'gate'>,
 noise_index=0, subindex=0)
Instruction('operator': X('qubit_count': 1), 'target': QubitSet([Qubit(0)]), 'control':
 QubitSet([]), 'control_state': (), 'power': 1)
```

You can also add gates to a circuit through `Moments`.

```
from braket.circuits import Instruction, Gate

new_circ = Circuit()
instructions = [Instruction(Gate.S(), 0),
                Instruction(Gate.CZ(), [1, 0]),
                Instruction(Gate.H(), 1)
                ]

new_circ.moments.add(instructions)
print(new_circ)
```

```
T  : #  0  #  1  #  2  #
      ##### #####
q0 : ## S ### Z ########
      ##### #####
              #   #####
q1 : ############# H ##
                  #####
T  : #  0  #  1  #  2  #
```

List of result types

Amazon Braket can return different types of results when a circuit is measured using `ResultType`. A circuit can return the following types of results.

- `AdjointGradient` returns the gradient (vector derivative) of the expectation value of a provided observable. This observable is acting on a provided target with respect to specified parameters using the adjoint differentiation method. You can only use this method when shots=0.
- `Amplitude` returns the amplitude of specified quantum states in the output wave function. It is available on the SV1 and local simulators only.
- `Expectation` returns the expectation value of a given observable, which can be specified with the `Observable` class introduced later in this chapter. The target qubits used to measure the observable must be specified, and the number of specified targets must equal the number of qubits on which the observable acts. If no targets are specified, the observable must operate only on 1 qubit and it is applied to all qubits in parallel.
- `Probability` returns the probabilities of measuring computational basis states. If no targets are specified, `Probability` returns the probability of measuring all basis states. If targets are specified, only the marginal probabilities of the basis vectors on the specified qubits are returned. Managed simulators and QPUs are limited to 15 qubits maximum, and local simulators are limited to the system's memory size.
- `Reduced density matrix` returns a density matrix for a subsystem of specified target qubits from a system of qubits. To limit the size of this result type, Braket limits the number of target qubits to a maximum of 8.
- `StateVector` returns the full state vector. It is available on the local simulator.
- `Sample` returns the measurement counts of a specified target qubit set and observable. If no targets are specified, the observable must operate only on 1 qubit and it is applied to all qubits in parallel. If targets are specified, the number of specified targets must equal the number of qubits on which the observable acts.
- `Variance` returns the variance ($\mathrm{mean}([x-\mathrm{mean}(x)]^2)$) of the specified target qubit set and observable as the requested result type. If no targets are specified, the observable must operate only on 1 qubit and it is applied to all qubits in parallel. Otherwise, the number of targets specified must equal the number of qubits to which the observable can be applied.

The supported result types for different devices:

	Local sim	SV1	DM1	TN1	Rigetti	IonQ	IQM
Adjoint Gradient	N	Y	N	N	N	N	N
Amplitude	Y	Y	N	N	N	N	N
Expectation	Y	Y	Y	Y	Y	Y	Y
Probability	Y	Y	Y	N	Y	Y	Y
Reduced density matrix	Y	N	Y	N	N	N	N
State vector	Y	N	N	N	N	N	N
Sample	Y	Y	Y	Y	Y	Y	Y
Variance	Y	Y	Y	Y	Y	Y	Y

You can check the supported result types by examining the device properties, as shown in the following example.

```
from braket.aws import AwsDevice

device = AwsDevice("arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3")

# Print the result types supported by this device
for iter in
 device.properties.action['braket.ir.openqasm.program'].supportedResultTypes:
    print(iter)
```

```
name='Sample' observables=['x', 'y', 'z', 'h', 'i'] minShots=10 maxShots=50000
name='Expectation' observables=['x', 'y', 'z', 'h', 'i'] minShots=10 maxShots=50000
```

```
name='Variance' observables=['x', 'y', 'z', 'h', 'i'] minShots=10 maxShots=50000
name='Probability' observables=None minShots=10 maxShots=50000
```

To call a `ResultType`, append it to a circuit, as shown in the following example.

```
from braket.circuits import Circuit, Observable

circ = Circuit().h(0).cnot(0, 1).amplitude(state=["01", "10"])
circ.probability(target=[0, 1])
circ.probability(target=0)
circ.expectation(observable=Observable.Z(), target=0)
circ.sample(observable=Observable.X(), target=0)
circ.state_vector()
circ.variance(observable=Observable.Z(), target=0)

# Print one of the result types assigned to the circuit
print(circ.result_types[0])
```

Note

Different quantum devices provide results in various formats. For example, Rigetti devices return measurements, while IonQ devices provide probabilities. The Amazon Braket SDK offers a measurements property for all results. However, for devices that return probabilities, these measurements are post-computed and based on the probabilities, as per-shot measurements are not available. To determine if a result has been post-computed, check the `measurements_copied_from_device` on the result object. This operation is detailed in the gate_model_quantum_task_result.py file in the Amazon Braket SDK GitHub repository.

Observables

Amazon Braket's `Observable` class allows you to measure a specific observable.

You can apply only one unique non-identity observable to each qubit. An error occurs If you specify two or more different non-identity observables to the same qubit. For this purpose, each factor of a tensor product counts as an individual observable. This means you can have multiple tensor products on the same qubit, as long as the factors acting on that qubit remain the same.

An observable can be scaled and add other observables (scaled or not). This creates a `Sum` which can be used in the `AdjointGradient` result type.

The `Observable` class includes the following observables.

```
import numpy as np

Observable.I()
Observable.H()
Observable.X()
Observable.Y()
Observable.Z()

# Get the eigenvalues of the observable
print("Eigenvalue:", Observable.H().eigenvalues)
# Or rotate the basis to be computational basis
print("Basis rotation gates:", Observable.H().basis_rotation_gates)

# Get the tensor product of the observable for the multi-qubit case
tensor_product = Observable.Y() @ Observable.Z()
# View the matrix form of an observable by using
print("The matrix form of the observable:\n", Observable.Z().to_matrix())
print("The matrix form of the tensor product:\n", tensor_product.to_matrix())

# Factorize an observable in the tensor form
print("Factorize an observable:", tensor_product.factors)

# Self-define observables, given it is a Hermitian
print("Self-defined Hermitian:", Observable.Hermitian(matrix=np.array([[0, 1], [1,
 0]])))

print("Sum of other (scaled) observables:", 2.0 * Observable.X() @ Observable.X() + 4.0
 * Observable.Z() @ Observable.Z())
```

```
Eigenvalue: [ 1. -1.]
Basis rotation gates: (Ry('angle': -0.7853981633974483, 'qubit_count': 1),)
The matrix form of the observable:
 [[ 1.+0.j  0.+0.j]
 [ 0.+0.j -1.+0.j]]
The matrix form of the tensor product:
 [[ 0.+0.j  0.+0.j  0.-1.j  0.+0.j]
 [ 0.+0.j -0.+0.j  0.+0.j  0.+1.j]
 [ 0.+1.j  0.+0.j  0.+0.j  0.+0.j]
```

```
 [ 0.+0.j  0.-1.j  0.+0.j -0.+0.j]]
Factorize an observable: (Y('qubit_count': 1), Z('qubit_count': 1))
Self-defined Hermitian: Hermitian('qubit_count': 1, 'matrix': [[0.+0.j 1.+0.j], [1.+0.j
 0.+0.j]])
Sum of other (scaled) observables: Sum(TensorProduct(X('qubit_count': 1),
 X('qubit_count': 1)), TensorProduct(Z('qubit_count': 1), Z('qubit_count': 1)))
```

Parameters

Circuits can incorporate free parameters. These free parameters only need to be constructed once to run multiple times, and can be used to compute gradients.

Each free parameter uses a string-encoded name that is used to:

- Set parameter values
- Identify which parameters to use

```
from braket.circuits import Circuit, FreeParameter, observables
from braket.parametric import FreeParameter

theta = FreeParameter("theta")
phi = FreeParameter("phi")
circ = Circuit().h(0).rx(0, phi).ry(0, phi).cnot(0, 1).xx(0, 1, theta)
```

Adjoint gradient

The SV1 device calculates the adjoint gradient of an observable expectation value, including multi-term Hamiltonian. To differentiate parameters, specify their name (in string format) or by direct reference.

```
from braket.aws import AwsDevice
from braket.devices import Devices

device = AwsDevice(Devices.Amazon.SV1)

circ.adjoint_gradient(observable=3 * Observable.Z(0) @ Observable.Z(1) - 0.5 *
 observables.X(0), parameters = ["phi", theta])
```

Passing fixed parameter values as arguments to a parameterized circuit will remove the free parameters. Running this circuit with `AdjointGradient` produces an error, because the free parameters no longer exist. The follow code example demonstrates the correct and incorrect usage:

```
# Will error, as no free parameters will be present
#device.run(circ(0.2), shots=0)

# Will succeed
device.run(circ, shots=0, inputs={'phi': 0.2, 'theta': 0.2})
```

Getting Expert advice

Connect with quantum computing experts directly in the Braket management console to get additional guidance around your workloads.

To explore Expert advice options through Braket Direct, open the Braket console, choose **Braket Direct** in the left pane, and navigate to the **Expert advice** section. The following Expert advice options are available:

- **Braket office hours:** Braket office hours are 1:1 sessions, first come first-serve, and take place every month. Each available office hour slot is 30 minutes and free of charge. Talking to Braket experts can help you get from ideation to execution faster by exploring use-case-to-device fit, identifying options to best use Braket for your algorithm, and getting recommendations for how to use certain Braket features like Amazon Braket Hybrid Jobs, Braket Pulse, or Analog Hamiltonian Simulation.
  - To sign up for Braket office hours, select **Sign up** and fill out contact information, workload details, and your desired discussion topics.
  - You will receive a calendar invitation to the next available slot through your email.

> **Note**
>
> For emergent issues or quick troubleshooting questions, we recommend reaching out to AWS Support. For non-urgent questions, you can also use the AWS re:Post forum or the Quantum Computing Stack Exchange, where you can browse previously answered questions and ask new ones.

- **Quantum hardware provider offerings:** IonQ, QuEra, and Rigetti each provide professional services offerings through AWS Marketplace.

    - To explore their offerings, select **Connect** and browse their listings.
    - To learn more about professional services offerings on the AWS Marketplace, see Professional services products.
- **Amazon Quantum Solutions Lab (QSL):** The QSL is a collaborative research and professional services team staffed with quantum computing experts who can help you effectively explore quantum computing and assess the current performance of this technology.
    - To contact the QSL, select **Connect**, and fill out contact information and use case details.
    - The QSL team will reach out to you through email with next steps.

Getting started with Amazon Braket Hybrid Jobs

This section provides information on components and instructions about how to set up your hybrid jobs in Amazon Braket.

You can access hybrid jobs in Braket using:

- The Amazon Braket Python SDK.
- The Amazon Braket console.
- The Amazon Braket API.

In this section:

- What is a Hybrid Job?
- When to use Amazon Braket Hybrid Jobs
- Inputs, outputs, environmental variables, and helper functions
- Define the environment for your algorithm script
- Using hyperparameters

What is a Hybrid Job?

Amazon Braket Hybrid Jobs offers a way for you to run hybrid quantum-classical algorithms requiring both classical AWS resources and quantum processing units (QPUs). Hybrid Jobs is designed to spin up the requested classical resources, run your algorithm, and release the instances after completion so you only pay for *what you use*.

Hybrid Jobs is ideal for long-running, iterative algorithms that involve the use of both classical computing resources and quantum computing resources. With Hybrid Jobs, after submitting your algorithm to run, Braket will run your algorithm in a scalable, containerized environment. Once the algorithm has completed, you can then retrieve the results.

Additionally, quantum tasks that are created from a hybrid job benefit from higher priority queueing to the target QPU device. This prioritization ensures that your quantum computations are processed and ran ahead of other tasks waiting in the queue. This is particularly advantageous for iterative hybrid algorithms, where the results of one quantum task depend on the outcomes of prior quantum tasks. Examples of such algorithms include the Quantum Approximate Optimization Algorithm (QAOA), variational quantum eigensolver, or quantum machine learning. You can also monitor your algorithm progress in near-real time, enabling you to keep track of costs, budget, or custom metrics such as training loss or expectation values.

When to use Amazon Braket Hybrid Jobs

Amazon Braket Hybrid Jobs enables you to run hybrid quantum-classical algorithms, such as the Variational Quantum Eigensolver (VQE) and the Quantum Approximate Optimization Algorithm (QAOA), that combine classical compute resources with quantum computing devices to optimize the performance of today's quantum systems. Amazon Braket Hybrid Jobs provides three main benefits:

1. **Performance**: Amazon Braket Hybrid Jobs provides better performance than running hybrid algorithms from your own environment. While your job is running, it has priority access to the selected target QPU. Tasks from your job run ahead of other tasks queued on the device. This results in shorter and more predictable runtimes for hybrid algorithms. Amazon Braket Hybrid Jobs also supports parametric compilation. You can submit a circuit using free parameters and Braket compiles the circuit once, without the need to recompile for subsequent parameter updates to the same circuit, resulting in even faster runtimes.
2. **Convenience**: Amazon Braket Hybrid Jobs simplifies setting up and managing your compute environment and keeping it running while your hybrid algorithm runs. You just provide your algorithm script and select a quantum device (either a quantum processing unit or a simulator) on which to run. Amazon Braket waits for the target device to become available, spins up the classical resources, runs the workload in pre-built container environments, returns the results to Amazon Simple Storage Service (Amazon S3), and releases the compute resources.

3. **Metrics**: Amazon Braket Hybrid Jobs provides on-the-fly insights into running algorithms and delivers customizable algorithm metrics in near real-time to Amazon CloudWatch and the Amazon Braket console so you can track the progress of your algorithms.

Inputs, outputs, environmental variables, and helper functions

In addition to the file or files that makes up your complete algorithm script, your hybrid job can have additional inputs and outputs. When your hybrid job starts, Amazon Braket copies inputs provided as part of the hybrid job creation into the container that runs the algorithm script. When the hybrid job completes, all outputs defined during the algorithm are copied to the Amazon S3 location specified.

> **Note**
>
> *Algorithm metrics* are reported in real time and do not follow this output procedure.

Amazon Braket also provides several environment variables and helper functions to simplify the interactions with container inputs and outputs.

This section explains the key concepts of the `AwsQuantumJob.create` function provided by the Amazon Braket Python SDK and their mapping to the container file structure.

In this section:

- Inputs
- Outputs
- Environmental variables
- Helper functions

Inputs

Input data: Input data can be provided to the hybrid algorithm by specifying the input data file, which is set up as a dictionary, with the `input_data` argument. The user defines the `input_data` argument within the `AwsQuantumJob.create` function in the SDK. This copies the input data to to the container file system at the location given by the environment variable `"AMZN_BRAKET_INPUT_DIR"`. For a couple examples of how input data is used in a hybrid algorithm, see the QAOA with Amazon Braket Hybrid Jobs and PennyLane and Quantum machine learning in Amazon Braket Hybrid Jobs Jupyter notebooks.

> **Note**
>
> When the input data is large (>1GB), there will be a long wait time before the hybrid job is submitted. This is due to the fact that the local input data will first be uploaded to an S3 bucket, then the S3 path will be added to the hybrid job request, and, finally, the hybrid job request is submitted to Braket service.

Hyperparameters: If you pass in `hyperparameters`, they are available under the environment variable `"AMZN_BRAKET_HP_FILE"`.

> **Note**
>
> For more information about how to create hyperparameters and input data and then pass this information to the hybrid job script, see the Use hyperparameters section and this github page.

Checkpoints: To specify a `job-arn` whose checkpoint you want to use in a new hybrid job, use the `copy_checkpoints_from_job` command. This command copies over the checkpoint data to the `checkpoint_configs3Uri` of the new hybrid job, making it available at the path given by the environment variable `AMZN_BRAKET_CHECKPOINT_DIR` while the job runs. The default is None, meaning checkpoint data from another hybrid job will not be used in the new hybrid job.

Outputs

Quantum Tasks: Quantum task results are stored in the S3 location `s3://amazon-braket-<region>-<accountID>/jobs/<job-name>/tasks`.

Job results: Everything that your algorithm script saves to the directory given by the environment variable `"AMZN_BRAKET_JOB_RESULTS_DIR"` is copied to the S3 location specified in `output_data_config`. If the value is not specified, it defaults to `s3://amazon-braket-<region>-<accountID>/jobs/<job-name>/<timestamp>/data`. We provide the SDK helper function **`save_job_result`**, which you can use to store results conveniently in the form of a dictionary when called from your algorithm script.

Checkpoints: If you want to use checkpoints, you can save them in the directory given by the environment variable `"AMZN_BRAKET_CHECKPOINT_DIR"`. You can also use the SDK helper function `save_job_checkpoint` instead.

Algorithm metrics: You can define algorithm metrics as part of your algorithm script that are emitted to Amazon CloudWatch and displayed in real time in the Amazon Braket console while your hybrid job is running. For an example of how to use algorithm metrics, see Use Amazon Braket Hybrid Jobs to run a QAOA algorithm.

Environmental variables

Amazon Braket provides several environment variables to simplify the interactions with container inputs and outputs. The folllowing code lists the environmental variables that Braket uses.

- `AMZN_BRAKET_INPUT_DIR` – The input data directory opt/braket/input/data.
- `AMZN_BRAKET_JOB_RESULTS_DIR` – The output directory opt/braket/model to write job results to.
- `AMZN_BRAKET_JOB_NAME` – The name of the job.
- `AMZN_BRAKET_CHECKPOINT_DIR` – The checkpoint directory.
- `AMZN_BRAKET_HP_FILE` – The file containing the hyperparameters.
- `AMZN_BRAKET_DEVICE_ARN` – The device ARN (AWS Resource Name).
- `AMZN_BRAKET_OUT_S3_BUCKET` – The output Amazon S3 bucket, as specified in the `CreateJob` request's `OutputDataConfig`.
- `AMZN_BRAKET_SCRIPT_ENTRY_POINT` – The entry point as specified in the `CreateJob` request's `ScriptModeConfig`.
- `AMZN_BRAKET_SCRIPT_COMPRESSION_TYPE` – The compression type as specified in the `CreateJob` request's `ScriptModeConfig`.
- `AMZN_BRAKET_SCRIPT_S3_URI` – The Amazon S3 location of the user's script as specified in the `CreateJob` request's `ScriptModeConfig`.
- `AMZN_BRAKET_TASK_RESULTS_S3_URI` – The Amazon S3 location where the SDK would store the quantum task results by default for the job.
- `AMZN_BRAKET_JOB_RESULTS_S3_PATH` – The Amazon S3 location where the job results would be stored, as specified in `CreateJob` request's `OutputDataConfig`.
- `AMZN_BRAKET_JOB_TOKEN` – The string that should be passed to `CreateQuantumTask`'s `jobToken` parameter for quantum tasks created in the job container.

Helper functions

Amazon Braket provides several helper functions to simplify the interactions with container inputs and outputs. These helper functions would be called from within the algorithm script that is used to run your Hybrid Job. The following example demonstrates how to use them.

```
get_checkpoint_dir() # Get the checkpoint directory
get_hyperparameters() # Get the hyperparameters as strings
get_input_data_dir() # Get the input data directory
get_job_device_arn() # Get the device specified by the hybrid job
get_job_name() # Get the name of the hybrid job.
get_results_dir() # Get the path to a results directory
save_job_result() # Save hybrid job results
save_job_checkpoint() # Save a checkpoint
load_job_checkpoint() # Load a previously saved checkpoint
```

Define the environment for your algorithm script

Amazon Braket supports environments defined by containers for your algorithm script:

- A base container (the default, if no `image_uri` is specified)
- A container with CUDA-Q
- A container with Tensorflow and PennyLane
- A container with PyTorch, PennyLane, and CUDA-Q

The following table provides details about the containers and the libraries they include.

Amazon Braket containers

Type	Base	CUDA-Q	TensorFlow	PyTorch
Image URI	292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-base-jobs:latest	292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-cudaq-jobs:latest	292282985366.dkr.ecr.us-east-1.amazonaws.com/amazon-braket-tensorflow-jobs:latest	292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-pytorch-jobs:latest

Type	Base	CUDA-Q	TensorFlow	PyTorch
Inherited Libraries		• amazon-braket-default-simulator • amazon-braket-pennylane-plugin • amazon-braket-schemas • amazon-braket-sdk • awscli • botocore • boto3 • dask • matplotlib • numpy • pandas • PennyLane • PennyLane-Lightning • qiskit-braket-provider • requests • sagemaker-training • scikit-learn • scipy	• awscli • numpy • pandas • scipy	• awscli • numpy • pandas • scipy

Type	Base	CUDA-Q	TensorFlow	PyTorch
Additional Libraries	• amazon-braket-default-simulator • amazon-braket-pennylane-plugin • amazon-braket-schemas • amazon-braket-sdk • awscli • boto3 • ipykernel • matplotlib • networkx • numpy • openbabel • pandas • PennyLane • protobuf • psi4 • rsa • scipy	• cudaq • cudaq-qec • cudaq-solvers	• amazon-braket-default-simulator • amazon-braket-pennylane-plugin • amazon-braket-schemas • amazon-braket-sdk • ipykernel • keras • matplotlib • networkx • openbabel • PennyLane • protobuf • psi4 • rsa • PennyLane-Lightning-gpu • cuQuantum	• amazon-braket-default-simulator • amazon-braket-pennylane-plugin • amazon-braket-schemas • amazon-braket-sdk • ipykernel • keras • matplotlib • networkx • openbabel • PennyLane • protobuf • psi4 • rsa • PennyLane-Lightning-gpu • cuQuantum • cudaq • cudaq-qec • cudaq-solvers

You can view and access the open source container definitions at [aws/amazon-braket-containers](). Choose the container that best matches your use case. You can use any of the available AWS Regions in Braket (us-east-1, us-west-1, us-west-2, eu-north-1, eu-west-2), but the container Region must match the Region for your hybrid job. Specify the container image when you create a hybrid job by adding one of the following three arguments to your `create(…)` call in the hybrid job script. You can install additional dependencies into the container you choose at runtime (at the cost of startup or runtime) because the Amazon Braket containers have internet connectivity. The following example is for the us-west-2 Region.

- **Base image:** image_uri="292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-base-jobs:latest"
- **CUDA-Q image:** image_uri="292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-cudaq-jobs:latest"
- **Tensorflow image:** image_uri="292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-tensorflow-jobs:latest"
- **PyTorch image:** image_uri="292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-pytorch-jobs:latest"

The `image-uris` can also be retrieved using the `retrieve_image()` function in the Amazon Braket SDK. The following example shows how to retrieve them from the us-west-2 AWS Region.

```
from braket.jobs.image_uris import retrieve_image, Framework

image_uri_base = retrieve_image(Framework.BASE, "us-west-2")
image_uri_cudaq = retrieve_image(Framework.CUDAQ, "us-west-2")
image_uri_tf = retrieve_image(Framework.PL_TENSORFLOW, "us-west-2")
image_uri_pytorch = retrieve_image(Framework.PL_PYTORCH, "us-west-2")
```

Using hyperparameters

You can define hyperparameters needed by your algorithm, such as the learning rate or step size, when you create a hybrid job. Hyperparameter values are typically used to control various aspects of the algorithm, and can often be tuned to optimize the algorithm's performance. To use hyperparameters in a Braket hybrid job, you need to specify their names and values explicitly as a dictionary. Note that the values must be of the string datatype. You specify the hyperparameter values that you want to test when searching for the optimal set of values. The first step to using hyperparameters is to set up and define the hyperparameters as a dictionary, which can be seen in the following code:

```
# Defining the number of qubits used
n_qubits = 8
```

```
# Defining the number of layers used
n_layers = 10
# Defining the number of iterations used for your optimization algorithm
n_iterations = 10

hyperparams = {
    "n_qubits": n_qubits,
    "n_layers": n_layers,
    "n_iterations": n_iterations
}
```

You would then pass the hyperparameters defined in the code snippet given above to be used in the algorithm of your choice with something that looks like the following:

```
import time
from braket.aws import AwsQuantumJob

# Name your job so that it can be later identified
job_name = f"qcbm-gaussian-training-{n_qubits}-{n_layers}-" + str(int(time.time()))

job = AwsQuantumJob.create(
    # Run this hybrid job on the SV1 simulator
    device="arn:aws:braket:::device/quantum-simulator/amazon/sv1",
    # The directory or single file containing the code to run.
    source_module="qcbm",
    # The main script or function the job will run.
    entry_point="qcbm.qcbm_job:main",
    # Set the job_name
    job_name=job_name,
    # Set the hyperparameters
    hyperparameters=hyperparams,
    # Define the file that contains the input data
    input_data="data.npy", # Or input_data=s3_path
    # wait_until_complete=False,
)
```

> **Note**
>
> In order to learn more about input data see the Inputs section.

The hyperparameters would then be loaded into the hybrid job script using the following code:

```
import json
import os

# Load the Hybrid Job hyperparameters
hp_file = os.environ["AMZN_BRAKET_HP_FILE"]
with open(hp_file, "r") as f:
    hyperparams = json.load(f)
```

> **Note**
>
> For more information about how to pass information like the input data and the device arn to the hybrid job script, see this github page.

A couple guides that are very useful for learning about how to use hyperparameters are given by the QAOA with Amazon Braket Hybrid Jobs and PennyLane and Quantum machine learning in Amazon Braket Hybrid Jobs tutorials.

Using PennyLane with Amazon Braket

Hybrid algorithms are algorithms that contain both classical and quantum instructions. The classical instructions are ran on classical hardware (an EC2 instance or your laptop), and the quantum instructions are ran either on a simulator or on a quantum computer. We recommend that you run hybrid algorithms using the Hybrid Jobs feature. For more information, see When to use Amazon Braket Jobs.

Amazon Braket enables you to set up and run hybrid quantum algorithms with the assistance of the **Amazon Braket PennyLane plugin**, or with the **Amazon Braket Python SDK** and example notebook repositories. Amazon Braket example notebooks, based on the SDK, enable you to set up and run certain hybrid algorithms without the PennyLane plugin. However, we recommend PennyLane because it provides a richer experience.

About hybrid quantum algorithms

Hybrid quantum algorithms are important to the industry today because contemporary quantum computing devices generally produce noise, and therefore, errors. Every quantum gate added to a computation increases the chance of adding noise; therefore, long-running algorithms can be overwhelmed by noise, which results in faulty computation.

Pure quantum algorithms such as Shor's (Quantum Phase Estimation example) or Grover's (Grover's example) require thousands, or millions, of operations. For this reason, they can be impractical for existing quantum devices, which are generally referred to as *noisy intermediate-scale quantum* (NISQ) devices.

In hybrid quantum algorithms, quantum processing units (QPUs) work as co-processors for classic CPUs, specifically to speed up certain calculations in a classical algorithm. Circuit executions become much shorter, within reach of the capabilities of today's devices.

In this section:

- Amazon Braket with PennyLane
- Hybrid algorithms in Amazon Braket example notebooks
- Hybrid algorithms with embedded PennyLane simulators
- Adjoint gradient on PennyLane with Amazon Braket simulators
- Using Hybrid Jobs and PennyLane to run a QAOA algorithm
- Run hybrid workloads with PennyLane embedded simulators

Amazon Braket with PennyLane

Amazon Braket provides support for PennyLane, an open-source software framework built around the concept of *quantum differentiable programming*. You can use this framework to train quantum circuits in the same way that you would train a neural network to find solutions for computational problems in quantum chemistry, quantum machine learning, and optimization.

The PennyLane library provides interfaces to familiar machine learning tools, including PyTorch and TensorFlow, to make training quantum circuits quick and intuitive.

- **The PennyLane Library** -– PennyLane is pre-installed in Amazon Braket notebooks. For access to Amazon Braket devices from PennyLane, open a notebook and import the PennyLane library with the following command.

```
import pennylane as qml
```

Tutorial notebooks help you get started quickly. Alternatively, you can use PennyLane on Amazon Braket from an IDE of your choice.

- **The Amazon Braket PennyLane plugin** — To use your own IDE, you can install the Amazon Braket PennyLane plugin manually. The plugin connects PennyLane with the Amazon Braket Python SDK, so you can run circuits in PennyLane on Amazon Braket devices. To install the the PennyLane plugin, use the following command.

```
pip install amazon-braket-pennylane-plugin
```

The following example demonstrates how to set up access to Amazon Braket devices in PennyLane:

```
# to use SV1
import pennylane as qml
sv1 = qml.device("braket.aws.qubit", device_arn="arn:aws:braket:::device/quantum-
simulator/amazon/sv1", wires=2)

# to run a circuit:
@qml.qnode(sv1)
def circuit(x):
    qml.RZ(x, wires=0)
    qml.CNOT(wires=[0,1])
    qml.RY(x, wires=1)
    return qml.expval(qml.PauliZ(1))

result = circuit(0.543)


#To use the local sim:
local = qml.device("braket.local.qubit", wires=2)
```

For tutorial examples and more information about PennyLane, see the Amazon Braket examples repository.

The Amazon Braket PennyLane plugin enables you to switch between Amazon Braket QPU and embedded simulator devices in PennyLane with a single line of code. It offers two Amazon Braket quantum devices to work with PennyLane:

- `braket.aws.qubit` for running with the Amazon Braket service's quantum devices, including QPUs and simulators
- `braket.local.qubit` for running with the Amazon Braket SDK's local simulator

The Amazon Braket PennyLane plugin is open source. You can install it from the PennyLane Plugin GitHub repository.

For more information about PennyLane, see the documentation on the PennyLane website.

Hybrid algorithms in Amazon Braket example notebooks

Amazon Braket does provide a variety of example notebooks that do not rely on the PennyLane plugin for running hybrid algorithms. You can get started with any of these Amazon Braket hybrid example notebooks that illustrate *variational methods*, such as the Quantum Approximate Optimization Algorithm (QAOA) or Variational Quantum Eigensolver (VQE).

The Amazon Braket example notebooks rely on the Amazon Braket Python SDK. The SDK provides a framework to interact with quantum computing hardware devices through Amazon Braket. It is an open source library that is designed to assist you with the quantum portion of your hybrid workflow.

You can explore Amazon Braket further with our example notebooks.

Hybrid algorithms with embedded PennyLane simulators

Amazon Braket Hybrid Jobs now comes with high performance CPU- and GPU-based embedded simulators from PennyLane. This family of embedded simulators can be embedded directly within your hybrid jobs container and includes the fast state-vector `lightning.qubit` simulator, the `lightning.gpu` simulator accelerated using NVIDIA's cuQuantum library, and others. These embedded simulators are ideally suited for variational algorithms such as quantum machine learning that can benefit from advanced methods such as the adjoint differentiation method. You can run these embedded simulators on one or multiple CPU or GPU instances.

With Hybrid Jobs, you can now run your variational algorithm code using a combination of a classical co-processor and a QPU, an Amazon Braket on-demand simulator such as SV1, or directly using the embedded simulator from PennyLane.

The embedded simulator is already available with the Hybrid Jobs container, you need to decorate your main Python function with the `@hybrid_job` decorator. To use the PennyLane `lightning.gpu` simulator, you also need to specify a GPU instance in the `InstanceConfig` as shown in the following code snippet:

```
import pennylane as qml
```

```
from braket.jobs import hybrid_job
from braket.jobs.config import InstanceConfig


@hybrid_job(device="local:pennylane/lightning.gpu",
 instance_config=InstanceConfig(instanceType="ml.p3.8xlarge"))
def function(wires):
    dev = qml.device("lightning.gpu", wires=wires)
    ...
```

Refer to the example notebook to get started with using a PennyLane embedded simulator with Hybrid Jobs.

Adjoint gradient on PennyLane with Amazon Braket simulators

With the PennyLane plugin for Amazon Braket, you can compute gradients using the adjoint differentiation method when running on the local state vector simulator or SV1.

Note: To use the adjoint differentiation method, you must specify `diff_method='device'` in your qnode, and **not** `diff_method='adjoint'`. See the following example.

```
device_arn = "arn:aws:braket:::device/quantum-simulator/amazon/sv1"
dev = qml.device("braket.aws.qubit", wires=wires, shots=0, device_arn=device_arn)

@qml.qnode(dev, diff_method="device")
def cost_function(params):
    circuit(params)
    return qml.expval(cost_h)

gradient = qml.grad(circuit)
initial_gradient = gradient(params0)
```

Note

Currently, PennyLane will compute grouping indices for QAOA Hamiltonians and use them to split the Hamiltonian into multiple expectation values. If you want to use SV1's adjoint differentiation capability when running QAOA from PennyLane, you will need reconstruct the cost Hamiltonian by removing the grouping indices, like so: `cost_h, mixer_h = qml.qaoa.max_clique(g, constrained=False) cost_h = qml.Hamiltonian(cost_h.coeffs, cost_h.ops)`

Using Hybrid Jobs and PennyLane to run a QAOA algorithm

In this section, you will use what you have learned to write an actual hybrid program using PennyLane with parametric compilation. You use the algorithm script to address a Quantum Approximate Optimization Algorithm (QAOA) problem. The program creates a cost function corresponding to a classical Max Cut optimization problem, specifies a parametrized quantum circuit, and uses a gradient descent method to optimize the parameters so that the cost function is minimized. In this example, we generate the problem graph in the algorithm script for simplicity, but for more typical use cases the best practice is to provide the problem specification through a dedicated channel in the input data configuration. The flag `parametrize_differentiable` defaults to `True` so you automatically get the benefits of improved runtime performance from parametric compilation on supported QPUs.

```
import os
import json
import time

from braket.jobs import save_job_result
from braket.jobs.metrics import log_metric

import networkx as nx
import pennylane as qml
from pennylane import numpy as np
from matplotlib import pyplot as plt

def init_pl_device(device_arn, num_nodes, shots, max_parallel):
    return qml.device(
        "braket.aws.qubit",
        device_arn=device_arn,
        wires=num_nodes,
        shots=shots,
        # Set s3_destination_folder=None to output task results to a default folder
        s3_destination_folder=None,
        parallel=True,
        max_parallel=max_parallel,
        parametrize_differentiable=True, # This flag is True by default.
    )

def start_here():
    input_dir = os.environ["AMZN_BRAKET_INPUT_DIR"]
    output_dir = os.environ["AMZN_BRAKET_JOB_RESULTS_DIR"]
    job_name = os.environ["AMZN_BRAKET_JOB_NAME"]
```

```
    checkpoint_dir = os.environ["AMZN_BRAKET_CHECKPOINT_DIR"]
    hp_file = os.environ["AMZN_BRAKET_HP_FILE"]
    device_arn = os.environ["AMZN_BRAKET_DEVICE_ARN"]

    # Read the hyperparameters
    with open(hp_file, "r") as f:
        hyperparams = json.load(f)

    p = int(hyperparams["p"])
    seed = int(hyperparams["seed"])
    max_parallel = int(hyperparams["max_parallel"])
    num_iterations = int(hyperparams["num_iterations"])
    stepsize = float(hyperparams["stepsize"])
    shots = int(hyperparams["shots"])

    # Generate random graph
    num_nodes = 6
    num_edges = 8
    graph_seed = 1967
    g = nx.gnm_random_graph(num_nodes, num_edges, seed=graph_seed)

    # Output figure to file
    positions = nx.spring_layout(g, seed=seed)
    nx.draw(g, with_labels=True, pos=positions, node_size=600)
    plt.savefig(f"{output_dir}/graph.png")

    # Set up the QAOA problem
    cost_h, mixer_h = qml.qaoa.maxcut(g)

    def qaoa_layer(gamma, alpha):
        qml.qaoa.cost_layer(gamma, cost_h)
        qml.qaoa.mixer_layer(alpha, mixer_h)

    def circuit(params, **kwargs):
        for i in range(num_nodes):
            qml.Hadamard(wires=i)
        qml.layer(qaoa_layer, p, params[0], params[1])

    dev = init_pl_device(device_arn, num_nodes, shots, max_parallel)

    np.random.seed(seed)
    cost_function = qml.ExpvalCost(circuit, cost_h, dev, optimize=True)
    params = 0.01 * np.random.uniform(size=[2, p])
```

```
    optimizer = qml.GradientDescentOptimizer(stepsize=stepsize)
    print("Optimization start")

    for iteration in range(num_iterations):
        t0 = time.time()

        # Evaluates the cost, then does a gradient step to new params
        params, cost_before = optimizer.step_and_cost(cost_function, params)
        # Convert cost_before to a float so it's easier to handle
        cost_before = float(cost_before)

        t1 = time.time()

        if iteration == 0:
            print("Initial cost:", cost_before)
        else:
            print(f"Cost at step {iteration}:", cost_before)

        # Log the current loss as a metric
        log_metric(
            metric_name="Cost",
            value=cost_before,
            iteration_number=iteration,
        )

        print(f"Completed iteration {iteration + 1}")
        print(f"Time to complete iteration: {t1 - t0} seconds")

    final_cost = float(cost_function(params))
    log_metric(
        metric_name="Cost",
        value=final_cost,
        iteration_number=num_iterations,
    )

    # We're done with the hybrid job, so save the result.
    # This will be returned in job.result()
    save_job_result({"params": params.numpy().tolist(), "cost": final_cost})
```

Note

Parametric compilation is supported on all superconducting, gate-based QPUs from Rigetti Computing with the exception of pulse level programs.

Run hybrid workloads with PennyLane embedded simulators

Lets look at how you can use embedded simulators from PennyLane on Amazon Braket Hybrid Jobs to run hybrid workloads. Pennylane's GPU-based embedded simulator, `lightning.gpu`, uses the [Nvidia cuQuantum library](#) to accelerate circuit simulations. The embedded GPU simulator is pre-configured in all of the Braket [job containers](#) that users can use out of the box. In this page, we show you how to use `lightning.gpu` to speed up your hybrid workloads.

Using `lightning.gpu` for QAOA workloads

Consider the Quantum Approximate Optimization Algorithm (QAOA) examples from this [notebook](#). To select an embedded simulator, you specify the `device` argument to be a string of the form: `"local:<provider>/<simulator_name>"`. For example, you would set `"local:pennylane/lightning.gpu"` for `lightning.gpu`. The device string you give to the Hybrid Job when you launch is passed to the job as the environment variable `"AMZN_BRAKET_DEVICE_ARN"`.

```
device_string = os.environ["AMZN_BRAKET_DEVICE_ARN"]
prefix, device_name = device_string.split("/")
device = qml.device(simulator_name, wires=n_wires)
```

In this page, compare the two embedded PennyLane state vector simulators `lightning.qubit` (which is CPU-based) and `lightning.gpu` (which is GPU-based). Provide the simulators with custom gate decompositions to compute various gradients.

Now you are ready to prepare the hybrid job launching script. Run the QAOA algorithm using two instance types: `m5.2xlarge` and `p3.2xlarge`. The `m5.2xlarge` instance type is comparable to a standard developer laptop. The `p3.2xlarge` is an accelerated computing instance that has a single NVIDIA Volta GPU with 16GB of memory.

The `hyperparameters` for all your hybrid jobs will be the same. All you need to do to try out different instances and simulators is change two lines as follows.

```
# Specify device that the hybrid job will primarily be targeting
```

```
device = "local:pennylane/lightning.qubit"
# Run on a CPU based instance with about as much power as a laptop
instance_config = InstanceConfig(instanceType='ml.m5.2xlarge')
```

or:

```
# Specify device that the hybrid job will primarily be targeting
device = "local:pennylane/lightning.gpu"
# Run on an inexpensive GPU based instance
instance_config = InstanceConfig(instanceType='ml.p3.2xlarge')
```

> **Note**
>
> If you specify the `instance_config` as using a GPU-based instance, but choose the `device` to be the embedded CPU-based simulator (`lightning.qubit`), the GPU **will not** be used. Make sure to use the embedded GPU-based simulator if you wish to target the GPU!

First, you can create two hybrid jobs and solve Max-Cut with QAOA on a graph with 18 vertices. This translates to an 18-qubit circuit—relatively small and feasible to run quickly on your laptop or the `m5.2xlarge` instance.

```
num_nodes = 18
num_edges = 24
seed = 1967

graph = nx.gnm_random_graph(num_nodes, num_edges, seed=seed)

# And similarly for the p3 job
m5_job = AwsQuantumJob.create(
    device=device,
    source_module="qaoa_source",
    job_name="qaoa-m5-" + str(int(time.time())),
    image_uri=image_uri,
    # Relative to the source_module
    entry_point="qaoa_source.qaoa_algorithm_script",
    copy_checkpoints_from_job=None,
    instance_config=instance_config,
    # general parameters
    hyperparameters=hyperparameters,
```

```
    input_data={"input-graph": input_file_path},
    wait_until_complete=True,
)
```

The mean iteration time for the `m5.2xlarge` instance is about 25 seconds, while for the `p3.2xlarge` instance it is about 12 seconds. For this 18-qubit workflow, the GPU instance gives us a 2x speedup. If you look at the Amazon Braket Hybrid Jobs [pricing page](), you can see that the cost per minute for an `m5.2xlarge` instance is $0.00768, while for the `p3.2xlarge` instance it is $0.06375. To run for 5 total iterations, as you did here, would cost $0.016 using the CPU instance or $0.06375 using the GPU instance — both pretty inexpensive!

Now solve a Max-Cut problem on a 24-vertex graph, which will translate to 24 qubits. Run the hybrid jobs again on the same two instances and compare the cost.

> **Note**
>
> Time to run this hybrid job on the CPU instance may be about five hours.

```
num_nodes = 24
num_edges = 36
seed = 1967

graph = nx.gnm_random_graph(num_nodes, num_edges, seed=seed)

# And similarly for the p3 job
m5_big_job = AwsQuantumJob.create(
    device=device,
    source_module="qaoa_source",
    job_name="qaoa-m5-big-" + str(int(time.time())),
    image_uri=image_uri,
    # Relative to the source_module
    entry_point="qaoa_source.qaoa_algorithm_script",
    copy_checkpoints_from_job=None,
    instance_config=instance_config,
    # general parameters
    hyperparameters=hyperparameters,
    input_data={"input-graph": input_file_path},
    wait_until_complete=True,
)
```

The mean iteration time for the `m5.2xlarge` instance is about an hour, while the `p3.2xlarge` instance is about two minutes. For this larger problem, the GPU instance is an order of magnitude faster. All you had to do to benefit from this speedup was to change two lines of code, swapping out the instance type and the local simulator used. To run for 5 total iterations, as was done here, would cost about $2.27072 using the CPU instance or about $0.775625 using the GPU instance. The CPU usage is not only more expensive, but also takes more time to run. Accelerating this workflow with a GPU instance available on AWS, using PennyLane's embedded simulator backed by NVIDIA CuQuantum, allows you to run workflows with intermediate qubit counts (between 20 and 30) for less total cost and in less time. This means you can experiment with quantum computing even for problems that are too big to run quickly on your laptop or a similarly-sized instance.

Quantum machine learning and data parallelism

If your workload type is quantum machine learning (QML) that trains on datasets, you can further accelerate your workload using data parallelism. In QML, the model contains one or more quantum circuits. The model may or may not also contain classical neural nets. When training the model with the dataset, the parameters in the model are updated to minimize the loss function. A loss function is usually defined for a single data point, and the total loss for the average loss over the whole dataset. In QML, the losses are usually computed in serial before averaging to total loss for gradient computations. This procedure is time consuming, especially when there are hundreds of data points.

Because the loss from one data point does not depend on other data points, the losses can be evaluated in parallel! Losses and gradients associated with different data points can be evaluated at the same time. This is known as data parallelism. With SageMaker's distributed data parallel library, Amazon Braket Hybrid Jobs make it easier for you to use data parallelism to accelerate your training.

Consider the following QML workload for data parallelism which uses the Sonar dataset dataset from the well-known UCI repository as an example for binary classification. The Sonar dataset have 208 data points each with 60 features that are collected from sonar signals bouncing off materials. Each data points is either labeled as "M" for mines or "R" for rocks. Our QML model consists of an input layer, a quantum circuit as a hidden layer, and an output layer. The input and output layers are classical neural nets implemented in PyTorch. The quantum circuit is integrated with the PyTorch neural nets using PennyLane's qml.qnn module. See our example notebooks for more detail about the workload. Like the QAOA example above, you can harness the power of GPU by using embedded GPU-based simulators like PennyLane's `lightning.gpu` to improve the performance over embedded CPU-based simulators.

To create a hybrid job, you can call `AwsQuantumJob.create` and specify the algorithm script, device, and other configurations through its keyword arguments.

```
instance_config = InstanceConfig(instanceType='ml.p3.2xlarge')

hyperparameters={"nwires": "10",
                 "ndata": "32",
                 ...
}

job = AwsQuantumJob.create(
    device="local:pennylane/lightning.gpu",
    source_module="qml_source",
    entry_point="qml_source.train_single",
    hyperparameters=hyperparameters,
    instance_config=instance_config,
    ...
)
```

In order to use data parallelism, you need to modify few lines of code in the algorithm script for the SageMaker distributed library to correctly parallelize the training. First, you import the `smdistributed` package which does most of the heavy-lifting for distributing your workloads across multiple GPUs and multiple instances. This package is preconfigured in the Braket PyTorch and TensorFlow containers. The `dist` module tells our algorithm script what the total number of GPUs for the training (`world_size`) is as well as the `rank` and `local_rank` of a GPU core. `rank` is the absolute index of a GPU across all instances, while `local_rank` is the index of a GPU within an instance. For example, if there are four instances each with eight GPUs allocated for the training, the `rank` ranges from 0 to 31 and the `local_rank` ranges from 0 to 7.

```
import smdistributed.dataparallel.torch.distributed as dist

dp_info = {
    "world_size": dist.get_world_size(),
    "rank": dist.get_rank(),
    "local_rank": dist.get_local_rank(),
}
batch_size //= dp_info["world_size"] // 8
batch_size = max(batch_size, 1)
```

Next, you define a `DistributedSampler` according to the `world_size` and `rank` and then pass it into the data loader. This sampler avoids GPUs accessing the same slice of a dataset.

```
train_sampler = torch.utils.data.distributed.DistributedSampler(
    train_dataset,
    num_replicas=dp_info["world_size"],
    rank=dp_info["rank"]
)
train_loader = torch.utils.data.DataLoader(
    train_dataset,
    batch_size=batch_size,
    shuffle=False,
    num_workers=0,
    pin_memory=True,
    sampler=train_sampler,
)
```

Next, you use the `DistributedDataParallel` class to enable data parallelism.

```
from smdistributed.dataparallel.torch.parallel.distributed import
 DistributedDataParallel as DDP

model = DressedQNN(qc_dev).to(device)
model = DDP(model)
torch.cuda.set_device(dp_info["local_rank"])
model.cuda(dp_info["local_rank"])
```

The above are the changes you need to use data parallelism. In QML, you often want to save results and print training progress. If each GPU runs the saving and printing command, the log will be flooded with the repeated information and the results will overwrite each other. To avoid this, you can only save and print from the GPU that has `rank` 0.

```
if dp_info["rank"]==0:
    print('elapsed time: ', elapsed)
    torch.save(model.state_dict(), f"{output_dir}/test_local.pt")
    save_job_result({"last loss": loss_before})
```

Amazon Braket Hybrid Jobs supports `ml.p3.16xlarge` instance types for the SageMaker distributed data parallel library. You configure the instance type through the `InstanceConfig` argument in Hybrid Jobs. For the SageMaker distributed data parallel library to know that data parallelism is enabled, you need to add two additional hyperparameters, `"sagemaker_distributed_dataparallel_enabled"` setting to `"true"` and `"sagemaker_instance_type"` setting to the instance type you are using. These two hyperparameters are used by `smdistributed` package. Your algorithm script does not need to explicitly use them. In Amazon Braket SDK, it provides a convenient keyword argument `distribution`. With `distribution="data_parallel"` in hybrid job creation, the Amazon Braket SDK automatically inserts the two hyperparameters for you. If you use the Amazon Braket API, you need to include these two hyperparameters.

With the instance and data parallelism configured, you can now submit your hybrid job. There are 8 GPUs in a `ml.p3.16xlarge` instance. When you set `instanceCount=1` , the workload is distributed across the 8 GPUs in the instance. When you set `instanceCount` greater than one, the workload is distributed across GPUs available in all instances. When using multiple instances, each instance incurs a charge based on how much time you use it. For example, when you use four instances, the billable time is four times the run time per instance because there are four instances running your workloads at the same time.

```
instance_config = InstanceConfig(instanceType='ml.p3.16xlarge',
                                 instanceCount=1,
)

hyperparameters={"nwires": "10",
                 "ndata": "32",
                 ...,
}

job = AwsQuantumJob.create(
    device="local:pennylane/lightning.gpu",
    source_module="qml_source",
    entry_point="qml_source.train_dp",
    hyperparameters=hyperparameters,
    instance_config=instance_config,
    distribution="data_parallel",
    ...
)
```

> **Note**
>
> In the above hybrid job creation, `train_dp.py` is the modified algorithm script for using data parallelism. Keep in mind that data parallelism only works correctly when you modify your algorithm script according to the above section. If the data parallelism option is

enabled without a correctly modified algorithm script, the hybrid job may throw errors, or each GPU may repeatedly process the same data slice, which is inefficient.

Compare the run time and cost in an example where when train a model with a 26-qubit quantum circuit for the binary classification problem mentioned above. The `ml.p3.16xlarge` instance used in this example costs $0.4692 per minute. Without data parallelism, it takes the simulator about 45 minutes to train the model for 1 epoch (that is, over 208 data points) and it costs about $20. With data parallelism across 1 instance and 4 instances, it only takes 6 minutes and 1.5 minutes respectively, which translates to roughly $2.8 for both. By using data parallelism across 4 instances, you not only improve the run time by 30x, but also reduce costs by an order of magnitude!

Using CUDA-Q with Amazon Braket

NVIDIA's CUDA-Q is a software library designed for programming hybrid quantum algorithms that combine CPUs, GPUs, and Quantum processing units (QPUs). It provides a unified programming model, allowing developers to express both classical and quantum instructions within a single program, streamlining workflows. CUDA-Q accelerates quantum program simulation and runtime with its built-in CPU and GPU simulators.

Using CUDA-Q on Amazon Braket Hybrid Jobs offers a flexible, on-demand computing environment. Computational instances run only for the duration of your workload, ensuring you pay only for what you use. Amazon Braket Hybrid Jobs also provides a scalable experience. Users can start with smaller instances for prototyping and testing, then scale up to larger instances capable of handling greater workloads for full experiments.

Amazon Braket Hybrid Jobs support GPUs that are essential for maximizing CUDA-Q's potential. GPUs significantly speed up quantum program simulations compared to CPU-based simulators, especially when working with high qubit count circuits. Parallelization becomes straightforward when using CUDA-Q on Amazon Braket Hybrid Jobs. Hybrid Jobs simplifies the distribution of circuit sampling and observable evaluations across multiple computational nodes. This seamless parallelization of CUDA-Q workloads allows users to focus more on developing their workloads rather than setting up infrastructure for large-scale experiments.

To get started, see the CUDA-Q starter example on the Amazon Braket examples Github to use a CUDA-Q hybrid jobs container provided by Braket.

The following code snippet is a `hello-world` example for running a CUDA-Q program with Amazon Braket Hybrid Jobs.

```
image_uri = retrieve_image(Framework.CUDAQ, AwsSession().region)

@hybrid_job(device='local:nvidia/qpp-cpu', image_uri=image_uri)
def hello_quantum():
    import cudaq

    # define the backend
    device=get_job_device_arn()
    cudaq.set_target(device.split('/')[-1])

    # define the Bell circuit
    kernel = cudaq.make_kernel()
    qubits = kernel.qalloc(2)
    kernel.h(qubits[0])
    kernel.cx(qubits[0], qubits[1])

    # sample the Bell circuit
    result = cudaq.sample(kernel, shots_count=1000)
    measurement_probabilities = dict(result.items())

    return measurement_probabilities
```

The above example simulates a Bell circuit on a CPU simulator. This example runs locally on your laptop or Braket Jupyter notebook. Because of the `local=True` setting, when you run this script, a container will start in your local environment to run the CUDA-Q program for testing and debugging. After you finish testing, you can remove the `local=True` flag and run your job AWS. To learn more, see [Getting started with Amazon Braket Hybrid Jobs](#).

If your workloads have a high qubit count, a large number of circuits or a large number of iterations, you can use more powerful CPU computing resources by specifying the `instance_config` setting. The following code snippet shows how to configure the `instance_config` setting in the `hybrid_job` decorator. For more information about supported instance types, see [Configure the hybrid job instance to run your script](#). For a list of instance types, see [Amazon EC2 Instance types](#).

```
@hybrid_job(
    device="local:nvidia/qpp-cpu",
    image_uri=image_uri,
    instance_config=InstanceConfig(instanceType="ml.c5.2xlarge"),
)
def my_job_script():
```

```
    ...
```

For more demanding workloads, you can run your workloads on a CUDA-Q GPU simulator. To enable a GPU simulator, use the backend name `nvidia`. The `nvidia` backend operates as a CUDA-Q GPU simulator. Next, select an Amazon EC2 instance type that supports an NVIDIA GPU. The following code snippet shows the GPU-configured `hybrid_job` decorator.

```
@hybrid_job(
    device="local:nvidia/nvidia",
    image_uri=image_uri,
    instance_config=InstanceConfig(instanceType="ml.p3.2xlarge"),
)
def my_job_script():
    ...
```

Amazon Braket Hybrid Jobs supports parallel GPU simulations with CUDA-Q. You can parallelize the evaluation of multiple observables or multiple circuits to boost the performance of your workload. To parallelize multiple observables, make the following changes to your algorithm script.

Set the `mgpu` option of the `nvidia` backend. This is required to parallelize the observables. The parallelization uses MPI for communication between GPUs, so MPI needs to be initialized before the execution and finalized after it.

Next, specify the execution mode by setting `execution=cudaq.parallel.mpi`. The following code snippet shows these changes.

```
cudaq.set_target("nvidia", option="mqpu")
cudaq.mpi.initialize()
result = cudaq.observe(
    kernel, hamiltonian, shots_count=n_shots, execution=cudaq.parallel.mpi
)
cudaq.mpi.finalize()
```

In the `hybrid_job` decorator specify an instance type that hosts multiple GPUs as shown in the following code snippet.

```
@hybrid_job(
    device="local:nvidia/nvidia-mqpu",
    instance_config=InstanceConfig(instanceType="ml.p3.8xlarge", instanceCount=1),
    image_uri=image_uri,
```

```
)
def parallel_observables_gpu_job(sagemaker_mpi_enabled=True):
    ...
```

The parallel simulations notebook in the Amazon Braket examples Github provide end-to-end examples that demonstrate how to run quantum program simulations on GPU backends and perform parallel simulations of observables and circuit batches.

Running your workloads on quantum computers

After completing simulator testing, you can transition to running experiments on QPUs. Just switch the target to an Amazon Braket QPU, such as the IQM, IonQ, or Rigetti devices. The following code snippet illustrates how to set the target to the IQM Garnet device. For a list of available QPUs, see the Amazon Braket Console.

```
device_arn = "arn:aws:braket:eu-north-1::device/qpu/iqm/Garnet"
cudaq.set_target("braket", machine=device_arn)
```

For more information about Amazon Braket Hybrid Jobs, see Working with Amazon Braket Hybrid Jobs in the developer guide. To learn more about CUDA-Q, see the CUDA-Q documentation.

Run your circuits with OpenQASM 3.0

Amazon Braket now supports OpenQASM 3.0 for gate-based quantum devices and simulators. This user guide provides information about the subset of OpenQASM 3.0 supported by Braket. Braket customers now have the choice of submitting Braket circuits with the SDK or by directly providing OpenQASM 3.0 strings to all gate-based devices with the Amazon Braket API and the Amazon Braket Python SDK.

The topics in this guide walk you through various examples of how to complete the following quantum tasks.

- Create and submit OpenQASM quantum tasks on different Braket devices
- Access the supported operations and result types
- Simulate noise with OpenQASM
- Use verbatim compilation with OpenQASM
- Troubleshoot OpenQASM issues

This guide also provides an introduction to certain hardware-specific features that can be implemented with OpenQASM 3.0 on Braket and links to further resources.

In this section:

- What is OpenQASM 3.0?
- When to use OpenQASM 3.0
- How OpenQASM 3.0 works
- Prerequisites
- What OpenQASM features does Braket support?
- Create and submit an example OpenQASM 3.0 quantum task
- Support for OpenQASM on different Braket devices
- Simulate noise with OpenQASM 3.0
- Qubit rewiring with OpenQASM 3.0
- Verbatim compilation with OpenQASM 3.0
- The Braket console
- Additional resources
- Computing gradients with OpenQASM 3.0
- Measuring specific qubits with OpenQASM 3.0

What is OpenQASM 3.0?

The Open Quantum Assembly Language (OpenQASM) is an intermediate representation for quantum instructions. OpenQASM is an open-source framework and is widely used for the specification of quantum programs for gate-based devices. With OpenQASM, users can program the quantum gates and measurement operations that form the building blocks of quantum computation. The previous version of OpenQASM (2.0) was used by a number of quantum programming libraries to describe basic programs.

The new version of OpenQASM (3.0) extends the previous version to include more features, such as pulse-level control, gate timing, and classical control flow to bridge the gap between end-user interface and hardware description language. Details and specification on the current version 3.0 are available on the GitHub OpenQASM 3.x Live Specification. OpenQASM's future development is governed by the OpenQASM 3.0 Technical Steering Committee, of which AWS is a member alongside IBM, Microsoft, and the University of Innsbruck.

When to use OpenQASM 3.0

OpenQASM provides an expressive framework to specify quantum programs through low-level controls that are not architecture specific, making it well suited as a representation across multiple gate-based devices. The Braket support for OpenQASM furthers its adoption as a consistent approach to developing gate-based quantum algorithms, reducing the need for users to learn and maintain libraries in multiple frameworks.

If you have existing libraries of programs in OpenQASM 3.0, you can adapt them for use with Braket rather than completely rewriting these circuits. Researchers and developers should also benefit from an increasing number of available third-party libraries with support for algorithm development in OpenQASM.

How OpenQASM 3.0 works

Support for OpenQASM 3.0 from Braket provides feature parity with the current Intermediate Representation. This means that anything you can do today on hardware devices and on-demand simulators with Braket, you can do with OpenQASM using the Braket API. You can run OpenQASM 3.0 programs by directly supplying OpenQASM strings to all gate-based devices in a manner that is similar to how circuits are currently supplied to devices on Braket. Braket users can also integrate third-party libraries that support OpenQASM 3.0. The rest of this guide details how to develop OpenQASM representations for use with Braket.

Prerequisites

To use OpenQASM 3.0 on Amazon Braket, you must have version v1.8.0 of the Amazon Braket Python Schemas and v1.17.0 or higher of the Amazon Braket Python SDK.

If you are a first time user of Amazon Braket, you need to enable Amazon Braket. For instructions, see Enable Amazon Braket.

What OpenQASM features does Braket support?

The following section lists the OpenQASM 3.0 data types, statements, and pragma instructions supported by Braket.

In this section:

- Supported OpenQASM data types
- Supported OpenQASM statements

- Braket OpenQASM pragmas
- Advanced feature support for OpenQASM on the Local Simulator
- Supported operations and grammar with OpenPulse

Supported OpenQASM data types

The following OpenQASM data types are supported by Amazon Braket.

- Non-negative integers are used for (virtual and physical) qubit indices:
  - `cnot q[0], q[1];`
  - `h $0;`
- Floating-point numbers or constants may be used for gate rotation angles:
  - `rx(-0.314) $0;`
  - `rx(pi/4) $0;`

> **Note**
>
> pi is a built-in constant in OpenQASM and cannot be used as a parameter name.

- Arrays of complex numbers (with the OpenQASM `im` notation for imaginary part) are allowed in result type pragmas for defining general hermitian observables and in unitary pragmas:
  - `#pragma braket unitary [[0, -1im], [1im, 0]] q[0]`
  - `#pragma braket result expectation hermitian([[0, -1im], [1im, 0]]) q[0]`

Supported OpenQASM statements

The following OpenQASM statements are supported by Amazon Braket.

- `Header: OPENQASM 3;`
- Classic bit declarations:
  - `bit b1;` (equivalently, `creg b1;`)
  - `bit[10] b2;` (equivalently, `creg b2[10];`)
- Qubit declarations:

  - `qubit b1;` (equivalently, `qreg b1;`)
  - `qubit[10] b2;` (equivalently, `qreg b2[10];`)
- Indexing within arrays: `q[0]`
- Input: `input float alpha;`
- specification of physical qubits: `$0`
- Supported gates and operations on a device:
  - `h $0;`
  - `iswap q[0], q[1];`

> **Note**
>
> A device's supported gates can be found in the device properties for OpenQASM actions; no gate definitions are needed to use these gates.

- Verbatim box statements. Currently, we do not support box duration notation. Native gates and physical qubits are required in verbatim boxes.

```
#pragma braket verbatim
box{
    rx(0.314) $0;
}
```

- Measurement and measurement assignment on qubits or a whole qubit register.
  - `measure $0;`
  - `measure q;`
  - `measure q[0];`
  - `b = measure q;`
  - `measure q # b;`

Braket OpenQASM pragmas

The following OpenQASM pragma instructions are supported by Amazon Braket.

- Noise pragmas
  - `#pragma braket noise bit_flip(0.2) q[0]`
  - `#pragma braket noise phase_flip(0.1) q[0]`
  - `#pragma braket noise pauli_channel`
- Verbatim pragmas
  - `#pragma braket verbatim`
- Result type pragmas
  - Basis invariant result types:
    - State vector: `#pragma braket result state_vector`
    - Density matrix: `#pragma braket result density_matrix`
  - Gradient computation pragmas:
    - Adjoint gradient: `#pragma braket result adjoint_gradient expectation(2.2 * x[0] @ x[1]) all`
  - Z basis result types:
    - Amplitude: `#pragma braket result amplitude "01"`
    - Probability: `#pragma braket result probability q[0], q[1]`
  - Basis rotated result types
    - Expectation: `#pragma braket result expectation x(q[0]) @ y([q1])`
    - Variance: `#pragma braket result variance hermitian([[0, -1im], [1im, 0]]) $0`
    - Sample: `#pragma braket result sample h($1)`

> **Note**
>
> OpenQASM 3.0 is backwards compatible with OpenQASM 2.0, so programs written using 2.0 can run on Braket. However the features of OpenQASM 3.0 supported by Braket do have some minor syntax differences, such as `qreg` vs `creg` and `qubit` vs `bit`. There are also differences in measurement syntax, and these need to be supported with their correct syntax.

Advanced feature support for OpenQASM on the Local Simulator

The `LocalSimulator` supports advanced OpenQASM features which are not offered as part of Braket's QPU's or on-demand simulators. The following list of features are only supported in the `LocalSimulator`:

- Gate modifiers
- OpenQASM built-in gates
- Classical variables
- Classical operations
- Custom gates
- Classical control
- QASM files
- Subroutines

For examples of each advanced feature, see this sample notebook. For the full OpenQASM specification, see the OpenQASM website.

Supported operations and grammar with OpenPulse

Supported OpenPulse Data Types

Cal blocks:

```
cal {
    ...
}
```

Defcal blocks:

```
// 1 qubit
defcal x $0 {
...
}

// 1 qubit w. input parameters as constants
defcal my_rx(pi) $0 {
...
```

```
}

// 1 qubit w. input parameters as free parameters
defcal my_rz(angle theta) $0 {
...
}

// 2 qubit (above gate args are also valid)
defcal cz $1, $0 {
...
}
```

Frames:

```
frame my_frame = newframe(port_0, 4.5e9, 0.0);
```

Waveforms:

```
// prebuilt
waveform my_waveform_1 = constant(1e-6, 1.0);

//arbitrary
waveform my_waveform_2 = {0.1 + 0.1im, 0.1 + 0.1im, 0.1, 0.1};
```

Custom Gate Calibration Example:

```
cal {
    waveform wf1 = constant(1e-6, 0.25);
}

defcal my_x $0 {
   play(wf1, q0_rf_frame);
}

defcal my_cz $1, $0 {
    barrier q0_q1_cz_frame, q0_rf_frame;
    play(q0_q1_cz_frame, wf1);
    delay[300ns] q0_rf_frame
    shift_phase(q0_rf_frame, 4.366186381749424);
    delay[300ns] q0_rf_frame;
    shift_phase(q0_rf_frame.phase, 5.916747563126659);
    barrier q0_q1_cz_frame, q0_rf_frame;
```

```
    shift_phase(q0_q1_cz_frame, 2.183093190874712);
}

bit[2] ro;
my_x $0;
my_cz $1,$0;
c[0] = measure $0;
```

Arbitrary pulse example:

```
bit[2] ro;
cal {
    waveform wf1 = {0.1 + 0.1im, 0.1 + 0.1im, 0.1, 0.1};
    barrier q0_drive, q0_q1_cross_resonance;
    play(q0_q1_cross_resonance, wf1);
    delay[300ns] q0_drive;
    shift_phase(q0_drive, 4.366186381749424);
    delay[300dt] q0_drive;
   barrier q0_drive, q0_q1_cross_resonance;
   play(q0_q1_cross_resonance, wf1);
    ro[0] = capture_v0(r0_measure);
    ro[1] = capture_v0(r1_measure);
}
```

Create and submit an example OpenQASM 3.0 quantum task

You can use the Amazon Braket Python SDK, Boto3, or the AWS CLI to submit OpenQASM 3.0 quantum tasks to an Amazon Braket device.

In this section:

- An example OpenQASM 3.0 program
- Use the Python SDK to create OpenQASM 3.0 quantum tasks
- Use Boto3 to create OpenQASM 3.0 quantum tasks
- Use the AWS CLI to create OpenQASM 3.0 tasks

An example OpenQASM 3.0 program

To create an OpenQASM 3.0 task, you can start with a basic OpenQASM 3.0 program (ghz.qasm) that prepares a GHZ state as shown in the following example.

```
// ghz.qasm
// Prepare a GHZ state
OPENQASM 3;

qubit[3] q;
bit[3] c;

h q[0];
cnot q[0], q[1];
cnot q[1], q[2];

c = measure q;
```

Use the Python SDK to create OpenQASM 3.0 quantum tasks

You can use the Amazon Braket Python SDK to submit this program to an Amazon Braket device with the following code. Be sure to replace the example Amazon S3 bucket location “amzn-s3-demo-bucket” with your own Amazon S3 bucket name.

```
with open("ghz.qasm", "r") as ghz:
    ghz_qasm_string = ghz.read()

# Import the device module
from braket.aws import AwsDevice
# Choose the Rigetti device
device = AwsDevice("arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3")
from braket.ir.openqasm import Program

program = Program(source=ghz_qasm_string)
my_task = device.run(program)

# Specify an optional s3 bucket location and number of shots
s3_location = ("amzn-s3-demo-bucket", "openqasm-tasks")
my_task = device.run(
    program,
    s3_location,
    shots=100,
)
```

Use Boto3 to create OpenQASM 3.0 quantum tasks

You can also use AWS Python SDK for Braket (Boto3) to create the quantum tasks using OpenQASM 3.0 strings, as shown in the following example. The following code snippet references ghz.qasm that prepares a GHZ state as shown above.

```
import boto3
import json

my_bucket = "amzn-s3-demo-bucket"
s3_prefix = "openqasm-tasks"

with open("ghz.qasm") as f:
    source = f.read()

action = {
    "braketSchemaHeader": {
        "name": "braket.ir.openqasm.program",
        "version": "1"
    },
    "source": source
}
device_parameters = {}
device_arn = "arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3"
shots = 100

braket_client = boto3.client('braket', region_name='us-west-1')
rsp = braket_client.create_quantum_task(
    action=json.dumps(
        action
    ),
    deviceParameters=json.dumps(
        device_parameters
    ),
    deviceArn=device_arn,
    shots=shots,
    outputS3Bucket=my_bucket,
    outputS3KeyPrefix=s3_prefix,
)
```

Use the AWS CLI to create OpenQASM 3.0 tasks

The AWS Command Line Interface (CLI) can also be used to submit OpenQASM 3.0 programs, as shown in the following example.

```
aws braket create-quantum-task \
    --region "us-west-1" \
    --device-arn "arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3" \
    --shots 100 \
    --output-s3-bucket "amzn-s3-demo-bucket" \
    --output-s3-key-prefix "openqasm-tasks" \
    --action '{
        "braketSchemaHeader": {
            "name": "braket.ir.openqasm.program",
            "version": "1"
        },
        "source": $(cat ghz.qasm)
    }'
```

Support for OpenQASM on different Braket devices

For devices supporting OpenQASM 3.0, the `action` field supports a new action through the `GetDevice` response, as shown in the following example for the Rigetti and IonQ devices.

```
//OpenQASM as available with the Rigetti device capabilities
{
    "braketSchemaHeader": {
        "name": "braket.device_schema.rigetti.rigetti_device_capabilities",
        "version": "1"
    },
    "service": {...},
    "action": {
        "braket.ir.jaqcd.program": {...},
        "braket.ir.openqasm.program": {
            "actionType": "braket.ir.openqasm.program",
            "version": [
                "1"
            ],
            ….
        }
    }
}
```

```
//OpenQASM as available with the IonQ device capabilities
{
    "braketSchemaHeader": {
        "name": "braket.device_schema.ionq.ionq_device_capabilities",
        "version": "1"
    },
    "service": {...},
    "action": {
        "braket.ir.jaqcd.program": {...},
        "braket.ir.openqasm.program": {
            "actionType": "braket.ir.openqasm.program",
            "version": [
                "1"
            ],
            ….
        }
    }
}
```

For devices that support pulse control, the `pulse` field is displayed in the `GetDevice` response. The following example show this `pulse` field for the Rigetti device.

```
// Rigetti
{
  "pulse": {
    "braketSchemaHeader": {
      "name": "braket.device_schema.pulse.pulse_device_action_properties",
      "version": "1"
    },
    "supportedQhpTemplateWaveforms": {
      "constant": {
        "functionName": "constant",
        "arguments": [
          {
            "name": "length",
            "type": "float",
            "optional": false
          },
          {
            "name": "iq",
            "type": "complex",
            "optional": false
```

```
          }
        ]
      },
      ...
    },
    "ports": {
      "q0_ff": {
        "portId": "q0_ff",
        "direction": "tx",
        "portType": "ff",
        "dt": 1e-9,
        "centerFrequencies": [
          375000000
        ]
      },
      ...
    },
    "supportedFunctions": {
      "shift_phase": {
        "functionName": "shift_phase",
        "arguments": [
          {
            "name": "frame",
            "type": "frame",
            "optional": false
          },
          {
            "name": "phase",
            "type": "float",
            "optional": false
          }
        ]
      },
     ...
    },
    "frames": {
      "q0_q1_cphase_frame": {
        "frameId": "q0_q1_cphase_frame",
        "portId": "q0_ff",
        "frequency": 462475694.24460185,
        "centerFrequency": 375000000,
        "phase": 0,
        "associatedGate": "cphase",
        "qubitMappings": [
```

```
          0,
          1
        ]
      },
      ...
    },
    "supportsLocalPulseElements": false,
    "supportsDynamicFrames": false,
    "supportsNonNativeGatesWithPulses": false,
    "validationParameters": {
      "MAX_SCALE": 4,
      "MAX_AMPLITUDE": 1,
      "PERMITTED_FREQUENCY_DIFFERENCE": 400000000
    }
  }
}
```

The preceding fields detail the following:

Ports:

Describes pre-made external (`extern`) device ports declared on the QPU in addition to the associated properties of the given port. All ports listed in this structure are pre-declared as valid identifiers within the `OpenQASM 3.0` program submitted by the user. The additional properties for a port include:

- Port id (portId)
  - The port name declared as an identifier in OpenQASM 3.0.
- Direction (direction)
  - The direction of the port. Drive ports transmit pulses (direction “tx”), while measurement ports receive pulses (direction “rx”).
- Port type (portType)
  - The type of action for which this port is responsible (for example, drive, capture, or ff - fast-flux).
- Dt (dt)
  - The time in seconds that represents a single sample time step on the given port.
- Qubit mappings (qubitMappings)
  - The qubits associated with the given port.

- Center frequencies (centerFrequencies)
  - A list of the associated center frequencies for all pre-declared or user-defined frames on the port. For more information, refer to Frames.
- QHP Specific Properties (qhpSpecificProperties)
  - An optional map detailing existing properties about the port specific to the QHP.

Frames:

Describes pre-made external frames declared on the QPU as well as associated properties about the frames. All frames listed in this structure are pre-declared as valid identifiers within the `OpenQASM 3.0` program submitted by the user. The additional properties for a frame include:

- Frame Id (frameId)
  - The frame name declared as an identifier in OpenQASM 3.0.
- Port Id (portId)
  - The associated hardware port for the frame.
- Frequency (frequency)
  - The default initial frequency of the frame.
- Center Frequency (centerFrequency)
  - The center of the frequency bandwidth for the frame. Typically, frames may only be adjusted to a certain bandwidth around the center frequency. As a result, frequency adjustments should stay within a given delta of the center frequency. You can find the bandwidth value in the validation parameters.
- Phase (phase)
  - The default initial phase of the frame.
- Associated Gate (associatedGate)
  - The gates associated with the given frame.
- Qubit Mappings (qubitMappings)
  - The qubits associated with the given frame.
- QHP Specific Properties (qhpSpecificProperties)
  - An optional map detailing existing properties about the frame specific to the QHP.

SupportsDynamicFrames:

Describes whether a frame can be declared in `cal` or `defcal` blocks through the OpenPulse `newframe` function. If this is false, only frames listed in the frame structure may be used within the program.

SupportedFunctions:

Describes the OpenPulse functions that are supported for the device in addition to the associated arguments, argument types, and return types for the given functions. To see examples of using the OpenPulse functions, see the OpenPulse specification. At this time, Braket supports:

- shift_phase
  - Shifts the phase of a frame by a specified value
- set_phase
  - Sets the phase of frame to the specified value
- swap_phases
  - Swaps the phases between two frames.
- shift_frequency
  - Shifts the frequency of a frame by a specified value
- set_frequency
  - Sets the frequency of frame to the specified value
- play
  - Schedules a waveform
- capture_v0
  - Returns the value on a capture frame to a bit register

SupportedQhpTemplateWaveforms:

Describes the pre-built waveform functions available on the device and the associated arguments and types. By default, Braket Pulse offers pre-built waveform routines on all devices, which are:

Constant

$$Constant(t, \tau, iq) = iq$$

`τ` is the length of the waveform and `iq` is a complex number.

```
def constant(length, iq)
```

Gaussian

$$Gaussian(t, \tau, \sigma, A = 1, ZaE = 0) = \frac{A}{1 - ZaE * \exp\left(-\frac{1}{2}\left(\frac{\tau}{2\sigma}\right)^2\right)} \left[\exp\left(-\frac{1}{2}\left(\frac{t - \frac{\tau}{2}}{\sigma}\right)^2\right) - ZaE * \exp\left(-\frac{1}{2}\left(\frac{\tau}{2\sigma}\right)^2\right)\right]$$

τ is the length of the waveform, σ is the width of the Gaussian, and A is the amplitude. If setting `ZaE` to `True`, the Gaussian is offset and rescaled such that it is equal to zero at the start and end of the waveform, and reaches A at maximum.

```
def gaussian(length, sigma, amplitude=1, zero_at_edges=False)
```

DRAG Gaussian

$$DRAG\_Gaussian(t, \tau, \sigma, \beta, A = 1, ZaE = 0) = \frac{A}{1 - ZaE * \exp\left(-\frac{1}{2}\left(\frac{\tau}{2\sigma}\right)^2\right)} \left(1 - i\beta\frac{t - \frac{\tau}{2}}{\sigma^2}\right) \left[\exp\left(-\frac{1}{2}\left(\frac{t - \frac{\tau}{2}}{\sigma}\right)^2\right) - ZaE * \exp\left(-\frac{1}{2}\left(\frac{\tau}{2\sigma}\right)^2\right)\right]$$

τ is the length of the waveform, σ is the width of the gaussian, β is a free parameter, and A is the amplitude. If setting `ZaE` to `True`, the Derivative Removal by Adiabatic Gate (DRAG) Gaussian is offset and rescaled such that it is equal to zero at the start and end of the waveform, and the real part reaches A at maximum. For more information about the DRAG waveform, see the paper [Simple Pulses for Elimination of Leakage in Weakly Nonlinear Qubits](#).

```
def drag_gaussian(length, sigma, beta, amplitude=1, zero_at_edges=False)
```

Erf Square

$$\text{Erf\_Square}(t, L, W, \sigma, A = 1, ZaE = 0) =$$

$$A \times \frac{\text{erf}((t - t_1)/\sigma) + \text{erf}(-(t - t_2)/\sigma)}{2 \times \text{erf}(W/2\sigma))}$$

Where L is the length, W is the width of the waveform, σ defines how fast the edges rise and fall, t_1=(L-W)/2 and $t_2$2=(L+W)/2, A is the amplitude. If setting ZaE to True, the Gaussian is offset and rescaled such that it is equal to zero at the start and end of the waveform, and reaches A at maximum. The following equation is the rescaled version of the waveform.

$$\mathrm{Erf\_Square}(..., ZaE = 1) = (a \times \mathrm{Erf\_Square}(..., ZaE = 0) - bA)/(a - b)$$

Where a=erf(W/2σ)and b=erf(-t_1/σ)/2+erf(t $_2$/σ)/2 .

```
def erf_square(length, width, sigma, amplitude=1, zero_at_edges=False)
```

SupportsLocalPulseElements:

Describes whether pulse elements, such as ports, frames, and waveforms may be defined locally in `defcal` blocks. If the value is `false`, elements must be defined in `cal` blocks.

SupportsNonNativeGatesWithPulses:

Describes whether we can or cannot use non-native gates in combination with pulse programs. For example, you cannot use a non-native gate like an `H` gate in a program without first defining the gate through `defcal` for the used qubit. You can find the list of native gates `nativeGateSet` key under the device capabilities.

ValidationParameters:

Describes pulse element validation boundaries, including:

- Maximum Scale / Maximum Amplitude values for waveforms (arbitrary and pre-built)
- Maximum frequency bandwidth from supplied center frequency in Hz
- Minimum pulse length/duration in seconds
- Maximum pulse length/duration in seconds

Supported Operations, Results and Result Types with OpenQASM

To find out which OpenQASM 3.0 features each device supports, you can refer to the `braket.ir.openqasm.program` key in the `action` field on the device capabilities output. For example, the following are the supported operations and result types available for the Braket State Vector simulator SV1.

```
...
  "action": {
    "braket.ir.jaqcd.program": {
      ...
    },
 "braket.ir.openqasm.program": {
      "version": [
        "1.0"
      ],
      "actionType": "braket.ir.openqasm.program",
      "supportedOperations": [
        "ccnot",
        "cnot",
        "cphaseshift",
        "cphaseshift00",
        "cphaseshift01",
        "cphaseshift10",
        "cswap",
        "cy",
        "cz",
        "h",
        "i",
        "iswap",
        "pswap",
        "phaseshift",
        "rx",
        "ry",
        "rz",
        "s",
        "si",
        "swap",
        "t",
        "ti",
        "v",
        "vi",
        "x",
        "xx",
        "xy",
        "y",
        "yy",
        "z",
        "zz"
      ],
```

```
      "supportedPragmas": [
        "braket_unitary_matrix"
      ],
      "forbiddenPragmas": [],
      "maximumQubitArrays": 1,
      "maximumClassicalArrays": 1,
      "forbiddenArrayOperations": [
        "concatenation",
        "negativeIndex",
        "range",
        "rangeWithStep",
        "slicing",
        "selection"
      ],
      "requiresAllQubitsMeasurement": true,
      "supportsPhysicalQubits": false,
      "requiresContiguousQubitIndices": true,
      "disabledQubitRewiringSupported": false,
      "supportedResultTypes": [
        {
          "name": "Sample",
          "observables": [
            "x",
            "y",
            "z",
            "h",
            "i",
            "hermitian"
          ],
          "minShots": 1,
          "maxShots": 100000
        },
        {
          "name": "Expectation",
          "observables": [
            "x",
            "y",
            "z",
            "h",
            "i",
            "hermitian"
          ],
          "minShots": 0,
          "maxShots": 100000
```

```
        },
        {
          "name": "Variance",
          "observables": [
            "x",
            "y",
            "z",
            "h",
            "i",
            "hermitian"
          ],
          "minShots": 0,
          "maxShots": 100000
        },
        {
          "name": "Probability",
          "minShots": 1,
          "maxShots": 100000
        },
        {
          "name": "Amplitude",
          "minShots": 0,
          "maxShots": 0
        }
        {
          "name": "AdjointGradient",
          "minShots": 0,
          "maxShots": 0
        }
      ]
    }
  },
...
```

Simulate noise with OpenQASM 3.0

To simulate noise with OpenQASM3, you use *pragma* instructions to add noise operators. For example, to simulate the noisy version of the GHZ program provided previously, you can submit the following OpenQASM program.

```
// ghz.qasm
// Prepare a GHZ state
OPENQASM 3;
```

```
qubit[3] q;
bit[3] c;

h q[0];
#pragma braket noise depolarizing(0.75) q[0] cnot q[0], q[1];
#pragma braket noise depolarizing(0.75) q[0]
#pragma braket noise depolarizing(0.75) q[1] cnot q[1], q[2];
#pragma braket noise depolarizing(0.75) q[0]
#pragma braket noise depolarizing(0.75) q[1]

c = measure q;
```

Specifications for all supported pragma noise operators are provided in the following list.

```
#pragma braket noise bit_flip(<float in [0,1/2]>) <qubit>
#pragma braket noise phase_flip(<float in [0,1/2]>) <qubit>
#pragma braket noise pauli_channel(<float>, <float>, <float>)  <qubit>
#pragma braket noise depolarizing(<float in [0,3/4]>) <qubit>
#pragma braket noise two_qubit_depolarizing(<float in [0,15/16]>) <qubit>, <qubit>
#pragma braket noise two_qubit_dephasing(<float in [0,3/4]>) <qubit>, <qubit>
#pragma braket noise amplitude_damping(<float in [0,1]>) <qubit>
#pragma braket noise generalized_amplitude_damping(<float in [0,1]> <float in [0,1]>)
 <qubit>
#pragma braket noise phase_damping(<float in [0,1]>) <qubit>
#pragma braket noise kraus([[<complex m0_00>, ], ...], [[<complex m1_00>, ], ...], ...)
 <qubit>[, <qubit>]     // maximum of 2 qubits and maximum of 4 matrices for 1 qubit,
 16 for 2
```

Kraus operator

To generate a Kraus operator, you can iterate through a list of matrices, printing each element of the matrix as a complex expression.

When using Kraus operators, remember the following:

- The number of qubits must not exceed 2. The current definition in the schemas sets this limit.
- The length of the argument list must be a multiple of 8. This means it must be composed only of 2x2 matrices.
- The total length does not exceed 2^{2*num_qubits} matrices. This means 4 matrices for 1 qubit and 16 for 2 qubits.

- All supplied matrices are completely positive trace preserving (CPTP).
- The product of the Kraus operators with their transpose conjugates need to add up to an identity matrix.

Qubit rewiring with OpenQASM 3.0

Amazon Braket supports the physical qubit notation within OpenQASM on Rigetti devices (to learn more see this page). When using physical qubits with the naive rewiring strategy, ensure that the qubits are connected on the selected device. Alternatively, if qubit registers are used instead, the PARTIAL rewiring strategy is enabled by default on Rigetti devices.

```
// ghz.qasm
// Prepare a GHZ state
OPENQASM 3;

h $0;
cnot $0, $1;
cnot $1, $2;

measure $0;
measure $1;
measure $2;
```

Verbatim compilation with OpenQASM 3.0

When you run a quantum circuit on quantum computers provided by vendors such as Rigetti, and IonQ, you can direct the compiler to run your circuits exactly as defined, without any modifications. This feature is known as *verbatim compilation*. With Rigetti devices, you can specify precisely what gets preserved—either an entire circuit or only specific parts of it. To preserve only specific parts of a circuit, you will need to use native gates within the preserved regions. Currently, IonQ only supports verbatim compilation for the entire circuit, so every instruction in the circuit needs to be enclosed in a verbatim box.

With OpenQASM, you can explicitly specify a verbatim pragma around a box of code that is then left untouched and not optimized by the low-level compilation routine of the hardware. The following code example shows how to use the `#pragma braket verbatim` directive to achieve this.

```
OPENQASM 3;
```

```
bit[2] c;

#pragma braket verbatim
box{
    rx(0.314159) $0;
    rz(0.628318) $0, $1;
    cz $0, $1;
}

c[0] = measure $0;
c[1] = measure $1;
```

For more detailed information on the process of verbatim compilation, including examples and best practices, see the Verbatim compilation sample notebook available in the amazon-braket-examples github repository.

The Braket console

OpenQASM 3.0 tasks are available and can be managed within the Amazon Braket console. On the console, you have the same experience submitting quantum tasks in OpenQASM 3.0 as you had submitting existing quantum tasks.

Additional resources

OpenQASM is available in all Amazon Braket Regions.

For an example notebook for getting started with OpenQASM on Amazon Braket, see Braket Tutorials GitHub.

Computing gradients with OpenQASM 3.0

Amazon Braket supports the computation of gradients on both on-demand and local simulators when running in the `shots=0` (exact) mode. This is achieved through the use of the adjoint differentiation method. To specify the gradient you want to compute, you can provide the appropriate pragma, as demonstrated in code in the following example.

```
OPENQASM 3.0;
input float alpha;

bit[2] b;
```

```
qubit[2] q;

h q[0];
h q[1];
rx(alpha) q[0];
rx(alpha) q[1];
b[0] = measure q[0];
b[1] = measure q[1];

#pragma braket result adjoint_gradient h(q[0]) @ i(q[1]) alpha
```

Instead of listing all the individual parameters explicitly, you can also specify the `all` keyword within the pragma. This will compute the gradient with respect to all of the `input` parameters listed, which can be a convenient option when the number of parameters is very large. In this case, the pragma will look like the code in the following example.

```
#pragma braket result adjoint_gradient h(q[0]) @ i(q[1]) all
```

All observable types are supported in Amazon Braket's OpenQASM 3.0 implementation, including individual operators, tensor products, Hermitian observables, and `Sum` observables. The specific operator you want to use when computing gradients must be wrapped within the `expectation()` function, and the qubits that each term of the observable acts upon must be explicitly specified.

Measuring specific qubits with OpenQASM 3.0

The local state vector simulator and local density matrix simulator provided by Amazon Braket support the submission of OpenQASM programs where a subset of the circuit's qubits can be selectively measured. This capability, often referred to as partial measurement, allows for more targeted and efficient quantum computations. For example, in the following code snippet, you can create a two-qubit circuit and choose to only measure the first qubit, while leaving the second qubit unmeasured.

```
partial_measure_qasm = """
OPENQASM 3.0;
bit[1] b;
qubit[2] q;
h q[0];
cnot q[0], q[1];
b[0] = measure q[0];
"""
```

In this example, we have a quantum circuit with two qubits, `q[0]` and `q[1]`, but we are only interested in measuring the state of the first qubit. This is achieved by the line `b[0] = measure q[0]`, which measures the state of qubit[0] and stores the result in the classical bit b[0]. To run this partial measurement scenario, we can run the following code on the local state vector simulator provided by Amazon Braket.

```
from braket.devices import LocalSimulator

local_sim = LocalSimulator()
partial_measure_local_sim_task =
 local_sim.run(OpenQASMProgram(source=partial_measure_qasm), shots = 10)
partial_measure_local_sim_result = partial_measure_local_sim_task.result()
print(partial_measure_local_sim_result.measurement_counts)
print("Measured qubits: ", partial_measure_local_sim_result.measured_qubits)
```

You can check whether a device supports partial measurement by inspecting the `requiresAllQubitsMeasurement` field in its action properties; if it is `False`, then partial measurement is supported.

```
from braket.devices import Devices

AwsDevice(Devices.Rigetti.Ankaa3).properties.action['braket.ir.openqasm.program'].requiresAllQu
```

Here, `requiresAllQubitsMeasurement` is `False`, which indicates that not all qubits must be measured.

Explore Experimental Capabilities

Experimental capabilities provide access to hardware with limited availability and emergent new features. For features for QuEra Aquila, you must request access directly in the Braket console to available experimental capabilities.

To request access to Experimental Capabilities for QuEra Aquila:

1. Navigate to the Amazon Braket console and select **Braket Direct** in the left menu, and then navigate to the **Experimental Capabilities** section.
2. Choose **Get Access** and fill out the requested information.
3. Provide details about the workload and where you plan to use this capability.

In this section:

- Access to local detuning on QuEra Aquila
- Access to tall geometries on QuEra Aquila
- Access to tight geometries on QuEra Aquila
- Dynamic circuits on IQM devices

Access to local detuning on QuEra Aquila

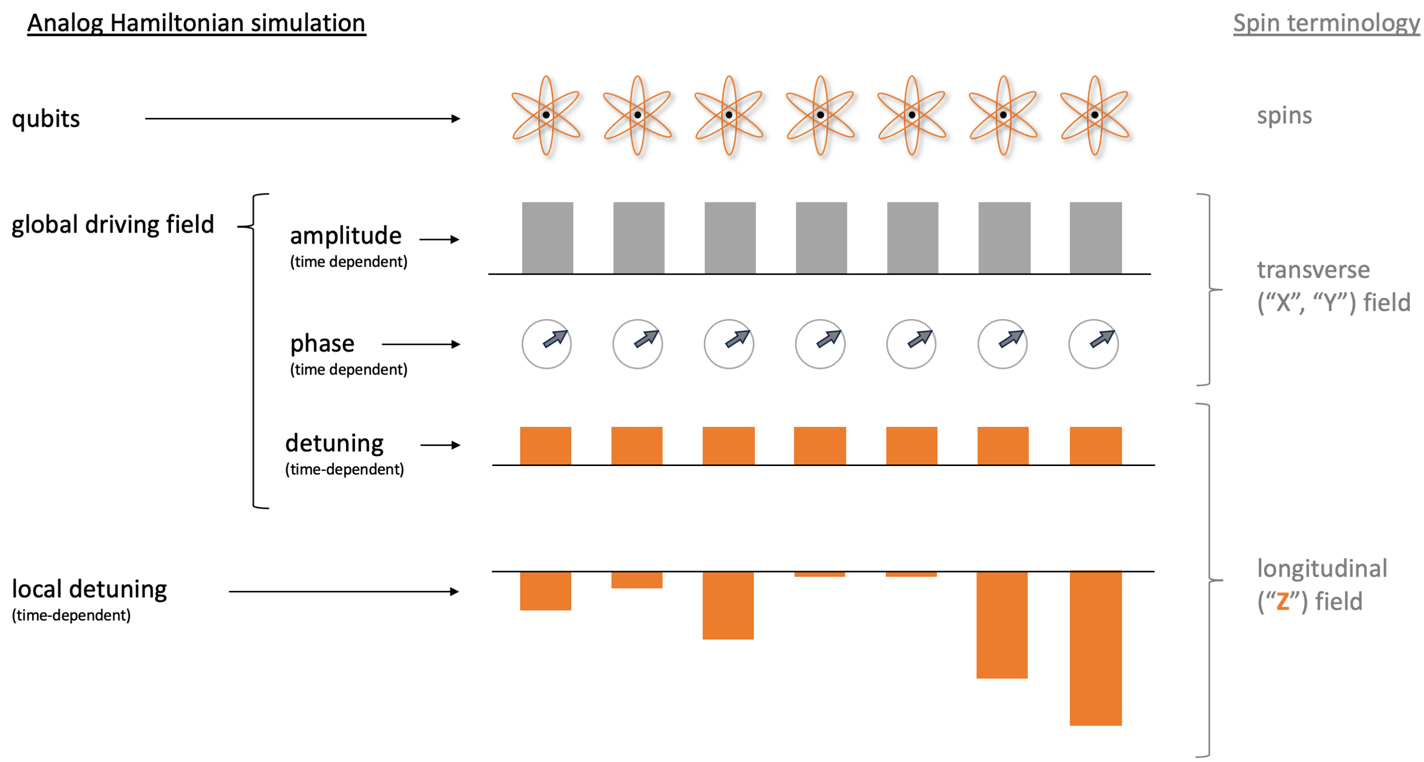
Local detuning (LD) is a new, time-dependent control field with a customizable spatial pattern. The LD field affects qubits according to a customizable spatial pattern, realizing different Hamiltonians for different qubits beyond what the uniform driving field and the Rydberg-Rydberg interaction can create.

Constraints:

The spatial pattern of the local detuning field is customizable for each AHS program, but it is constant over the course of a program. The time series of the local detuning field must start and end at zero with all values being less than or equal to zero. Additionally, the parameters of the local detuning field are limited by numerical constraints, which can be viewed through the Braket SDK in the specific device properties section - `aquila_device.properties.paradigm.rydberg.rydbergLocal`.

Limitations:

When running quantum programs that use the local detuning field (even if its magnitude is set to constant zero in the Hamiltonian), the device experiences faster decoherence than the T2 time listed in the performance section of Aquila's properties. When unnecessary, it is best practice to omit the local detuning field from the Hamiltonian of the AHS program.



Examples:

1. **Simulating the effect of non-uniform longitudinal magnetic field in spin systems**

   While the amplitude and phase of the driving field have the same effect on the qubits as the transverse magnetic field on spins, the sum of the driving field's detuning and the local detuning produces the same effect on the qubits as the longitudinal field on spins. With the spatial control over the local detuning field, more complex spin systems can be simulated.

2. **Preparing non-equilibrium initial states**

   The example notebook Simulating lattice gauge theory with Rydberg atoms shows how to suppress the central atom of a 9-atom linear arrangement from being excited when annealing the system towards the Z2 ordered phase. After the preparation step, the local detuning field is ramped down, and the AHS program continues to simulate the time evolution of the system starting from this particular non-equilibrium state.

3. **Solving weighted optimization problems**

   The example notebook Maximum weight independent set (MWIS) shows how to solve a MWIS problem on Aquila. The local detuning field is used to define the weights on the nodes of the unit disk graph, whose edges are realized by the Rybderg-blockage effect. Starting from the uniform ground state, and gradually ramping up the local detuning field makes the system transition into the ground state of the MWIS Hamiltonian to find solutions to the problem.

Access to tall geometries on QuEra Aquila

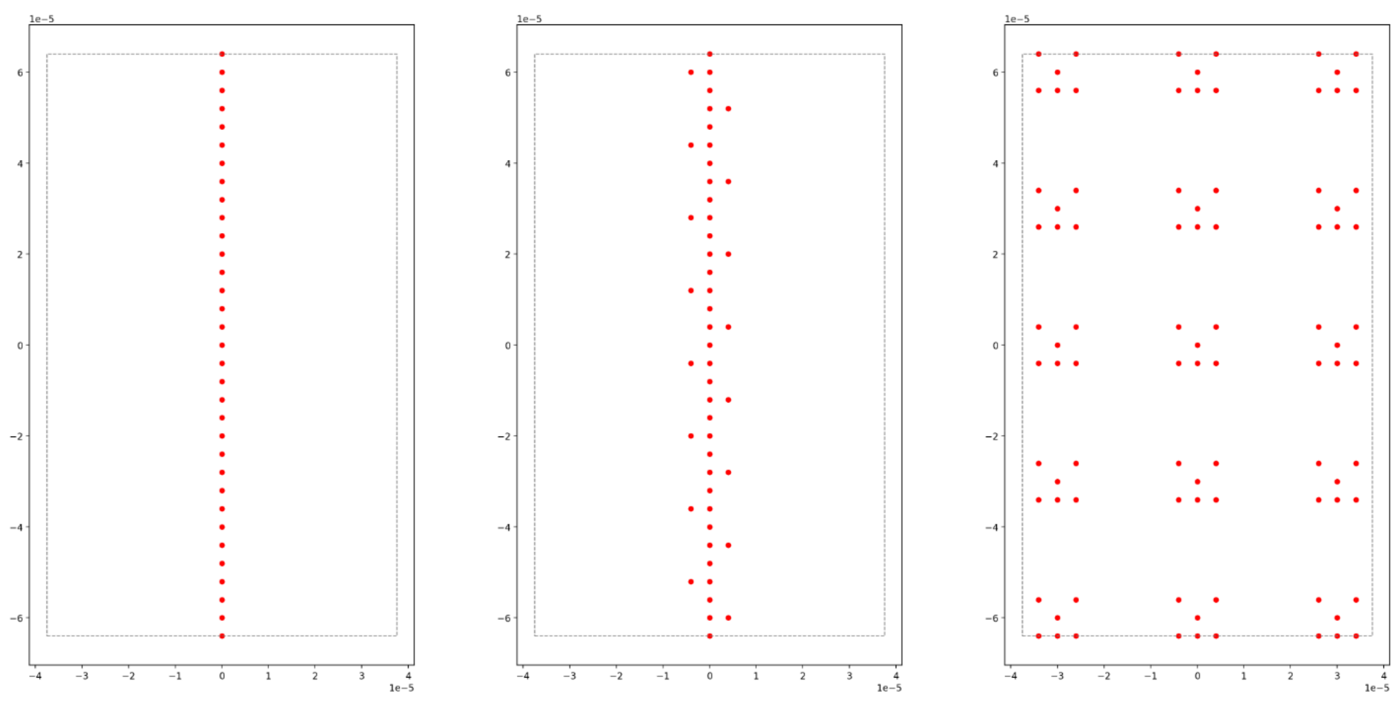
The tall geometries feature allows you to specify geometries with increased height. With this capability, the atom arrangements of your AHS programs can span an additional length in the y direction beyond Aquila's regular capabilities.

Constraints:

The max height for tall geometries is 0.000128 m (128 um).

Limitations:

When this experimental capability is enabled for your account, the capabilities shown on the device properties page and the `GetDevice` call will continue to reflect the regular, lower limit on the height. When an AHS program uses atom arrangements that go beyond the regular capabilities, the filling error is expected to increase. You will find an elevated number of unexpected 0s in the `pre_sequence` part of the task result, in turn, lowering the chance to get a perfectly initialized arrangement. This effect is strongest in rows with many atoms.



Examples:

1. **Bigger 1d and quasi-1d arrangements**

   Atom chains and ladder-like arrangements can be extended to higher atom numbers. By orienting the long direction parallel to y allows for programming longer instances of these models.

2. **More room for multiplexing the execution of tasks with small geometries**

   The example notebook [Parallel quantum tasks on Aquila]() shows how to make the most out of the available area: by placing multiplexed copies of the geometry in question in one atom arrangement. With the more available area, more copies can be placed.

Access to tight geometries on QuEra Aquila

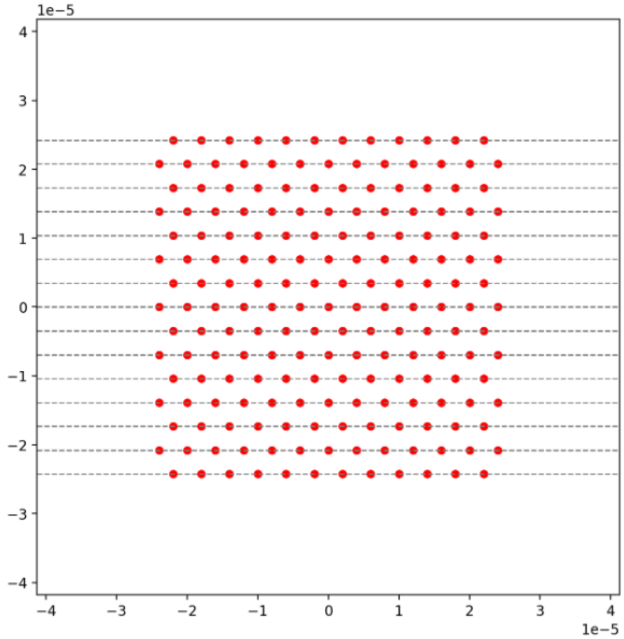
The tight geometries feature allows you to specify geometries with shorter spacing between neighboring rows. In an AHS program, atoms are arranged in rows, separated by a minimal vertical spacing. The y coordinate of any two atom sites must be either zero (same row), or differ by more than the minimal row spacing (different row). With the tight geometries capability, the minimal row spacing is reduced, enabling the creation of tighter atom arrangements. While this extension does not change the minimal Euclidean distance requirement between atoms, it allows the creation of lattices where distant atoms occupy neighboring rows closer to each other, a notable example is the triangle lattice.

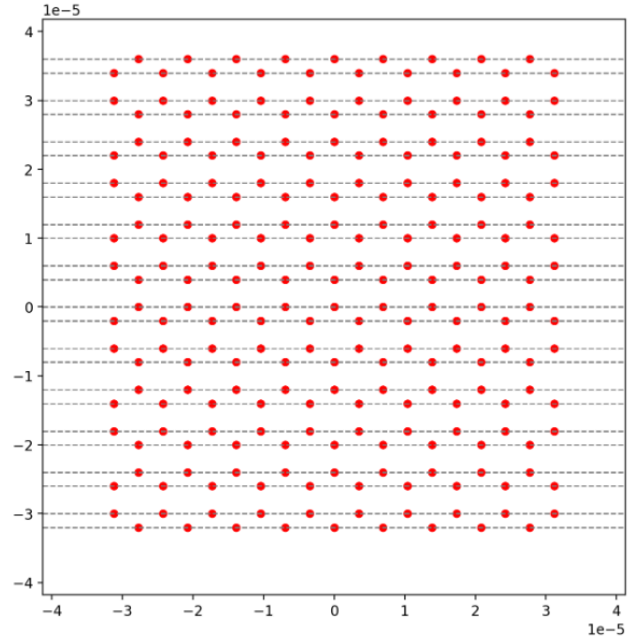
Constraints:

The minimal row spacing for tight geometries is 0.000002 m (2 um).

Limitations:

When this experimental capability is enabled for your account, the capabilities shown on the device properties page and the `GetDevice` call will continue to reflect the regular, lower limit on the height. When an AHS program uses atom arrangements that go beyond the regular capabilities, the filling error is expected to increase. Customers will find an elevated number of unexpected 0s in the `pre_sequence` part of the task result, in turn, lowering the chance to get a perfectly initialized arrangement. This effect is strongest in rows with many atoms.





Examples:

1. **Non-rectangular lattices with small lattice constants**

   Tighter row spacing allows the creation of lattices where the closest neighbor to some atoms are in the diagonal direction. Notable examples are triangular, hexagonal, and Kagome lattices and some quasi-crystals.

2. **Tunable family of lattices**

   In AHS programs, interactions are tuned by adjusting the distance between pairs of atoms. Tighter row spacing allow tuning the interactions of different atom pairs relative to each other with more freedom, since the angles and distances that define the atom structure are less limited by the minimal row spacing constraint. A notable example is the family of Shastry-Sutherland lattices with different bond lengths.

Dynamic circuits on IQM devices

Dynamic circuits on IQM devices enable mid-circuit measurements (MCM) and feed-forward operations. These features allow quantum researchers and developers to implement advanced quantum algorithms with conditional logic and qubit reuse capabilities. This experimental feature helps explore quantum algorithms with improved resource efficiency and study quantum error mitigation and error correction schemes.

Key instructions:

- `measure_ff`: Implements measurement for feed-forward control, measuring a qubit and storing the result with a feedback key.
- `cc_prx`: Implements a classically-controlled rotation that applies only when the result associated with the given feedback key measures a $|1\rangle$ state.

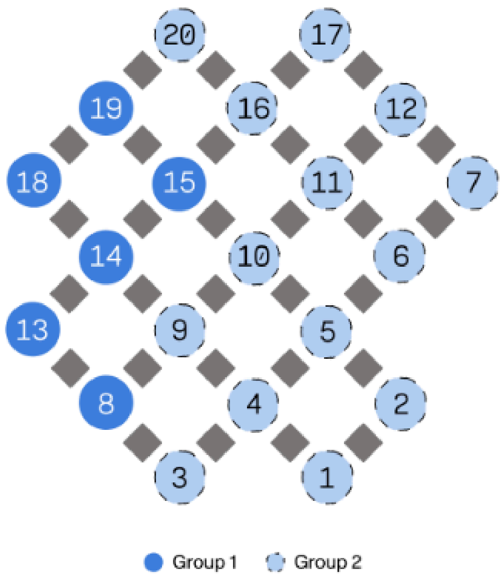
Amazon Braket supports dynamic circuits through OpenQASM, the Amazon Braket SDK, and the Amazon Braket Qiskit Provider.

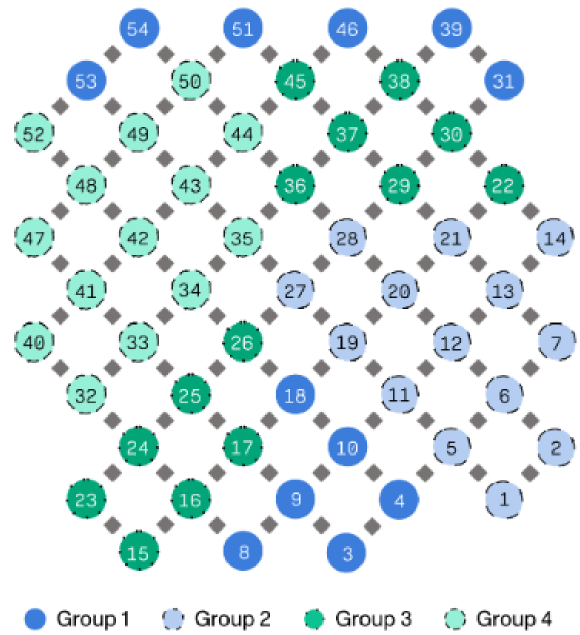
Constraints:

1. Feedback keys in the `measure_ff` instructions must be unique.
2. A `cc_prx` must happen after `measure_ff` with the same feedback key.
3. In a single circuit, the feed-forward on a qubit can only be controlled by one qubit, either by itself or by another qubit. In different circuits, you can have different pairs of control.
    a. For example, if qubit 1 is controlled by qubit 2, it cannot be controlled by qubit 3 in the same circuit. There is no constraint on how many times the control is applied between qubit 1 and qubit 2. Qubit 2 can be controlled by qubit 3 (or qubit 1), unless an active reset was performed on qubit 2.
4. Control can only be applied to qubits within the same group. The qubit groups for the IQM Garnet and Emerald devices are in the following images.
5. Programs with these capabilities must be submitted as verbatim programs. To learn more about verbatim programs, see Verbatim compilation with OpenQASM 3.0.

Limitations:

MCM can only be use for feed-forward control in a program. The MCM outcomes (0 or 1) are not returned as part of a task result.





These images display the qubit groupings for both IQM devices. The Garnet 20-qubit device contains 2 groups of qubits, while the Emerald 54-qubit device contains 4 groups of qubits.

Examples:

1. **Qubit reuse through active reset**

   MCM with conditional reset operations enable qubit reuse within a single circuit execution. This reduces circuit depth requirements and improves quantum device resource utilization.

2. **Active bit flip protection**

   Dynamic circuits detect bit flip errors and apply corrective operations based on measurement outcomes. This implementation serves as a quantum error detection experiment.

3. **Teleportation experiments**

   State teleportation transfers qubit states using local quantum operations and classical information from MCMs. Gate teleportation implements gates between qubits without direct quantum operations. These experiments demonstrate foundational subroutines in three key areas: quantum error correction, measurement-based quantum computing, and quantum communication.

4. **Open quantum systems simulation**

Dynamic circuits model noise in quantum systems through data qubit and environment entanglement, and environmental measurements. This approach uses specific qubits to represent data and environment elements. A Noise channel can be designed by the gates and measurements applied on the environment.

For more information on using dynamic circuits, see additional examples in the Amazon Braket notebook repository.

Pulse control on Amazon Braket

Pulses are the analog signals that control the qubits in a quantum computer. With certain devices on Amazon Braket, you can access the pulse control feature to submit circuits using pulses. You can access pulse control through the Braket SDK, using OpenQASM 3.0, or directly through the Braket APIs. First, introduce some key concepts for pulse control in Braket.

In this section:

- Frames
- Ports
- Waveforms
- Roles of frames and ports
- Working with Hello Pulse
- Accessing native gates using pulses

Frames

A frame is a software abstraction that acts as both a clock within the quantum program and a phase. The clock time is incremented on each usage and a stateful carrier signal that is defined by a frequency. When transmitting signals to the qubit, a frame determines the qubit's carrier frequency, phase offset, and the time at which the waveform envelope is emitted. In Braket Pulse, constructing frames depends on the device, frequency, and phase. Depending on the device, you can either choose a predefined frame or instantiate new frames by providing a port.

```
from braket.aws import AwsDevice
from braket.pulse import Frame, Port
```

```
# Predefined frame from a device
device = AwsDevice("arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3")
drive_frame = device.frames["Transmon_5_charge_tx"]

# Create a custom frame
readout_frame = Frame(frame_id="r0_measure", port=Port("channel_0", dt=1e-9),
 frequency=5e9, phase=0)
```

Ports

A port is a software abstraction representing any input/output hardware component controlling qubits. It helps hardware vendors provide an interface with which users can interact to manipulate and observe qubits. Ports are characterized by a single string that represents the name of the connector. This string also exposes a minimum time increment that specifies how finely we can define the waveforms.

```
from braket.pulse import Port
Port0 = Port("channel_0", dt=1e-9)
```

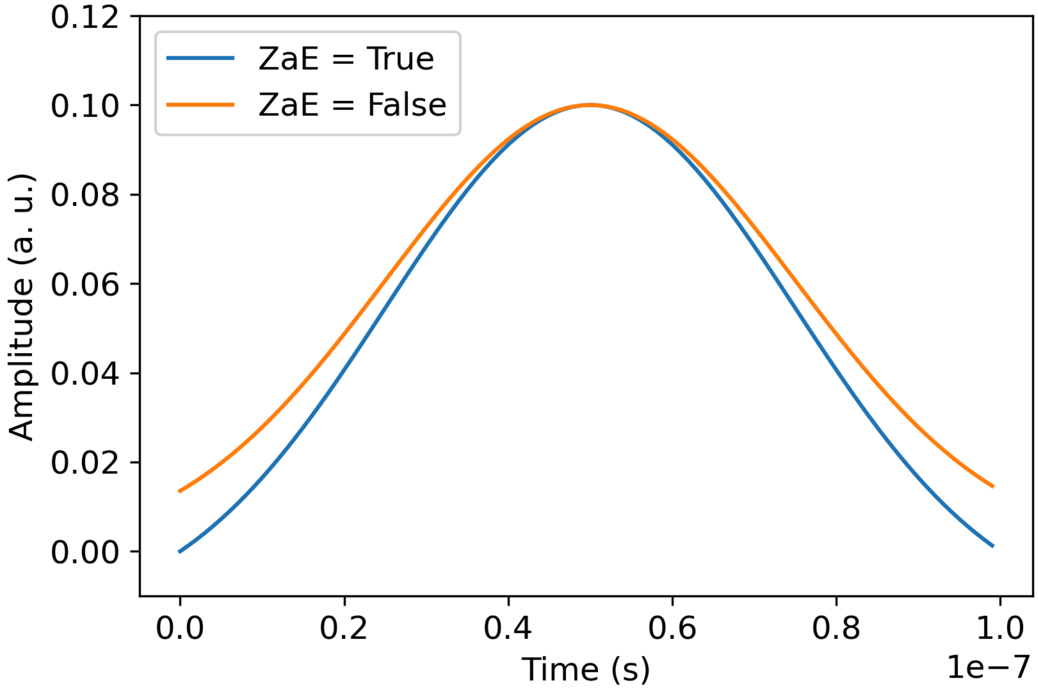
Waveforms

A waveform is a time-dependent envelope that we can use to emit signals on an output port or capture signals through an input port. You can specify your waveforms directly either through a list of complex numbers or by using a waveform template to generate a list from the hardware provider.

```
from braket.pulse import ArbitraryWaveform, ConstantWaveform
cst_wfm = ConstantWaveform(length=1e-7, iq=0.1)
arb_wf = ArbitraryWaveform(amplitudes=np.linspace(0, 100))
```

Braket Pulse provides a standard library of waveforms, including a constant waveform, a Gaussian waveform, and a Derivative Removal by Adiabatic Gate (DRAG) waveform. You can retrieve the waveform data through the `sample` function to draw the shape of the waveform as shown in the following example.

```
gaussian_waveform = GaussianWaveform(1e-7, 25e-9, 0.1)
x = np.arange(0, gaussian_waveform.length, drive_frame.port.dt)
plt.plot(x, gaussian_waveform.sample(drive_frame.port.dt))
```



The preceding image depicts the Gaussian waveforms created from `GaussianWaveform`. We chose a pulse length of 100 ns, a width of 25 ns, and an amplitude of 0.1 (arbitrary units). The waveforms are centered in the pulse window. `GaussianWaveform` accepts a boolean argument `zero_at_edges` (ZaE in the legend). When set to `True`, this argument offsets the Gaussian waveform such that the points at t=0 and t=`length` are at zero and rescales its amplitude such that the maximum value corresponds to the `amplitude` argument.

Roles of frames and ports

This section describes the predefined frames and ports available for each device. We will also briefly discuss the mechanisms involved when pulses are played on certain frames.

Rigetti frames

Rigetti devices support predefined frames that have their frequency and phase calibrated to be on resonance with the associated qubit. The naming convention is `q{i}[_q{j}]_{role}_frame` where `{i}` refers to the first qubit number, `{j}` refers to the second qubit number in case the frame serves to activate a two-qubit interaction, and `{role}` refers to the role of the frame. The roles are as follows:

- `rf` is the frame to drive the 0-1 transition of the qubit. Pulses are transmitted as microwave transient signals of frequency and phase previously provided through the `set` and `shift` functions. The time-dependent amplitude of the signal is given by the waveform played on the frame. The frame plugs a single-qubit, off-diagonal interaction. For more information, see Krantz et al. and Rahamim et al..
- `rf_f12` is similar to `rf` and its parameters target the 1-2 transition.
- `ro_rx` is used to achieve dispersive readout of the qubit through a coupled coplanar waveguide. The frequency, phase, and full set of parameters for the readout waveform are precalibrated. It is used through the `capture_v0`, which does not require any argument besides the frame identifier.
- `ro_tx` is for transmitting signals from the resonator. It is currently unused.
- `cz` is a frame calibrated to enable the two-qubit `cz` gate. As with all the frames associated with an `ff` port, it turns on an entangling interaction through the flux line by modulating the tunable qubit of the pair on resonance with its neighbor. For more information about the entangling mechanism, see Reagor et al., Caldwell et al., and Didier et al..
- `cphase` is a frame calibrated to enable the two-qubit `cphaseshift` gate and is linked to an `ff` port. For more information about the entangling mechanism, see the description for the `cz` frame.
- `xy` is a frame calibrated to enable the two-qubit XY(θ) gates and is linked to an `ff` port. For more information about the entangling mechanism and how to achieve XY gates, see the description for the `cz` frame and Abrams et al..

As frames based on the `ff` port shift the frequency of the tunable qubit, all the other driving frames related to the qubit will be dephased by an amount that is related to the amplitude and the duration of the frequency shift. Consequently, you must compensate for this effect by adding a corresponding phase shift to the frames of the neighboring qubits.

Ports

The Rigetti devices provide a list of ports that you can inspect through the device capabilities. Port names follow the convention `q{i}_{type}` where `{i}` refers to the qubit number and `{type}` refers to the type of the port. Note that not all of the qubits have a complete set of ports. The types of ports are as follows:

- `rf` represents the main interface to drive the single-qubit transition. It is associated with the `rf` and `rf_f12` frames. It is capacitively coupled to the qubit, allowing microwave driving in the gigahertz range.
- `ro_tx` serves to transmit signals to the readout resonator capacitively coupled to the qubit. Readout signal delivery is multiplexed eight-fold by octagon.
- `ro_rx` serves to receive signals from the readout resonator coupled to the qubit.
- `ff` represents the fast-flux line inductively coupled to the qubit. We can use this to tune the frequency of the transmon. Only qubits designed to be highly tunable have an `ff` port. This port serves to activate qubit-qubit interaction as there is a static capacitive coupling between each pair of neighboring transmons.

For more information about the architecture, see Valery et al..

Working with Hello Pulse

In this section, you will learn how to characterize and construct a single qubit gate directly using pulse on a Rigetti device. Applying an electromagnetic field to a qubit leads to Rabi oscillation, switching qubits between its 0 state and 1 state. With calibrated length and phase of the pulse, the Rabi oscillation can calculate a single qubit gates. Here, we will determine the optimal pulse length to measure a pi/2 pulse, an elementary block used to build more complex pulse sequences.

First, to build a pulse sequence, import the `PulseSequence` class.

```
from braket.aws import AwsDevice
from braket.circuits import FreeParameter
from braket.devices import Devices
from braket.pulse import PulseSequence, GaussianWaveform

import numpy as np
```

Next, instantiate a new Braket device using the Amazon Resource Name (ARN) of the QPU. The following code block uses Rigetti Ankaa-3.

```
device = AwsDevice(Devices.Rigetti.Ankaa3)
```

The following pulse sequence includes two components: Playing a waveform and measuring a qubit. Pulse sequence can usually be applied to frames. With some exceptions such as barrier and delay, which can be applied to qubits. Before constructing the pulse sequence you must retrieve the available frames. The drive frame is used for applying the pulse for Rabi oscillation, and the readout frame is for measuring the qubit state. This example, uses the frames of qubit 25. For more information about frames, see Roles of frames and ports.

```
drive_frame = device.frames["Transmon_25_charge_tx"]
readout_frame = device.frames["Transmon_25_readout_rx"]
```

Now, create the waveform that will play in the drive frame. The goal is to characterize the behavior of the qubits for different pulse lengths. You will play a waveform with different lengths each time. Instead of instantiating a new waveform each time, use the Braket-supported `FreeParameter` in pulse sequence. You are able to create the waveform and the pulse sequence once with a free parameters, and then run the same pulse sequence with different input values.

```
waveform = GaussianWaveform(FreeParameter("length"), FreeParameter("length") * 0.25,
 0.2, False)
```

Finally, put them together as a pulse sequence. In the pulse sequence, `play` plays the specified waveform on the drive frame, and the `capture_v0` measures the state from the readout frame.

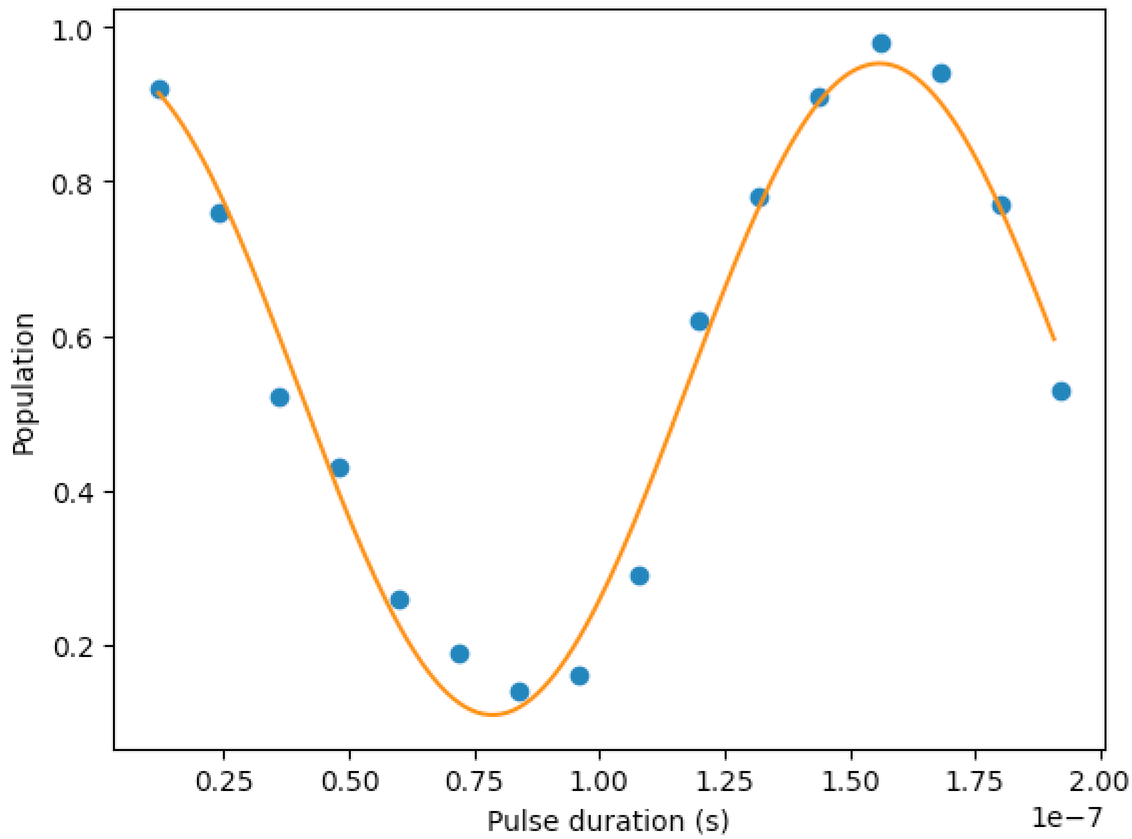
```
pulse_sequence = (
    PulseSequence()
    .play(drive_frame, waveform)
    .capture_v0(readout_frame)
)
```

Scan across a range of pulse length and submit them to the QPU.

```
start_length = 12e-9
end_length = 2e-7
lengths = np.arange(start_length, end_length, 12e-9)
N_shots = 100

tasks = [
    device.run(pulse_sequence(length=length), shots=N_shots)
    for length in lengths
]

probability_of_zero = [
    task.result().measurement_counts['0']/N_shots
    for task in tasks
```

```
]
```

The statistics of the qubit measurement exhibits the oscillatory dynamics of the qubit that oscillates between the 0 state and the 1 state. From the measurement data, you can extract the Rabi frequency and fine tune the length of the pulse to implement a particular 1-qubit gate. For example, from the data in figure below, the periodicity is about 154 ns. So a pi/2 rotation gate would correspond to the pulse sequence with length=38.5ns.



Hello Pulse using OpenPulse

OpenPulse is a language for specifying pulse-level control of a general quantum device and is part of the OpenQASM 3.0 specification. Amazon Braket supports OpenPulse for directly programming pulses using the OpenQASM 3.0 representation.

Braket uses OpenPulse as the underlying intermediate representation for expressing pulses in native instructions. OpenPulse supports the addition of instruction calibrations in the form of `defcal` (short for “define calibration”) declarations. With these declarations, you can specify an implementation of a gate instruction within a lower-level control grammar.

You can view the OpenPulse program of a Braket `PulseSequence` using the following command.

```
print(pulse_sequence.to_ir())
```

You can also construct an OpenPulse program directly.

```
from braket.ir.openqasm import Program

openpulse_script = """
OPENQASM 3.0;
cal {
    bit[1] psb;
    waveform my_waveform = gaussian(12.0ns, 3.0ns, 0.2, false);
    play(Transmon_25_charge_tx, my_waveform);
    psb[0] = capture_v0(Transmon_25_readout_rx);
}
"""
```

Create a `Program` object with your script. Then, submit the program to a QPU.

```
from braket.aws import AwsDevice
from braket.devices import Devices
from braket.ir.openqasm import Program

program = Program(source=openpulse_script)

device = AwsDevice(Devices.Rigetti.Ankaa3)
task = device.run(program, shots=100)
```

Accessing native gates using pulses

Researchers often need to know exactly how the *native* gates supported by a particular QPU are implemented as pulses. Pulse sequences are carefully calibrated by hardware providers, but accessing them provides researchers the opportunity to design better gates or explore protocols for error mitigation such as zero noise extrapolation by stretching the pulses of specific gates.

Amazon Braket supports programmatic access to native gates from Rigetti.

```
import math
from braket.aws import AwsDevice
from braket.circuits import Circuit, GateCalibrations, QubitSet
from braket.circuits.gates import Rx
```

```
device = AwsDevice("arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3")

calibrations = device.gate_calibrations
print(f"Downloaded {len(calibrations)} calibrations.")
```

Note

Hardware providers periodically calibrate the QPU, often more than once a day. The Braket SDK enables you to obtain the latest gate calibrations.

```
device.refresh_gate_calibrations()
```

To retrieve a given native gate, such as the RX or XY gate, you need to pass the `Gate` object and the qubits of interest. For example, you can inspect the pulse implementation of the RX(π/2) applied on qubit 0.

```
rx_pi_2_q0 = (Rx(math.pi/2), QubitSet(0))

pulse_sequence_rx_pi_2_q0 = calibrations.pulse_sequences[rx_pi_2_q0]
```

You can create a filtered set of calibrations using the `filter` function. You pass a list of gates or a list of `QubitSet`. The following code creates two sets that contain all of the calibrations for RX(π/2) and for qubit 0.

```
rx_calibrations = calibrations.filter(gates=[Rx(math.pi/2)])
q0_calibrations = calibrations.filter(qubits=QubitSet([0])
```

Now you can provide or modify the action of native gates by attaching a custom calibration set. For example, consider the following circuit.

```
bell_circuit = (
Circuit()
.rx(0,math.pi/2)
.rx(1,math.pi/2)
.iswap(0,1)
.rx(1,-math.pi/2)
)
```

You can run it with a custom gate calibration for the `rx` gate on `qubit 0` by passing a dictionary of `PulseSequence` objects to the `gate_definitions` keyword argument. You can construct a dictionary from the attribute `pulse_sequences` of the `GateCalibrations` object. All gates not specified are replaced with the quantum hardware provider's pulse calibration.

```
nb_shots = 50
custom_calibration = GateCalibrations({rx_pi_2_q0: pulse_sequence_rx_pi_2_q0})
task=device.run(bell_circuit, gate_definitions=custom_calibration.pulse_sequences,
 shots=nb_shots)
```

Analog Hamiltonian Simulation

Analog Hamiltonian Simulation (AHS) is an emerging paradigm in quantum computing that differs significantly from the traditional quantum circuit model. Instead of a sequence of gates, where each circuit acts only on a couple of qubits at a time. An AHS program is defined by the time-dependent and space-dependent parameters of the Hamiltonian in question. The Hamiltonian of a system encodes its energy levels and the effects of external forces, which together govern the time evolution of its states. For an N-qubit systems, the Hamiltonian can be represented by a 2^NX2^N square matrix of complex numbers.

Quantum devices capable of performing AHS are designed to closely approximate the time evolution of a quantum system under a custom Hamiltonian by carefully tuning their internal control parameters. Such as, adjusting the amplitude and detuning parameters of a coherent driving field. The AHS paradigm is well-suited for simulating the static and dynamic properties of quantum systems with many interacting particles, such as in condensed matter physics or quantum chemistry. Purpose-built quantum processing units (QPUs), like the Aquila device from QuEra, have been developed to use the power of AHS and tackle problems beyond the reach of conventional digital quantum computing approaches in innovative ways.

In this section:

- Hello AHS: Run your first Analog Hamiltonian Simulation
- Submit an analog program using QuEra Aquila

Hello AHS: Run your first Analog Hamiltonian Simulation

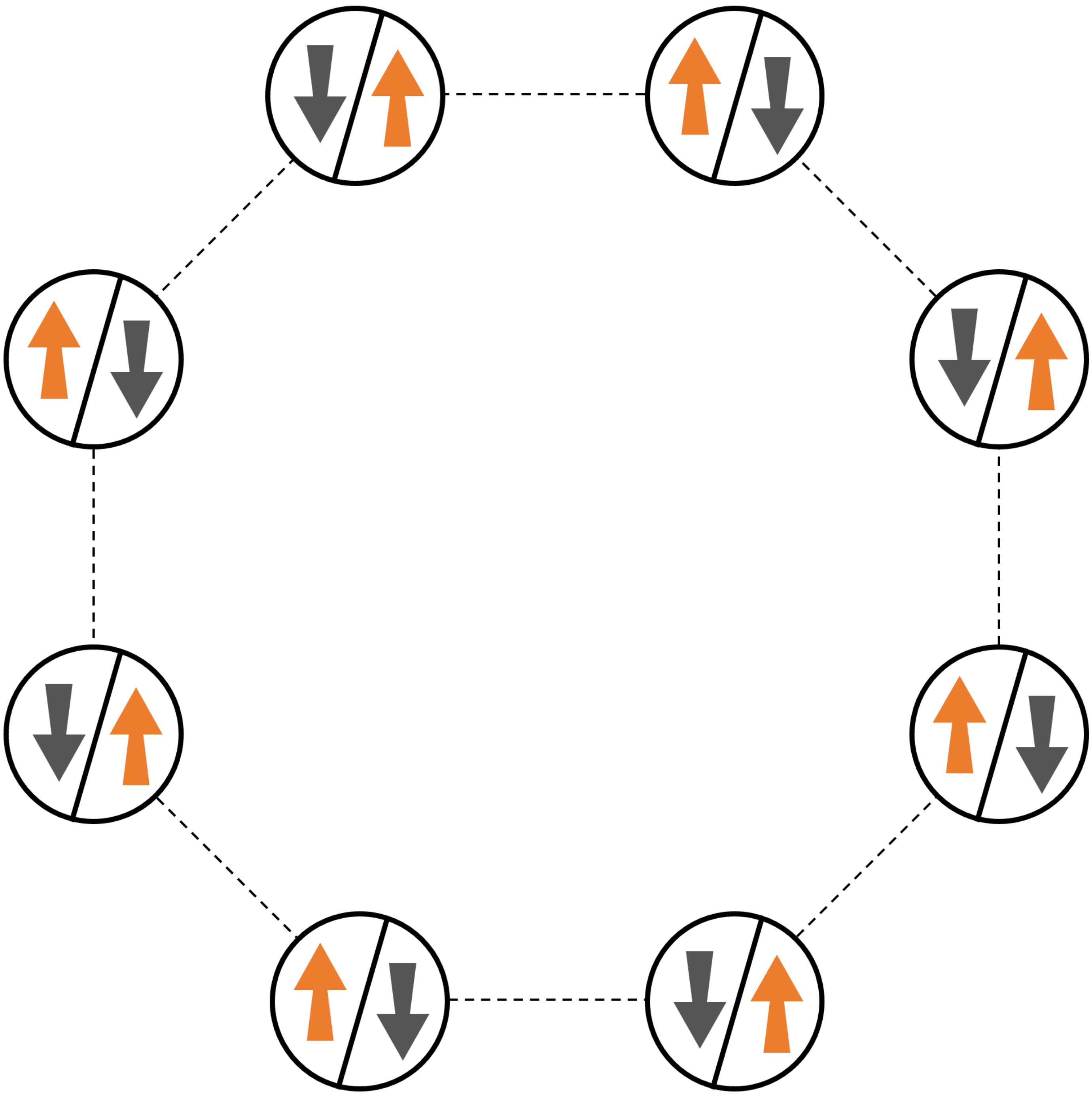
This section provides information on running your first Analog Hamiltonian Simulation.

In this section:

- Interacting spin chain
- Arrangement
- Interaction
- Driving field
- AHS program
- Running on local simulator
- Analyzing simulator results
- Running on QuEra's Aquila QPU
- Analyzing QPU results
- Next steps

Interacting spin chain

For a canonical example of a system of many interacting particles, let us consider a ring of eight spins (each of which can be in “up” |↑# and “down” |↓# states). Albeit small, this model system already exhibits a handful of interesting phenomena of naturally occurring magnetic materials. In this example, we will show how to prepare a so-called anti-ferromagnetic order, where consecutive spins point in opposite directions.



Arrangement

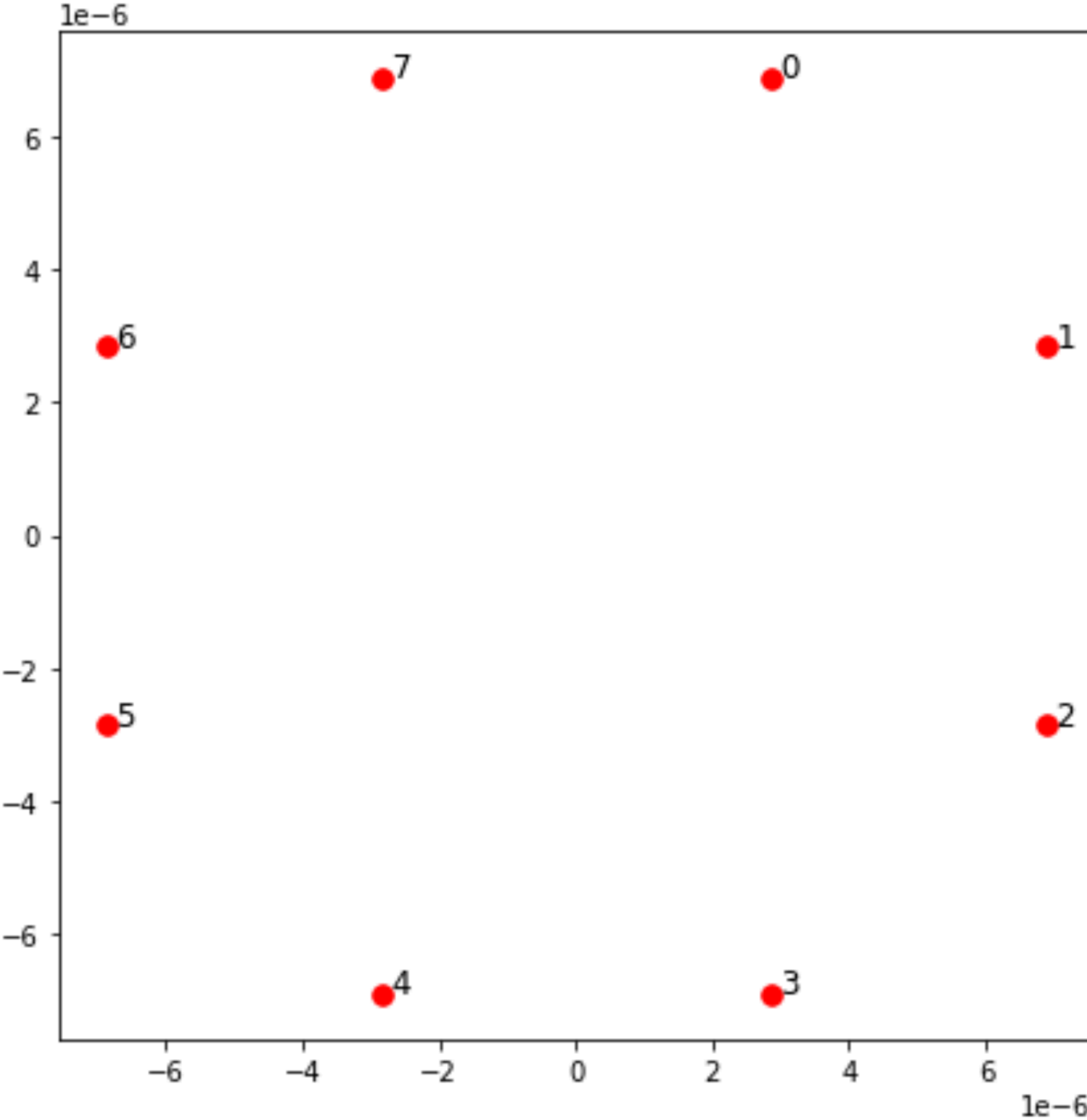
We will use one neutral atom to stand for each spin, and the “up” and “down” spin states will be encoded in excited Rydberg state and ground state of the atoms, respectively. First, we create the 2-d arrangement. We can program the above ring of spins with the following code.

Prerequisites: You need to pip install the Braket SDK. (If you are using a Braket hosted notebook instance, this SDK comes pre-installed with the notebooks.) To reproduce the plots, you also need to separately install matplotlib with the shell command `pip install matplotlib`.

```
import numpy as np
   import matplotlib.pyplot as plt  # Required for plotting

   from braket.ahs.atom_arrangement import AtomArrangement

   a = 5.7e-6  # Nearest-neighbor separation (in meters)

   register = AtomArrangement()
   register.add(np.array([0.5, 0.5 + 1/np.sqrt(2)]) * a)
   register.add(np.array([0.5 + 1/np.sqrt(2), 0.5]) * a)
   register.add(np.array([0.5 + 1/np.sqrt(2), - 0.5]) * a)
   register.add(np.array([0.5, - 0.5 - 1/np.sqrt(2)]) * a)
   register.add(np.array([-0.5, - 0.5 - 1/np.sqrt(2)]) * a)
   register.add(np.array([-0.5 - 1/np.sqrt(2), - 0.5]) * a)
   register.add(np.array([-0.5 - 1/np.sqrt(2), 0.5]) * a)
   register.add(np.array([-0.5, 0.5 + 1/np.sqrt(2)]) * a)
```

which we can also plot with

```
fig, ax = plt.subplots(1, 1, figsize=(7,7))
   xs, ys = [register.coordinate_list(dim) for dim in (0, 1)]
   ax.plot(xs, ys, 'r.', ms=15)
   for idx, (x, y) in enumerate(zip(xs, ys)):
       ax.text(x, y, f" {idx}", fontsize=12)
   plt.show()  # This will show the plot below in an ipython or jupyter session
```



Interaction

To prepare the anti-ferromagnetic phase, we need to induce interactions between neighboring spins. We use the [van der Waals interaction](https://en.wikipedia.org/wiki/Van_der_Waals_force) for this, which is natively implemented by neutral atom devices (such as the Aquila device from QuEra). Using the spin-representation, the Hamiltonian term for this interaction can be expressed as a sum over all spin pairs (j,k).

$$H_{\text{interaction}} = \sum_{j=1}^{N-1} \sum_{k=j+1}^{N} V_{j,k} n_j n_k$$

Here, nj=|↑$_j$⟩⟨#$_j$| is an operator that takes the value of 1 only if spin j is in the “up” state, and 0 otherwise. The strength is $V_{j,k}=C_6/(d_{j,k})^6$, where C_6 is the fixed coefficient, and $d_{j,k}$ is the Euclidean distance between spins j and k. The immediate effect of this interaction term is that any state where both spin j and spin k are “up” have elevated energy (by the amount $V_{j,k}$). By carefully designing the rest of the AHS program, this interaction will prevent neighboring spins from both being in the “up” state, an effect commonly known as "Rydberg blockade."

Driving field

At the beginning of the AHS program, all spins (by default) start in their “down” state, they are in a so-called ferromagnetic phase. Keeping an eye on our goal to prepare the anti-ferromagnetic phase, we specify a time-dependent coherent driving field that smoothly transitions the spins from this state to a many-body state where the “up” states are preferred. The corresponding Hamiltonian can be written as

$$H_{\text{drive}}(t) = \sum_{k=1}^{N} \frac{1}{2}\Omega(t) \left[e^{i\phi(t)} S_{-,k} + e^{-i\phi(t)} S_{+,k}\right] - \sum_{k=1}^{N} \Delta(t) n_k$$

where Ω(t),ϕ(t),Δ(t) are the time-dependent, global amplitude (aka [Rabi frequency](https://en.wikipedia.org/wiki/Rabi_frequency)), phase, and detuning of the driving field affecting all spins uniformly. Here $S_{-,k}$=|↓$_k$⟩⟨#$_k$|and $S_{+,k}=(S_{-,k})^\dagger$=|↑$_k$⟩⟨#$_k$ | are the lowering and raising operators of spin k, respectively, and n_k=|↑$_k$⟩⟨#$_k$| is the same operator as before. The Ω part of the driving field coherently couples the “down” and the “up” states of all spins simultaneously, while the Δ part controls the energy reward for “up” states.

To program a smooth transition from the ferromagnetic phase to the anti-ferromagnetic phase, we specify the driving field with the following code.

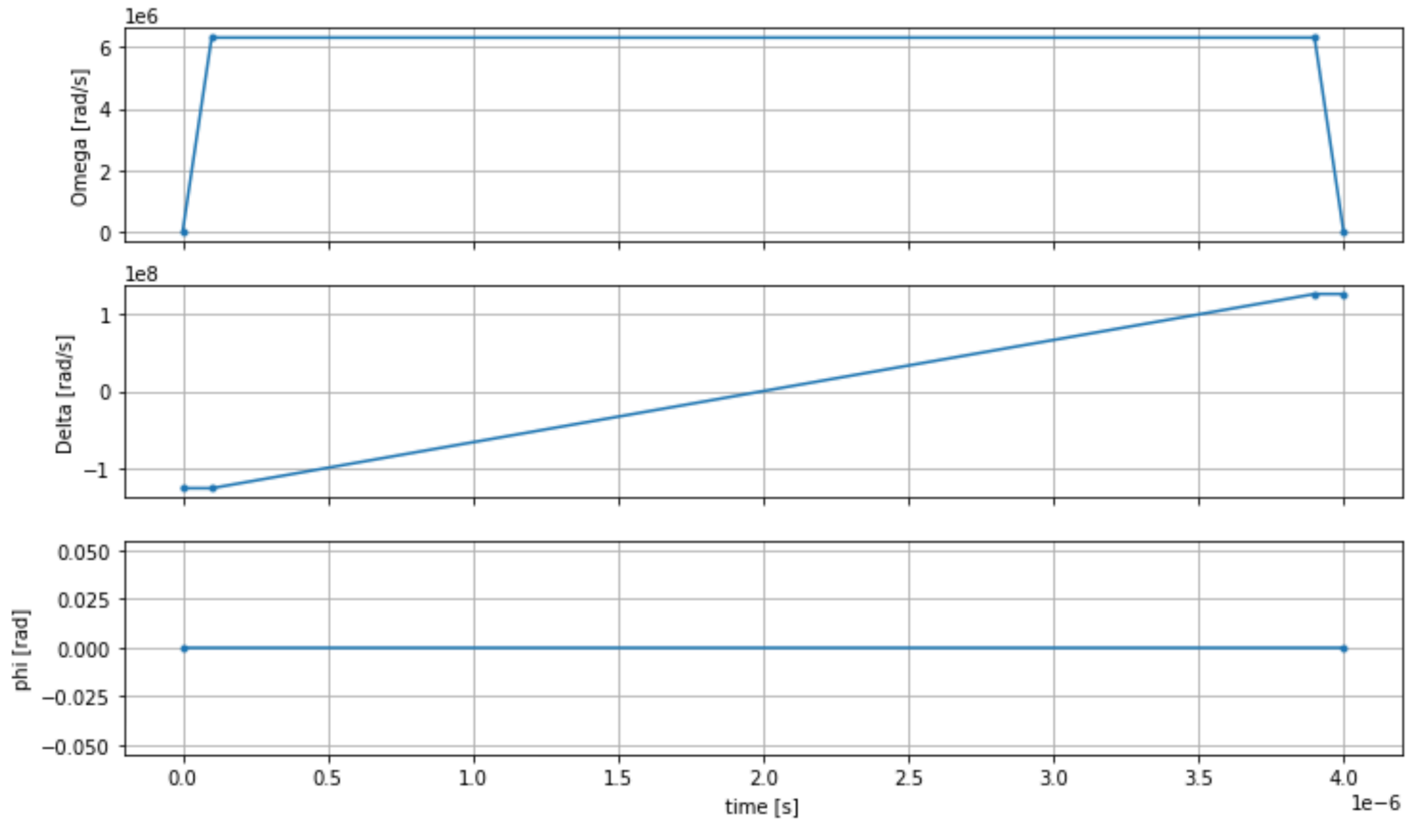
```
from braket.timings.time_series import TimeSeries
   from braket.ahs.driving_field import DrivingField

   # Smooth transition from "down" to "up" state
   time_max = 4e-6  # seconds
   time_ramp = 1e-7  # seconds
   omega_max = 6300000.0  # rad / sec
   delta_start = -5 * omega_max
```

```
    delta_end = 5 * omega_max

    omega = TimeSeries()
    omega.put(0.0, 0.0)
    omega.put(time_ramp, omega_max)
    omega.put(time_max - time_ramp, omega_max)
    omega.put(time_max, 0.0)

    delta = TimeSeries()
    delta.put(0.0, delta_start)
    delta.put(time_ramp, delta_start)
    delta.put(time_max - time_ramp, delta_end)
    delta.put(time_max, delta_end)

    phi = TimeSeries().put(0.0, 0.0).put(time_max, 0.0)

    drive = DrivingField(
        amplitude=omega,
        phase=phi,
        detuning=delta
    )
```

We can visualize the time series of the driving field with the following script.

```
fig, axes = plt.subplots(3, 1, figsize=(12, 7), sharex=True)

    ax = axes[0]
    time_series = drive.amplitude.time_series
    ax.plot(time_series.times(), time_series.values(), '.-');
    ax.grid()
    ax.set_ylabel('Omega [rad/s]')

    ax = axes[1]
    time_series = drive.detuning.time_series
    ax.plot(time_series.times(), time_series.values(), '.-');
    ax.grid()
    ax.set_ylabel('Delta [rad/s]')

    ax = axes[2]
    time_series = drive.phase.time_series
    # Note: time series of phase is understood as a piecewise constant function
    ax.step(time_series.times(), time_series.values(), '.-', where='post');
    ax.set_ylabel('phi [rad]')
```

```
    ax.grid()
    ax.set_xlabel('time [s]')

    plt.show()  # This will show the plot below in an ipython or jupyter session
```



AHS program

The register, the driving field, (and the implicit van der Waals interactions) make up the Analog Hamiltonian Simulation program `ahs_program`.

```
from braket.ahs.analog_hamiltonian_simulation import AnalogHamiltonianSimulation

    ahs_program = AnalogHamiltonianSimulation(
        register=register,
        hamiltonian=drive
    )
```

Running on local simulator

Since this example is small (less than 15 spins), before running it on an AHS-compatible QPU, we can run it on the local AHS simulator which comes with the Braket SDK. Since the local simulator is available for free with the Braket SDK, this is best practice to ensure that our code can correctly execute.

Here, we can set the number of shots to a high value (say, 1 million) because the local simulator tracks the time evolution of the quantum state and draws samples from the final state; hence, increasing the number of shots, while increasing the total runtime only marginally.

```
from braket.devices import LocalSimulator
   device = LocalSimulator("braket_ahs")

   result_simulator = device.run(
       ahs_program,
       shots=1_000_000
   ).result()  # Takes about 5 seconds
```

Analyzing simulator results

We can aggregate the shot results with the following function that infers the state of each spin (which may be “d” for “down”, “u” for “up”, or “e” for empty site), and counts how many times each configuration occurred across the shots.

```
from collections import Counter

   def get_counts(result):
       """Aggregate state counts from AHS shot results

       A count of strings (of length = # of spins) are returned, where
       each character denotes the state of a spin (site):
          e: empty site
          u: up state spin
          d: down state spin

       Args:
          result
 (braket.tasks.analog_hamiltonian_simulation_quantum_task_result.AnalogHamiltonianSimulationQua

       Returns
           dict: number of times each state configuration is measured

       """
       state_counts = Counter()
       states = ['e', 'u', 'd']
```

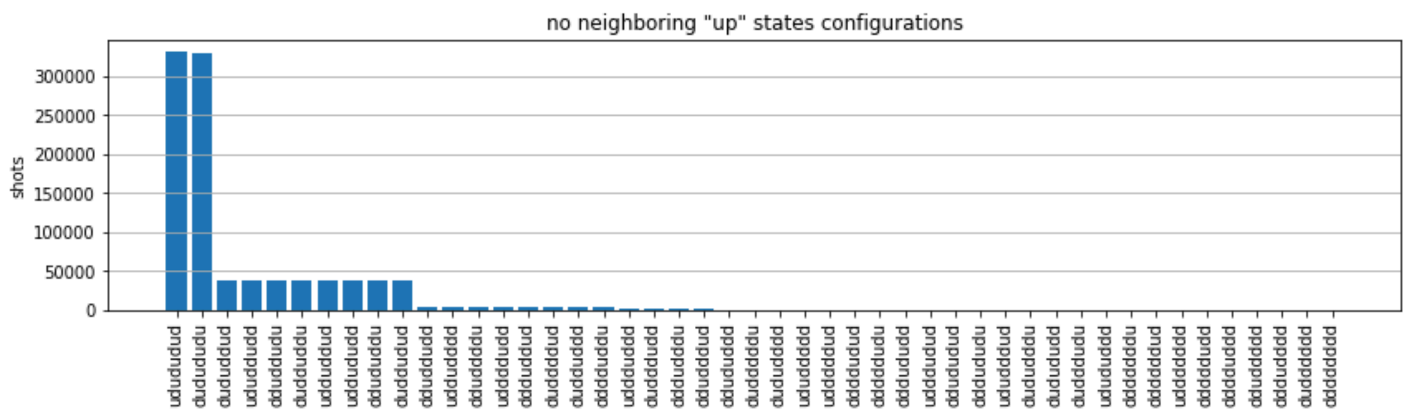
```
        for shot in result.measurements:
            pre = shot.pre_sequence
            post = shot.post_sequence
            state_idx = np.array(pre) * (1 + np.array(post))
            state = "".join(map(lambda s_idx: states[s_idx], state_idx))
            state_counts.update((state,))
        return dict(state_counts)

    counts_simulator = get_counts(result_simulator)  # Takes about 5 seconds
    print(counts_simulator)
```

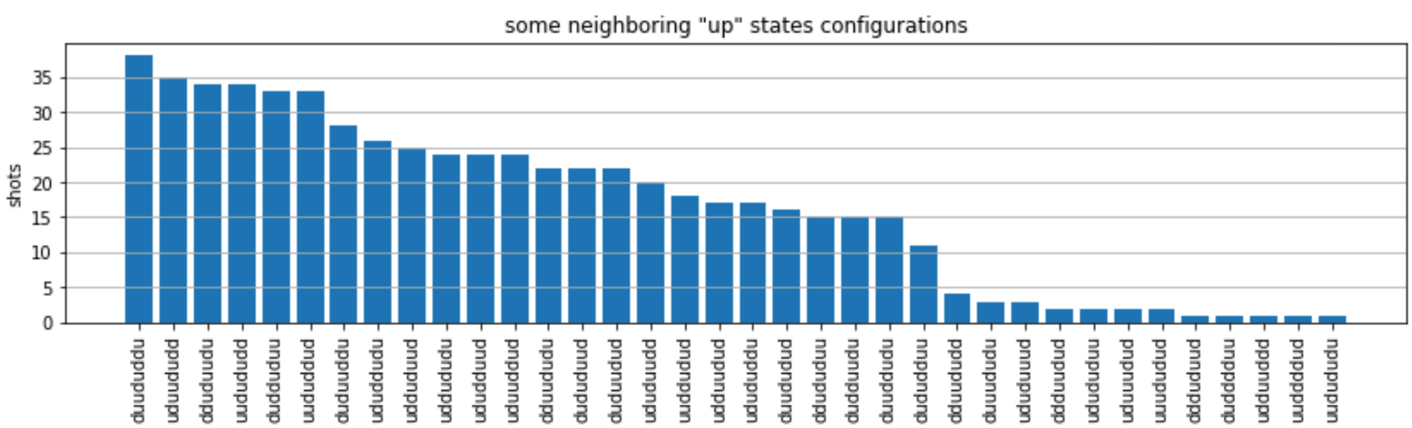
```
{'udududud': 330944, 'dudududu': 329576, 'dududdud': 38033, ...}
```

Here `counts` is a dictionary that counts the number of times each state configuration is observed across the shots. We can also visualize them with the following code.

```
from collections import Counter

    def has_neighboring_up_states(state):
        if 'uu' in state:
            return True
        if state[0] == 'u' and state[-1] == 'u':
            return True
        return False

    def number_of_up_states(state):
        return Counter(state)['u']

    def plot_counts(counts):
        non_blockaded = []
        blockaded = []
        for state, count in counts.items():
            if not has_neighboring_up_states(state):
                collection = non_blockaded
            else:
                collection = blockaded
            collection.append((state, count, number_of_up_states(state)))

        blockaded.sort(key=lambda _: _[1], reverse=True)
        non_blockaded.sort(key=lambda _: _[1], reverse=True)

        for configurations, name in zip((non_blockaded,
                                        blockaded),
```

```
                                        ('no neighboring "up" states',
                                        'some neighboring "up" states')):
        plt.figure(figsize=(14, 3))
        plt.bar(range(len(configurations)), [item[1] for item in configurations])
        plt.xticks(range(len(configurations)))
        plt.gca().set_xticklabels([item[0] for item in configurations], rotation=90)
        plt.ylabel('shots')
        plt.grid(axis='y')
        plt.title(f'{name} configurations')
        plt.show()

plot_counts(counts_simulator)
```





From the plots, we can read the following observations the verify that we successfully prepared the anti-ferromagnetic phase.

1. Generally, non-blockaded states (where no two neighboring spins are in the “up” state) are more common than states where at least one pair of neighboring spins are both in “up” states.
2. Generally, states with more "up" excitations are favored, unless the configuration is blockaded.

3. The most common states are indeed the perfect anti-ferromagnetic states "dudududu" and "udududud".
4. The second most common states are the ones where there is only 3 “up” excitations with consecutive separations of 1, 2, 2. This shows that the van der Waals interaction has an affect (albeit much smaller) on next-nearest neighbors too.

Running on QuEra's Aquila QPU

Prerequisites: Apart from pip installing the Braket SDK, if you are new to Amazon Braket, make sure that you have completed the necessary Get Started steps.

> **Note**
>
> If you are using a Braket hosted notebook instance, the Braket SDK comes pre-installed with the instance.

With all dependencies installed, we can connect to the Aquila QPU.

```
from braket.aws import AwsDevice

   aquila_qpu = AwsDevice("arn:aws:braket:us-east-1::device/qpu/quera/Aquila")
```

To make our AHS program suitable for the QuEra machine, we need to round all values to comply with the levels of precision allowed by the Aquila QPU. (These requirements are governed by the device parameters with “Resolution” in their name. We can see them by executing `aquila_qpu.properties.dict()` in a notebook. For more details of capabilities and requirements of Aquila, see the Introduction to Aquila notebook.) We can do this by calling the `discretize` method.

```
discretized_ahs_program = ahs_program.discretize(aquila_qpu)
```

Now we can run the program (running only 100 shots for now) on the Aquila QPU.

> **Note**
>
> Running this program on the Aquila processor will incur a cost. The Amazon Braket SDK includes a Cost Tracker that enables customers to set cost limits as well as track their costs in near real-time.

```
task = aquila_qpu.run(discretized_ahs_program, shots=100)

   metadata = task.metadata()
   task_arn = metadata['quantumTaskArn']
   task_status = metadata['status']

   print(f"ARN: {task_arn}")
   print(f"status: {task_status}")
```

```
task ARN: arn:aws:braket:us-east-1:123456789012:quantum-task/12345678-90ab-
cdef-1234-567890abcdef
   task status: CREATED
```

Due to the large variance of how long a quantum task may take to run (depending on availability windows and QPU utilization), it is a good idea to note down the quantum task ARN, so we can check its status at a later time with the following code snippet.

```
# Optionally, in a new python session

   from braket.aws import AwsQuantumTask

   SAVED_TASK_ARN = "arn:aws:braket:us-east-1:123456789012:quantum-task/12345678-90ab-
cdef-1234-567890abcdef"

   task = AwsQuantumTask(arn=SAVED_TASK_ARN)
   metadata = task.metadata()
   task_arn = metadata['quantumTaskArn']
   task_status = metadata['status']

   print(f"ARN: {task_arn}")
   print(f"status: {task_status}")
```

```
*[Output]*
```

```
    task ARN: arn:aws:braket:us-east-1:123456789012:quantum-task/12345678-90ab-
cdef-1234-567890abcdef
    task status: COMPLETED
```

Once the status is COMPLETED (which can also be checked from the quantum tasks page of the Amazon Braket console), we can query the results with:

```
result_aquila = task.result()
```

Analyzing QPU results

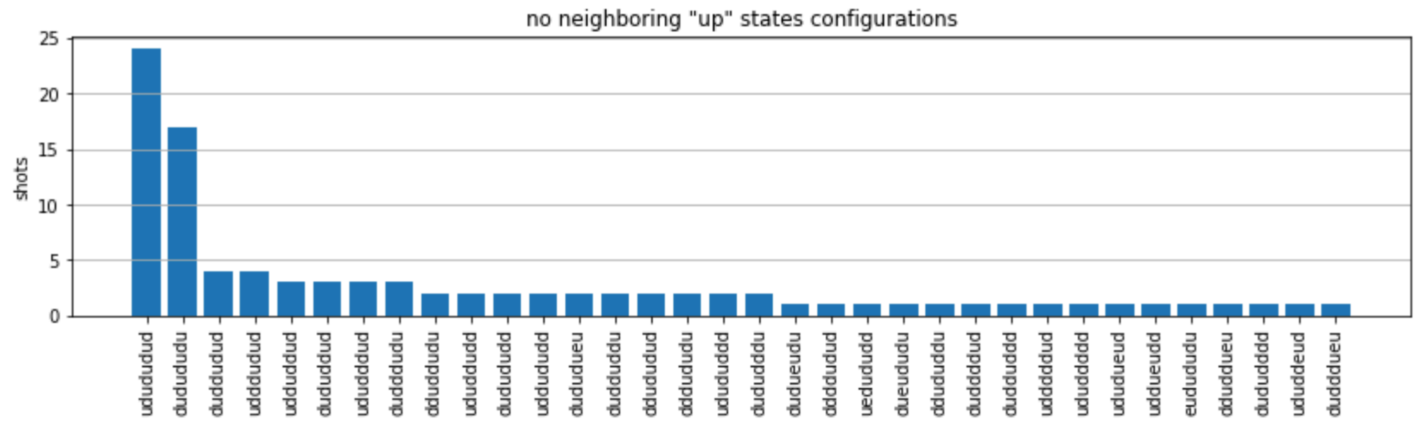
Using the same `get_counts` functions as before, we can compute the counts:

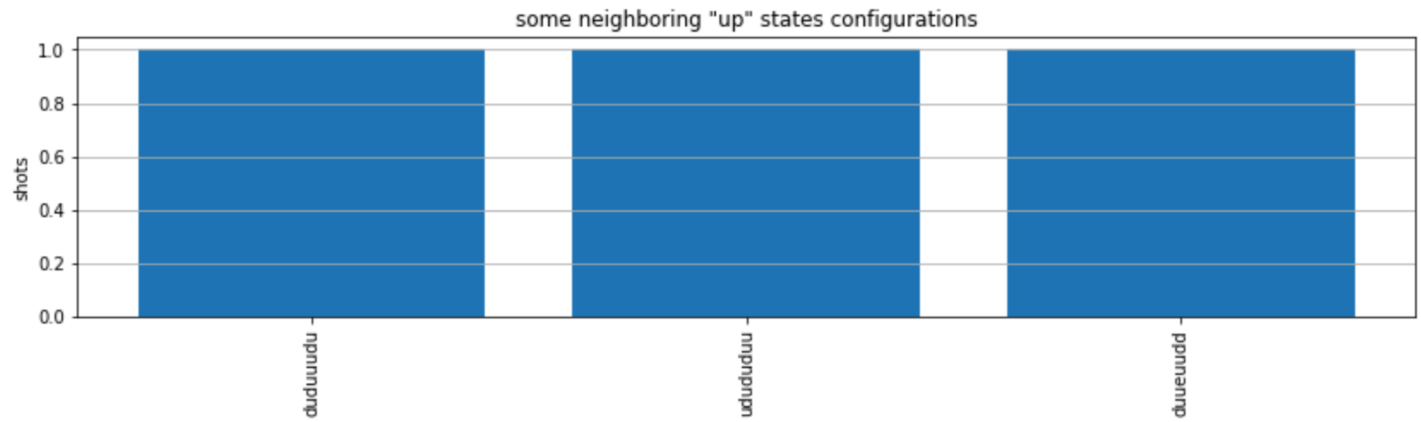
```
counts_aquila = get_counts(result_aquila)
    print(counts_aquila)
```

```
*[Output]*
    {'udududud': 24, 'dudududu': 17, 'dududdud': 3, ...}
```

and plot them with `plot_counts`:

```
plot_counts(counts_aquila)
```





Note that a small fraction of shots have empty sites (marked with “e”). This is due to a 1—2% per atom preparation imperfections of the Aquila QPU. Apart from this, the results match with the simulation within the expected statistical fluctuation due to small number of shots.

Next steps

Congratulations, you have now run your first AHS workload on Amazon Braket using the local AHS simulator and the Aquila QPU.

To learn more about Rydberg physics, Analog Hamiltonian Simulation and the Aquila device, refer to our example notebooks.

Submit an analog program using QuEra Aquila

This page provides a comprehensive documentation about the capabilities of the Aquila machine from QuEra. Details covered here are the following:

1. The parameterized Hamiltonian simulated by Aquila
2. AHS program parameters

3. AHS result content
4. Aquila capabilities parameter

In this section:

- Hamiltonian
- Braket AHS program schema
- Braket AHS task result schema
- QuEra device properties schema

Hamiltonian

The Aquila machine from QuEra simulates the following (time-dependent) Hamiltonian natively:

$$H(t) = \sum_{k=1}^{N} H_{\text{drive},k}(t) + \sum_{k=1}^{N} H_{\text{local detuning},k}(t) + \sum_{k=1}^{N-1} \sum_{l=k+1}^{N} V_{\text{vdw},k,l}$$

> **Note**
>
> Access to local detuning is an Experimental capability and is available by request through Braket Direct.

where

- $H_{\text{drive},k}(t) = ({}^{1}/_{2}\ \Omega(t) e^{i\phi(t)} S_{-,k} + {}^{1}/_{2}\ \Omega(t) e^{-i\phi(t)} S_{+,k}) + (-\Delta_{\text{global}}(t) n_k)$,
  - $\Omega(t)$ is the time-dependent, global driving amplitude (aka Rabi frequency), in units of (rad / s)
  - $\phi(t)$ is the time-dependent, global phase, measured in radians
  - $S_{-,k}$ and $S_{+,k}$ are the spin lowering and raising operators of atom k (in the basis |#⟩=|g⟩, |#⟩=|r⟩, they are $S_- = |g\rangle\langle r|$, $S_+ = (S_-)^\dagger = |r\rangle\langle g|$)
  - $\Delta_{\text{global}}(t)$ is the time-dependent, global detuning
  - n_k is the projection operator on the Rydberg state of atom k (that is, $n = |r\rangle\langle r|$)
- $H_{\text{local detuning},k}(t) = -\Delta_{\text{local}}(t) h_k n_k$
  - $\Delta_{\text{local}}(t)$ is the time-dependent factor of the local frequency shift, in units of (rad / s)
  - h_k is the site-dependent factor, a dimensionless number between 0.0 and 1.0

- $V_{vdw,k,l}=C_6/(d_{k,l})^6 n_k n_l$,
  - C_6 is the van der Waals coefficient, in units of (rad / s) * (m)^6
  - $d_{k,l}$ is the Euclidean distance between atom k and l, measured in meters.

Users have control over the following parameters through the Braket AHS program schema.

- 2-d atom arrangement (x_k and y_k coordinates of each atom k, in units of um), which controls the pairwise atomic distances $d_{k,l}$ with k,l=1,2,…N
- Ω(t), the time-dependent, global Rabi frequency, in units of (rad / s)
- ϕ(t), the time-dependent, global phase, in units of (rad)
- Δ_{global}(t), the time-dependent, global detuning, in units of (rad / s)
- Δ_{local}(t), the time-dependent (global) factor of the magnitude of local detuning, in units of (rad / s)
- h_k, the (static) site-dependent factor of the magnitude of local detuning, a dimensionless number between 0.0 and 1.0

Note

The user cannot control which levels are involved (that is, S_-,S_+, n operators are fixed) nor the strength of the Rydberg-Rydberg interaction coefficient (C_6).

Braket AHS program schema

braket.ir.ahs.program_v1.Program object (example)

Note

If the local detuning feature is not enabled for your account, use `localDetuning=[]` in the following example.

```
Program(
    braketSchemaHeader=BraketSchemaHeader(
        name='braket.ir.ahs.program',
        version='1'
```

```
    ),
    setup=Setup(
        ahs_register=AtomArrangement(
            sites=[
                [Decimal('0'), Decimal('0')],
                [Decimal('0'), Decimal('4e-6')],
                [Decimal('4e-6'), Decimal('0')]
            ],
            filling=[1, 1, 1]
        )
    ),
    hamiltonian=Hamiltonian(
        drivingFields=[
            DrivingField(
                amplitude=PhysicalField(
                    time_series=TimeSeries(
                        values=[Decimal('0'), Decimal('15700000.0'),
 Decimal('15700000.0'), Decimal('0')],
                        times=[Decimal('0'), Decimal('0.000001'), Decimal('0.000002'),
 Decimal('0.000003')]
                    ),
                    pattern='uniform'
                ),
                phase=PhysicalField(
                    time_series=TimeSeries(
                        values=[Decimal('0'), Decimal('0')],
                        times=[Decimal('0'), Decimal('0.000003')]
                    ),
                    pattern='uniform'
                ),
                detuning=PhysicalField(
                    time_series=TimeSeries(
                        values=[Decimal('-54000000.0'), Decimal('54000000.0')],
                        times=[Decimal('0'), Decimal('0.000003')]
                    ),
                    pattern='uniform'
                )
            )
        ],
        localDetuning=[
            LocalDetuning(
                magnitude=PhysicalField(
                    times_series=TimeSeries(
```

```
                        values=[Decimal('0'), Decimal('25000000.0'),
 Decimal('25000000.0'), Decimal('0')],
                        times=[Decimal('0'), Decimal('0.000001'), Decimal('0.000002'),
 Decimal('0.000003')]
                    ),
                    pattern=Pattern([Decimal('0.8'), Decimal('1.0'), Decimal('0.9')])
                )
            )
        ]
    )
)
```

JSON (example)

> **Note**
>
> If the [local detuning](#) feature is not enabled for your account, use `"localDetuning": []` in the following example.

```
{
    "braketSchemaHeader": {
        "name": "braket.ir.ahs.program",
        "version": "1"
    },
    "setup": {
        "ahs_register": {
            "sites": [
                [0E-7, 0E-7],
                [0E-7, 4E-6],
                [4E-6, 0E-7]
            ],
            "filling": [1, 1, 1]
        }
    },
    "hamiltonian": {
        "drivingFields": [
            {
                "amplitude": {
                    "time_series": {
                        "values": [0.0, 15700000.0, 15700000.0, 0.0],
                        "times": [0E-9, 0.000001000, 0.000002000, 0.000003000]
```

```
                    },
                    "pattern": "uniform"
                },
                "phase": {
                    "time_series": {
                        "values": [0E-7, 0E-7],
                        "times": [0E-9, 0.000003000]
                    },
                    "pattern": "uniform"
                },
                "detuning": {
                    "time_series": {
                        "values": [-54000000.0, 54000000.0],
                        "times": [0E-9, 0.000003000]
                    },
                    "pattern": "uniform"
                }
            }
        ],
        "localDetuning": [
            {
                "magnitude": {
                    "time_series": {
                        "values": [0.0, 25000000.0, 25000000.0, 0.0],
                        "times": [0E-9, 0.000001000, 0.000002000, 0.000003000]
                    },
                    "pattern": [0.8, 1.0, 0.9]
                }
            }
        ]
    }
}
```

Main fields

Program field	type	description
setup.ahs_register.sites	List[List[Decimal]]	List of 2-d coordinates where the tweezers trap atoms

Program field	type	description
setup.ahs_register.filling	List[int]	Marks atoms that occupy the trap sites with 1, and empty sites with 0
hamiltonian.drivingFields[].amplitude.time_series.times	List[Decimal]	time points of driving amplitude, Omega(t)
hamiltonian.drivingFields[].amplitude.time_series.values	List[Decimal]	values of driving amplitude, Omega(t)
hamiltonian.drivingFields[].amplitude.pattern	str	spatial pattern of driving amplitude, Omega(t); must be 'uniform'
hamiltonian.drivingFields[].phase.time_series.times	List[Decimal]	time points of driving phase, phi(t)
hamiltonian.drivingFields[].phase.time_series.values	List[Decimal]	values of driving phase, phi(t)
hamiltonian.drivingFields[].phase.pattern	str	spatial pattern of driving phase, phi(t); must be 'uniform'
hamiltonian.drivingFields[].detuning.time_series.times	List[Decimal]	time points of driving detuning, Delta_global(t)
hamiltonian.drivingFields[].detuning.time_series.values	List[Decimal]	values of driving detuning, Delta_global(t)

Program field	type	description
hamiltonian.drivingFields[].detuning.pattern	str	spatial pattern of driving detuning, Delta_global(t); must be 'uniform'
hamiltonian.localDetuning[].magnitude.time_series.times	List[Decimal]	time points of the time-dependent factor of the local detuning magnitude, Delta_local(t)
hamiltonian.localDetuning[].magnitude.time_series.values	List[Decimal]	values of the time-dependent factor of the local detuning magnitude, Delta_local(t)
hamiltonian.localDetuning[].magnitude.pattern	List[Decimal]	site-dependent factor of the local detuning magnitude, h_k (values corresponds to sites in setup.ahs_register.sites)

Metadata fields

Program field	type	description
braketSchemaHeader.name	str	name of the schema; must be 'braket.ir.ahs.program'

Program field	type	description
braketSchemaHeader.version	str	version of the schema

Braket AHS task result schema

braket.tasks.analog_hamiltonian_simulation_quantum_task_result.AnalogHamiltonianSimulationQua (example)

```
AnalogHamiltonianSimulationQuantumTaskResult(
    task_metadata=TaskMetadata(
        braketSchemaHeader=BraketSchemaHeader(
            name='braket.task_result.task_metadata',
            version='1'
        ),
        id='arn:aws:braket:us-east-1:123456789012:quantum-task/12345678-90ab-
cdef-1234-567890abcdef',
        shots=2,
        deviceId='arn:aws:braket:us-east-1::device/qpu/quera/Aquila',
        deviceParameters=None,
        createdAt='2022-10-25T20:59:10.788Z',
        endedAt='2022-10-25T21:00:58.218Z',
        status='COMPLETED',
        failureReason=None
    ),
    measurements=[
        ShotResult(
            status=<AnalogHamiltonianSimulationShotStatus.SUCCESS: 'Success'>,

            pre_sequence=array([1, 1, 1, 1]),
            post_sequence=array([0, 1, 1, 1])
        ),

        ShotResult(
            status=<AnalogHamiltonianSimulationShotStatus.SUCCESS: 'Success'>,

            pre_sequence=array([1, 1, 0, 1]),
            post_sequence=array([1, 0, 0, 0])
        )
    ]
)
```

JSON (example)


```
{
    "braketSchemaHeader": {
        "name": "braket.task_result.analog_hamiltonian_simulation_task_result",
        "version": "1"
    },
    "taskMetadata": {
        "braketSchemaHeader": {
            "name": "braket.task_result.task_metadata",
            "version": "1"
        },
        "id": "arn:aws:braket:us-east-1:123456789012:quantum-task/12345678-90ab-
cdef-1234-567890abcdef",
        "shots": 2,
        "deviceId": "arn:aws:braket:us-east-1::device/qpu/quera/Aquila",

        "createdAt": "2022-10-25T20:59:10.788Z",
        "endedAt": "2022-10-25T21:00:58.218Z",
        "status": "COMPLETED"

    },
    "measurements": [
        {
            "shotMetadata": {"shotStatus": "Success"},
            "shotResult": {
                "preSequence": [1, 1, 1, 1],
                "postSequence": [0, 1, 1, 1]
            }
        },
        {
            "shotMetadata": {"shotStatus": "Success"},
            "shotResult": {
                "preSequence": [1, 1, 0, 1],
                "postSequence": [1, 0, 0, 0]
            }
        }
    ],
    "additionalMetadata": {
        "action": {...}
        "queraMetadata": {
            "braketSchemaHeader": {
                "name": "braket.task_result.quera_metadata",
                "version": "1"
```

```
            },
            "numSuccessfulShots": 100
        }
    }
}
```

Main fields

Task result field	type	description
measurements[].shotResult.preSequence	List[int]	Pre-sequence measurement bits (one for each atomic site) for each shot: 0 if site is empty, 1 if site is filled, measured before the sequences of pulses that run the quantum evolution
measurements[].shotResult.postSequence	List[int]	Post-sequence measurement bits for each shot: 0 if atom is in Rydberg state or site is empty, 1 if atom is in ground state, measured at the end of the sequences of pulses that run the quantum evolution

Metadata fields

Task result field	type	description
braketSchemaHeader.name	str	name of the schema; must be 'braket.task_result.analog_hamiltonian_simulation_task_result'

Task result field	type	description
braketSchemaHeader.version	str	version of the schema
taskMetadata.braketSchemaHeader.name	str	name of the schema; must be ‘braket.task_result.task_metadata'
taskMetadata.braketSchemaHeader.version	str	version of the schema
taskMetadata.id	str	The ID of the quantum task. For AWS quantum tasks, this is the quantum task ARN.
taskMetadata.shots	int	The number of shots for the quantum task
taskMetadata.shots.deviceId	str	The ID of the device on which the quantum task ran. For AWS devices, this is the device ARN.

Task result field	type	description
taskMetadata.shots.createdAt	str	The timestamp of creation; the format must be in ISO-8601/ RFC3339 string format YYYY-MM-D DTHH:mm:s s.sssZ. Default is None.
taskMetadata.shots.endedAt	str	The timestamp of when the quantum task ended; the format must be in ISO-8601/ RFC3339 string format YYYY-MM-D DTHH:mm:s s.sssZ. Default is None.
taskMetadata.shots.status	str	The status of the quantum task (CREATED, QUEUED, RUNNING, COMPLETED , FAILED). Default is None.

Task result field	type	description
taskMetadata.shots.failureReason	str	The failure reason of the quantum task. Default is None.
additionalMetadata.action	braket.ir.ahs.program_v1.Program	(See the Braket AHS program schema section)
additionalMetadata.action.braketSchemaHeader.queraMetadata.name	str	name of the schema; must be 'braket.task_result.quera_metadata'
additionalMetadata.action.braketSchemaHeader.queraMetadata.version	str	version of the schema
additionalMetadata.action.numSuccessfulShots	int	number of completely successful shots; must be equal to the requested number of shots

Task result field	type	description
measurements[].shotMetadata.shotStatus	int	The status of the shot, (Success, Partial success, Failure); must be "Success"

QuEra device properties schema

braket.device_schema.quera.quera_device_capabilities_v1.QueraDeviceCapabilities (example)

```
QueraDeviceCapabilities(
    service=DeviceServiceProperties(
        braketSchemaHeader=BraketSchemaHeader(
            name='braket.device_schema.device_service_properties',
            version='1'
            ),
            executionWindows=[
                DeviceExecutionWindow(
                    executionDay=<ExecutionDay.MONDAY: 'Monday'>,
                    windowStartHour=datetime.time(1, 0),
                    windowEndHour=datetime.time(23, 59, 59)
                ),
                DeviceExecutionWindow(
                    executionDay=<ExecutionDay.TUESDAY: 'Tuesday'>,
                    windowStartHour=datetime.time(0, 0),
                    windowEndHour=datetime.time(12, 0)
                ),
                DeviceExecutionWindow(
                    executionDay=<ExecutionDay.WEDNESDAY: 'Wednesday'>,
                    windowStartHour=datetime.time(0, 0),
                    windowEndHour=datetime.time(12, 0)
                ),
                DeviceExecutionWindow(
                    executionDay=<ExecutionDay.FRIDAY: 'Friday'>,
                    windowStartHour=datetime.time(0, 0),
                    windowEndHour=datetime.time(23, 59, 59)
                ),
                DeviceExecutionWindow(
```

```
                    executionDay=<ExecutionDay.SATURDAY: 'Saturday'>,
                    windowStartHour=datetime.time(0, 0),
                    windowEndHour=datetime.time(23, 59, 59)
                ),
                DeviceExecutionWindow(
                    executionDay=<ExecutionDay.SUNDAY: 'Sunday'>,
                    windowStartHour=datetime.time(0, 0),
                    windowEndHour=datetime.time(12, 0)
                )
            ],
            shotsRange=(1, 1000),
            deviceCost=DeviceCost(
                price=0.01,
                unit='shot'
            ),
            deviceDocumentation=
                DeviceDocumentation(
                    imageUrl='https://
a.b.cdn.console.awsstatic.com/59534b58c709fc239521ef866db9ea3f1aba73ad3ebcf60c23914ad8c5c5c878/
a6cfc6fca26cf1c2e1c6.png',
                    summary='Analog quantum processor based on neutral atom arrays',
                    externalDocumentationUrl='https://www.quera.com/aquila'
                ),
                deviceLocation='Boston, USA',
                updatedAt=datetime.datetime(2024, 1, 22, 12, 0,
 tzinfo=datetime.timezone.utc),
                getTaskPollIntervalMillis=None
    ),
    action={
        <DeviceActionType.AHS: 'braket.ir.ahs.program'>: DeviceActionProperties(
                version=['1'],
                actionType=<DeviceActionType.AHS: 'braket.ir.ahs.program'>
            )
    },
    deviceParameters={},
    braketSchemaHeader=BraketSchemaHeader(
        name='braket.device_schema.quera.quera_device_capabilities',
        version='1'
    ),
    paradigm=QueraAhsParadigmProperties(
        ...
        # See https://github.com/amazon-braket/amazon-braket-schemas-python/blob/main/
src/braket/device_schema/quera/quera_ahs_paradigm_properties_v1.py
        ...
```

```
    )
)
```

JSON (example)

```
{
    "service": {
        "braketSchemaHeader": {
            "name": "braket.device_schema.device_service_properties",
            "version": "1"
        },
        "executionWindows": [
            {
                "executionDay": "Monday",
                "windowStartHour": "01:00:00",
                "windowEndHour": "23:59:59"
            },
            {
                "executionDay": "Tuesday",
                "windowStartHour": "00:00:00",
                "windowEndHour": "12:00:00"
            },
            {
                "executionDay": "Wednesday",
                "windowStartHour": "00:00:00",
                "windowEndHour": "12:00:00"
            },
            {
                "executionDay": "Friday",
                "windowStartHour": "00:00:00",
                "windowEndHour": "23:59:59"
            },
            {
                "executionDay": "Saturday",
                "windowStartHour": "00:00:00",
                "windowEndHour": "23:59:59"
            },
            {
                "executionDay": "Sunday",
                "windowStartHour": "00:00:00",
                "windowEndHour": "12:00:00"
            }
        ],
```

```
        "shotsRange": [
            1,
            1000
        ],
        "deviceCost": {
            "price": 0.01,
            "unit": "shot"
        },
        "deviceDocumentation": {
            "imageUrl": "https://
a.b.cdn.console.awsstatic.com/59534b58c709fc239521ef866db9ea3f1aba73ad3ebcf60c23914ad8c5c5c878/
a6cfc6fca26cf1c2e1c6.png",
            "summary": "Analog quantum processor based on neutral atom arrays",
            "externalDocumentationUrl": "https://www.quera.com/aquila"
        },
        "deviceLocation": "Boston, USA",
        "updatedAt": "2024-01-22T12:00:00+00:00"
    },
    "action": {
        "braket.ir.ahs.program": {
            "version": [
                "1"
            ],
            "actionType": "braket.ir.ahs.program"
        }
    },
    "deviceParameters": {},
    "braketSchemaHeader": {
        "name": "braket.device_schema.quera.quera_device_capabilities",
        "version": "1"
    },
    "paradigm": {
        ...
        # See Aquila device page > "Calibration" tab > "JSON" page
        ...
    }
}
```

Service properties fields

Service properties field	type	description
service.executionWindows[].executionDay	ExecutionDay	Days of the execution window; must be 'Everyday', 'Weekdays ', 'Weekend', 'Monday', 'Tuesday', 'Wednesda y', Thursday', 'Friday', 'Saturday' or 'Sunday'
service.executionWindows[].windowStartHour	datetime.time	UTC 24-hour format of the time when the execution window starts
service.executionWindows[].windowEndHour	datetime.time	UTC 24-hour format of the time when the execution window ends
service.qpu_capabilities.service.shotsRange	Tuple[int, int]	Minimum and maximum number of shots for the device
service.qpu_capabilities.service.deviceCost.p rice	float	Price of the device in terms of US dollars
service.qpu_capabilities.service.deviceCost.u nit	str	unit for charging the price, e.g: 'minute', 'hour', 'shot', 'task'

Metadata fields

Metadata field	type	description
action[].version	str	version of the AHS program schema

Metadata field	type	description
action[].actionType	ActionType	AHS program schema name; must be 'braket.ir.ahs.pro gram'
service.braketSchemaHeader.name	str	name of the schema; must be 'braket.d evice_schema.devic e_service_properties'
service.braketSchemaHeader.version	str	version of the schema
service.deviceDocumentation.imageUrl	str	URL for the image of the device
service.deviceDocumentation.summary	str	brief description on the device
service.deviceDocumentation.externalDocumenta tionUrl	str	external documenta tion URL
service.deviceLocation	str	geographic location fo the device
service.updatedAt	datetime	time when the device properties were last updated

Working with AWS Boto3

Boto3 is the AWS SDK for Python. With Boto3, Python developers can create, configure, and manage AWS services, such as Amazon Braket. Boto3 provides an object-oriented API, as well as low-level access to Amazon Braket.

Follow the instructions in the Boto3 Quickstart guide to learn how to install and configure Boto3.

Boto3 provides the core functionality that works along with the Amazon Braket Python SDK to help you configure and run your quantum tasks. Python customers always need to install Boto3, because that is the core implementation. If you want to make use of additional helper methods, you also need to install the Amazon Braket SDK.

For example, when you call `CreateQuantumTask`, the Amazon Braket SDK submits the request to Boto3, which then calls the AWS API.

In this section:

- Turn on the Amazon Braket Boto3 client
- Configure AWS CLI profiles for Boto3 and the Braket SDK

Turn on the Amazon Braket Boto3 client

To use Boto3 with Amazon Braket, you must import Boto3 and then define a client that you use to connect to the Amazon Braket API. In the following example, the Boto3 client is named `braket`.

```
import boto3
import botocore

braket = boto3.client("braket")
```

> **Note**
>
> Braket supports IPv6. If you are using an IPv6-only network or wish to ensure your workload uses IPv6 traffic, use the dual-stack endpoints as outlined in the Dual-stack and FIPS endpoints guide.

Now that you have a `braket` client established, you can make requests and process responses from the Amazon Braket service. You can get more detail on request and response data in the API Reference.

The following examples show how to work with devices and quantum tasks.

- Search for devices
- Retrieve a device
- Create a quantum task

- Retrieve a quantum task
- Search for quantum tasks
- Cancel quantum task

Search for devices

- `search_devices(**kwargs)`

Search for devices using the specified filters.

```
# Pass search filters and optional parameters when sending the
# request and capture the response
response = braket.search_devices(filters=[{
    'name': 'deviceArn',
    'values': ['arn:aws:braket:::device/quantum-simulator/amazon/sv1']
}], maxResults=10)

print(f"Found {len(response['devices'])} devices")

for i in range(len(response['devices'])):
    device = response['devices'][i]
    print(device['deviceArn'])
```

Retrieve a device

- `get_device(deviceArn)`

Retrieve the devices available in Amazon Braket.

```
# Pass the device ARN when sending the request and capture the repsonse
response = braket.get_device(deviceArn='arn:aws:braket:::device/quantum-simulator/
amazon/sv1')

print(f"Device {response['deviceName']} is {response['deviceStatus']}")
```

Create a quantum task

- `create_quantum_task(**kwargs)`

Create a quantum task.

```
# Create parameters to pass into create_quantum_task()
kwargs = {
    # Create a Bell pair
    'action': '{"braketSchemaHeader": {"name": "braket.ir.jaqcd.program", "version":
 "1"}, "results": [], "basis_rotation_instructions": [], "instructions": [{"type": "h",
 "target": 0}, {"type": "cnot", "control": 0, "target": 1}]}',
    # Specify the SV1 Device ARN
    'deviceArn': 'arn:aws:braket:::device/quantum-simulator/amazon/sv1',
    # Specify 2 qubits for the Bell pair
    'deviceParameters': '{"braketSchemaHeader": {"name":
 "braket.device_schema.simulators.gate_model_simulator_device_parameters",
 "version": "1"}, "paradigmParameters": {"braketSchemaHeader": {"name":
 "braket.device_schema.gate_model_parameters", "version": "1"}, "qubitCount": 2}}',
    # Specify where results should be placed when the quantum task completes.
    # You must ensure the S3 Bucket exists before calling create_quantum_task()
    'outputS3Bucket': 'amazon-braket-examples',
    'outputS3KeyPrefix': 'boto-examples',
    # Specify number of shots for the quantum task
    'shots': 100
}

# Send the request and capture the response
response = braket.create_quantum_task(**kwargs)

print(f"Quantum task {response['quantumTaskArn']} created")
```

Retrieve a quantum task

- `get_quantum_task(quantumTaskArn)`

Retrieve the specified quantum task.

```
# Pass the quantum task ARN when sending the request and capture the response
response = braket.get_quantum_task(quantumTaskArn='arn:aws:braket:us-
west-1:123456789012:quantum-task/ce78c429-cef5-45f2-88da-123456789012')

print(response['status'])
```

Search for quantum tasks

- `search_quantum_tasks(**kwargs)`

Search for quantum tasks that match the specified filter values.

```
# Pass search filters and optional parameters when sending the
# request and capture the response
response = braket.search_quantum_tasks(filters=[{
    'name': 'deviceArn',
    'operator': 'EQUAL',
    'values': ['arn:aws:braket:::device/quantum-simulator/amazon/sv1']
}], maxResults=25)

print(f"Found {len(response['quantumTasks'])} quantum tasks")

for n in range(len(response['quantumTasks'])):
    task = response['quantumTasks'][n]
    print(f"Quantum task {task['quantumTaskArn']} for {task['deviceArn']} is
 {task['status']}")
```

Cancel quantum task

- `cancel_quantum_task(quantumTaskArn)`

Cancel the specified quantum task.

```
# Pass the quantum task ARN when sending the request and capture the response
response = braket.cancel_quantum_task(quantumTaskArn='arn:aws:braket:us-
west-1:123456789012:quantum-task/ce78c429-cef5-45f2-88da-123456789012')

print(f"Quantum task {response['quantumTaskArn']} is {response['cancellationStatus']}")
```

Configure AWS CLI profiles for Boto3 and the Braket SDK

The Amazon Braket SDK relies upon the default AWS CLI credentials, unless you explicitly specify otherwise. We recommend that you keep the default when you run on a managed Amazon Braket notebook because you must provide an IAM role that has permissions to launch the notebook instance.

Optionally, if you run your code locally (on an Amazon EC2 instance, for example), you can establish named AWS CLI profiles. You can give each profile a different permission set, rather than regularly overwriting the default profile.

This section provides a brief explanation of how to configure such a CLI `profile` and how to incorporate that profile into Amazon Braket so that API calls are made with the permissions from that profile.

In this section:

- Step 1: Configure a local AWS CLI profile
- Step 2: Establish a Boto3 session object
- Step 3: Incorporate the Boto3 session into the Braket AwsSession

Step 1: Configure a local AWS CLI `profile`

It is beyond the scope of this document to explain how to create a user and how to configure a non-default profile. For information on these topics, see:

- Getting started
- Configuring the AWS CLI to use AWS IAM Identity Center

To use Amazon Braket, you must provide this user — and the associated CLI `profile` — with the necessary Braket permissions. For instance, you can attach the **AmazonBraketFullAccess** policy.

Step 2: Establish a Boto3 session object

In order to establish a Boto3 session object, utilize the following code example.

```
from boto3 import Session

# Insert CLI profile name here
boto_sess = Session(profile_name=`profile`)
```

Note

If the expected API calls have Region-based restrictions that are not aligned with your `profile` default Region, you can specify a Region for the Boto3 session as shown in the following example.

```
# Insert CLI profile name _and_ region
boto_sess = Session(profile_name=`profile`, region_name=`region`)
```

For the argument designated as `region`, substitute a value that corresponds to one of the AWS Regions in which Amazon Braket is available such as `us-east-1`, `us-west-1`, and so forth.

Step 3: Incorporate the Boto3 session into the Braket AwsSession

The following example shows how to initialize a Boto3 Braket session and instantiate a device in that session.

```
from braket.aws import AwsSession, AwsDevice

# Initialize Braket session with Boto3 Session credentials
aws_session = AwsSession(boto_session=boto_sess)

# Instantiate any Braket QPU device with the previously initiated AwsSession
sim_arn = 'arn:aws:braket:::device/quantum-simulator/amazon/sv1'
device = AwsDevice(sim_arn, aws_session=aws_session)
```

After this setup is complete, you can submit quantum tasks to that instantiated `AwsDevice` object (by calling the `device.run(…)` command for example). All API calls made by that device can use the IAM credentials associated with the CLI profile that you previously designated as `profile`.

Testing your quantum tasks with Amazon Braket

Amazon Braket provides a variety of high-performance quantum circuit simulators to help you test and validate your quantum algorithms before running them on actual quantum hardware. These simulators handle the complex underlying software and infrastructure, and Amazon Elastic Compute Cloud (Amazon EC2) clusters to take away the burden of simulating quantum circuits on classical high performance computing (HPC) infrastructure. These resources allow you to focus on developing and optimizing your quantum applications.

With Braket's simulators, you can thoroughly test your quantum circuits and algorithms without the constraints and limitations of physical quantum devices. This enables you to explore a wide range of quantum computing concepts, from basic quantum gates and circuits to more advanced quantum algorithms and error mitigation techniques.

The Braket SDK simplifies submitting your quantum tasks to the simulators, allowing you to control the simulation parameters, such as the number of shots and the noise model, to better understand the behavior of your quantum algorithms. You can also use Amazon Braket Hybrid Job capabilities to combine classical and quantum computing elements, further expanding the scope of your testing and validation.

By thoroughly testing your quantum tasks on Braket's simulators, you can gain valuable insights, refine your algorithms, and ensure their correctness before deploying them on real quantum hardware. This helps to reduce development time, minimize errors, and ultimately accelerate your progress in the field of quantum computing.

In this section:

- Submitting quantum tasks to simulators
- Local quantum device emulator
- Working with Amazon Braket Hybrid Jobs

Submitting quantum tasks to simulators

Amazon Braket provides access to several simulators that can test your quantum tasks. You can submit quantum tasks individually or you can set up quantum task batching.

Simulators

- **Density matrix simulator, DM1** : `arn:aws:braket:::device/quantum-simulator/amazon/dm1`
- **State vector simulator, SV1** : `arn:aws:braket:::device/quantum-simulator/amazon/sv1`
- **Tensor network simulator, TN1** : `arn:aws:braket:::device/quantum-simulator/amazon/tn1`
- **The local simulator** : `LocalSimulator()`

> **Note**
>
> You can cancel quantum tasks in the `CREATED` state for QPUs and on-demand simulators. You can cancel quantum tasks in the `QUEUED` state on a best-effort basis for on-demand simulators and QPUs. Note that QPU `QUEUED` quantum tasks are unlikely to be cancelled successfully during QPU availability windows.

In this section:

- Local state vector simulator (braket_sv)
- Local density matrix simulator (braket_dm)
- Local AHS simulator (braket_ahs)
- State vector simulator (SV1)
- Density matrix simulator (DM1)
- Tensor network simulator (TN1)
- About embedded simulators
- Compare Amazon Braket simulators
- Example quantum tasks on Amazon Braket
- Testing a quantum task with the local simulator
- Quantum task batching

Local state vector simulator (`braket_sv`)

The local state vector simulator (`braket_sv`) is part of the Amazon Braket SDK that runs locally in your environment. It is well-suited for rapid prototyping on small circuits (up to 25 qubits) depending on the hardware specifications of your Braket notebook instance or your local environment.

The local simulator supports all gates in the Amazon Braket SDK, but QPU devices support a smaller subset. You can find the supported gates of a device in the device properties.

Note

The local simulator supports advanced OpenQASM features which may not be supported on QPU devices or other simulators. For more information on supported features, see the examples provided in the OpenQASM Local Simulator notebook.

For more information about how to work with simulators, see the Amazon Braket examples.

Local density matrix simulator (`braket_dm`)

The local density matrix simulator (`braket_dm`) is part of the Amazon Braket SDK that runs locally in your environment. It is well-suited for rapid prototyping on small circuits with noise (up to 12 qubits) depending on the hardware specifications of your Braket notebook instance or your local environment.

You can build common noisy circuits from the ground up using gate noise operations such as bit-flip and depolarizing error. You can also apply noise operations to specific qubits and gates of existing circuits that are intended to run both with and without noise.

The `braket_dm` local simulator can provide the following results, given the specified number of shots:

- Reduced density matrix: Shots = 0

Note

The local simulator supports advanced OpenQASM features, which may not be supported on QPU devices or other simulators. For more information about supported features, see the examples provided in the OpenQASM Local Simulator notebook.

To learn more about the local density matrix simulator, see the Braket introductory noise simulator example.

Local AHS simulator (braket_ahs)

The local AHS (Analog Hamiltonian Simulation) simulator (`braket_ahs`) is part of the Amazon Braket SDK that runs locally in your environment. It can be used to simulate results from an AHS program. It is well-suited for prototyping on small registers (up to 10-12 atoms) depending on the hardware specifications of your Braket notebook instance or your local environment.

The local simulator supports AHS programs with one uniform driving field, one (non-uniform) shifting field, and arbitrary atom arrangements. For details, refer to the Braket AHS class and the Braket AHS program schema.

To learn more about the local AHS simulator, see the Hello AHS: Run your first Analog Hamiltonian Simulation page and the Analog Hamiltonian Simulation example notebooks.

State vector simulator (SV1)

SV1 is an on-demand, high-performance, universal state vector simulator. It can simulate circuits of up to 34 qubits. You can expect a 34-qubit, dense, and square circuit (circuit depth = 34) to take approximately 1–2 hours to complete, depending on the type of gates used and other factors. Circuits with all-to-all gates are well suited for SV1. It returns results in forms such as a full state vector or an array of amplitudes.

SV1 has a maximum runtime of 6 hours. It has a default of 35 concurrent quantum tasks, and a maximum of 100 (50 in us-west-1 and eu-west-2) concurrent quantum tasks.

SV1 results

SV1 can provide the following results, given the specified number of shots:

- Sample: Shots > 0
- Expectation: Shots >= 0
- Variance: Shots >= 0
- Probability: Shots > 0
- Amplitude: Shots = 0
- Adjoint Gradient: Shots = 0

For more about results, see Result types.

SV1 is always available, it runs your circuits on demand, and it can run multiple circuits in parallel. The runtime scales linearly with the number of operations and exponentially with the number of qubits. The number of shots has a small impact on the runtime. To learn more, visit Compare simulators.

Simulators support all gates in the Braket SDK, but QPU devices support a smaller subset. You can find the supported gates of a device in the device properties.

Density matrix simulator (DM1)

DM1 is an on-demand, high-performance, density matrix simulator. It can simulate circuits of up to 17 qubits.

DM1 has a maximum runtime of 6 hours, a default of 35 concurrent quantum tasks, and a maximum of 50 concurrent quantum tasks.

DM1 results

DM1 can provide the following results, given the specified number of shots:

- Sample: Shots > 0
- Expectation: Shots >= 0
- Variance: Shots >= 0
- Probability: Shots > 0
- Reduced density matrix: Shots = 0, up to max 8 qubits

For more information about results, see Result types.

DM1 is always available, it runs your circuits on demand, and it can run multiple circuits in parallel. The runtime scales linearly with the number of operations and exponentially with the number of qubits. The number of shots has a small impact on the runtime. To learn more, see Compare simulators.

Noise gates and limitations

```
AmplitudeDamping
    Probability has to be within [0,1]
BitFlip
```

```
    Probability has to be within [0,0.5]
Depolarizing
    Probability has to be within [0,0.75]
GeneralizedAmplitudeDamping
    Probability has to be within [0,1]
PauliChannel
    The sum of the probabilities has to be within [0,1]
Kraus
    At most 2 qubits
    At most 4 (16) Kraus matrices for 1 (2) qubit
PhaseDamping
    Probability has to be within [0,1]
PhaseFlip
    Probability has to be within [0,0.5]
TwoQubitDephasing
    Probability has to be within [0,0.75]
TwoQubitDepolarizing
    Probability has to be within [0,0.9375]
```

Tensor network simulator (TN1)

TN1 is an on-demand, high-performance, tensor network simulator. TN1 can simulate certain circuit types with up to 50 qubits and a circuit depth of 1,000 or smaller. TN1 is particularly powerful for sparse circuits, circuits with local gates, and other circuits with special structure, such as quantum Fourier transform (QFT) circuits. TN1 operates in two phases. First, the *rehearsal phase* attempts to identify an efficient computational path for your circuit, so TN1 can estimate the runtime of the next stage, which is called the *contraction phase*. If the estimated contraction time exceeds the TN1 simulation runtime limit, TN1 does not attempt contraction.

TN1 has a runtime limit of 6 hours. It is limited to a maximum of 10 (5 in eu-west-2) concurrent quantum tasks.

TN1 results

The contraction phase consists of a series of matrix multiplications. The series of multiplications continues until a result is reached or until it is determined that a result cannot be reached.

Note: Shots must be > 0.

Result types include:

- Sample

- Expectation
- Variance

For more about results, see Result types.

TN1 is always available, it runs your circuits on demand, and it can run multiple circuits in parallel. To learn more, see Compare simulators.

Simulators support all gates in the Braket SDK, but QPU devices support a smaller subset. You can find the supported gates of a device in the device properties.

Visit the Amazon Braket GitHub repository for a TN1 example notebook to help you get started with TN1.

Best practices for working with TN1

- Avoid all-to-all circuits.
- Test a new circuit or class of circuits with a small number of shots, to learn the circui's "hardness" for TN1.
- Split large shot simulations over multiple quantum tasks.

About embedded simulators

Embedded simulators operate by having the simulation embedded directly within the algorithm code. Also, it is contained within the same container and runs the simulation directly on the hybrid job instance. This approach is useful for removing bottlenecks typically associated with communicating between the simulation and a remote device. By keeping all computations in a single, cohesive environment, embedded simulators can greatly reduce memory requirements and decrease the number of circuit executions needed to achieve a target result. This can lead to substantial performance improvements, often by a factor of ten or more, as compared to traditional setups that rely on remote simulation. For more information about how embedded simulators enhance performance and enable streamlined hybrid jobs, refer to the Run a hybrid job with Amazon Braket Hybrid Jobs documentation page.

PennyLane's lightning simulators

You can use PennyLane's lightning simulators as embedded simulators on Braket. With PennyLane's lightning simulators, you can use advanced gradient computation methods, such as adjoint differentiation, to evaluate gradients faster. The lightning.qubit simulator is available as a device through Braket NBIs and as an embedded simulator, whereas the lightning.gpu simulator needs to be run as an embedded simulator with a GPU instance. See the Embedded simulators in Braket Hybrid Jobs notebook for an example of using lightning.gpu.

Compare Amazon Braket simulators

This section helps you select the Amazon Braket simulator that is best suited for your quantum task, by describing some concepts, limitations, and use cases.

Choosing between local simulators and on-demand simulators (SV1, TN1, DM1)

The performance of *local simulators* depends on the hardware that hosts the local environment, such as a Braket notebook instance, used to run your simulator. *On-demand simulators* run in the AWS cloud and are designed to scale beyond typical local environments. On-demand simulators are optimized for larger circuits, but add some latency overhead per quantum task or batch of quantum tasks. This can imply a trade-off if many quantum tasks are involved. Given these general performance characteristics, the following guidance can help you choose how to run simulations, including ones with noise.

For **simulations**:

- When employing fewer than 18 qubits, use a local simulator.
- When employing 18–24 qubits, choose a simulator based on the workload.
- When employing more than 24 qubits, use an on-demand simulator.

For **noise simulations**:

- When employing fewer than 9 qubits, use a local simulator.
- When employing 9–12 qubits, choose a simulator based on the workload.
- When employing more than 12 qubits, use DM1.

What is a state vector simulator?

SV1 is a universal state vector simulator. It stores the full wave function of the quantum state and sequentially applies gate operations to the state. It stores all possibilities, even the extremely unlikely ones. The SV1 simulator's run time for a quantum task increases linearly with the number of gates in the circuit.

What is a density matrix simulator?

DM1 simulates quantum circuits with noise. It stores the full density matrix of the system and sequentially applies the gates and noise operations of the circuit. The final density matrix contains complete information about the quantum state after the circuit runs. The runtime generally scales linearly with the number of operations and exponentially with the number of qubits.

What is a tensor network simulator?

TN1 encodes quantum circuits into a structured graph.

- The nodes of the graph consist of quantum gates, or qubits.
- The edges of the graph represent connections between gates.

As a result of this structure, TN1 can find simulated solutions for relatively large and complex quantum circuits.

TN1 requires two phases

Typically, TN1 operates in a two-phase approach to simulating quantum computation.

- **The rehearsal phase:** In this phase, TN1 comes up with a way to traverse the graph in an efficient manner, which involves visiting every node so that you can obtain the measurement you desire. As a customer, you do not see this phase because TN1 performs both phases together for you. It completes the first phase and determines whether to perform the second phase on its own based on practical constraints. You have no input into that decision after the simulation has begun.
- **The contraction phase:** This phase is analogous to the execution phase of a computation in a classical computer. The phase consists of a series of matrix multiplications. The order of these multiplications has a great effect on the difficulty of the computation. Therefore, the rehearsal phase is accomplished first to find the most effective computation paths across the graph. After it finds the contraction path during the rehearsal phase, TN1 contracts together the gates of your circuit to produce the results of the simulation.

TN1 graphs are analogous to a map

Metaphorically, you can compare the underlying TN1 graph to the streets of a city. In a city with a planned grid, it is easy to find a route to your destination using a map. In a city with unplanned streets, duplicate street names, and so forth, it can be difficult to find a route to your destination by looking at a map.

If TN1 did not perform the rehearsal phase, it would be like walking around the streets of the city to find your destination, instead of looking at a map first. It can really pay off in terms of walking time to spend more time looking at the map. Similarly, the rehearsal phase provides valuable information.

You might say that the TN1 has a certain “awareness” of the structure of the underlying circuit that it traverses. It gains this awareness during the rehearsal phase.

Types of problems best suited for each of these types of simulators

SV1 is well-suited for any class of problems that rely primarily on having a certain number of qubits and gates. Generally, the time required grows linearly with the number of gates, while it does not depend on the number of shots. SV1 is generally faster than TN1 for circuits under 28 qubits.

SV1 can be slower for higher qubit numbers because it actually simulates all possibilities, even the extremely unlikely ones. It has no way to determine which outcomes are likely. Thus, for a 30-qubit evaluation, SV1 must calculate 2^30 configurations. The limit of 34 qubits for the Amazon Braket SV1 simulator is a practical constraint due to memory and storage limitations. You can think of it like this: Each time you add a qubit to SV1, the problem becomes twice as hard.

For many classes of problems, TN1 can evaluate much larger circuits in realistic time than SV1 because TN1 takes advantage of the structure of the graph. It essentially tracks the evolution of solutions from its starting place and it retains only the configurations that contribute to an efficient traversal. Put another way, it saves the configurations to create an ordering of matrix multiplication that results in a simpler evaluation process.

For TN1, the number of qubits and gates matters, but the structure of the graph matters a lot more. For example, TN1 is very good at evaluating circuits (graphs) in which the gates are short-range (that is, each qubit is connected by gates only to its nearest neighbour qubits), and circuits (graphs) in which the connections (or gates) have similar range. A typical range for TN1 is having each qubit talk only to other qubits that are 5 qubits away. If most of the structure can be decomposed into simpler relationships such as these, which can be represented in *more*, *smaller*, or *more uniform* matrices, TN1 performs the evaluation efficiently.

Limitations of TN1

TN1 can be slower than SV1 depending on the graph's structural complexity. For certain graphs, TN1 terminates the simulation after the rehearsal stage, and shows a status of `FAILED`, for either of these two reasons:

- **Cannot find a path** — If the graph is too complex, it is too difficult to find a good traversal path and the simulator gives up on the computation. TN1 cannot perform the contraction. You may see an error message similar to this one: `No viable contraction path found.`
- **Contraction stage is too difficult** — In some graphs, TN1 can find a traversal path, but it is very long and extremely time-consuming to evaluate. In this case, the contraction is so expensive that the cost would be prohibitive and instead, TN1 exits after the rehearsal phase. You may see an error message similar to this one: `Predicted runtime based on best contraction path found exceeds TN1 limit.`

> **Note**
>
> You are billed for the rehearsal stage of TN1 even if contraction is not performed and you see a `FAILED` status.

The predicted runtime also depends on the shot count. In worst-case scenarios, TN1 contraction time depends linearly on the shot count. The circuit may be contractable with fewer shots. For example, you might submit a quantum task with 100 shots, which TN1 decides is uncontractable, but if you resubmit with only 10, the contraction proceeds. In this situation, to attain 100 samples, you could submit 10 quantum tasks of 10 shots for the same circuit and combine the results in the end.

As a best practice, we recommend that you always test your circuit or circuit class with a few shots (for example, 10) to find out how hard your circuit is for TN1, before you proceed with a higher number of shots.

> **Note**
>
> The series of multiplications that forms the contraction phase begins with small, NxN matrices. For example, a 2-qubit gate requires a 4x4 matrix. The intermediate matrices required during a contraction that is adjudged to be too difficult are gigantic. Such a computation would require days to complete. That is why Amazon Braket does not attempt extremely complex contractions.

Concurrency

All Braket simulators give you the ability to run multiple circuits concurrently. Concurrency limits vary by simulator and region. For more information on concurrency limits, see the Quotas page.

Example quantum tasks on Amazon Braket

This section walks through the stages of running an example quantum task, from selecting the device to viewing the result. As a best practice for Amazon Braket, we recommend that you begin by running the circuit on a simulator, such as SV1.

In this section:

- Specify the device
- Submit an example quantum task
- Submit a parametrized task
- Specify shots
- Poll for results
- View the example results

Specify the device

First, select and specify the device for your quantum task. This example shows how to choose the simulator, SV1.

```
# choose the on-demand simulator to run the circuit
from braket.aws import AwsDevice
device = AwsDevice("arn:aws:braket:::device/quantum-simulator/amazon/sv1")
```

You can view some of the properties of this device as follows:

```
print (device.name)
for iter in device.properties.action['braket.ir.jaqcd.program']:
    print(iter)
```

```
SV1
('version', ['1.0', '1.1'])
```

```
('actionType', <DeviceActionType.JAQCD: 'braket.ir.jaqcd.program'>)
('supportedOperations', ['ccnot', 'cnot', 'cphaseshift', 'cphaseshift00',
 'cphaseshift01', 'cphaseshift10', 'cswap', 'cy', 'cz', 'h', 'i', 'iswap', 'pswap',
 'phaseshift', 'rx', 'ry', 'rz', 's', 'si', 'swap', 't', 'ti', 'unitary', 'v', 'vi',
 'x', 'xx', 'xy', 'y', 'yy', 'z', 'zz'])
('supportedResultTypes', [ResultType(name='Sample', observables=['x', 'y', 'z', 'h',
 'i', 'hermitian'], minShots=1, maxShots=100000), ResultType(name='Expectation',
 observables=['x', 'y', 'z', 'h', 'i', 'hermitian'], minShots=0, maxShots=100000),
 ResultType(name='Variance', observables=['x', 'y', 'z', 'h', 'i', 'hermitian'],
 minShots=0, maxShots=100000), ResultType(name='Probability', observables=None,
 minShots=1, maxShots=100000), ResultType(name='Amplitude', observables=None,
 minShots=0, maxShots=0)])
```

Submit an example quantum task

Submit an example quantum task to run on the on-demand simulator.

```
# create a circuit with a result type
circ = Circuit().rx(0, 1).ry(1, 0.2).cnot(0,2).variance(observable=Observable.Z(),
 target=0)
# add another result type
circ.probability(target=[0, 2])

# set up S3 bucket (where results are stored)
my_bucket = "amzn-s3-demo-bucket" # the name of the bucket
my_prefix = "your-folder-name" # the name of the folder in the bucket
s3_location = (my_bucket, my_prefix)

# submit the quantum task to run
my_task = device.run(circ, s3_location, shots=1000, poll_timeout_seconds = 100,
 poll_interval_seconds = 10)
# the positional argument for the S3 bucket is optional if you want to specify a bucket
 other than the default

# get results of the quantum task
result = my_task.result()
```

The `device.run()` command creates a quantum task through the `CreateQuantumTask` API. After a short initialization time, the quantum task is queued until capacity exists to run the quantum task on a device. In this case, the device is SV1. After the device completes the computation, Amazon Braket writes the results to the Amazon S3 location specified in the call. The positional argument `s3_location` is required for all devices except the local simulator.

> **Note**
>
> The Braket quantum task action is limited to 3MB in size.

Submit a parametrized task

Amazon Braket on-demand and local simulators and QPUs also support specifying values of free parameters at task submission. You can do this by using the `inputs` argument to `device.run()`, as shown in the following example. The `inputs` must be a dictionary of string-float pairs, where the keys are the parameter names.

Parametric compilation can improve the performance of executing parametric circuits on certain QPUs. When submitting a parametric circuit as a quantum task to a supported QPU, Braket will compile the circuit once, and cache the result. There is no recompilation for subsequent parameter updates to the same circuit, resulting in faster runtimes for tasks that use the same circuit. Braket automatically uses the updated calibration data from the hardware provider when compiling your circuit to ensure the highest quality results.

> **Note**
>
> Parametric compilation is supported on all superconducting, gate-based QPUs from Rigetti Computing with the exception of pulse level programs.

```
from braket.circuits import Circuit, FreeParameter, Observable

# create the free parameters
alpha = FreeParameter('alpha')
beta = FreeParameter('beta')

# create a circuit with a result type
circ = Circuit().rx(0, alpha).ry(1, alpha).cnot(0,2).xx(0, 2, beta)
circ.variance(observable=Observable.Z(), target=0)
# add another result type
circ.probability(target=[0, 2])
# submit the quantum task to run
my_task = device.run(circ, inputs={'alpha': 0.1, 'beta':0.2})
```

Specify shots

The shots argument refers to the number of desired measurement shots. Simulators such as SV1 support two simulation modes.

- For shots = 0, the simulator performs an exact simulation, returning the true values for all result types. (Not available on TN1.)
- For non-zero values of shots, the simulator samples from the output distribution to emulate the shot noise of real QPUs. QPU devices only allow shots > 0.

For information about the maximum number of shots per quantum task, refer to [Braket Quotas](#).

Poll for results

When executing `my_task.result()`, the SDK begins polling for a result with the parameters you define upon quantum task creation:

- `poll_timeout_seconds` is the number of seconds to poll the quantum task before it times out when running the quantum task on the on-demand simulator and or QPU devices. The default value is 432,000 seconds, which is 5 days.
- **Note:** For QPU devices such as Rigetti and IonQ, we recommend that you allow a few days. If your polling timeout is too short, results may not be returned within the polling time. For example, when a QPU is unavailable, a local timeout error is returned.
- `poll_interval_seconds` is the frequency with which the quantum task is polled. It specifies how often you call the Braket API to get the status when the quantum task is run on the on-demand simulator and on QPU devices. The default value is 1 second.

This asynchronous execution facilitates the interaction with QPU devices that are not always available. For example, a device could be unavailable during a regular maintenance window.

The returned result contains a range of metadata associated with the quantum task. You can check the measurement result with the following commands:

```
print('Measurement results:\n',result.measurements)
print('Counts for collapsed states:\n',result.measurement_counts)
print('Probabilities for collapsed states:\n',result.measurement_probabilities)
```

```
Measurement results:
```

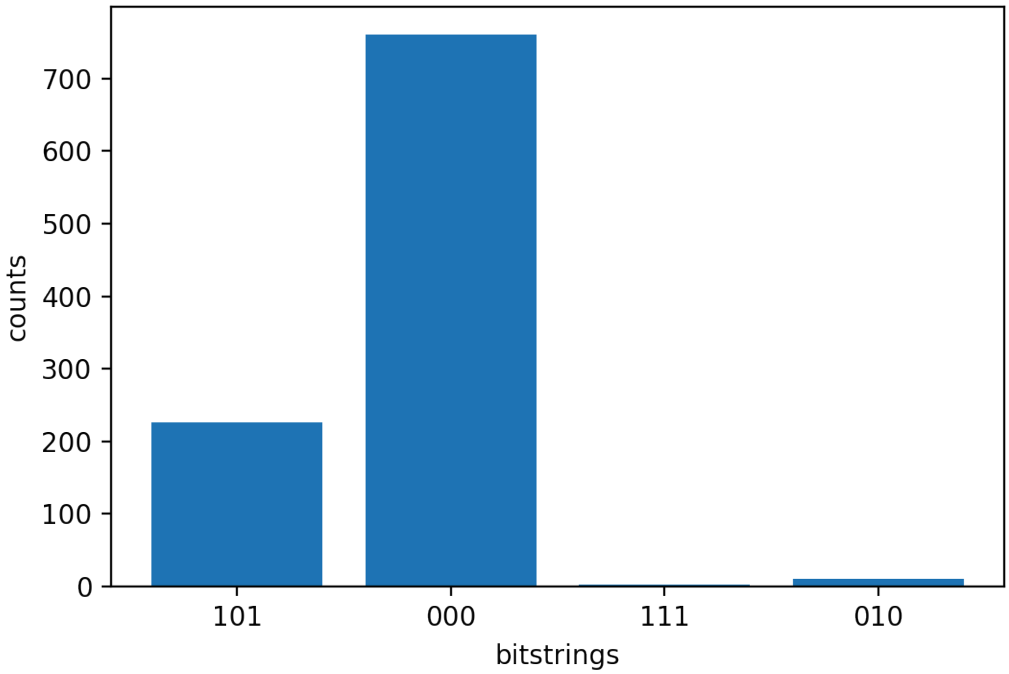
```
 [[1 0 1]
 [0 0 0]
 [1 0 1]
 ...
 [0 0 0]
 [0 0 0]
 [0 0 0]]
Counts for collapsed states:
 Counter({'000': 761, '101': 226, '010': 10, '111': 3})
Probabilities for collapsed states:
 {'101': 0.226, '000': 0.761, '111': 0.003, '010': 0.01}
```

View the example results

Because you've also specified the `ResultType`, you can view the returned results. The result types appear in the order in which they were added to the circuit.

```
print('Result types include:\n', result.result_types)
print('Variance=',result.values[0])
print('Probability=',result.values[1])

# you can plot the result and do some analysis
import matplotlib.pyplot as plt
plt.bar(result.measurement_counts.keys(), result.measurement_counts.values());
plt.xlabel('bitstrings');
plt.ylabel('counts');
```

```
Result types include:
 [ResultTypeValue(type={'observable': ['z'], 'targets': [0], 'type': 'variance'},
 value=0.7062359999999999), ResultTypeValue(type={'targets': [0, 2], 'type':
 'probability'}, value=array([0.771, 0.   , 0.   , 0.229]))]
Variance= 0.7062359999999999
Probability= [0.771 0.    0.    0.229]
```



Testing a quantum task with the local simulator

You can send quantum tasks directly to a local simulator for rapid prototyping and testing. This simulator runs in your local environment, so you do not need to specify an Amazon S3 location. The results are computed directly in your session. To run a quantum task on the local simulator, you must only specify the shots parameter.

> **Note**
>
> The execution speed and maximum number of qubits the local simulator can process depends on the Amazon Braket notebook instance type, or on your local hardware specifications.

The following commands are all identical and instantiate the state vector (noise free) local simulator.

```
# import the LocalSimulator module
from braket.devices import LocalSimulator
# the following are identical commands
```

```
device = LocalSimulator()
device = LocalSimulator("default")
device = LocalSimulator(backend="default")
device = LocalSimulator(backend="braket_sv")
```

Then run a quantum task with the following.

```
my_task = device.run(circ, shots=1000)
```

To instantiate the local density matrix (noise) simulator customers change the backend as follows.

```
# import the LocalSimulator module
from braket.devices import LocalSimulator
device = LocalSimulator(backend="braket_dm")
```

Measuring specific qubits on the local simulator

The local state vector simulator and local density matrix simulator support running circuits where a subset of the circuit's qubits can be measured, which is often called *partial measurement*.

For example, in the following code you can create a two-qubit circuit and only measure the first qubit by adding a `measure` instruction with the target qubits to the end of the circuit.

```
# Import the LocalSimulator module
from braket.devices import LocalSimulator

# Use the local simulator device
device = LocalSimulator()

# Define a bell circuit and only measure
circuit = Circuit().h(0).cnot(0, 1).measure(0)

# Run the circuit
task = device.run(circuit, shots=10)

# Get the results
result = task.result()

# Print the measurement counts for qubit 0
print(result.measurement_counts)
```

Quantum task batching

Quantum task batching is available on every Amazon Braket device, except the local simulator. Batching is especially useful for quantum tasks you run on the on-demand simulators (TN1 or SV1) because they can process multiple quantum tasks in parallel. To help you set up various quantum tasks, Amazon Braket provides example notebooks.

Batching allows you to launch quantum tasks in parallel. For example, if you wish to make a calculation that requires 10 quantum tasks and the circuits in those quantum tasks are independent of each other, it is a good idea to use batching. That way, you don't have to wait for one quantum task to be complete before another task begins.

The following example shows how to run a batch of quantum tasks:

```
circuits = [bell for _ in range(5)]
batch = device.run_batch(circuits, s3_folder, shots=100)
print(batch.results()[0].measurement_counts)  # The result of the first quantum task in
 the batch
```

For more information, see the Amazon Braket examples on GitHub or Quantum task batching, which has more specific information about batching.

In this section:

- About quantum task batching and costs
- Quantum task batching and PennyLane
- Task batching and parametrized circuits

About quantum task batching and costs

A few caveats to keep in mind regarding quantum task batching and billing costs:

- By default, quantum task batching retries all time out or fail quantum tasks 3 times.
- A batch of long running quantum tasks, such as 34 qubits for SV1, can incur large costs. Be sure to double check the `run_batch` assignment values carefully before you start a batch of quantum tasks. We do not recommend using TN1 with `run_batch`.
- TN1 can incur costs for failed rehearsal phase tasks (see the TN1 description for more information). Automatic retries can add to the cost and so we recommend setting the number of 'max_retries' on batching to 0 when using TN1 (see Quantum Task Batching, Line 186).

Quantum task batching and PennyLane

Take advantage of batching when you're using PennyLane on Amazon Braket by setting `parallel = True` when you instantiate an Amazon Braket device, as shown in the following example.

```
device = qml.device("braket.aws.qubit",device_arn="arn:aws:braket:::device/quantum-simulator/amazon/sv1",wires=wires,s3_destination_folder=s3_folder,parallel=True,)
```

For more information about batching with PennyLane, see Parallelized optimization of quantum circuits.

Task batching and parametrized circuits

When submitting a quantum task batch that contains parametrized circuits, you can either provide an `inputs` dictionary, which is used for all quantum tasks in the batch, or a `list` of input dictionaries, in which case the `i`-th dictionary is paired with the `i`-th task, as shown in the following example.

```
from braket.circuits import Circuit, FreeParameter, Observable
from braket.aws import AwsQuantumTaskBatch

# create the free parameters
alpha = FreeParameter('alpha')
beta = FreeParameter('beta')

# create two circuits
circ_a = Circuit().rx(0, alpha).ry(1, alpha).cnot(0,2).xx(0, 2, beta)
circ_a.variance(observable=Observable.Z(), target=0)

circ_b = Circuit().rx(0, alpha).rz(1, alpha).cnot(0,2).zz(0, 2, beta)
circ_b.expectation(observable=Observable.Z(), target=2)

# use the same inputs for both circuits in one batch

tasks = device.run_batch([circ_a, circ_b], inputs={'alpha': 0.1, 'beta':0.2})

# or provide each task its own set of inputs

inputs_list = [{'alpha': 0.3,'beta':0.1}, {'alpha': 0.1,'beta':0.4}]

tasks = device.run_batch([circ_a, circ_b], inputs=inputs_list)
```

You can also prepare a list of input dictionaries for a single parametric circuit and submit them as a quantum task batch. If there is N input dictionaries in the list, the batch contains N quantum task. The `i`-th quantum task corresponds to the circuit executed with `i`-th input dictionary.

```
from braket.circuits import Circuit, FreeParameter

# create a parametric circuit
circ = Circuit().rx(0, FreeParameter('alpha'))

# provide a list of inputs to execute with the circuit
inputs_list = [{'alpha': 0.1}, {'alpha': 0.2}, {'alpha': 0.3}]

tasks = device.run_batch(circ, inputs=inputs_list)
```

Local quantum device emulator

With Amazon Braket's local emulator tool, you can emulate your verbatim quantum programs locally before running them on actual quantum hardware. The emulator uses device calibration data to validate the verbatim circuits, allowing you to catch compatibility issues earlier.

Additionally, the local emulator simulates quantum hardware noise through the following process:

- Using device calibration data to construct the noise model
- Applying depolarizing noise to each gate in your circuit
- Applying readout error at the end of your circuit
- Simulating the noisy circuit with a local density matrix simulator

For more information on using a local emulator, see Local emulation for verbatim circuits on Amazon Braket in the amazon-braket-examples GitHub repository.

In this section:

- Benefits of local emulation
- Create a local emulator

Benefits of local emulation

- Validate verbatim circuits against device constraints using real-time or historical calibration data.

- Debug issues before submitting tasks to quantum hardware.
- Compare noiseless and noisy emulations with hardware results to understand noise effects.
- Streamline the workflow of developing noise aware quantum algorithms.

Create a local emulator

A local quantum device emulator can be created directly from a quantum device or a set of device properties. When directly emulating a device, the emulator uses the most recent calibration data from the instantiated device. The following code example demonstrates how to directly emulate Rigetti's Ankaa-3 device.

```
from braket.aws.aws_device import AwsDevice

ankaa3 = AwsDevice("arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3")
ankaa3_emulator = ankaa3.emulator()
```

The following example shows creating a local device emulator from a set of Ankaa-3 device properties in the JSON format.

```
from braket.aws import AwsDevice
from braket.emulation.local_emulator import LocalEmulator
import json

# Instantiate the device
ankaa3 = AwsDevice("arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3")
ankaa3_properties = ankaa3.properties

# Put the Ankaa-3 properties in a file named ankaa3_device_properties.json
with open("ankaa3_device_properties.json", "w") as f:
    json.dump(ankaa3_properties.json(), f)

# Load the json into the ankaa3_data_json variable
with open("ankaa3_device_properties.json", "r") as json_file:
    ankaa3_data_json = json.load(json_file)

# Create the Ankaa-3 local emulator from the json file you created
ankaa3_emulator = LocalEmulator.from_json(ankaa3_data_json)
```

You can customize the example by:

- Using properties from a different QPU device
- Specifying a different JSON file for the emulator
- Changing the values of the device properties before instantiating the emulator

Working with Amazon Braket Hybrid Jobs

This section provides instructions about the basics of creating and running hybrid jobs in Amazon Braket.

You can access hybrid jobs in Braket using:

- The Amazon Braket Python SDK.
- The Amazon Braket console.
- The Amazon Braket API.

In this section:

- Running your local code as a hybrid job
- Running a hybrid job with Amazon Braket Hybrid Jobs
- Create your first Hybrid Job
- Saving your job results
- Saving and restarting hybrid jobs using checkpoints
- Building and debugging a hybrid job with local mode

Running your local code as a hybrid job

Amazon Braket Hybrid Jobs provides a fully managed orchestration of hybrid quantum-classical algorithms, combining Amazon EC2 compute resources with Amazon Braket Quantum Processing Unit (QPU) access. Quantum tasks created in a hybrid job have priority queueing over individual quantum tasks so that your algorithms won't be interrupted by fluctuations in the quantum task queue. Each QPU maintains a separate hybrid jobs queue, ensuring that only one hybrid job can run at any given time.

In this section:

- Create a hybrid job from local Python code

- Install additional Python packages and source code
- Save and load data into a hybrid job instance
- Best practices for hybrid job decorators

Create a hybrid job from local Python code

You can run your local Python code as an Amazon Braket Hybrid Job. You can do this by annotating your code with an `@hybrid_job` decorator, as shown in the following code example. For custom environments, you can opt to use a custom container from Amazon Elastic Container Registry (ECR).

> **Note**
>
> Only Python 3.10 is supported by default.

You can use the `@hybrid_job` decorator to annotate a function. Braket transforms the code inside the decorator into a Braket hybrid job algorithm script. The hybrid job then invokes the function inside the decorator on an Amazon EC2 instance. You can monitor the progress of the job with `job.state()` or with the Braket console. The following code example shows how to run a sequence of five states on the State Vector Simulator (SV1) device.

```
from braket.aws import AwsDevice
from braket.circuits import Circuit, FreeParameter, Observable
from braket.devices import Devices
from braket.jobs.hybrid_job import hybrid_job
from braket.jobs.metrics import log_metric

device_arn = Devices.Amazon.SV1


@hybrid_job(device=device_arn)  # choose priority device
def run_hybrid_job(num_tasks=1):
    device = AwsDevice(device_arn)  # declare AwsDevice within the hybrid job

    # create a parametric circuit
    circ = Circuit()
    circ.rx(0, FreeParameter("theta"))
    circ.cnot(0, 1)
```

```
    circ.expectation(observable=Observable.X(), target=0)

    theta = 0.0  # initial parameter

    for i in range(num_tasks):
        task = device.run(circ, shots=100, inputs={"theta": theta})  # input parameters
        exp_val = task.result().values[0]

        theta += exp_val  # modify the parameter (possibly gradient descent)

        log_metric(metric_name="exp_val", value=exp_val, iteration_number=i)

    return {"final_theta": theta, "final_exp_val": exp_val}
```

You create the hybrid job by invoking the function as you would normal Python functions. However, the decorator function returns the hybrid job handle rather than the result of the function. To retrieve the results after it has completed, use `job.result()`.

```
job = run_hybrid_job(num_tasks=1)
result = job.result()
```

The device argument in the `@hybrid_job` decorator specifies the device that the hybrid job has priority access to - in this case, the SV1 simulator. To get QPU priority, you must ensure that the device ARN used within the function matches that specified in the decorator. For convenience, you can use the helper function `get_job_device_arn()` to capture the device ARN declared in `@hybrid_job`.

Note

Each hybrid job has at least a one minute startup time since it creates a containerized environment on Amazon EC2. So for very short workloads, such as a single circuit or a batch of circuits, it may suffice for you to use quantum tasks.

Hyperparameters

The `run_hybrid_job()` function takes the argument `num_tasks` to control the number of quantum tasks created. The hybrid job automatically captures this as a hyperparameter.

Note

Hyperparameters are displayed in the Braket console as strings, that are limited to 2500 characters.

Metrics and logging

Within the `run_hybrid_job()` function, metrics from iterative algorithms are recorded with `log_metrics`. Metrics are automatically plotted in the Braket console page under the hybrid job tab. You can use metrics to track the quantum task costs in near-real time during the hybrid job run with the Braket cost tracker. The example above uses the metric name “probability” that records the first probability from the result type.

Retrieving results

After the hybrid job has completed, you use `job.result()` to retrieve the hybrid jobs results. Any objects in the return statement are automatically captured by Braket. Note that the objects returned by the function must be a tuple with each element being serializable. For example, the following code shows a working, and a failing example.

```
@hybrid_job(device=Devices.Amazon.SV1)
def passing():
    np_array = np.random.rand(5)
    return np_array  # serializable

@hybrid_job(device=Devices.Amazon.SV1)
def failing():
    return MyObject() # not serializable
```

Job name

By default, the name for this hybrid job is inferred from the function name. You may also specify a custom name up to 50 characters long. For example, in the following code the job name is "my-job-name".

```
@hybrid_job(device=Devices.Amazon.SV1, job_name="my-job-name")
def function():
    pass
```

Local mode

Local jobs are be created by adding the argument `local=True` to the decorator. This runs the hybrid job in a containerized environment on your local compute environment, such as your laptop. Local jobs **do not** have priority queueing for quantum tasks. For advanced cases such as multi-node or MPI, local jobs may have access to the required Braket environment variables. The following code creates a local hybrid job with the device as the SV1 simulator.

```
@hybrid_job(device=Devices.Amazon.SV1, local=True)
def run_hybrid_job(num_tasks = 1):
    return ...
```

All other hybrid job options are supported. For a list of options see the braket.jobs.quantum_job_creation module.

Install additional Python packages and source code

You can customize your runtime environment to use your preferred Python packages. You can use either a `requirements.txt` file, a list of package names, or bring your own container (BYOC). To customize a runtime environment using a `requirements.txt` file, refer to the following code example.

```
@hybrid_job(device=Devices.Amazon.SV1, dependencies="requirements.txt")
def run_hybrid_job(num_tasks = 1):
    return ...
```

For example, the `requirements.txt` file may include other packages to install.

```
qiskit
pennylane >= 0.31
mitiq == 0.29
```

Alternatively, you may supply the package names as a Python list as follows.

```
@hybrid_job(device=Devices.Amazon.SV1, dependencies=["qiskit", "pennylane>=0.31",
 "mitiq==0.29"])
def run_hybrid_job(num_tasks = 1):
    return ...
```

Additional source code can be specified either as a list of modules, or a single module as in the following code example.

```
@hybrid_job(device=Devices.Amazon.SV1, include_modules=["my_module1", "my_module2"])
def run_hybrid_job(num_tasks = 1):
    return ...
```

Save and load data into a hybrid job instance

Specifying input training data

When you create a hybrid job, you may provide an input training datasets by specifying an Amazon Simple Storage Service (Amazon S3) bucket. You may also specify a local path, then Braket automatically uploads the data to Amazon S3 at `s3://<default_bucket_name>/jobs/<job_name>/<timestamp>/data/<channel_name>`. If you specify a local path, the channel name defaults to “input”. The following code shows a numpy file from the local path `data/file.npy`.

```
@hybrid_job(device=Devices.Amazon.SV1, input_data="data/file.npy")
def run_hybrid_job(num_tasks = 1):
    data = np.load("data/file.npy")
    return ...
```

For S3, you must use the `get_input_data_dir()` helper funciton.

```
s3_path = "s3://amazon-braket-us-west-1-961591465522/job-data/file.npy"

@hybrid_job(device=None, input_data=s3_path)
def job_s3_input():
    np.load(get_input_data_dir() + "/file.npy")

@hybrid_job(device=None, input_data={"channel": s3_path})
def job_s3_input_channel():
    np.load(get_input_data_dir("channel") + "/file.npy")
```

You can specify multiple input data sources by providing a dictionary of channel values and S3 URIs or local paths.

```
input_data = {
    "input": "data/file.npy",
    "input_2": "s3://amzn-s3-demo-bucket/data.json"
```

```
}


@hybrid_job(device=None, input_data=input_data)
def multiple_input_job():
    np.load(get_input_data_dir("input") + "/file.npy")
    np.load(get_input_data_dir("input_2") + "/data.json")
```

> **Note**
>
> When the input data is large (>1GB), there is a long wait time before the job is created. This is due to the local input data when it is first uploaded to an S3 bucket, then the S3 path is added to the job request. Finally, the job request is submitted to the Braket service.

Saving results to S3

To save results not included in the return statement of the decorated function, you must append the correct directory to all file writing operations. The following example, shows saving a numpy array and a matplotlib figure.

```
@hybrid_job(device=Devices.Amazon.SV1)
def run_hybrid_job(num_tasks = 1):
    result = np.random.rand(5)

    # save a numpy array
    np.save("result.npy", result)

    # save a matplotlib figure
    plt.plot(result)
    plt.savefig("fig.png")
    return ...
```

All results are compressed into a file named `model.tar.gz`. You can download the results with the Python function `job.result()`, or by navigating to the results folder from the hybrid job page in the Braket management console.

Saving and resuming from checkpoints

For long-running hybrid jobs, its recommended to periodically save the intermediate state of the algorithm. You can use the built-in `save_job_checkpoint()` helper function, or save files to the `AMZN_BRAKET_JOB_RESULTS_DIR` path. The later is available with the helper function `get_job_results_dir()`.

The following is a minimal working example for saving and loading checkpoints with a hybrid job decorator:

```
from braket.jobs import save_job_checkpoint, load_job_checkpoint, hybrid_job

@hybrid_job(device=None, wait_until_complete=True)
def function():
    save_job_checkpoint({"a": 1})

job = function()
job_name = job.name
job_arn = job.arn

@hybrid_job(device=None, wait_until_complete=True, copy_checkpoints_from_job=job_arn)
def continued_function():
    load_job_checkpoint(job_name)

continued_job = continued_function()
```

In the first hybrid job, `save_job_checkpoint()` is called with a dictionary containing the data we want to save. By default, every value must be serializable as text. For checkpointing more complex Python objects, such as numpy arrays, you can set `data_format = PersistedJobDataFormat.PICKLED_V4`. This code creates and overwrites a checkpoint file with default name `<jobname>.json` in your hybrid job artifacts under a subfolder called "checkpoints".

To create a new hybrid job to continue from the checkpoint, we need to pass `copy_checkpoints_from_job=job_arn` where `job_arn` is the hybrid job ARN of the previous job. Then we use `load_job_checkpoint(job_name)` to load from the checkpoint.

Best practices for hybrid job decorators

Embrace asynchronicity

Hybrid jobs created with the decorator annotation are asynchronous - they run once the classical and quantum resources are available. You monitor the progress of the algorithm using the Braket Management Console or Amazon CloudWatch. When you submit your algorithm to run, Braket runs your algorithm in a scalable containerized environment and results are retrieved when the algorithm is complete.

Run iterative variational algorithms

Hybrid jobs gives you the tools to run iterative quantum-classical algorithms. For purely quantum problems, use quantum tasks or a batch of quantum tasks. The priority access to certain QPUs is most beneficial for long-running variational algorithms requiring multiple iterative calls to the QPUs with classical processing in between.

Debug using local mode

Before you run a hybrid job on a QPU, its recommended to first run on the simulator SV1 to confirm it runs as expected. For small scale tests, you can run with local mode for rapid iteration and debugging.

Improve reproducibility with Bring your own container (BYOC)

Create a reproducible experiment by encapsulating your software and its dependencies within a containerized environment. By packaging all your code, dependencies, and settings in a container, you prevent potential conflicts and versioning issues.

Multi-instance distributed simulators

To run a large number of circuits, consider using built-in MPI support to run local simulators on multiple instances within a single hybrid job. For more information, see embedded simulators.

Use parametric circuits

Parametric circuits that you submit from a hybrid job are automatically compiled on certain QPUs using parametric compilation to improve the runtimes of your algorithms.

Checkpoint periodically

For long-running hybrid jobs, its recommended to periodically save the intermediate state of the algorithm.

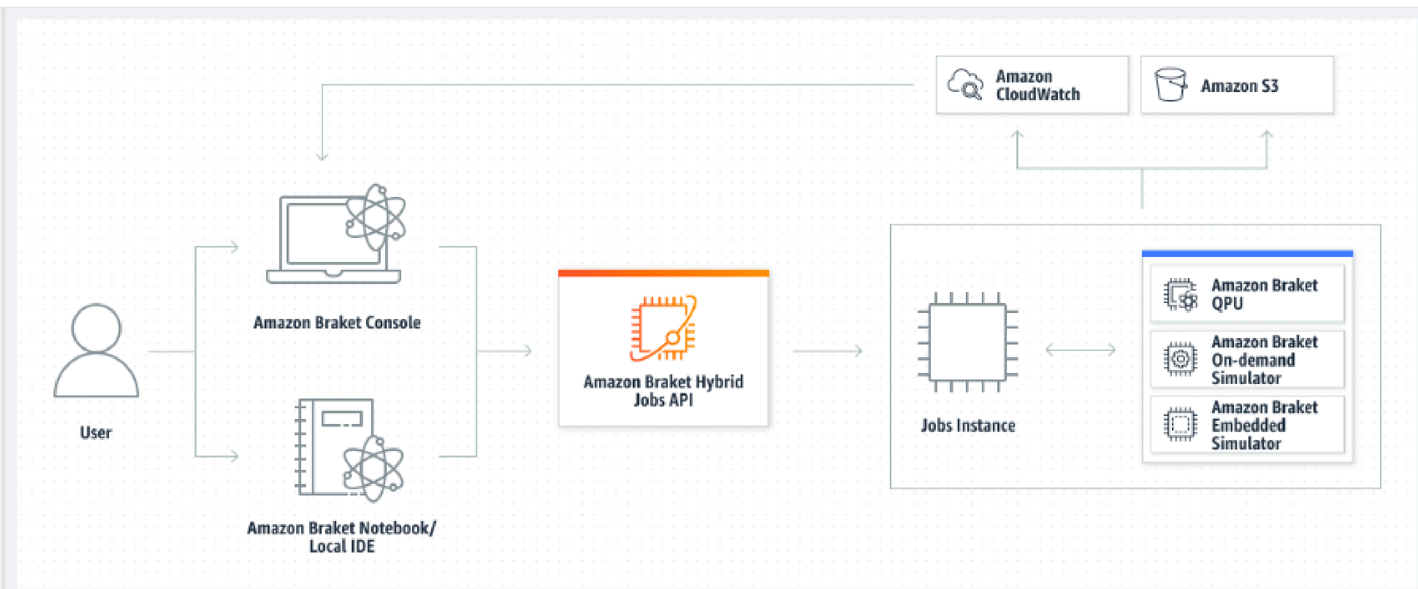
For further examples, use cases, and best-practices, see Amazon Braket examples GitHub.

Running a hybrid job with Amazon Braket Hybrid Jobs

To run a hybrid job with Amazon Braket Hybrid Jobs, you first need to define your algorithm. You can define it by writing the *algorithm script* and, optionally, other dependency files using the

Amazon Braket Python SDK or PennyLane. If you want to use other (open source or proprietary) libraries, you can define your own custom container image using Docker, which includes these libraries. For more information, see Bring your own container (BYOC).

In either case, next you create a hybrid job using the Amazon Braket API, where you provide your algorithm script or container, select the target quantum device the hybrid job is to use, and then choose from a variety of optional settings. The default values provided for these optional settings work for the majority of use cases. For the target device to run your Hybrid Job, you have a choice between a QPU, an on-demand simulator (such as SV1, DM1 or TN1), or the classical hybrid job instance itself. With an on-demand simulator or QPU, your hybrid jobs container makes API calls to a remote device. With the embedded simulators, the simulator is embedded in the same container as your algorithm script. The lightning simulators from PennyLane are embedded with the default pre-built hybrid jobs container for you to use. If you run your code using an embedded PennyLane simulator or a custom simulator, you can specify an instance type as well as how many instances you wish to use. Refer to the Amazon Braket Pricing page for the costs associated with each choice.



If your target device is an on-demand or embedded simulator, Amazon Braket starts running the hybrid job right away. It spins up the hybrid job instance (you can customize the instance type in the API call), runs your algorithm, writes the results to Amazon S3, and releases your resources. This release of resources ensures that you only pay for what you use.

The total number of concurrent hybrid jobs per quantum processing unit (QPU) is restricted. Today, only one hybrid job can run on a QPU at any given time. Queues are used to control the number of hybrid jobs allowed to run so as not to exceed the limit allowed. If your target device is a QPU, your hybrid job first enters the *job queue* of the selected QPU. Amazon Braket spins up the hybrid job instance needed and runs your hybrid job on the device. For the duration of your algorithm, your hybrid job has priority access, meaning that quantum tasks from your hybrid job run ahead of other Braket quantum tasks queued up on the device, provided the job quantum tasks are submitted to the QPU once every few minutes. Once your hybrid job is complete, resources are released, meaning you only pay for what you use.

> **Note**
>
> Devices are regional and your hybrid job runs in the same AWS Region as your primary device.

In both the simulator and QPU target scenarios, you have the option to define custom algorithm metrics, such as the energy of your Hamiltonian, as part of your algorithm. These metrics are automatically reported to Amazon CloudWatch and from there, they display in near real-time in the Amazon Braket console.

> **Note**
>
> If you wish to use a GPU based instance, be sure to use one of the GPU-based simulators available with the embedded simulators on Braket (for example, `lightning.gpu`). If you choose one of the CPU-based embedded simulators (for example, `lightning.qubit`, or `braket:default-simulator`), the GPU will not be used and you may incur unnecessary costs.

Create your first Hybrid Job

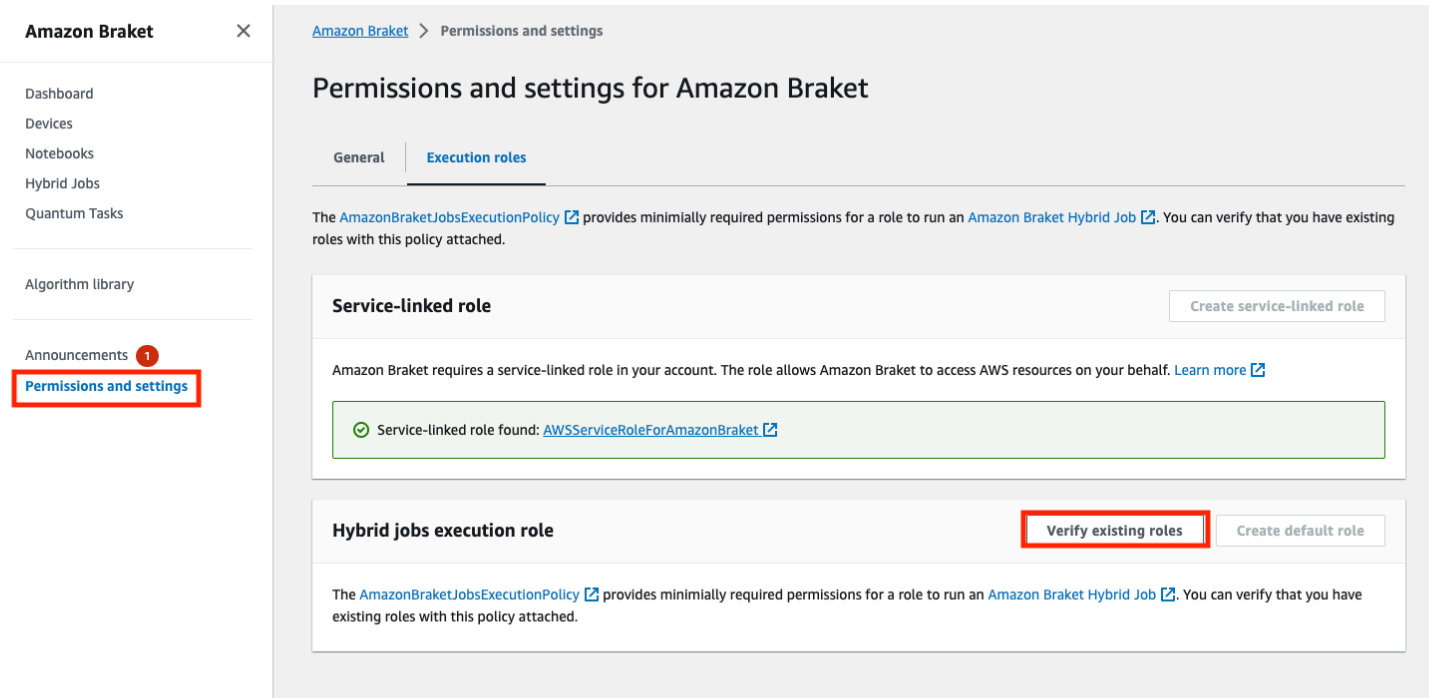
This section shows you how to create a Hybrid Job using a Python script. Alternatively, to create a hybrid job from local Python code, such as your preferred integrated development environment (IDE) or a Braket notebook, see Running your local code as a hybrid job.

In this section:

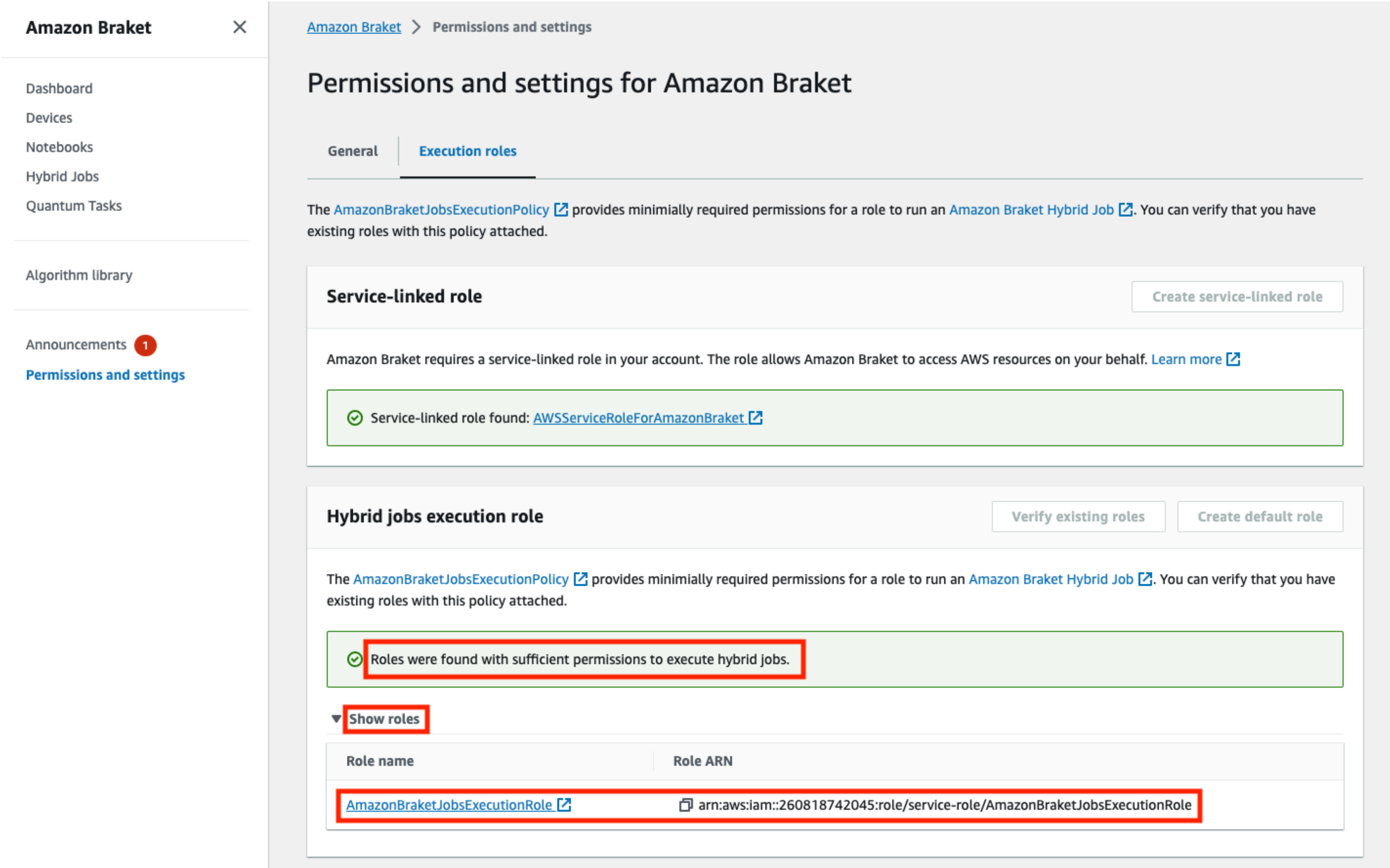
- Set permissions
- Create and run
- Monitor results

Set permissions

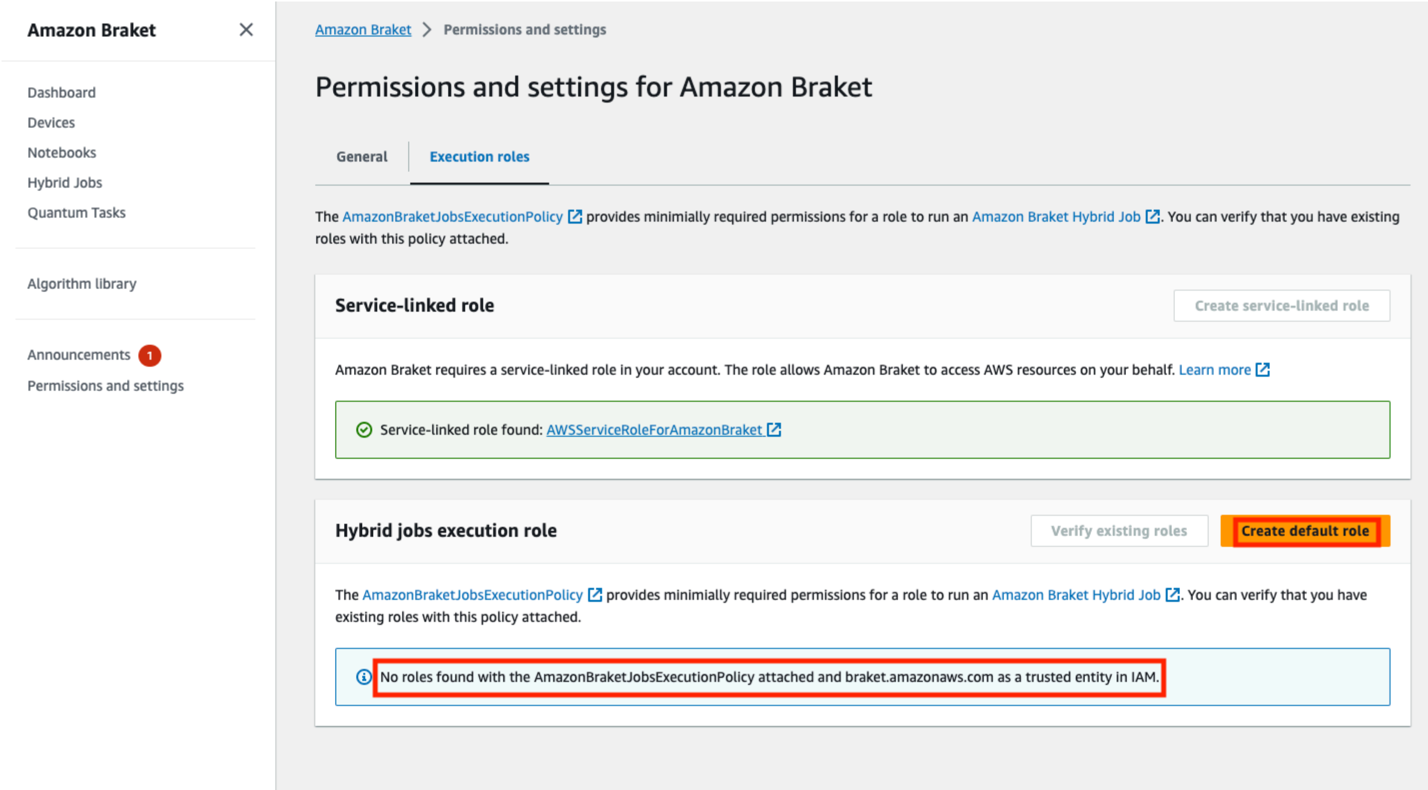
Before you run your first hybrid job, you must ensure that you have sufficient permissions to proceed with this task. To determine that you have the correct permissions, select **Permissions** from the menu on left side of the Braket Console. The **Permissions management for Amazon Braket** page helps you verify whether one of your existing roles has permissions that are sufficient to run your hybrid job or guides you through the creation of a default role that can be used to run your hybrid job if you do not already have such a role.



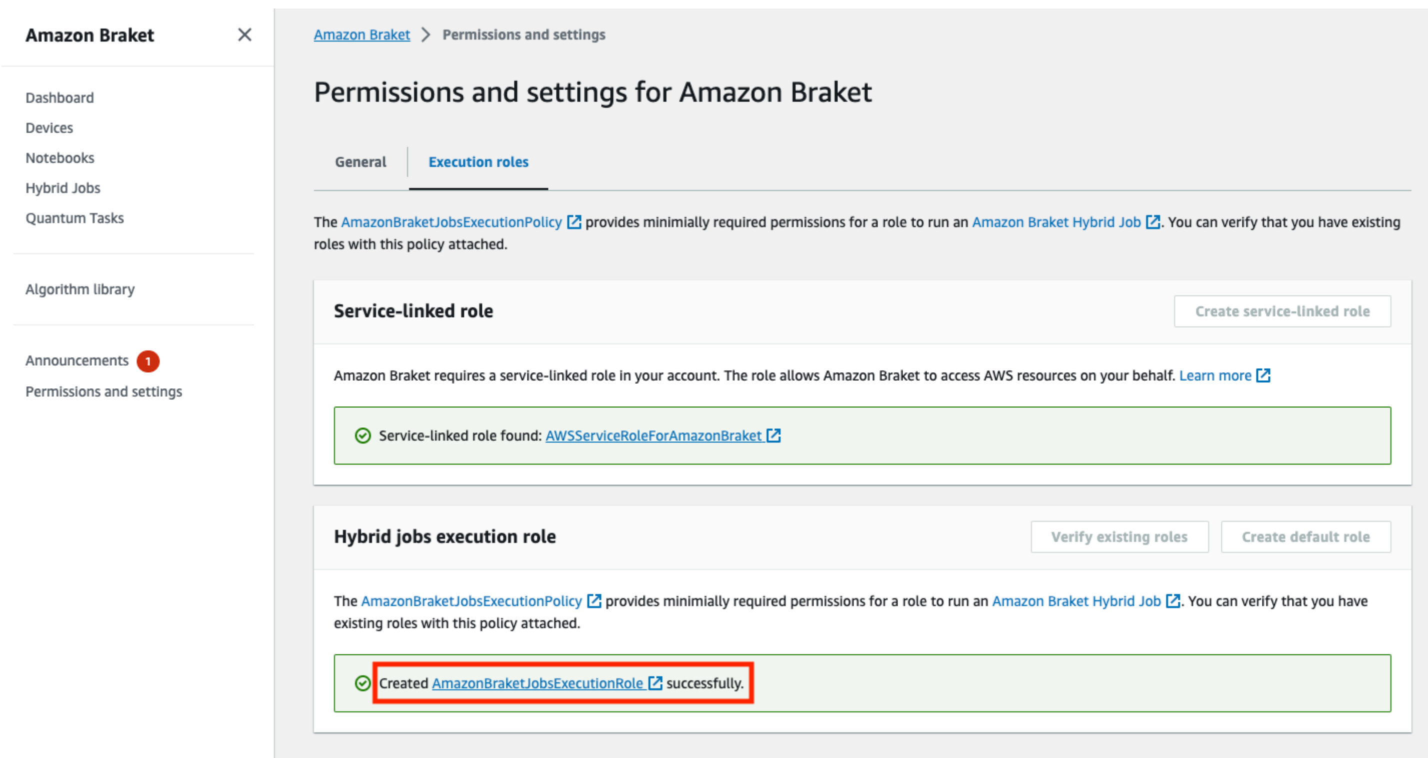
To verify that you have roles with sufficient permissions to run a hybrid job, select the **Verify existing role** button. If you do, you get a message that the roles were found. To see the names of the roles and their role ARNs, select the **Show roles** button.



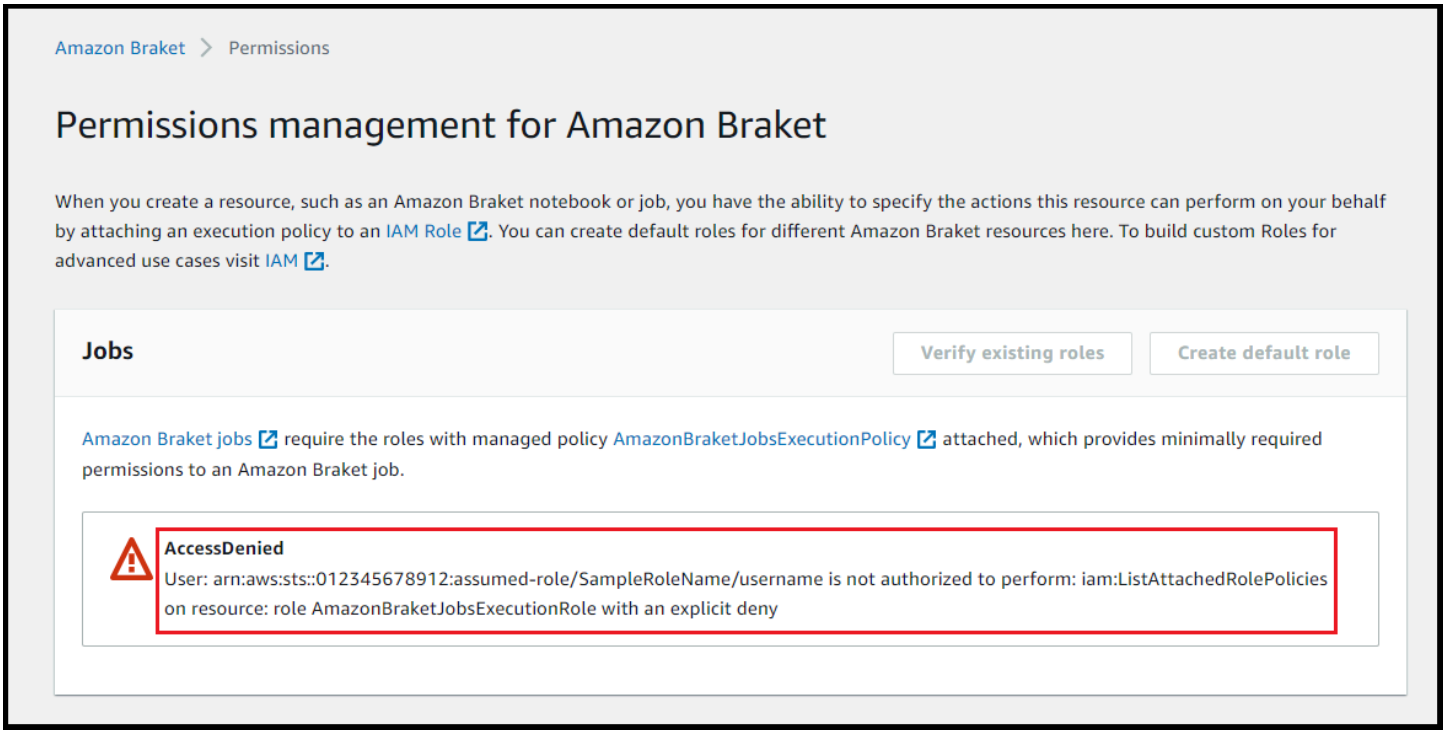
If you do not have a role with sufficient permissions to run a hybrid job, you get a message that no such role was found. Select the **Create default role** button to obtain a role with sufficient permissions.



If the role was created successfully, you get a message confirming this.

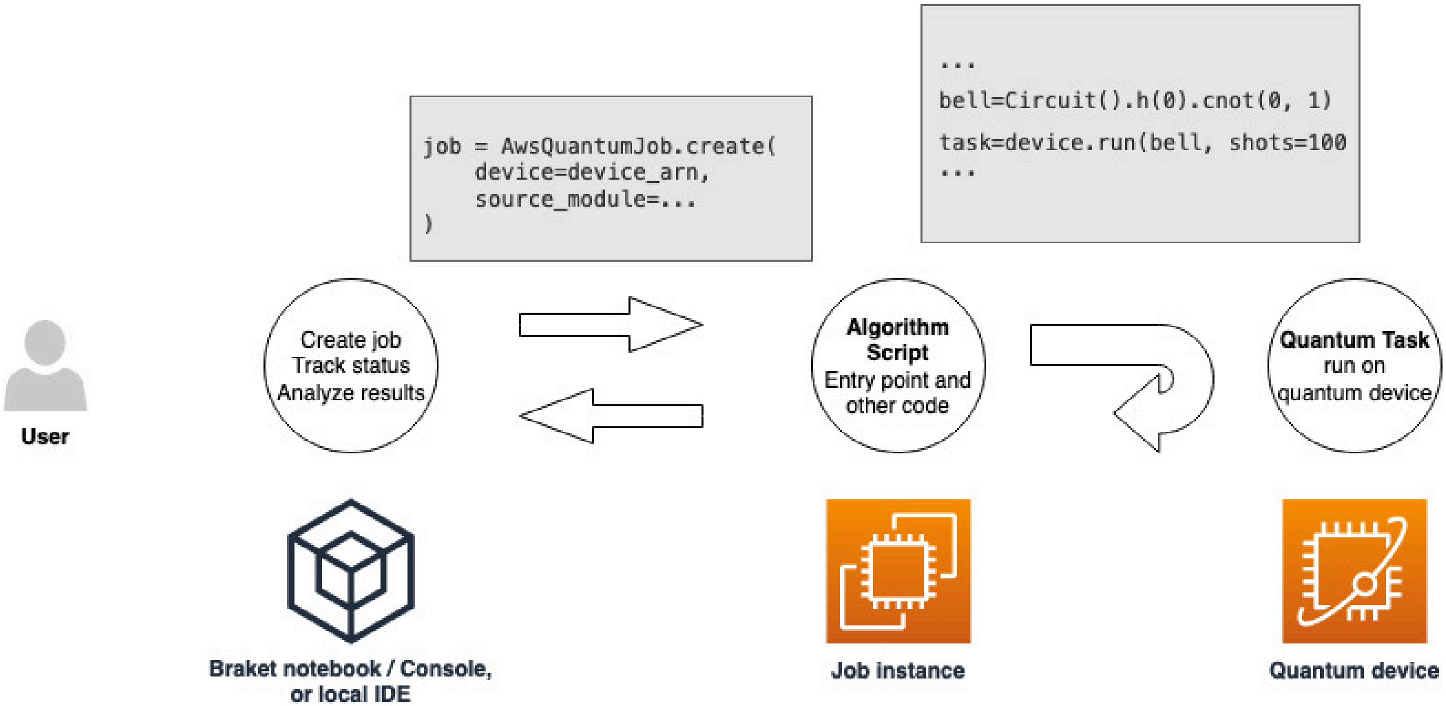


If you do not have permissions to make this inquiry, you will be denied access. In this case, contact your internal AWS administrator.



Create and run

Once you have a role with permissions to run a hybrid job, you are ready to proceed. The key piece of your first Braket hybrid job is the *algorithm script*. It defines the algorithm you want to run and contains the classical logic and quantum tasks that are part of your algorithm. In addition to your algorithm script, you can provide other dependency files. The algorithm script together with its dependencies is called the *source module*. The *entry point* defines the first file or function to run in your source module when the hybrid job starts.



First, consider the following basic example of an algorithm script that creates five bell states and prints the corresponding measurement results.

```
import os

from braket.aws import AwsDevice
from braket.circuits import Circuit

def start_here():

    print("Test job started!")

    # Use the device declared in the job script
    device = AwsDevice(os.environ["AMZN_BRAKET_DEVICE_ARN"])

    bell = Circuit().h(0).cnot(0, 1)
    for count in range(5):
        task = device.run(bell, shots=100)
        print(task.result().measurement_counts)

    print("Test job completed!")
```

Save this file with the name *algorithm_script.py* in your current working directory on your Braket notebook or local environment. The algorithm_script.py file has `start_here()` as the planned entry point.

Next, create a Python file or Python notebook in the same directory as the algorithm_script.py file. This script kicks off the hybrid job and handles any asynchronous processing, such as printing the status or key outcomes that we are interested in. At a minimum, this script needs to specify your hybrid job script and your primary device.

> **Note**
>
> For more information about how to create a Braket notebook or upload a file, such as the *algorithm_script.py* file, in the same directory as the notebooks, see Run your first circuit using the Amazon Braket Python SDK

For this basic first case, you target a simulator. Whichever type of quantum device you target, a simulator or an actual quantum processing unit (QPU), the device you specify with `device` in the following script is used to schedule the hybrid job and is available to the algorithm scripts as the environment variable `AMZN_BRAKET_DEVICE_ARN`.

> **Note**
>
> You can only use devices that are available in the AWS Region of your hybrid job. The Amazon Braket SDK auto selects this AWS Region. For example, a hybrid job in us-east-1 can use IonQ, SV1, DM1, and TN1 devices, but not Rigetti devices.

If you choose a quantum computer instead of a simulator, Braket schedules your hybrid jobs to run all of their quantum tasks with priority access.

```
from braket.aws import AwsQuantumJob
from braket.devices import Devices

job = AwsQuantumJob.create(
    Devices.Amazon.SV1,
    source_module="algorithm_script.py",
    entry_point="algorithm_script:start_here",
    wait_until_complete=True
```

```
)
```

The parameter `wait_until_complete=True` sets a verbose mode so that your job prints output from the actual job as it's running. You should see an output similar to the following example.

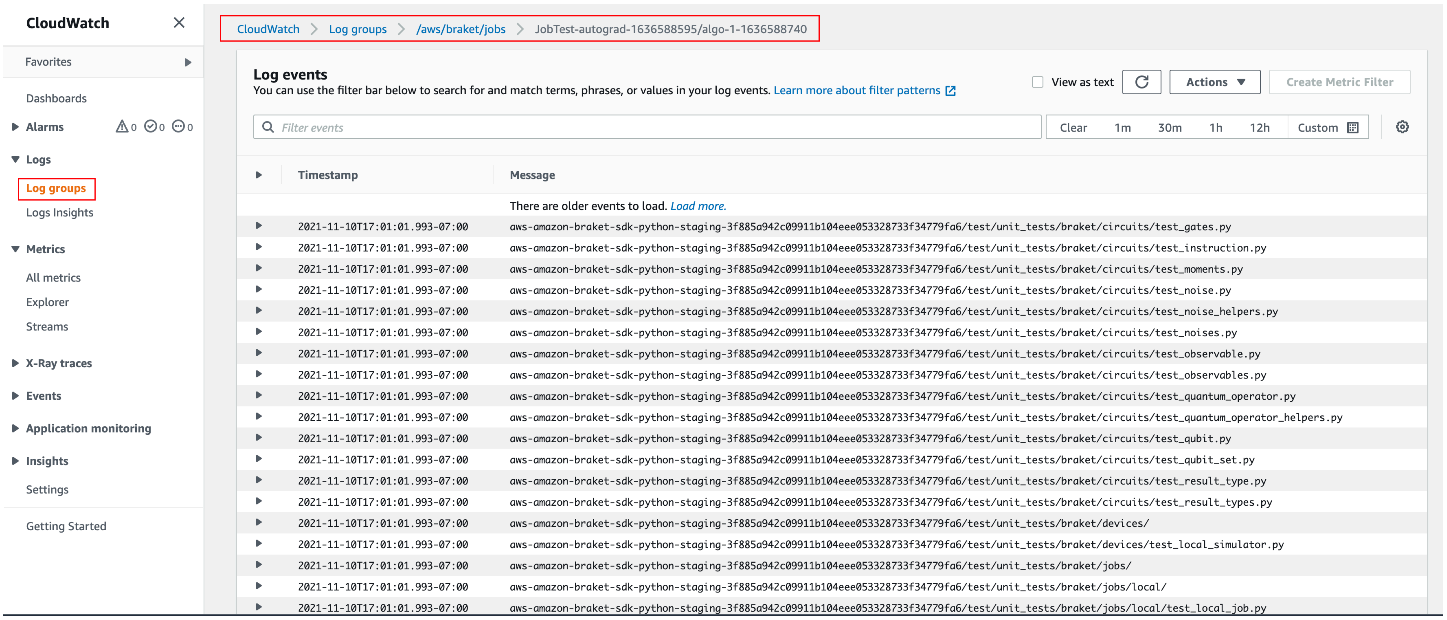
```
job = AwsQuantumJob.create(
    Devices.Amazon.SV1,
    source_module="algorithm_script.py",
    entry_point="algorithm_script:start_here",
    wait_until_complete=True,
)
Initializing Braket Job: arn:aws:braket:us-west-2:<accountid>:job/<UUID>
.........................................
.
.
.

Completed 36.1 KiB/36.1 KiB (692.1 KiB/s) with 1 file(s) remaining#015download:
 s3://braket-external-assets-preview-us-west-2/HybridJobsAccess/models/
braket-2019-09-01.normal.json to ../../braket/additional_lib/original/
braket-2019-09-01.normal.json
Running Code As Process
Test job started!!!!!
Counter({'00': 55, '11': 45})
Counter({'11': 59, '00': 41})
Counter({'00': 55, '11': 45})
Counter({'00': 58, '11': 42})
Counter({'00': 55, '11': 45})
Test job completed!!!!!
Code Run Finished
2021-09-17 21:48:05,544 sagemaker-training-toolkit INFO     Reporting training SUCCESS
```

Note

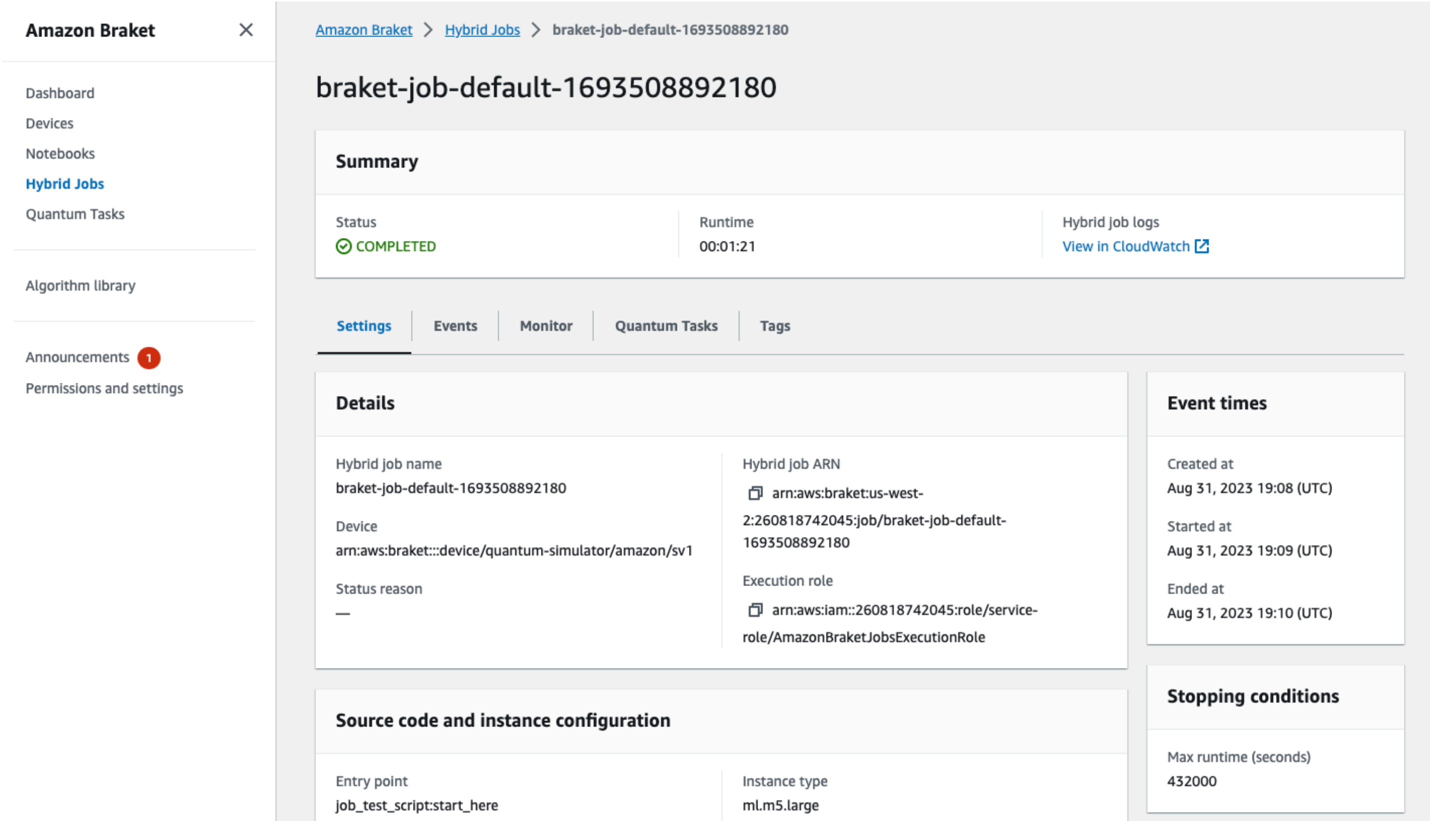
You can also use your custom-made module with the [AwsQuantumJob.create](https://amazon-braket-sdk-python.readthedocs.io/en/latest/_apidoc/braket.aws.aws_quantum_job.html#braket.aws.aws_quantum_job.AwsQuantumJob.create) method by passing its location (either the path to a local directory or file, or an S3 URI of a tar.gz file). For a working example, see Parallelize_training_for_QML.ipynb file in the hybrid jobs folder in the Amazon Braket examples Github repo.

Monitor results

Alternatively, you can access the log output from Amazon CloudWatch. To do this, go to the **Log groups** tab on the left menu of the job detail page, select the log group `aws/braket/jobs`, and then choose the log stream that contains the job name. In the example above, this is `braket-job-default-1631915042705/algo-1-1631915190`.



You can also view the status of the hybrid job in the console by selecting the **Hybrid Jobs** page and then choose **Settings**.



Your hybrid job produces some artifacts in Amazon S3 while it runs. The default S3 bucket name is amazon-braket-<region>-<accountid> and the content is in the jobs/<jobname>/<timestamp> directory. You can configure the S3 locations where these artifacts are stored by specifying a different code_location when the hybrid job is created with the Braket Python SDK.

Note

This S3 bucket must be located in the same AWS Region as your job script.

The jobs/<jobname>/<timestamp> directory contains a subfolder with the output from the entry point script in a model.tar.gz file. There is also a directory called script that contains your algorithm script artifacts in a source.tar.gz file. The results from your actual quantum tasks are in the directory named jobs/<jobname>/tasks.

Saving your job results

You can save the results generated by the algorithm script so that they are available from the hybrid job object in the hybrid job script as well as from the output folder in Amazon S3 (in a tar-zipped file named model.tar.gz).

The output must be saved in a file using a JavaScript Object Notation (JSON) format. If the data can not be readily serialized to text, as in the case of a numpy array, you could pass in an option to serialize using a pickled data format. See the braket.jobs.data_persistence module for more details.

To save the results of the hybrid jobs, you add the following lines commented with #ADD to the algorithm script.

```
from braket.aws import AwsDevice
from braket.circuits import Circuit
from braket.jobs import save_job_result #ADD

def start_here():

    print("Test job started!!!!!")

    device = AwsDevice(os.environ['AMZN_BRAKET_DEVICE_ARN'])

    results = []  #ADD

    bell = Circuit().h(0).cnot(0, 1)
    for count in range(5):
        task = device.run(bell, shots=100)
        print(task.result().measurement_counts)
        results.append(task.result().measurement_counts)  #ADD

        save_job_result({ "measurement_counts": results })  #ADD

    print("Test job completed!!!!!")
```

You can then display the results of the job from your job script by appending the line **`print(job.result())`** commented with #ADD.

```
import time
from braket.aws import AwsQuantumJob
```

```
job = AwsQuantumJob.create(
    source_module="algorithm_script.py",
    entry_point="algorithm_script:start_here",
    device_arn="arn:aws:braket:::device/quantum-simulator/amazon/sv1",
)

print(job.arn)
while job.state() not in AwsQuantumJob.TERMINAL_STATES:
    print(job.state())
    time.sleep(10)

print(job.state())
print(job.result())   #ADD
```

In this example, we have removed `wait_until_complete=True` to suppress verbose output. You can add it back in for debugging. When you run this hybrid job, it outputs the identifier and the `job-arn`, followed by the state of the hybrid job every 10 seconds until the hybrid job is `COMPLETED`, after which it shows you the results of the bell circuit. See the following example.

```
arn:aws:braket:us-west-2:111122223333:job/braket-job-default-1234567890123
INITIALIZED
RUNNING
RUNNING
RUNNING
RUNNING
RUNNING
RUNNING
RUNNING
RUNNING
RUNNING
RUNNING
...
RUNNING
RUNNING
COMPLETED
{'measurement_counts': [{'11': 53, '00': 47},..., {'00': 51, '11': 49}]}
```

Saving and restarting hybrid jobs using checkpoints

You can save intermediate iterations of your hybrid jobs using checkpoints. In the algorithm script example from the previous section, you would add the following lines commented with #ADD to create checkpoint files.

```
from braket.aws import AwsDevice
from braket.circuits import Circuit
from braket.jobs import save_job_checkpoint  #ADD
import os

def start_here():

    print("Test job starts!!!!!")

    device = AwsDevice(os.environ["AMZN_BRAKET_DEVICE_ARN"])

    #ADD the following code
    job_name = os.environ["AMZN_BRAKET_JOB_NAME"]
     save_job_checkpoint(
     checkpoint_data={"data": f"data for checkpoint from {job_name}"},
     checkpoint_file_suffix="checkpoint-1",
     ) #End of ADD

    bell = Circuit().h(0).cnot(0, 1)
    for count in range(5):
        task = device.run(bell, shots=100)
        print(task.result().measurement_counts)

    print("Test hybrid job completed!!!!!")
```

When you run the hybrid job, it creates the file *<jobname>-checkpoint-1.json* in your hybrid job artifacts in the checkpoints directory with a default `/opt/jobs/checkpoints` path. The hybrid job script remains unchanged unless you want to change this default path.

If you want to load a hybrid job from a checkpoint generated by a previous hybrid job, the algorithm script uses `from braket.jobs import load_job_checkpoint`. The logic to load in your algorithm script is as follows.

```
checkpoint_1 = load_job_checkpoint(
    "previous_job_name",
    checkpoint_file_suffix="checkpoint-1",
)
```

After loading this checkpoint, you can continue your logic based on the content loaded to `checkpoint-1`.

Note

The *checkpoint_file_suffix* must match the suffix previously specified when creating the checkpoint.

Your orchestration script needs to specify the `job-arn` from the previous hybrid job with the line commented with #ADD.

```
job = AwsQuantumJob.create(
    source_module="source_dir",
    entry_point="source_dir.algorithm_script:start_here",
    device_arn="arn:aws:braket:::device/quantum-simulator/amazon/sv1",
    copy_checkpoints_from_job="<previous-job-ARN>", #ADD
    )
```

Building and debugging a hybrid job with local mode

When you are building a new hybrid algorithm, local mode helps you to debug and test your algorithm script. Local mode is a feature that allows you to run code you plan to use in Amazon Braket Hybrid Jobs, but without needing Braket to manage the infrastructure for running the hybrid job. Instead, run hybrid jobs locally on your Amazon Braket Notebook instance or on a preferred client, such as a laptop or desktop computer.

In local mode, you can still send quantum tasks to actual devices, but you do not get the performance benefits when running against an actual Quantum processing unit (QPU) while in local mode.

To use local mode, modify `AwsQuantumJob` to `LocalQuantumJob` wherever it occurs inside of your program. For instance, to run the example from Create your first hybrid job, edit the hybrid job script in the code as follows.

```
from braket.jobs.local import LocalQuantumJob

job = LocalQuantumJob.create(
    device="arn:aws:braket:::device/quantum-simulator/amazon/sv1",
    source_module="algorithm_script.py",
    entry_point="algorithm_script:start_here",
)
```

Note

Docker, which is already pre-installed in the Amazon Braket notebooks, needs to be installed in your local environment to use this feature. Instructions for installing Docker can be found on the Get Docker page. In addition, not all parameters are supported in local mode.

Running your quantum tasks with Amazon Braket

Braket provides secure, on-demand access to different types of quantum computers. You have access to gate-based quantum computers from IonQ, IQM, and Rigetti, as well as an Analog Hamiltonian Simulator from QuEra. You also have no upfront commitment, and no need to procure access through individual providers.

- The Amazon Braket Console provides device information and status to help you create, manage, and monitor your resources and quantum tasks.
- Submit and run quantum tasks through the Amazon Braket Python SDK, as well as through the console. The SDK is accessible through preconfigured Amazon Braket notebooks.
- The Amazon Braket API is accessible through the Amazon Braket Python SDK and notebooks. You can make calls directly to the API if you are building applications that work with quantum computing programmatically.

The examples throughout this section demonstrate how you can work with the Amazon Braket API directly using the Amazon Braket Python SDK along with the AWS Python SDK for Braket (Boto3).

More about the Amazon Braket Python SDK

To work with the Amazon Braket Python SDK, first install the AWS Python SDK for Braket (Boto3) so that you can communicate with the AWS API. You can think of the Amazon Braket Python SDK as a convenient wrapper around Boto3 for quantum customers.

- Boto3 contains interfaces you need to tap into the AWS API. (Note that Boto3 is a large Python SDK that talks to the AWS API. Most AWS services support a Boto3 interface.)
- The Amazon Braket Python SDK contains software modules for circuits, gates, devices, result types, and other parts of a quantum task. Each time you create a program, you import the modules you need for that quantum task.
- The Amazon Braket Python SDK is accessible through notebooks, which are pre-loaded with all of the modules and dependencies you need for running quantum tasks.
- You can import modules from the Amazon Braket Python SDK into any Python script if you do not wish to work with notebooks.

After you have installed Boto3, an overview of steps for creating a quantum task through the Amazon Braket Python SDK resembles the following:

1. (Optionally) Open your notebook.
2. Import the SDK modules you need for your circuits.
3. Specify a QPU or simulator.
4. Instantiate the circuit.
5. Run the circuit.
6. Collect the results.

The examples in this section show details of each step.

For more examples, see the Amazon Braket Examples repository on GitHub.

In this section:

- Submitting quantum tasks to QPUs
- When will my quantum task run?
- Managing your Amazon Braket Hybrid Job
- Working with reservations
- Error mitigation techniques

Submitting quantum tasks to QPUs

Amazon Braket provides access to several devices that can run quantum tasks. You can submit quantum tasks individually or you can set up quantum task batching.

QPUs

You can submit quantum tasks to QPUs at any time, but the task runs within certain availability windows that are displayed on the **Devices** page of the Amazon Braket console. You can retrieve the results of the quantum task with the quantum task ID, which is introduced in the next section.

- **IonQ Aria-1** : `arn:aws:braket:us-east-1::device/qpu/ionq/Aria-1`
- **IonQ Aria-2** : `arn:aws:braket:us-east-1::device/qpu/ionq/Aria-2`
- **IonQ Forte-1** : `arn:aws:braket:us-east-1::device/qpu/ionq/Forte-1`
- **IonQ Forte-Enterprise-1** : `arn:aws:braket:us-east-1::device/qpu/ionq/Forte-Enterprise-1`
- **IQM Garnet** : `arn:aws:braket:eu-north-1::device/qpu/iqm/Garnet`

- **IQM Emerald** : `arn:aws:braket:eu-north-1::device/qpu/iqm/Emerald`
- **QuEra Aquila** : `arn:aws:braket:us-east-1::device/qpu/quera/Aquila`
- **Rigetti Ankaa-3** : `arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3`

> **Note**
>
> You can cancel quantum tasks in the `CREATED` state for QPUs and on-demand simulators. You can cancel quantum tasks in the `QUEUED` state on a best-effort basis for on-demand simulators and QPUs. Note that QPU `QUEUED` quantum tasks are unlikely to be cancelled successfully during QPU availability windows.

In this section:

- IonQ
- IQM
- Rigetti
- QuEra
- Example: Submitting a quantum task to a QPU
- Inspecting compiled circuits

IonQ

IonQ offers gate-based QPUs based on ion trap technology. IonQ's trapped ion QPUs are built on a chain of trapped 171Yb+ ions that are spatially confined by means of a microfabricated surface electrode trap within a vacuum chamber.

IonQ devices support the following quantum gates.

```
'x', 'y', 'z', 'rx', 'ry', 'rz', 'h', 'cnot', 's', 'si', 't', 'ti', 'v', 'vi', 'xx',
 'yy', 'zz', 'swap'
```

With verbatim compilation, the IonQ QPUs support the following native gates.

```
'gpi', 'gpi2', 'ms'
```

If you only specify two phase parameters when using the native MS gate, a fully- entangling MS gate runs. A fully-entangling MS gate always performs a π/2 rotation. To specify a different angle and run a partially-entangling MS gate, you specify the desired angle by adding a third parameter. For more information, see the braket.circuits.gate module.

These native gates can only be used with verbatim compilation. To learn more about verbatim compilation, see Verbatim Compilation.

IQM

IQM quantum processors are universal gate-model devices based on superconducting transmon qubits. The IQM Garnet is a 20-qubit device, while IQM Emerald is a 54-qubit device. Both these devices use a square lattice topology, also known as a Crystal lattice topology.

The IQM devices support the following quantum gates.

```
"ccnot", "cnot", "cphaseshift", "cphaseshift00", "cphaseshift01", "cphaseshift10",
 "cswap", "swap", "iswap", "pswap", "ecr", "cy", "cz", "xy", "xx", "yy", "zz", "h",
 "i", "phaseshift", "rx", "ry", "rz", "s", "si", "t", "ti", "v", "vi", "x", "y", "z"
```

With verbatim compilation, the IQM devices support the following native gates.

```
'cz', 'prx'
```

Rigetti

Rigetti quantum processors are universal, gate-model machines based on all-tunable superconducting qubits.

- The Ankaa-3 system is an 84-qubit device that utilizes scalable multi-chip technology.

The Rigetti device supports the following quantum gates.

```
'cz', 'xy', 'ccnot', 'cnot', 'cphaseshift', 'cphaseshift00', 'cphaseshift01',
 'cphaseshift10', 'cswap', 'h', 'i', 'iswap', 'phaseshift', 'pswap', 'rx', 'ry', 'rz',
 's', 'si', 'swap', 't', 'ti', 'x', 'y', 'z'
```

With verbatim compilation, Ankaa-3 supports the following native gates.

```
'rx', 'rz', 'iswap'
```

Rigetti superconducting quantum processors can run the 'rx' gate with only the angles of ±π/2 or ±π.

Pulse-level control is available on the Rigetti devices, which support a set of predefined frames of the following types for the Ankaa-3 system.

```
`flux_tx`, `charge_tx`, `readout_rx`, `readout_tx`
```

For more information about these frames, see Roles of frames and ports.

QuEra

QuEra offers neutral-atom based devices that can run Analog Hamiltonian Simulation (AHS) quantum tasks. These special-purpose devices faithfully reproduce the time-dependent quantum dynamics of hundreds of simultaneously interacting qubits.

One can program these devices in the paradigm of Analog Hamiltonian Simulation by prescribing the layout of the qubit register and the temporal and spatial dependence of the manipulating fields. Amazon Braket provides utilities to construct such programs through the AHS module of the python SDK, `braket.ahs`.

For more information, see the Analog Hamiltonian Simulation example notebooks or the Submit an analog program using QuEra's Aquila page.

Example: Submitting a quantum task to a QPU

Amazon Braket allows you to run a quantum circuit on a QPU device. The following example shows how to submit a quantum task to Rigetti or IonQ devices.

Choose the Rigetti Ankaa-3 device, then look at the associated connectivity graph

```
# import the QPU module
from braket.aws import AwsDevice
# choose the Rigetti device
device = AwsDevice("arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3")

# take a look at the device connectivity graph
```

```
device.properties.dict()['paradigm']['connectivity']
```

```
{'fullyConnected': False,
 'connectivityGraph': {'0': ['1', '7'],
  '1': ['0', '2', '8'],
  '2': ['1', '3', '9'],
  '3': ['2', '4', '10'],
  '4': ['3', '5', '11'],
  '5': ['4', '6', '12'],
  '6': ['5', '13'],
  '7': ['0', '8', '14'],
  '8': ['1', '7', '9', '15'],
  '9': ['2', '8', '10', '16'],
  '10': ['3', '9', '11', '17'],
  '11': ['4', '10', '12', '18'],
  '12': ['5', '11', '13', '19'],
  '13': ['6', '12', '20'],
  '14': ['7', '15', '21'],
  '15': ['8', '14', '22'],
  '16': ['9', '17', '23'],
  '17': ['10', '16', '18', '24'],
  '18': ['11', '17', '19', '25'],
  '19': ['12', '18', '20', '26'],
  '20': ['13', '19', '27'],
  '21': ['14', '22', '28'],
  '22': ['15', '21', '23', '29'],
  '23': ['16', '22', '24', '30'],
  '24': ['17', '23', '25', '31'],
  '25': ['18', '24', '26', '32'],
  '26': ['19', '25', '33'],
  '27': ['20', '34'],
  '28': ['21', '29', '35'],
  '29': ['22', '28', '30', '36'],
  '30': ['23', '29', '31', '37'],
  '31': ['24', '30', '32', '38'],
  '32': ['25', '31', '33', '39'],
  '33': ['26', '32', '34', '40'],
  '34': ['27', '33', '41'],
  '35': ['28', '36', '42'],
  '36': ['29', '35', '37', '43'],
  '37': ['30', '36', '38', '44'],
  '38': ['31', '37', '39', '45'],
  '39': ['32', '38', '40', '46'],
```

```
    '40': ['33', '39', '41', '47'],
    '41': ['34', '40', '48'],
    '42': ['35', '43', '49'],
    '43': ['36', '42', '44', '50'],
    '44': ['37', '43', '45', '51'],
    '45': ['38', '44', '46', '52'],
    '46': ['39', '45', '47', '53'],
    '47': ['40', '46', '48', '54'],
    '48': ['41', '47', '55'],
    '49': ['42', '56'],
    '50': ['43', '51', '57'],
    '51': ['44', '50', '52', '58'],
    '52': ['45', '51', '53', '59'],
    '53': ['46', '52', '54'],
    '54': ['47', '53', '55', '61'],
    '55': ['48', '54', '62'],
    '56': ['49', '57', '63'],
    '57': ['50', '56', '58', '64'],
    '58': ['51', '57', '59', '65'],
    '59': ['52', '58', '60', '66'],
    '60': ['59'],
    '61': ['54', '62', '68'],
    '62': ['55', '61', '69'],
    '63': ['56', '64', '70'],
    '64': ['57', '63', '65', '71'],
    '65': ['58', '64', '66', '72'],
    '66': ['59', '65', '67'],
    '67': ['66', '68'],
    '68': ['61', '67', '69', '75'],
    '69': ['62', '68', '76'],
    '70': ['63', '71', '77'],
    '71': ['64', '70', '72', '78'],
    '72': ['65', '71', '73', '79'],
    '73': ['72', '80'],
    '75': ['68', '76', '82'],
    '76': ['69', '75', '83'],
    '77': ['70', '78'],
    '78': ['71', '77', '79'],
    '79': ['72', '78', '80'],
    '80': ['73', '79', '81'],
    '81': ['80', '82'],
    '82': ['75', '81', '83'],
    '83': ['76', '82']}}
```

The preceding dictionary `connectivityGraph` lists the neighboring qubits for each qubit in the Rigetti device.

Choose the IonQ Aria-1 device

For the IonQ Aria-1 device, the `connectivityGraph` is empty, as shown in the following example, because the device offers *all-to-all* connectivity. Therefore, a detailed `connectivityGraph` is not needed.

```
# or choose the IonQ Aria-1 device
device = AwsDevice("arn:aws:braket:us-east-1::device/qpu/ionq/Aria-1")

# take a look at the device connectivity graph
device.properties.dict()['paradigm']['connectivity']
```

```
{'fullyConnected': True, 'connectivityGraph': {}}
```

As shown in the following example, you have the option to adjust the shots (default=1000), the `poll_timeout_seconds` (default = 432000 = 5 days), the `poll_interval_seconds` (default = 1), and the location of the S3 bucket (`s3_location`) where your results will be stored if you choose to specify a location other than the default bucket.

```
my_task = device.run(circ, s3_location = 'amazon-braket-my-folder', shots=100,
 poll_timeout_seconds = 100, poll_interval_seconds = 10)
```

The IonQ and Rigetti devices compile the provided circuit into their respective native gate sets automatically, and they map the abstract qubit indices to physical qubits on the respective QPU.

Note

QPU devices have limited capacity. You can expect longer wait times when capacity is reached.

Amazon Braket can run QPU quantum tasks within certain availability windows, but you can still submit quantum tasks any time (24/7) because all corresponding data and metadata are stored reliably in the appropriate S3 bucket. As shown in the next section, you can recover your quantum task using `AwsQuantumTask` and your unique quantum task ID.

Inspecting compiled circuits

When a quantum circuit needs to be run on a hardware device, such as a quantum processing unit (QPU), the circuit must first be compiled into an acceptable format that the device can understand and process. For example, transpiling the high-level quantum circuit down to the specific native gates supported by the target QPU hardware. Inspecting the actual compiled output of the quantum circuit can be extremely useful for debugging and optimization purposes. This knowledge can help identify potential issues, bottlenecks, or opportunities for improving the performance and efficiency of the quantum application. You can view and analyze the compiled output of your quantum circuits for both Rigetti and IQM quantum computing devices using the code provided below.

```
task = AwsQuantumTask(arn=task_id, aws_session=session)
# After the task has finished running
task_result = task.result()
compiled_circuit = task_result.get_compiled_circuit()
```

Note

Currently, viewing the compiled circuit output for IonQ devices is not supported.

When will my quantum task run?

When you submit a circuit, Amazon Braket sends it to the device you specify. Quantum Processing Unit (QPU) and on-demand simulator quantum tasks are queued and processed in the order they are received. The time required to process your quantum task after you submit it varies depending on the number and complexity of tasks submitted by other Amazon Braket customers and the availability of the QPU selected.

In this section:

- QPU availability windows and status
- Queue visibility
- Set up email or SMS notifications

QPU availability windows and status

QPU availability varies from device to device.

In the **Devices** page of the Amazon Braket console, you can see the current and upcoming availability windows and device status. Additionally, each device page shows individual queue depths for quantum tasks and hybrid jobs.

A device is considered *offline* if is not available to customers, regardless of availability window. For example, it could be offline due to scheduled maintenance, upgrades, or operational issues.

Queue visibility

Before submitting a quantum task or hybrid job, you can view how many quantum tasks or hybrid jobs are in front of you by checking device queue depth.

Queue depth

Queue depth refers to the number of quantum tasks and hybrid jobs queued for a particular device. A device's quantum task and hybrid job queue count are accessible through the Braket Software Development Kit (SDK) or Amazon Braket Management Console.

1. *Task queue depth* refers to the total number of quantum tasks currently waiting to run in normal priority.
2. *Priority task queue depth* refers to the total number of submitted quantum tasks waiting to run through Amazon Braket Hybrid Jobs. These tasks run before standalone tasks.
3. *Hybrid jobs queue depth* refers to the total number of hybrid jobs currently queued on a device. Quantum tasks submitted as part of a hybrid job have priority, and are aggregated in the Priority Task Queue.

Customers wishing to view queue depth through the Braket SDK can modify the following code snippet to get the queue position of their quantum task or hybrid job:

```
device = AwsDevice("arn:aws:braket:us-east-1::device/qpu/ionq/Aria-1")

# returns the number of quantum tasks queued on the device
print(device.queue_depth().quantum_tasks)
{<QueueType.NORMAL: 'Normal'>: '0', <QueueType.PRIORITY: 'Priority'>: '0'}
```

```
# returns the number of hybrid jobs queued on the device
print(device.queue_depth().jobs)
'3'
```

Submitting a quantum task or hybrid job to a QPU may result in your workload being in a QUEUED state. Amazon Braket provides customers visibility into their quantum task and hybrid job queue position.

Queue position

Queue position refers to the current position of your quantum task or hybrid job within a respective device queue. It can be obtained for quantum tasks or hybrid jobs through the Braket Software Development Kit (SDK) or Amazon Braket Management Console.

Customers wishing to view queue position through the Braket SDK can modify the following code snippet to get the queue position of their quantum task or hybrid job:

```
# choose the device to run your circuit
device = AwsDevice("arn:aws:braket:eu-north-1::device/qpu/iqm/Garnet")

#execute the circuit
task = device.run(bell, s3_folder, shots=100)

# retrieve the queue position information
print(task.queue_position().queue_position)

# Returns the number of Quantum Tasks queued ahead of you
'2'


from braket.aws import AwsQuantumJob

job = AwsQuantumJob.create(
    "arn:aws:braket:eu-north-1::device/qpu/iqm/Garnet",
    source_module="algorithm_script.py",
    entry_point="algorithm_script:start_here",
    wait_until_complete=False
)
```

```
# retrieve the queue position information
print(job.queue_position().queue_position)
'3' # returns the number of hybrid jobs queued ahead of you
```

Set up email or SMS notifications

Amazon Braket sends events to Amazon EventBridge when the availability of a QPU changes or when the state of your quantum task changes. Follow these steps to receive device and quantum task status change notifications by email or SMS message:

1. Create an Amazon SNS topic and a subscription to email or SMS. Availability of email or SMS depends on your Region. For more information, see Getting started with Amazon SNS and Sending SMS messages.
2. Create a rule in EventBridge that triggers the notifications to your SNS topic. For more information, see Monitoring Amazon Braket with Amazon EventBridge.

(Optional) Set up SNS notifications

You can set up notifications through the Amazon Simple Notification Service (SNS) so that you receive an alert when your Amazon Braket quantum task is complete. Active notifications are useful if you expect a long wait time; for example, when you submit a large quantum task or when you submit a quantum task outside of a device's availability window. If you do not want to wait for the quantum task to complete, you can set up an SNS notification.

An Amazon Braket notebook walks you through the setup steps. For more information, see the Amazon Braket examples on GitHub and, specifically, the example notebook for setting up notifications.

Managing your Amazon Braket Hybrid Job

This section provides instructions about how to manage hybrid jobs in Amazon Braket.

You can access hybrid jobs in Braket using:

- The Amazon Braket Python SDK.
- The Amazon Braket console.
- The Amazon Braket API.

In this section:

- Configure the hybrid job instance to run your script
- How to cancel a Hybrid Job
- Using parametric compilation to speed up Hybrid Jobs
- Bring your own container (BYOC)
- Interact with hybrid jobs directly using the API

Configure the hybrid job instance to run your script

Depending on your algorithm, you may have different requirements. By default, Amazon Braket runs your algorithm script on an `ml.m5.large` instance. However, you can customize this instance type when you create a hybrid job using the following import and configuration argument.

```
from braket.jobs.config import InstanceConfig

job = AwsQuantumJob.create(
    ...
    instance_config=InstanceConfig(instanceType="ml.p3.8xlarge"), # Use NVIDIA Tesla
 V100 instance with 4 GPUs.
    ...
    ),
```

If you are running an embedded simulation and have specified a local device in the device configuration, you can additionally request more than one instance in the `InstanceConfig` by specifying the `instanceCount` and setting it to be greater than one. The upper limit is 5. For instance, you can choose 3 instances as follows.

```
from braket.jobs.config import InstanceConfig
job = AwsQuantumJob.create(
    ...
    instance_config=InstanceConfig(instanceType="ml.p3.8xlarge", instanceCount=3), #
 Use 3 NVIDIA Tesla V100
    ...
    ),
```

When you use multiple instances, consider distributing your hybrid job using the data parallel feature. See the following example notebook for more details on how-to see this Parallelize training for QML example.

The following three tables list the available instance types and specs for standard, high performance, and GPU accelerated instances.

Note

To view the default classical compute instance quotas for Hybrid Jobs, see the Amazon Braket Quotas page.

Standard Instances	vCPU	Memory (GiB)
ml.m5.large (default)	4	16
ml.m5.xlarge	4	16
ml.m5.2xlarge	8	32
ml.m5.4xlarge	16	64
ml.m5.12xlarge	48	192
ml.m5.24xlarge	96	384

High performance Instances	vCPU	Memory (GiB)
ml.c5.xlarge	4	8
ml.c5.2xlarge	8	16
ml.c5.4xlarge	16	32
ml.c5.9xlarge	36	72
ml.c5.18xlarge	72	144
ml.c5n.xlarge	4	10.5
ml.c5n.2xlarge	8	21

High performance Instances	vCPU	Memory (GiB)
ml.c5n.4xlarge	16	32
ml.c5n.9xlarge	36	72
ml.c5n.18xlarge	72	192

GPU accelerated Instances	GPUs	vCPU	Memory (GiB)	GPU Memory (GiB)
ml.p3.2xlarge	1	8	61	16
ml.p3.8xlarge	4	32	244	64
ml.p3.16xlarge	8	64	488	128
ml.p4d.24xlarge	8	96	1152	320
ml.g4dn.xlarge	1	4	16	16
ml.g4dn.2xlarge	1	8	32	16
ml.g4dn.4xlarge	1	16	64	16
ml.g4dn.8xlarge	1	32	128	16
ml.g4dn.12xlarge	4	48	192	64
ml.g4dn.16xlarge	1	64	256	16

Note

p3 instances are not available in us-west-1. If your hybrid job is unable to provision requested ML compute capacity, use another Region.

Each instance uses a default configuration of data storage (SSD) of 30 GB. But you can adjust the storage in the same way that you configure the `instanceType`. The following example shows how to increase the total storage to 50 GB.

```
from braket.jobs.config import InstanceConfig

job = AwsQuantumJob.create(
    ...
    instance_config=InstanceConfig(
        instanceType="ml.p3.8xlarge",
        volumeSizeInGb=50,
    ),
    ...
    ),
```

Configure the default bucket in `AwsSession`

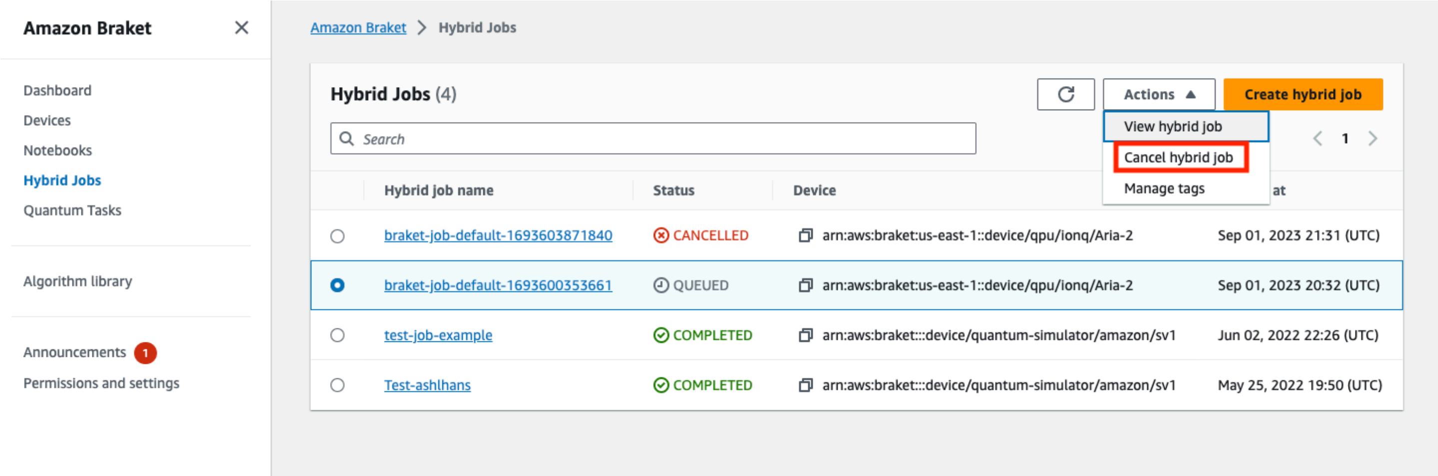
Utilizing your own `AwsSession` instance provides you with enhanced flexibility, such as the ability to specify a custom location for your default Amazon S3 bucket. By default, an `AwsSession` has a pre-configured Amazon S3 bucket location of `"amazon-braket-{id}-{region}"`. However, you have the option to override the default Amazon S3 bucket location when creating an `AwsSession`. Users can optionally pass in an `AwsSession` object into the `AwsQuantumJob.create()` method, by providing the `aws_session` parameter as demonstrated in the following code example.

```
aws_session = AwsSession(default_bucket="amazon-braket-s3-demo-bucket")

# Then you can use that AwsSession when creating a hybrid job
job = AwsQuantumJob.create(
    ...
    aws_session=aws_session
)
```

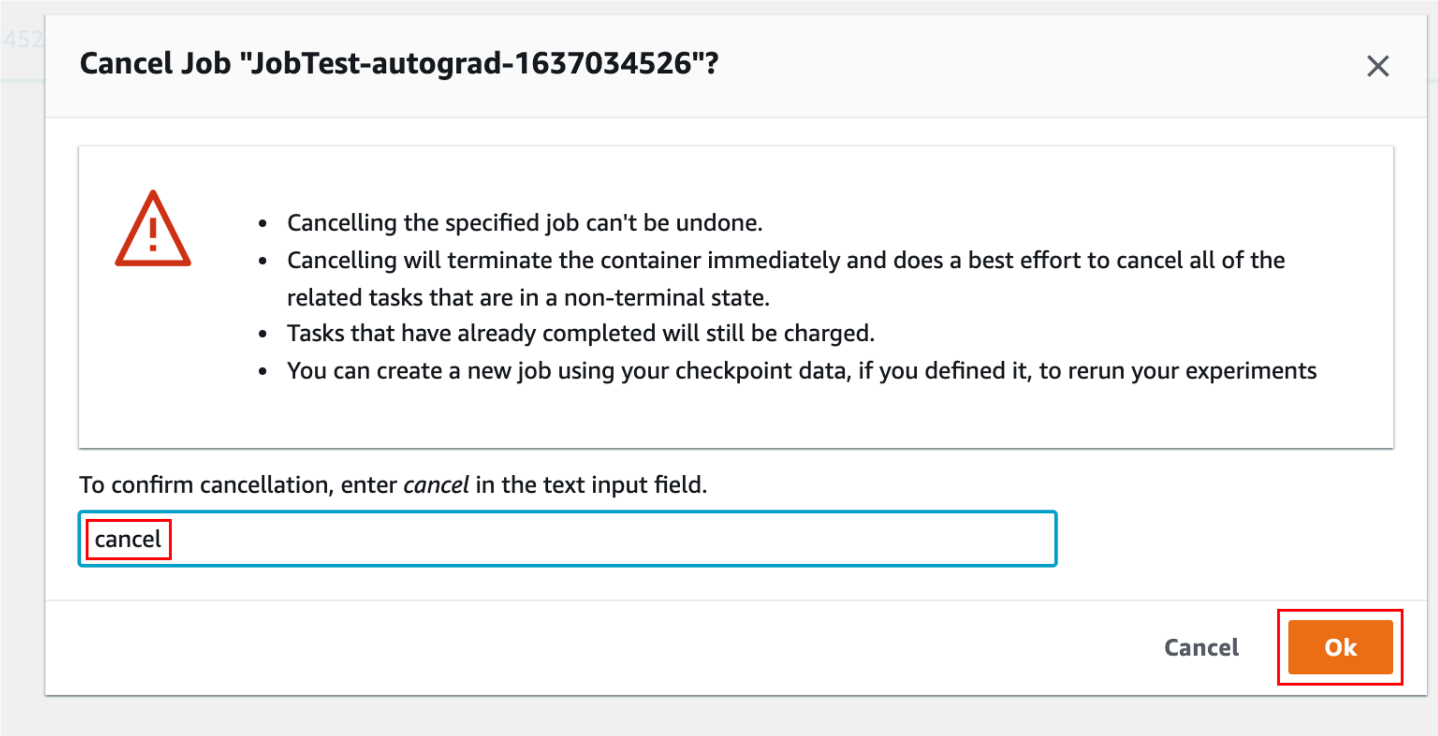
How to cancel a Hybrid Job

You may need to cancel a hybrid job in a non-terminal state. This can be done either in the console or with code.

To cancel your hybrid job in the console, select the hybrid job to cancel from the **Hybrid Jobs** page and then select **Cancel hybrid job** from the **Actions** dropdown menu.



To confirm the cancellation, enter *cancel* into the input field when prompted and then select **OK**.



To cancel your hybrid job using code from the Braket Python SDK, use the `job_arn` to identify the hybrid job and then call the `cancel` command on it as shown in following code.

```
job = AwsQuantumJob(arn=job_arn)
job.cancel()
```

The `cancel` command terminates the classical hybrid job container immediately and does a best effort to cancel all of the related quantum tasks that are still in a non-terminal state.

Using parametric compilation to speed up Hybrid Jobs

Amazon Braket supports parametric compilation on certain QPUs. This enables you to reduce the overhead associated with the computationally expensive compilation step by compiling a circuit only once and not for every iteration in your hybrid algorithm. This can improve runtimes dramatically for Hybrid Jobs, since you avoid the need to recompile your circuit at each step. Just submit parametrized circuits to one of our supported QPUs as a Braket Hybrid Job. For long running hybrid jobs, Braket automatically uses the updated calibration data from the hardware provider when compiling your circuit to ensure the highest quality results.

To create a parametric circuit, you first need to provide parameters as inputs in your algorithm script. In this example, we use a small parametric circuit and ignore any classical processing between each iteration. For typical workloads, you would submit many circuits in batch and perform classical processing such as updating the parameters in each iteration.

```
import os

from braket.aws import AwsDevice
from braket.circuits import Circuit, FreeParameter

def start_here():

    print("Test job started.")

    # Use the device declared in the job script
    device = AwsDevice(os.environ["AMZN_BRAKET_DEVICE_ARN"])

    circuit = Circuit().rx(0, FreeParameter("theta"))
    parameter_list = [0.1, 0.2, 0.3]

    for parameter in parameter_list:
        result = device.run(circuit, shots=1000, inputs={"theta": parameter})

    print("Test job completed.")
```

You can submit the algorithm script to run as a Hybrid Job with the following job script. When running the Hybrid Job on a QPU that supports parametric compilation, the circuit is compiled only on the first run. In following runs, the compiled circuit is reused, increasing the runtime performance of the Hybrid Job without any additional lines of code.

```
from braket.aws import AwsQuantumJob
```

```
job = AwsQuantumJob.create(
    device=device_arn,
    source_module="algorithm_script.py",
)
```

> **Note**
>
> Parametric compilation is supported on all superconducting, gate-based QPUs from Rigetti Computing with the exception of pulse level programs.

Bring your own container (BYOC)

Amazon Braket Hybrid Jobs provides three pre-built containers for running code in different environments. If one of these containers supports your use case, you only have to provide your algorithm script when you create a hybrid job. Minor missing dependencies can be added from your algorithm script or from a `requirements.txt` file using `pip`.

If none of these containers support your use case, or if you wish to expand on them, Braket Hybrid Jobs supports running hybrid jobs with your own custom Docker container image, or bring your own container (BYOC). Make sure it is the right feature for your use case.

In this section:

- When is bringing my own container the right decision?
- Recipe for bringing your own container
- Running Braket hybrid jobs in your own container

When is bringing my own container the right decision?

Bringing your own container (BYOC) to Braket Hybrid Jobs offers the flexibility to use your own software by installing it in a packaged environment. Depending on your specific needs, there may be ways to achieve the same flexibility without having to go through the full BYOC Docker build - Amazon ECR upload - custom image URI cycle.

> **Note**
>
> BYOC may not be the right choice if you want to add a small number of additional Python packages (generally fewer than 10) which are publicly available. For example, if you're using PyPi.

In this case, you can use one of the pre-built Braket images, and then include a `requirements.txt` file in your source directory at the job submission. The file is automatically read, and `pip` will install the packages with the specified versions as normal. If you're installing a large number of packages, the runtime of your jobs may be substantially increased. Check the Python and, if applicable, CUDA version of the prebuilt container you want to use to test if your software will work.

BYOC is necessary when you want to use a non-Python language (like C++ or Rust) for your job script, or if you want to use a Python version not available through the Braket pre-built containers. It is also a good choice if:

- You're using software with a license key, and you need to authenticate that key against a licensing server to run the software. With BYOC, you can embed the license key in your Docker image and include code to authenticate it.
- You are using software that is not publicly available. For example, the software is hosted on a private GitLab or GitHub repository that you need a particular SSH key to access.
- You need to install a large suite of software that is not packaged in the Braket provided containers. BYOC will allow you to eliminate long startup times for your hybrid jobs containers due to software installation.

BYOC also enables you to make your custom SDK or algorithm available to customers by building a Docker container with your software and making it available to your users. You can do this by setting appropriate permissions in Amazon ECR.

> **Note**
>
> You must comply with all applicable software licenses.

Recipe for bringing your own container

In this section, we provide a step-by-step guide of what you need to bring your own container (BYOC) to Braket Hybrid Jobs — the scripts, files, and steps to combine them to get up and running with your custom Docker images. The recipes for two common cases:

1. Install additional software in a Docker image and use only Python algorithm scripts in your jobs.
2. Use algorithm scripts written in a non-Python language with Hybrid Jobs, or a CPU architecture besides x86.

Defining the *container entry script* is more complex for case 2.

When Braket runs your Hybrid Job, it launches the requested number and type of Amazon EC2 instances, then runs the Docker image specified by the image URI input to job creation on them. When using the BYOC feature, you specify an image URI hosted in a private Amazon ECR repository that you have Read access to. Braket Hybrid Jobs uses that custom image to run the job.

The specific components you need to build a Docker image that can be used with Hybrid Jobs. If you are unfamiliar with writing and building `Dockerfiles`, refer to the Dockerfile documentation and the Amazon ECR CLI documentation.

Requirements:

- A base image for your Dockerfile
- (Optional) A modified container entry point script
- Install needed software and container script with Dockerfile

A base image for your Dockerfile

If you are using Python and want to install software on top of what is provided in the Braket provided containers, an option for a base image is one of the Braket container images, hosted in our GitHub repo and on Amazon ECR. You will need to authenticate to Amazon ECR to pull the image and build on top of it. For example, the first line of your BYOC Docker file could be: `FROM [IMAGE_URI_HERE]`

Next, fill out the rest of the Dockerfile to install and set up the software that you want to add to the container. The pre-built Braket images will already contain the appropriate container entry point script, so you do not need to worry about including that.

If you want to use a non-Python language, such as C++, Rust, or Julia, or if you want to build an image for a non-x86 CPU architecture, like ARM, you may need to build on top of a barebones public image. You can find many such images at the Amazon Elastic Container Registry Public Gallery. Make sure you choose one that is appropriate for the CPU architecture, and if necessary, the GPU you want to use.

(Optional) A modified container entry point script

> **Note**
>
> If you're only adding additional software to a pre-built Braket image, you can skip this section.

To run non-Python code as part of your hybrid job, modify the Python script which defines the container entry point. For example, the `braket_container.py` python script on the Amazon Braket Github. This is the script the images pre-built by Braket use to launch your algorithm script and set appropriate environment variables. The container entry point script itself **must** be in Python, but can launch non-Python scripts. In the pre-built example, you can see that Python algorithm scripts are launched either as a Python subprocess or as a fully new process. By modifying this logic, you can enable the entry point script to launch non-Python algorithm scripts. For example, you could modify `thekick_off_customer_script()` function to launch Rust processes dependent on the file extension ending.

You can also choose to write a completely new `braket_container.py`. It should copy input data, source archives, and other necessary files from Amazon S3 into the container, and define the appropriate environment variables.

Install needed software and container script with `Dockerfile`

> **Note**
>
> If you use a pre-built Braket image as your Docker base image, the container script is already present.

If you created a modified container script in the previous step, you'll need to copy it into the container **and** define the environment variable `SAGEMAKER_PROGRAM` to `braket_container.py`, or what you have named your new container entry point script.

The following is an example of a `Dockerfile` that allows you to use Julia on GPU-accelerated Jobs instances:

```
FROM nvidia/cuda:12.2.0-devel-ubuntu22.04


 ARG DEBIAN_FRONTEND=noninteractive
 ARG JULIA_RELEASE=1.8
 ARG JULIA_VERSION=1.8.3


 ARG PYTHON=python3.11
 ARG PYTHON_PIP=python3-pip
 ARG PIP=pip


 ARG JULIA_URL = https://julialang-s3.julialang.org/bin/linux/x64/${JULIA_RELEASE}/
 ARG TAR_NAME = julia-${JULIA_VERSION}-linux-x86_64.tar.gz


 ARG PYTHON_PKGS = # list your Python packages and versions here


 RUN curl -s -L ${JULIA_URL}/${TAR_NAME} | tar -C /usr/local -x -z --strip-components=1
 -f -


 RUN apt-get update \

    && apt-get install -y --no-install-recommends \

    build-essential \

    tzdata \

    openssh-client \

    openssh-server \

    ca-certificates \

    curl \
```

```
    git \

    libtemplate-perl \

    libssl1.1 \

    openssl \

    unzip \

    wget \

    zlib1g-dev \

    ${PYTHON_PIP} \

    ${PYTHON}-dev \



 RUN ${PIP} install --no-cache --upgrade ${PYTHON_PKGS}


 RUN ${PIP} install --no-cache --upgrade sagemaker-training==4.1.3


 # Add EFA and SMDDP to LD library path
 ENV LD_LIBRARY_PATH="/opt/conda/lib/python${PYTHON_SHORT_VERSION}/site-packages/
smdistributed/dataparallel/lib:$LD_LIBRARY_PATH"
 ENV LD_LIBRARY_PATH=/opt/amazon/efa/lib/:$LD_LIBRARY_PATH


 # Julia specific installation instructions
 COPY Project.toml /usr/local/share/julia/environments/v${JULIA_RELEASE}/
 RUN JULIA_DEPOT_PATH=/usr/local/share/julia \

    julia -e 'using Pkg; Pkg.instantiate(); Pkg.API.precompile()'
 # generate the device runtime library for all known and supported devices
 RUN JULIA_DEPOT_PATH=/usr/local/share/julia \

    julia -e 'using CUDA; CUDA.precompile_runtime()'
```

```
# Open source compliance scripts
RUN HOME_DIR=/root \

&& curl -o ${HOME_DIR}/oss_compliance.zip https://aws-dlinfra-
utilities.s3.amazonaws.com/oss_compliance.zip \

&& unzip ${HOME_DIR}/oss_compliance.zip -d ${HOME_DIR}/ \

&& cp ${HOME_DIR}/oss_compliance/test/testOSSCompliance /usr/local/bin/
testOSSCompliance \

&& chmod +x /usr/local/bin/testOSSCompliance \

&& chmod +x ${HOME_DIR}/oss_compliance/generate_oss_compliance.sh \

&& ${HOME_DIR}/oss_compliance/generate_oss_compliance.sh ${HOME_DIR} ${PYTHON} \

&& rm -rf ${HOME_DIR}/oss_compliance*


# Copying the container entry point script
COPY braket_container.py /opt/ml/code/braket_container.py
ENV SAGEMAKER_PROGRAM braket_container.py
```

This example, downloads and runs scripts provided by AWS to ensure compliance with all relevant Open-Source licenses. For example, by properly attributing any installed code governed by an MIT license.

If you need to include non-public code, for instance code that is hosted in a private GitHub or GitLab repository, **do not** embed SSH keys in the Docker image to access it. Instead, use Docker Compose when you build to allow Docker to access SSH on the host machine it is built on. For more information, see the Securely using SSH keys in Docker to access private Github repositories guide.

Building and uploading your Docker image

With a properly defined `Dockerfile`, you are now ready to follow the steps to create a private Amazon ECR repository, if one does not already exist. You can also build, tag, and upload your container image to the repository.

You are ready to build, tag, and push the image. See the Docker build documentation for a full explanation of options to `docker build` and some examples.

For the sample file defined above, you could run:

```
aws ecr get-login-password --region ${your_region} | docker login --username AWS --
password-stdin ${aws_account_id}.dkr.ecr.${your_region}.amazonaws.com
 docker build -t braket-julia .
 docker tag braket-julia:latest ${aws_account_id}.dkr.ecr.${your_region}.amazonaws.com/
braket-julia:latest
 docker push ${aws_account_id}.dkr.ecr.${your_region}.amazonaws.com/braket-julia:latest
```

Assigning appropriate Amazon ECR permissions

Braket Hybrid Jobs Docker images must be hosted in private Amazon ECR repositories. By default, a private Amazon ECR repo does **not** provide read access to the Braket Hybrid Jobs IAM role or to any other users that want to use your image, such as a collaborator or student. You must set a repository policy to grant the appropriate permissions. In general, only give permission to those specific users and IAM roles you want to access your images, rather than allowing anyone with the image URI to pull them.

Running Braket hybrid jobs in your own container

To create a hybrid job with your own container, call `AwsQuantumJob.create()` with the argument `image_uri` specified. You can use a QPU, an on-demand simulator, or run your code locally on the classical processor available with Braket Hybrid Jobs. We recommend testing your code out on a simulator like SV1, DM1, or TN1 before running on a real QPU.

To run your code on the classical processor, specify the `instanceType` and the `instanceCount` you use by updating the `InstanceConfig`. Note that if you specify an `instance_count` > 1, you need to make sure that your code can run across multiple hosts. The upper limit for the number of instances you can choose is 5. For example:

```
job = AwsQuantumJob.create(
    source_module="source_dir",
    entry_point="source_dir.algorithm_script:start_here",
    image_uri="111122223333.dkr.ecr.us-west-2.amazonaws.com/my-byoc-container:latest",
    instance_config=InstanceConfig(instanceType="ml.p3.8xlarge", instanceCount=3),
    device="local:braket/braket.local.qubit",
    # ...)
```

Note

Use the device ARN to track the simulator you used as hybrid job metadata. Acceptable values must follow the format `device = "local:<provider>/<simulator_name>"`.

Remember that <provider> and <simulator_name> must consist only of letters, numbers, _, -, and `.` . The string is limited to 256 characters.
If you plan to use BYOC and you're not using the Braket SDK to create quantum tasks, you should pass the value of the environmental variable `AMZN_BRAKET_JOB_TOKEN` to the `jobToken` parameter in the `CreateQuantumTask` request. If you don't, the quantum tasks don't get priority and are billed as regular standalone quantum tasks.

Interact with hybrid jobs directly using the API

You can access and interact with Amazon Braket Hybrid Jobs directly using the API. However, defaults and convenience methods are not available when using the API directly.

> **Note**
>
> We strongly recommend that you interact with Amazon Braket Hybrid Jobs using the Amazon Braket Python SDK. It offers convenient defaults and protections that help your hybrid jobs run successfully.

This topic covers the basics of using the API. If you choose to use the API, keep in mind that this approach can be more complex and be prepared for several iterations to get your hybrid job to run.

To use the API, your account should have a role with the `AmazonBraketFullAccess` managed policy.

> **Note**
>
> For more information on how to obtain a role with the `AmazonBraketFullAccess` managed policy, see the Enable Amazon Braket page.

Additionally, you need an **execution role**. This role will be passed to the service. You can create the role using the **Amazon Braket console**. Use the **Execution roles** tab on the **Permissions and settings** page to create a default role for hybrid jobs.

The `CreateJob` API requires that you specify all the required parameters for the hybrid job. To use Python, compress your algorithm script files to a tar bundle, such as an input.tar.gz file, and run the following script. Update the parts of the code within angled brackets (<>) to match your account information and entry point that specify the path, file, and method where your hybrid job starts.

```
from braket.aws import AwsDevice, AwsSession
import boto3
from datetime import datetime

s3_client = boto3.client("s3")
client = boto3.client("braket")

project_name = "job-test"
job_name = project_name + "-" + datetime.strftime(datetime.now(), "%Y%m%d%H%M%S")
bucket = "amazon-braket-<your_bucket>"
s3_prefix = job_name

job_script = "input.tar.gz"
job_object = f"{s3_prefix}/script/{job_script}"
s3_client.upload_file(job_script, bucket, job_object)

input_data = "inputdata.csv"
input_object = f"{s3_prefix}/input/{input_data}"
s3_client.upload_file(input_data, bucket, input_object)

job = client.create_job(
    jobName=job_name,
    roleArn="arn:aws:iam::<your_account>:role/service-role/
AmazonBraketJobsExecutionRole",  # https://docs.aws.amazon.com/braket/latest/
developerguide/braket-manage-access.html#about-amazonbraketjobsexecution
    algorithmSpecification={
        "scriptModeConfig": {
            "entryPoint": "<your_execution_module>:<your_execution_method>",
            "containerImage": {"uri": "292282985366.dkr.ecr.us-west-1.amazonaws.com/
amazon-braket-base-jobs:1.0-cpu-py37-ubuntu18.04"},   # Change to the specific region
 you are using
            "s3Uri": f"s3://{bucket}/{job_object}",
            "compressionType": "GZIP"
        }
    },
    inputDataConfig=[
        {
            "channelName": "hellothere",
            "compressionType": "NONE",
            "dataSource": {
```

```
                "s3DataSource": {
                    "s3Uri": f"s3://{bucket}/{s3_prefix}/input",
                    "s3DataType": "S3_PREFIX"
                }
            }
        }
    ],
    outputDataConfig={
        "s3Path": f"s3://{bucket}/{s3_prefix}/output"
    },
    instanceConfig={
        "instanceType": "ml.m5.large",
        "instanceCount": 1,
        "volumeSizeInGb": 1
    },
    checkpointConfig={
        "s3Uri":  f"s3://{bucket}/{s3_prefix}/checkpoints",
        "localPath": "/opt/omega/checkpoints"
    },
    deviceConfig={
        "priorityAccess": {
            "devices": [
                "arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3"
            ]
        }
    },
    hyperParameters={
        "hyperparameter key you wish to pass": "<hyperparameter value you wish to
 pass>",
    },
    stoppingCondition={
        "maxRuntimeInSeconds": 1200,
        "maximumTaskLimit": 10
    },
)
```

Once you create your hybrid job, you can access the hybrid job details through the `GetJob` API or the console. To get the hybrid job details from the Python session in which you ran the `createJob` code as in the previous example, use the following Python command.

```
getJob = client.get_job(jobArn=job["jobArn"])
```

To cancel a hybrid job, call the `CancelJob` API with the Amazon Resource Name of the job ('JobArn').

```
cancelJob = client.cancel_job(jobArn=job["jobArn"])
```

You can specify checkpoints as part of the `createJob` API using the `checkpointConfig` parameter.

```
    checkpointConfig = {
        "localPath" : "/opt/omega/checkpoints",
        "s3Uri": f"s3://{bucket}/{s3_prefix}/checkpoints"
    },
```

> **Note**
>
> The localPath of `checkpointConfig` cannot start with any of the following reserved paths: `/opt/ml`, `/opt/braket`, `/tmp`, or `/usr/local/nvidia`.

Working with reservations

Reservations give you exclusive access to the quantum device of your choice. You can schedule a reservation at your convenience, so you know exactly when your workload starts and ends execution. Reservations are available in 1-hour increments and can be cancelled up to 48 hours in advance, at no additional charge. You can choose to queue quantum tasks and hybrid jobs for an upcoming reservation in advance, or submit workloads during your reservation.

The cost of dedicated device access is based on the duration of your reservation, regardless of how many quantum tasks and hybrid jobs you run on the Quantum Processing Unit (QPU).

The following quantum computers are available for reservations:

- IonQ's Aria and Forte
- Rigetti's Ankaa-3
- IQM's Garnet and Emerald
- QuEra's Aquila

Note

When using a direct reservation with an IonQ device, there is no gateshot limit, and a minimum of 500 shots for Error mitigation tasks.

When to use a reservation

Leveraging dedicated device access with reservations provides you with the convenience and predictability of knowing exactly when your quantum workload starts and ends execution. Compared to submitting tasks and hybrid jobs on-demand, you do not have wait in a queue with other customer tasks. Because you have exclusive access to the device during your reservation, only your workloads run on the device for the entirety of the reservation.

We recommend using on-demand access for the design and prototyping phase of your research, enabling quick and cost-efficient iteration of your algorithms. Once you are ready to produce final experiment results, consider scheduling a device reservation at your convenience to ensure that you can meet project or publication deadlines. We also recommend using reservations when you desire task execution during specific times, such as when you're running a live demo or workshop on a quantum computer.

In this section:

- How to create a reservation
- Running quantum tasks during a reservation
- Running hybrid jobs during a reservation
- What happens at the end of your reservation
- Cancel or reschedule an existing reservation

How to create a reservation

To create a reservation, contact the Braket team by following these steps:

1. Open the Amazon Braket console.
2. Choose **Braket Direct** in the left pane, and then in the **Reservations** section, choose **Reserve device** .
3. Select the **Device** that you would like to reserve.

4. Provide your contact information including **Name** and **Email**. Be sure to provide a valid email address that you regularly check.
5. Under **Tell us about your workload**, provide any details about the workload to run using your reservation. For example, desired reservation length, relevant constraints, or desired schedule.
6. If you are interested in connecting with a Braket expert for a reservation prep session after your reservation is confirmed, optionally select **I'm interested in a prep session**.

You can also contact us to create a reservation by following these steps:

1. Open the Amazon Braket console.
2. Choose **Devices** in the left pane and choose the device that you would like to reserve.
3. In the **Summary** section, choose **Reserve device**.
4. Follow steps 4-6 in the previous procedure.

After you submit the form, you receive an email from the Braket team with the next steps to create your reservation. Once your reservation is confirmed, you will receive the reservation ARN through your email.

Note

Your reservation is only confirmed once you receive the reservation ARN.

Reservations are available in minimum 1-hour increments and certain devices might have additional reservation length constraints (including minimum and maximum reservation durations). The Braket team shares any relevant information with you prior to confirming the reservation.

If you indicated interest in a reservation prep session, the Braket team will contact you through your email to arrange a 30-minute session with a Braket expert.

Running quantum tasks during a reservation

After obtaining a valid reservation ARN from Create a reservation, you can create quantum tasks to run during the reservation. These tasks remain in the QUEUED state until your reservation begins.

> **Note**
>
> Reservations are AWS account and device-specific. Only the AWS account that created the reservation can use your reservation ARN.

> **Note**
>
> There is no queue visibility for tasks and jobs submitted with a reservation ARN because only your tasks run during your reservation.

You can create quantum tasks using Python SDKs such as Braket, Qiskit, PennyLane, or directly with boto3 (Working with Boto3). To use reservations, you must have version v1.79.0 or higher of the Amazon Braket Python SDK. You can update to the latest Braket SDK, Qiskit provider and PennyLane plugin with the following code.

```
pip install --upgrade amazon-braket-sdk amazon-braket-pennylane-plugin qiskit-braket-provider
```

Run tasks with the `DirectReservation` context manager

The recommended way to run a task within your scheduled reservation is to use the `DirectReservation` context manager. By specifying your target device and reservation ARN, the context manager ensures that all tasks created within the Python `with` statement are run with exclusive access to the device.

First, define a quantum circuit and the device. Then use the reservation context and run the task.

```
from braket.aws import AwsDevice, DirectReservation
from braket.circuits import Circuit
from braket.devices import Devices

bell = Circuit().h(0).cnot(0, 1)
device = AwsDevice(Devices.IonQ.Aria1)

# run the circuit in a reservation
with DirectReservation(device, reservation_arn="<my_reservation_arn>"):
    task = device.run(bell, shots=100)
```

You can create quantum tasks in a reservation using PennyLane and Qiskit plugins, as long as the `DirectReservation` context is active while creating quantum tasks. For example, with the Qiskit-Braket provider, you can run tasks as follows.

```
from braket.devices import Devices
from braket.aws import DirectReservation
from qiskit import QuantumCircuit
from qiskit_braket_provider import BraketProvider


qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0, 1)

aria = BraketProvider().get_backend("Aria 1")

# run the circuit in a reservation
with DirectReservation(Devices.IonQ.Aria1, reservation_arn="<my_reservation_arn>"):
    aria_task = aria.run(qc, shots=10)
```

Similarly, the following code runs a circuit during a reservation using the Braket-PennyLane plugin.

```
from braket.devices import Devices
from braket.aws import DirectReservation
import pennylane as qml


dev = qml.device("braket.aws.qubit", device_arn=Devices.IonQ.Aria1.value, wires=2,
 shots=10)

@qml.qnode(dev)
def bell_state():
    qml.Hadamard(wires=0)
    qml.CNOT(wires=[0, 1])
    return qml.probs(wires=[0, 1])

# run the circuit in a reservation
with DirectReservation(Devices.IonQ.Aria1, reservation_arn="<my_reservation_arn>"):
    probs = bell_state()
```

Manually setting the reservation context

Alternatively, you can manually set the reservation context with the following code.

```
# set reservation context
reservation = DirectReservation(device, reservation_arn="<my_reservation_arn>").start()

# run circuit during reservation
task = device.run(bell, shots=100)
```

This is ideal for Jupyter notebooks where the context can be run in the first cell and all subsequent tasks will run in the reservation.

> **Note**
>
> The cell containing the `.start()` call should *only be run once*.

To switch back to the on-demand mode: Restart the Jupyter notebook, or call the following to change the context back to on-demand mode.

```
reservation.stop()  # unset reservation context
```

> **Note**
>
> Reservations have a scheduled start and end time (see Create a reservation). The `reservation.start()` and `reservation.stop()` methods **do not begin or terminate a reservation**. These are methods to modify all subsequent quantum tasks to run during the reservation. These methods have no effect on the scheduled reservation time.

Explicitly pass the reservation ARN when creating task

Another way to create tasks during a reservation is to explicitly pass the reservation ARN when calling `device.run()`.

```
task = device.run(bell, shots=100, reservation_arn="<my_reservation_arn>")
```

This method directly associates the quantum task with the reservation ARN, ensuring it runs during the reserved period. For this option, add the reservation ARN to each task you plan to run during a reservation. Additionally, check that tasks created in Qiskit or PennyLane are using the correct reservation ARN. Due to these additional considerations, the prior two ways are recommended.

When directly using boto3, pass the reservation ARN as an association when creating a task.

```
import boto3

braket_client = boto3.client("braket")


kwargs["associations"] = [
    {
        "arn": "<my_reservation_arn>",
        "type": "RESERVATION_TIME_WINDOW_ARN",
    }
]

response = braket_client.create_quantum_task(**kwargs)
```

Running hybrid jobs during a reservation

Once you have a Python function to run as a hybrid job, you can run the hybrid job in a reservation by passing the `reservation_arn` keyword argument. All the tasks within the hybrid job use the reservation ARN. Importantly, the hybrid job with `reservation_arn` *only spins up the classical compute once your reservation starts*.

> **Note**
>
> A hybrid job running during a reservation *only successfully runs* quantum tasks on the reserved device. Attempting to use a different on-demand Braket device will result in an error. If you need to run tasks on both an on-demand simulator and the reserved device within the same hybrid job, use `DirectReservation` instead.

The following code demonstrates how to run a hybrid job during a reservation.

```
from braket.aws import AwsDevice
from braket.devices import Devices
from braket.jobs import get_job_device_arn, hybrid_job
```

```
@hybrid_job(device=Devices.IonQ.Aria1, reservation_arn="<my_reservation_arn>")
def example_hybrid_job():
    # declare AwsDevice within the hybrid job
    device = AwsDevice(get_job_device_arn())
    bell = Circuit().h(0).cnot(0, 1)

    task = device.run(bell, shots=10)
```

For hybrid jobs that use a Python script (see the section on Creating your first Hybrid Job in the developer guide), you can run them within the reservation by passing the `reservation_arn` keyword argument when creating the job.

```
from braket.aws import AwsQuantumJob
from braket.devices import Devices

job = AwsQuantumJob.create(
    Devices.IonQ.Aria1,
    source_module="algorithm_script.py",
    entry_point="algorithm_script:start_here",
    reservation_arn="<my_reservation_arn>"
)
```

What happens at the end of your reservation

After your reservation ends, you no longer have dedicated access to the device. Any remaining workloads that are queued with this reservation are automatically canceled.

> **Note**
>
> Any job that was in `RUNNING` status when the reservation ends is canceled. We recommend using checkpoints to save and restart jobs at your convenience.

An ongoing reservation, such as after reservation start and before reservation end, can't be extended because each reservation represents standalone dedicated device access. For example, two back-to-back reservations are considered separate and any pending tasks from the first reservation are automatically canceled. They do not resume in the second reservation.

Note

Reservations represent dedicated device access for your AWS account. Even if the device remains idle, no other customers can use it. Therefore, you are charged for the length of the reserved time, regardless of the utilized time.

Cancel or reschedule an existing reservation

You can cancel your reservation no less than 48 hours before the scheduled reservation start time. To cancel, respond to the reservation confirmation email you received with your cancellation request.

To reschedule, you have to cancel your existing reservation, and then create a new one.

Error mitigation techniques

Quantum error mitigation is a set of techniques aimed at reducing the effects of errors in quantum computers.

Quantum devices are subject to environmental noise that degrades the quality of computations performed. While fault-tolerant quantum computing promises a solution to this problem, current quantum devices are limited by the number of qubits and relatively high error rates. To combat this in the near-term, researchers are investigating methods to improve the accuracy of noisy quantum computation. This approach, known as *quantum error mitigation*, involves using various techniques to extract the best signal from noisy measurement data.

In this section:

- Error mitigation techniques on IonQ devices

Error mitigation techniques on IonQ devices

Error mitigation involves running multiple physical circuits and combining their measurements to give an improved result.

> **Note**
>
> For all IonQ's devices: When using an on-demand model, there is a 1 Million gateshot limit, and a minimum of 2500 shots for Error mitigation tasks. For a direct reservation, there is no gateshot limit, and a minimum of 500 shots for Error mitigation tasks.

Debiasing

IonQ devices features an error mitigation method called *debiasing*.

Debiasing maps a circuit into multiple variants that act on different qubit permutations or with different gate decompositions. This reduces the effect of systematic errors such as gate over-rotations or a single faulty qubit by using different implementations of a circuit that could otherwise bias measurement results. This comes at the expense of extra overhead to calibrate multiple qubits and gates.

For more information on debiasing, see Enhancing quantum computer performance through symmetrization.

> **Note**
>
> Using debiasing requires a minimum of 2500 shots.

You can run a quantum task with debiasing on an IonQ device using the following code:

```
from braket.aws import AwsDevice
from braket.circuits import Circuit
from braket.error_mitigation import Debias

# choose an IonQ device
device = AwsDevice("arn:aws:braket:us-east-1::device/qpu/ionq/Aria-1")
circuit = Circuit().h(0).cnot(0, 1)

task = device.run(circuit, shots=2500, device_parameters={"errorMitigation": Debias()})

result = task.result()
print(result.measurement_counts)
>>> {"00": 1245, "01": 5, "10": 10 "11": 1240} # result from debiasing
```

When the quantum task is complete, you can see the measurement probabilities and any result types from the quantum task. The measurement probabilities and counts from all variants are aggregated into a single distribution. Any result types specified in the circuit, such as expectation values, are computed using the aggregate measurement counts.

Sharpening

You can also access measurement probabilities computed with a different post-processing strategy called *sharpening*. Sharpening compares the results of each variant and discards inconsistent shots, favoring the most likely measurement outcome across variants. For more information, see Enhancing quantum computer performance through symmetrization.

Importantly, sharpening assumes the form of the output distribution to be sparse with few high-probability states and many zero-probability states. It may distort the probability distribution if this assumption is not valid.

You can access the probabilities from a sharpened distribution in the `additional_metadata` field on the `GateModelTaskResult` in the Braket Python SDK. Note that sharpening does not return the measurement counts, but instead returns a re-normalized probability distribution. The following code snippet shows how to access the distribution after sharpening.

```
print(result.additional_metadata.ionqMetadata.sharpenedProbabilities)
>>> {"00": 0.51, "11": 0.549} # sharpened probabilities
```

Troubleshooting Amazon Braket

Use the troubleshooting information and solutions in this section to help resolve issues with Amazon Braket.

In this section:

- AccessDeniedException
- An error occurred (ValidationException) when calling the CreateQuantumTask operation
- An SDK feature does not work
- Hybrid job fails due to ServiceQuotaExceededException
- Components stopped working in notebook instance
- Troubleshooting OpenQASM

AccessDeniedException

If you receive an **AccessDeniedException** when enabling or using Braket, you are likely attempting to enable or use Braket in a region where your restricted role does not have access.

In such cases, contact your internal AWS administrator to understand which of the following conditions apply:

- If there are role restrictions preventing access to a region.
- If the role you are attempting to use is permitted to use Braket.

If your role does not have access to a given region when using Braket, then you will be unable to use devices in that particular region.

An error occurred (ValidationException) when calling the CreateQuantumTask operation

If you receive an error similar to: `An error occurred (ValidationException) when calling the CreateQuantumTask operation: Caller doesn't have access to amazon-braket-…` Check that you are referring to an existing s3_folder. Braket does not auto create new Amazon S3 buckets and prefixes for you.

If you are accessing the API directly and receiving an error similar to: `Failed to create quantum task: Caller doesn't have access to s3://MY_BUCKET` Check that you are not including `s3://` in the Amazon S3 bucket path.

An SDK feature does not work

Your Python version must be 3.9 or above. For Amazon Braket Hybrid Jobs, we recommend Python 3.10.

Verify your SDK and schemas are up-to-date. To update the SDK from the notebook or your python editor, run the following command:

```
pip install amazon-braket-sdk --upgrade --upgrade-strategy eager
```

To update the schemas, run the following command:

```
pip install amazon-braket-schemas --upgrade
```

If you are accessing Amazon Braket from your own client, verify your AWS Region is set to a region supported by Amazon Braket.

Hybrid job fails due to ServiceQuotaExceededException

A hybrid job running quantum tasks against the Amazon Braket simulators can fail to be created if you exceed the concurrent quantum task limit for the simulator device you are targeting. For more information on the service limits, see the Quotas topic.

If you are running concurrent tasks against a simulator device in multiple hybrid jobs from your account, you could encounter this error.

To see the number of concurrent quantum tasks against a specific simulator device, use the `search-quantum-tasks` API, as shown in the following code example.

```
DEVICE_ARN=arn:aws:braket:::device/quantum-simulator/amazon/sv1
task_list=""
for status_value in "CREATED" "QUEUED" "RUNNING" "CANCELLING"; do
    tasks=$(aws braket search-quantum-tasks --filters
 name=status,operator=EQUAL,values=${status_value}
```

```
 name=deviceArn,operator=EQUAL,values=$DEVICE_ARN --max-results 100 --query
 'quantumTasks[*].quantumTaskArn' --output text)
    task_list="$task_list $tasks"
done;
echo "$task_list" | tr -s ' \t' '[\n*]' | sort | uniq
```

You can also view the created quantum tasks against a device using Amazon CloudWatch metrics: **Braket** > **By Device**.

To avoid running into these errors:

1. Request a service quota increase for the number of concurrent quantum tasks for the simulator device. This is only applicable to the SV1 device.
2. Handle `ServiceQuotaExceeded` exceptions in your code and retry.

Components stopped working in notebook instance

If some components of your notebook stop working, try the following:

1. Download any notebooks you created or modified to a local drive.
2. Stop your notebook instance.
3. Delete your notebook instance.
4. Create new notebook instance with a different name.
5. Upload the notebooks to the new instance.

Troubleshooting OpenQASM

This section provides troubleshooting pointers that might be useful when encountering errors using OpenQASM 3.0.

In this section:

- Include statement error
- Non-contiguous qubits error
- Mixing physical qubits with virtual qubits error
- Requesting result types and measuring qubits in the same program error
- Classical and qubit register limits exceeded error

- Box not preceded by a verbatim pragma error
- Verbatim boxes missing native gates error
- Verbatim boxes missing physical qubits error
- The verbatim pragma is missing "braket" error
- Single qubits cannot be indexed error
- The physical qubits in a two qubit gate are not connected error
- Local simulator support warning

Include statement error

Braket currently doesn't have a standard gate library file to be included in OpenQASM programs. For example, the following example raises a parser error.

```
OPENQASM 3;
include "standardlib.inc";
```

This code generates the error message: `No terminal matches '"' in the current parser context, at line 2 col 17.`

Non-contiguous qubits error

Using non-contiguous qubits on devices that `requiresContiguousQubitIndices` be set to `true` in the device capability result in an error.

When running quantum tasks on simulators and IonQ, the following program triggers the error.

```
OPENQASM 3;

qubit[4] q;

h q[0];
cnot q[0], q[2];
cnot q[0], q[3];
```

This code generates the error message: `Device requires contiguous qubits. Qubit register q has unused qubits q[1], q[4].`

Mixing physical qubits with virtual qubits error

Mixing physical qubits with virtual qubits in the same program is not allowed and results in an error. The following code generates the error.

```
OPENQASM 3;

qubit[2] q;
cnot q[0], $1;
```

This code generates the error message: `[line 4] mixes physical qubits and qubits registers.`

Requesting result types and measuring qubits in the same program error

Requesting result types and that qubits are explicitly measured in the same program results in an error. The following code generates the error.

```
OPENQASM 3;

qubit[2] q;

h q[0];
cnot q[0], q[1];
measure q;

#pragma braket result expectation x(q[0]) @ z(q[1])
```

This code generates the error message: `Qubits should not be explicitly measured when result types are requested.`

Classical and qubit register limits exceeded error

Only one classical register and one qubit register are allowed. The following code generates the error.

```
OPENQASM 3;
```

```
qubit[2] q0;
qubit[2] q1;
```

This code generates the error message: `[line 4] cannot declare a qubit register. Only 1 qubit register is supported.`

Box not preceded by a verbatim pragma error

All boxes must be preceded by a verbatim pragma. The following code generates the error.

```
box{
rx(0.5) $0;
}
```

This code generates the error message: `In verbatim boxes, native gates are required. x is not a device native gate.`

Verbatim boxes missing native gates error

Verbatim boxes should have native gates and physical qubits. The following code generates the native gates error.

```
#pragma braket verbatim
box{
x $0;
}
```

This code generates the error message: `In verbatim boxes, native gates are required. x is not a device native gate.`

Verbatim boxes missing physical qubits error

Verbatim boxes must have physical qubits. The following code generates the missing physical qubits error.

```
qubit[2] q;

#pragma braket verbatim
box{
rx(0.1) q[0];
```

```
}
```

This code generates the error message: `Physical qubits are required in verbatim box.`

The verbatim pragma is missing "braket" error

You must include “braket” in the verbatim pragma. The following code generates the error.

```
#pragma braket verbatim            // Correct
#pragma verbatim                   // wrong
```

This code generates the error message: `You must include “braket” in the verbatim pragma`

Single qubits cannot be indexed error

Single qubits cannot be indexed. The following code generates the error.

```
OPENQASM 3;

qubit q;
h q[0];
```

This code generates the error: `[line 4] single qubit cannot be indexed.`

However, single qubit arrays can be indexed as follows:

```
OPENQASM 3;

qubit[1] q;
h q[0];   // This is valid
```

The physical qubits in a two qubit gate are not connected error

To use physical qubits, first confirm that the device uses physical qubits by checking `device.properties.action[DeviceActionType.OPENQASM].supportPhysicalQubits` and then verify the connectivity graph by checking `device.properties.paradigm.connectivity.connectivityGraph` or `device.properties.paradigm.connectivity.fullyConnected`.

```
OPENQASM 3;

cnot $0, $14;
```

This code generates the error message: `[line 3] has disconnected qubits 0 and 14`

Local simulator support warning

The `LocalSimulator` supports advanced features in OpenQASM that may not be available on QPUs or on-demand simulators. If your program contains language features specific only to the `LocalSimulator`, as seen in the following example, you will receive a warning.

```
qasm_string = """
qubit[2] q;

h q[0];
ctrl @ x q[0], q[1];
"""
qasm_program = Program(source=qasm_string)
```

This code generates the warning: `This program uses OpenQASM language features only supported in the LocalSimulator. Some of these features may not be supported on QPUs or on-demand simulators.

For more information on supported OpenQASM features, explore the page Advanced feature support for OpenQASM on the Local Simulator.

Security in Amazon Braket

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from data centers and network architectures that are built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The shared responsibility model describes this as security *of* the cloud and security *in* the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. Third-party auditors regularly test and verify the effectiveness of our security as part of the AWS Compliance Programs. To learn about the compliance programs that apply to Amazon Braket, see AWS Services in Scope by Compliance Program.
- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your company's requirements, and applicable laws and regulations.

This documentation helps you understand how to apply the shared responsibility model when using Braket. The following topics show you how to configure Braket to meet your security and compliance objectives. You also learn how to use other AWS services that help you to monitor and secure your Braket resources.

In this section:

- Shared responsibility for security
- Data protection
- Data retention
- Managing access to Amazon Braket
- Amazon Braket service-linked role
- Compliance validation for Amazon Braket
- Infrastructure Security in Amazon Braket
- Security of Amazon Braket Hardware Providers
- Amazon VPC endpoints for Amazon Braket

Shared responsibility for security

Security is a shared responsibility between AWS and you. The shared responsibility model describes this as security *of* the cloud and security *in* the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. Third-party auditors regularly test and verify the effectiveness of our security as part of the AWS Compliance Programs. To learn about the compliance programs that apply to Amazon Braket, see AWS Services in Scope by Compliance Program.
- **Security in the cloud** – You are responsible for maintaining control over your content that is hosted on this AWS infrastructure. This content includes the security configuration and management tasks for the AWS services that you use.

Data protection

The AWS shared responsibility model applies to data protection in Amazon Braket. As described in this model, AWS is responsible for protecting the global infrastructure that runs all of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on this infrastructure. You are also responsible for the security configuration and management tasks for the AWS services that you use. For more information about data privacy, see the Data Privacy FAQ. For information about data protection in Europe, see the AWS Shared Responsibility Model and GDPR blog post on the *AWS Security Blog*.

For data protection purposes, we recommend that you protect AWS account credentials and set up individual users with AWS IAM Identity Center or AWS Identity and Access Management (IAM). That way, each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with AWS resources. We require TLS 1.2 and recommend TLS 1.3.
- Set up API and user activity logging with AWS CloudTrail. For information about using CloudTrail trails to capture AWS activities, see Working with CloudTrail trails in the *AWS CloudTrail User Guide*.
- Use AWS encryption solutions, along with all default security controls within AWS services.

- Use advanced managed security services such as Amazon Macie, which assists in discovering and securing sensitive data that is stored in Amazon S3.
- If you require FIPS 140-3 validated cryptographic modules when accessing AWS through a command line interface or an API, use a FIPS endpoint. For more information about the available FIPS endpoints, see Federal Information Processing Standard (FIPS) 140-3.

We strongly recommend that you never put confidential or sensitive information, such as your customers' email addresses, into tags or free-form text fields such as a **Name** field. This includes when you work with Amazon Braket or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into tags or free-form text fields used for names may be used for billing or diagnostic logs. If you provide a URL to an external server, we strongly recommend that you do not include credentials information in the URL to validate your request to that server.

Data retention

After 90 days, Amazon Braket automatically removes all quantum task IDs and other metadata associated with your quantum tasks. As a result of this data retention policy, these tasks and results are no longer retrievable by search from the Amazon Braket console, although they remain stored in your S3 bucket.

If you need access to historical quantum tasks and results that are stored in your S3 bucket for longer than 90 days, you must keep a separate record of your task ID and other metadata associated with that data. Be sure to save the information prior to 90 days. You can use that saved information to retrieve the historical data.

Managing access to Amazon Braket

This chapter describes the permissions that are required to run Amazon Braket, or to restrict the access of specific users and roles. You can grant (or deny) the required permissions to any user or role in your account. To do so, attach the appropriate Amazon Braket policy to that user or role in your account as described in the following sections.

As a prerequisite, you must enable Amazon Braket. To enable Braket, be sure to sign in as a user or role that has (1) administrator permissions or (2) is assigned the **AmazonBraketFullAccess** policy and has permissions to create Amazon Simple Storage Service (Amazon S3) buckets.

In this section:

- Amazon Braket resources
- Notebooks and roles
- AWS managed policies for Amazon Braket
- Restrict user access to certain devices
- Restrict user access to certain notebook instances
- Restrict user access to certain S3 buckets

Amazon Braket resources

Braket creates one type of resource: the *quantum-task* resource. The AWS Resource Name (ARN) for this resource type is as follows:

- **Resource Name:** *AWS::Service::Braket*
- **ARN Regex:** *arn:${Partition}:braket:${Region}:${Account}:quantum-task/${RandomId}*

Notebooks and roles

You can use the noteboook resource type in Braket. A notebook is an Amazon SageMaker AI resource that Braket is able to share. To use a notebook with Braket, you must specify an IAM role with a name that begins with `AmazonBraketServiceSageMakerNotebook`.

To create a notebook, you must use a role with admin permissions or that has the following inline policy attached to it.

To create the role, follow the steps given in the Create a notebook page or have your administrator create it for you. Ensure that the **AmazonBraketFullAccess** policy is attached.

After you've created the role, you can reuse that role for all notebooks you launch in the future.

AWS managed policies for Amazon Braket

An AWS managed policy is a standalone policy that is created and administered by AWS. AWS managed policies are designed to provide permissions for many common use cases so that you can start assigning permissions to users, groups, and roles.

Keep in mind that AWS managed policies might not grant least-privilege permissions for your specific use cases because they're available for all AWS customers to use. We recommend that you reduce permissions further by defining customer managed policies that are specific to your use cases.

You cannot change the permissions defined in AWS managed policies. If AWS updates the permissions defined in an AWS managed policy, the update affects all principal identities (users, groups, and roles) that the policy is attached to. AWS is most likely to update an AWS managed policy when a new AWS service is launched or new API operations become available for existing services.

For more information, see AWS managed policies in the *IAM User Guide*.

Topics

- AWS managed policy: AmazonBraketFullAccess
- AWS managed policy: AmazonBraketJobsExecutionPolicy
- AWS managed policy: AmazonBraketServiceRolePolicy
- Amazon Braket updates to AWS managed policies

AWS managed policy: AmazonBraketFullAccess

The **AmazonBraketFullAccess** policy grants permissions for Amazon Braket operations, including permissions for these tasks:

- **Download containers from Amazon Elastic Container Registry** – To read and download container images that are used for the Amazon Braket Hybrid Jobs feature. The containers must conform to the format "arn:aws:ecr:::repository/amazon-braket".
- **Keep AWS CloudTrail logs** – For all *describe*, *get*, and *list* actions in addition to starting and stopping queries, testing metrics filters, and filtering log events. The AWS CloudTrail log file contains a record of all Amazon Braket API activity that occurs in your account.
- **Utilize roles to control resources** – To create a service-linked role in your account. The service-linked role has access to AWS resources on your behalf. It can be used only by the Amazon Braket service. Also, to pass in IAM roles to the Amazon Braket `CreateJob` API and to create a role and attach a policy scoped to AmazonBraketFullAccess to the role.
- **Create log groups, log events, and query log groups in order to maintain usage log files for your account** – To create, store, and view logging information about Amazon Braket usage in your account. Query metrics on hybrid jobs log groups. Encompass the proper Braket path and allow putting log data. Put metric data in CloudWatch.

- **Create and store data in Amazon S3 buckets, and list all buckets** – To create S3 buckets, list the S3 buckets in your account, and put objects into and get objects from any bucket in your account whose name begins with *amazon-braket-*. These permissions are required for Braket to put files containing results from processed quantum tasks into the bucket and to retrieve them from the bucket.
- **Pass IAM roles** – To pass in IAM roles to the `CreateJob` API.
- **Amazon SageMaker AI Notebook** – To create and manage SageMaker notebook instances scoped to the resource from "arn:aws:sagemaker:::notebook-instance/amazon-braket-".
- **Validate service quotas** – To create SageMaker AI notebooks and Amazon Braket Hybrid jobs, your resource counts cannot exceed quotas for your account.
- **View product pricing** – Review and plan quantum hardware costs before submitting your workloads.

To view the permissions for this policy, see AmazonBraketFullAccess in the *AWS Managed Policy Reference*.

AWS managed policy: AmazonBraketJobsExecutionPolicy

The **AmazonBraketJobsExecutionPolicy** policy grants permissions for execution roles used in Amazon Braket Hybrid Jobs as follows:

- **Download containers from Amazon Elastic Container Registry** - Permissions to read and download container images that are used for the Amazon Braket Hybrid Jobs feature. Containers must conform to the format "arn:aws:ecr:*:*:repository/amazon-braket*".
- **Create log groups and log events and query log groups in order to maintain usage log files for your account** – Create, store, and view logging information about Amazon Braket usage in your account. Query metrics on hybrid jobs log groups. Encompass the proper Braket path and allow putting log data. Put metric data in CloudWatch.
- **Store data in Amazon S3 buckets** – List the S3 buckets in your account, put objects into and get objects from any bucket in your account that starts with *amazon-braket-* in its name. These permissions are required for Braket to put files containing results from processed quantum tasks into the bucket, and to retrieve them from the bucket.
- **Pass IAM roles** – Passing in IAM roles to the CreateJob API. Roles must conform to the format arn:aws:iam::*:role/service-role/AmazonBraketJobsExecutionRole*.

To view the permissions for this policy, see AmazonBraketJobsExecutionPolicy in the *AWS Managed Policy Reference*.

AWS managed policy: AmazonBraketServiceRolePolicy

The **AmazonBraketServiceRolePolicy** policy grants permissions for Amazon Braket operations, including permissions for these tasks:

- **Amazon S3 –** permissions to list the buckets in your account, and put objects into and get objects from any bucket in your account with a name that starts with `amazon-braket-`.
- **Amazon CloudWatch Logs –** permissions to list and create log groups, create the associated log streams, and put events into the log group created for Amazon Braket.

For more information on service-linked roles, see Amazon Braket service-linked role.

To view the permissions for this policy, see AmazonBraketServiceRolePolicy in the *AWS Managed Policy Reference*.

Amazon Braket updates to AWS managed policies

The following table provides details about updates to AWS managed policies for Amazon Braket from the time this service began tracking these changes.

Change	Description	Date
AmazonBraketServiceRolePolicy - Resource management policy	Added the **"aws:ResourceAccount" : "${aws:PrincipalAccount}"** condition scope to Amazon S3 and CloudWatch logs actions.	July 11, 2025
AmazonBraketFullAccess - Full access policy for Braket	Added the **"pricing:GetProducts"** action.	April 14, 2025
AmazonBraketFullAccess - Full access policy for Braket	Added the **"aws:ResourceAccount": "${aws:PrincipalAccount}"** condition scope to S3 actions.	March 7, 2025

Change	Description	Date
AmazonBraketFullAccess - Full access policy for Braket	Added the **servicequotas:GetServiceQuota** and **cloudwatch:GetMetricData** actions.	March 24, 2023
AmazonBraketFullAccess - Full access policy for Braket	Added the **s3:ListAllMyBuckets** permissions to view and inspect the used Amazon S3 buckets.	March 31, 2022
AmazonBraketFullAccess - Full access policy for Braket	Braket adjusted iam:PassRole permissions for AmazonBraketFullAccess to include the `service-role/` path.	November 29, 2021
AmazonBraketJobsExecutionPolicy - Hybrid jobs execution policy for Amazon Braket Hybrid Jobs	Braket updated the hybrid jobs execution role ARN to include the `service-role/` path.	November 29, 2021
Braket started tracking changes	Braket started tracking changes for its AWS managed policies.	November 29, 2021

Restrict user access to certain devices

To restrict user access for certain Braket devices, you can add a *deny permissions* policy to a specific IAM role.

The following actions can be restricted:

- `CreateQuantumTask` - to deny quantum task creation on specified devices.
- `CreateJob` - to deny hybrid job creation on specified devices.
- `GetDevice` - to deny getting details of specified devices.

The following example restricts access to all QPUs for the AWS account `123456789012`.

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
    "Effect": "Deny",
    "Action": [
      "braket:CreateQuantumTask",
      "braket:CreateJob",
      "braket:GetDevice"
    ],
    "Resource": [
    "arn:aws:braket:*:*:device/qpu/*"
    ],
    "Condition": {
             "StringEquals": {
                 "aws:PrincipalAccount": "123456789012"
             }
      }
    }
  ]
}
```

> **Note**
>
> Exclude the `braket:GetDevice` Action from the policy to enable a user's Read access to the device properties such as device availability, calibration data, and pricing through the Braket console.

To adapt this code, substitute the Amazon Resource Number (ARN) of the restricted device for the string shown in the previous example. This string provides the **Resource** value. In Braket, a device represents a QPU or simulator that you can call to run quantum tasks. The devices available are listed on the Devices page. There are two schemas used to specify access to these devices:

- `arn:aws:braket:<region>:*:device/qpu/<provider>/<device_id>`
- `arn:aws:braket:<region>:*:device/quantum-simulator/<provider>/<device_id>`

Here are examples for various types of device access

- To select all QPUs across all regions: `arn:aws:braket:*:*:device/qpu/*`
- To select all QPUs in the us-west-2 region ONLY: `arn:aws:braket:us-west-2:*:device/qpu/*`
- Equivalently, to select all QPUs in the us-west-2 region ONLY (since devices are a service resource, not a customer resource): `arn:aws:braket:us-west-2:*:device/qpu/*`
- To restrict access to all on-demand simulator devices: `arn:aws:braket:*:*:device/quantum-simulator/*`
- To restrict access to devices from a certain provider (for example, to Rigetti QPU devices): `arn:aws:braket:*:*:device/qpu/rigetti/*`
- To restrict access to the TN1 device: `arn:aws:braket:*:*:device/quantum-simulator/amazon/tn1`
- To restrict access to all `Create` actions: `braket:Create*`

Restrict user access to certain notebook instances

To restrict access for certain users to specific Braket notebook instances, you can add a *deny permissions* policy to a specific role, user, or group.

The following example uses policy variables to efficiently restrict permissions to start, stop, and access specific notebook instances in the AWS account `123456789012`, which is named according to the user who should have access (for example, user `Alice` would have access to a notebook instance named `amazon-braket-Alice`).

Restrict user access to certain S3 buckets

To restrict access for certain users to specific Amazon S3 buckets, you can add a deny policy to a specific role, user, or group.

The following example restricts permissions to retrieve and place objects into a specific S3 bucket (`arn:aws:s3:::amazon-braket-us-east-1-123456789012-Alice`) and also restricts the listing of those objects.

JSON

```
{
```

```
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Action": [
        "s3:ListBucket"
      ],
      "NotResource": [
        "arn:aws:s3:::amazon-braket-us-east-1-123456789012-Alice"
      ]
    },
    {
      "Effect": "Deny",
      "Action": [
        "s3:GetObject"
      ],
      "NotResource": [
        "arn:aws:s3:::amazon-braket-us-east-1-123456789012-Alice/*"
      ]
    }
  ]
}
```

To restrict access to the bucket for a certain notebook instance, you can add the preceding policy to the notebook execution role.

Amazon Braket service-linked role

When you enable Amazon Braket, a *service-linked role* is created in your account.

A service-linked role is a unique type of IAM role that, in this case, is linked directly to Amazon Braket. The Amazon Braket service-linked role is predefined to include all the permissions that Braket requires when calling other AWS services on your behalf.

A service-linked role makes setting up Amazon Braket easier because you don't have to add the necessary permissions manually. Amazon Braket defines the permissions of its service-linked roles. Unless you change these definitions, only Amazon Braket can assume its roles. The defined permissions include the *trust policy* and the *permissions policy*. The permissions policy cannot be attached to any other IAM entity.

The service-linked role that Amazon Braket sets up is part of the AWS Identity and Access Management (IAM) service-linked roles capability. For information about other AWS services that support service-linked roles, see AWS Services That Work with IAM and look for the services that have **Yes** in the **Service-Linked Role** column. Choose a **Yes** with a link to view the service-linked role documentation for that service.

For more information on the AWS managed policy for service-linked roles, see AmazonBraketServiceRolePolicy.

Compliance validation for Amazon Braket

> **Note**
>
> AWS compliance reports don't cover QPUs from third-party hardware providers who can choose to go through their own independent audits.

To learn whether an AWS service is within the scope of specific compliance programs, see AWS services in Scope by Compliance Program and choose the compliance program that you are interested in. For general information, see AWS Compliance Programs.

You can download third-party audit reports using AWS Artifact. For more information, see Downloading Reports in AWS Artifact.

Your compliance responsibility when using AWS services is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. AWS provides the following resources to help with compliance:

- Security Compliance & Governance – These solution implementation guides discuss architectural considerations and provide steps for deploying security and compliance features.
- HIPAA Eligible Services Reference – Lists HIPAA eligible services. Not all AWS services are HIPAA eligible.
- AWS Compliance Resources – This collection of workbooks and guides might apply to your industry and location.
- AWS Customer Compliance Guides – Understand the shared responsibility model through the lens of compliance. The guides summarize the best practices for securing AWS services and map the guidance to security controls across multiple frameworks (including National Institute of

Standards and Technology (NIST), Payment Card Industry Security Standards Council (PCI), and International Organization for Standardization (ISO)).

- Evaluating Resources with Rules in the *AWS Config Developer Guide* – The AWS Config service assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations.
- AWS Security Hub – This AWS service provides a comprehensive view of your security state within AWS. Security Hub uses security controls to evaluate your AWS resources and to check your compliance against security industry standards and best practices. For a list of supported services and controls, see Security Hub controls reference.
- Amazon GuardDuty – This AWS service detects potential threats to your AWS accounts, workloads, containers, and data by monitoring your environment for suspicious and malicious activities. GuardDuty can help you address various compliance requirements, like PCI DSS, by meeting intrusion detection requirements mandated by certain compliance frameworks.
- AWS Audit Manager – This AWS service helps you continuously audit your AWS usage to simplify how you manage risk and compliance with regulations and industry standards.

Infrastructure Security in Amazon Braket

As a managed service, Amazon Braket is protected by AWS global network security. For information about AWS security services and how AWS protects infrastructure, see AWS Cloud Security. To design your AWS environment using the best practices for infrastructure security, see Infrastructure Protection in *Security Pillar AWS Well‐Architected Framework*.

You use AWS published API calls to access Amazon Braket through the network. Clients must support the following:

- Transport Layer Security (TLS). We require TLS 1.2 and recommend TLS 1.3.
- Cipher suites with perfect forward secrecy (PFS) such as DHE (Ephemeral Diffie-Hellman) or ECDHE (Elliptic Curve Ephemeral Diffie-Hellman). Most modern systems such as Java 7 and later support these modes.

Additionally, requests must be signed by using an access key ID and a secret access key that is associated with an IAM principal. Or you can use the AWS Security Token Service (AWS STS) to generate temporary security credentials to sign requests.

You can call these API operations from any network location, but Braket does support resource-based access policies, which can include restrictions based on the source IP address. You can also use Braket policies to control access from specific Amazon Virtual Private Cloud (Amazon VPC) endpoints or specific VPCs. Effectively, this isolates network access to a given Braket resource from only the specific VPC within the AWS network.

Security of Amazon Braket Hardware Providers

QPUs on Amazon Braket are hosted by third-party hardware providers. When you run your quantum task on a QPU, Amazon Braket uses the DeviceARN as an identifier when sending the circuit to the specified QPU for processing.

If you use Amazon Braket for access to quantum computing hardware operated by one of the third-party hardware providers, your circuit and its associated data are processed by hardware providers outside of facilities operated by AWS. Information about the physical location and AWS Region where each QPU is available can be found in the **Device Details** section of the Amazon Braket console.

Your content is anonymized. Only the content necessary to process the circuit is sent to third parties. AWS account information is not transmitted to third parties.

All data is encrypted at rest and in transit. Data is decrypted for processing only. Amazon Braket third-party providers are not permitted to store or use your content for purposes other than processing your circuit. Once the circuit completes, the results are returned to Amazon Braket and stored in your S3 bucket.

The security of Amazon Braket third-party quantum hardware providers is audited periodically, to ensure that standards of network security, access control, data protection, and physical security are met.

Amazon VPC endpoints for Amazon Braket

You can establish a private connection between your VPC and Amazon Braket by creating an interface VPC endpoint. Interface endpoints are powered by AWS PrivateLink, a technology that enables access to Braket APIs without an internet gateway, NAT device, VPN connection, or AWS Direct Connect connection. Instances in your VPC don't need public IP addresses to communicate with Braket APIs.

Each interface endpoint is represented by one or more Elastic Network Interfaces in your subnets.

With AWS PrivateLink, traffic between your VPC and Braket does not leave the Amazon network, which increases the security of data that you share with cloud-based applications, because it reduces your data's exposure to the public internet. For more information, see Access an AWS service using an interface VPC endpoint in the Amazon VPC User Guide.

In this section:

- Considerations for Amazon Braket VPC endpoints
- Set up Braket and PrivateLink
- Additional information about creating an endpoint
- Control access with Amazon VPC endpoint policies

Considerations for Amazon Braket VPC endpoints

Before you set up an interface VPC endpoint for Braket, ensure that you review Interface endpoint prerequisites in the *Amazon VPC User Guide*.

Braket supports making calls to all of its API actions from your VPC.

By default, full access to Braket is allowed through the VPC endpoint. You can control access if you specify VPC endpoint policies. For more information, see Control access to VPC endpoints using endpoint policies in the *Amazon VPC User Guide*.

Set up Braket and PrivateLink

To use AWS PrivateLink with Amazon Braket, you must create an Amazon Virtual Private Cloud (Amazon VPC) endpoint as an interface, and then connect to the endpoint through the Amazon Braket API service.

Here are the general steps of this process, which are explained in detail in later sections.

- Configure and launch an Amazon VPC to host your AWS resources. If you already have a VPC, you can skip this step.
- Create an Amazon VPC endpoint for Braket
- Connect and run Braket quantum tasks through your endpoint

Step 1: Launch an Amazon VPC if needed

Remember that you can skip this step if your account already has a VPC in operation.

A VPC controls your network settings, such as the IP address range, subnets, route tables, and network gateways. Essentially, you are launching your AWS resources in a custom virtual network. For more information about VPCs, see the Amazon VPC User Guide.

Open the Amazon VPC console and create a new VPC with subnets, security groups, and network gateways.

Step 2: Create an interface VPC endpoint for Braket

You can create a VPC endpoint for the Braket service using either the Amazon VPC console or the AWS Command Line Interface (AWS CLI). For more information, see Create a VPC endpoint in the *Amazon VPC User Guide*.

To create a VPC endpoint in the console, open the Amazon VPC console, open the **Endpoints** page, and proceed to create the new endpoint. Make note of the endpoint ID for later reference. It is required as part of the –`endpoint-url` flag when you are making certain calls to the Braket API.

Create the VPC endpoint for Braket using the following service name:

- `com.amazonaws.substitute_your_region.braket`

For more information, see Access an AWS service using an interface VPC endpoint in the *Amazon VPC User Guide*.

Step 3: Connect and run Braket quantum tasks through your endpoint

After you have created a VPC endpoint, you can run CLI commands that include the `endpoint-url` parameter to specify interface endpoints to the API or runtime, such as the following example:

```
aws braket search-quantum-tasks --endpoint-url
 VPC_Endpoint_ID.braket.substituteYourRegionHere.vpce.amazonaws.com
```

If you enable private DNS hostnames for your VPC endpoint, you don't need to specify the endpoint as a URL in your CLI commands. Instead, the Amazon Braket API DNS hostname, which the CLI and Braket SDK use by default, resolves to your VPC endpoint. It has the form shown in the following example:

```
https://braket.substituteYourRegionHere.amazonaws.com
```

The blog post called Direct access to Amazon SageMaker AI notebooks from Amazon VPC by using an AWS PrivateLink endpoint provides an example of how to set up an endpoint to make secure connections to SageMaker notebooks, which are similar to Amazon Braket notebooks.

If you're following the steps in the blog post, remember to substitute the name **Amazon Braket** for **Amazon SageMaker AI**. For **Service Name** enter `com.amazonaws.us-east-1.braket` or substitute your correct AWS Region name into that string, if your Region is not *us-east-1*.

Additional information about creating an endpoint

- For information about how to create a VPC with private subnets, see Create a VPC with private subnets.
- For information about creating and configuring an endpoint using the Amazon VPC console or the AWS CLI, see Create a VPC endpoint in the *Amazon VPC User Guide*.
- For information about creating and configuring an endpoint using AWS CloudFormation, see the AWS::EC2::VPCEndpoint resource in the *AWS CloudFormation User Guide*.

Control access with Amazon VPC endpoint policies

To control connectivity access to Amazon Braket, you can attach an AWS Identity and Access Management (IAM) endpoint policy to your Amazon VPC endpoint. The policy specifies the following information:

- The principal (user or role) that can perform actions.
- The actions that can be performed.
- The resources on which actions can be performed.

For more information, see Control access to VPC endpoints using endpoint policies in the *Amazon VPC User Guide*.

Example: VPC endpoint policy for Braket actions

The following example shows an endpoint policy for Braket. When attached to an endpoint, this policy grants access to the listed Braket actions for all principals on all resources.

```
{
 "Statement":[
```

```
  {
    "Principal":"*",
    "Effect":"Allow",
    "Action":[
      "braket:action-1",
      "braket:action-2",
      "braket:action-3"
      ],
    "Resource":"*"
    }
   ]
}
```

You can create complex IAM rules by attaching multiple endpoint policies. For more information and examples, see:

- Amazon Virtual Private Cloud Endpoint Policies for Step Functions
- Creating Granular IAM Permissions for Non-Admin Users
- Control access to VPC endpoints using endpoint policies

Logging and monitoring

After you submit a quantum task through the Amazon Braket service, you can closely monitor the status and progression of that task through the Amazon Braket SDK and console. This provides you with a centralized interface to track the implementation of your workloads, identify any potential bottlenecks or issues, and take appropriate actions to optimize the performance and reliability of your quantum applications. When the quantum task completes, Braket saves the results in your specified Amazon S3 location. Completion time for quantum tasks can vary, especially for those running on quantum processing unit (QPU) devices. This is largely due to the length of the execution queue, as quantum hardware resources are shared among multiple users.

List of status types:

- `CREATED` – Amazon Braket received your quantum task.
- `QUEUED` – Amazon Braket processed your quantum task and it is now waiting to run on the device.
- `RUNNING` – Your quantum task is running on a QPU or on-demand simulator.
- `COMPLETED` – Your quantum task finished running on the QPU or on-demand simulator.
- `FAILED` – Your quantum task attempted to run and failed. Depending on the reason your quantum task failed, try submitting your quantum task again.
- `CANCELLED` – You cancelled the quantum task. The quantum task did not run.

In this section:

- Tracking quantum tasks from the Amazon Braket SDK
- Monitoring quantum tasks through the Amazon Braket console
- Tagging Amazon Braket resources
- Monitoring your quantum tasks with EventBridge
- Monitoring your metrics with CloudWatch
- Logging your quantum tasks with CloudTrail
- Advanced logging with Amazon Braket

Tracking quantum tasks from the Amazon Braket SDK

The command `device.run(…)` defines a quantum task with a unique quantum task ID. You can query and track the status with `task.state()` as shown in the following example.

Note: `task = device.run()` is an asynchronous operation, which means that you can keep working while the system processes your quantum task in the background.

Retrieve a result

When you call `task.result()`, the SDK begins polling Amazon Braket to see whether the quantum task is complete. The SDK uses the polling parameters you defined in `.run()`. After the quantum task is complete, the SDK retrieves the result from the S3 bucket and returns it as a `QuantumTaskResult` object.

```
# create a circuit, specify the device and run the circuit
circ = Circuit().rx(0, 0.15).ry(1, 0.2).cnot(0,2)
device = AwsDevice("arn:aws:braket:::device/quantum-simulator/amazon/sv1")
task = device.run(circ, s3_location, shots=1000)

# get ID and status of submitted task
task_id = task.id
status = task.state()
print('ID of task:', task_id)
print('Status of task:', status)
# wait for job to complete
while status != 'COMPLETED':
    status = task.state()
    print('Status:', status)
```

```
ID of task:
arn:aws:braket:us-west-2:123412341234:quantum-task/b68ae94b-1547-4d1d-aa92-1500b82c300d
Status of task: QUEUED
Status: QUEUED
Status: QUEUED
Status: QUEUED
Status: QUEUED
Status: QUEUED
Status: QUEUED
Status: QUEUED
Status: RUNNING
Status: RUNNING
```

```
Status: COMPLETED
```

Cancel a quantum task

To cancel a quantum task, call the `cancel()` method, as shown in the following example.

```
# cancel quantum task
task.cancel()
status = task.state()
print('Status of task:', status)
```

```
Status of task: CANCELLING
```

Check the metadata

You can check the metadata of the finished quantum task, as shown in the following example.

```
# get the metadata of the quantum task
metadata = task.metadata()
# example of metadata
shots = metadata['shots']
date = metadata['ResponseMetadata']['HTTPHeaders']['date']
# print example metadata
print("{} shots taken on {}.".format(shots, date))

# print name of the s3 bucket where the result is saved
results_bucket = metadata['outputS3Bucket']
print('Bucket where results are stored:', results_bucket)
# print the s3 object key (folder name)
results_object_key = metadata['outputS3Directory']
print('S3 object key:', results_object_key)

# the entire look-up string of the saved result data
look_up = 's3://'+results_bucket+'/'+results_object_key
print('S3 URI:', look_up)
```

```
1000 shots taken on Wed, 05 Aug 2020 14:44:22 GMT.
Bucket where results are stored: amazon-braket-123412341234
S3 object key: simulation-output/b68ae94b-1547-4d1d-aa92-1500b82c300d
S3 URI: s3://amazon-braket-123412341234/simulation-output/b68ae94b-1547-4d1d-
aa92-1500b82c300d
```

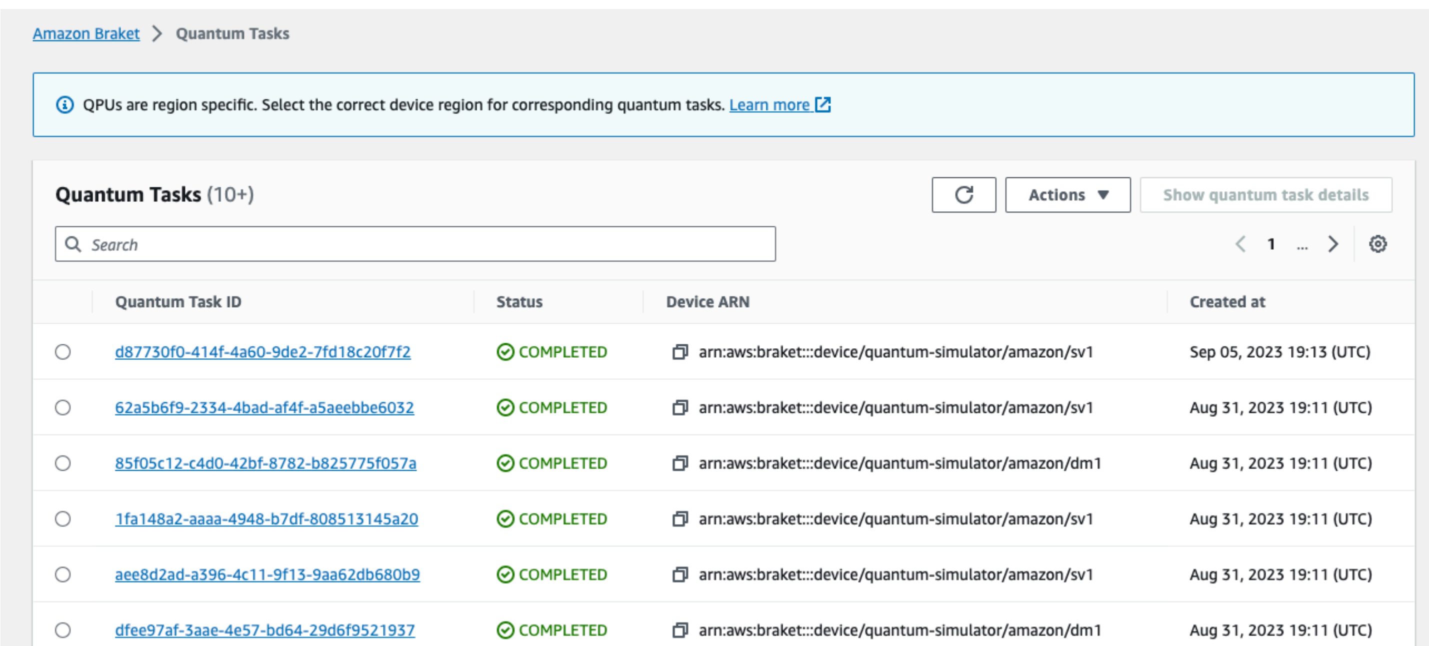
Retrieve a quantum task or result

If your kernel dies after you submit the quantum task or if you close your notebook or computer, you can reconstruct the `task` object with its unique ARN (quantum task ID). Then you can call `task.result()` to get the result from the S3 bucket where it is stored.

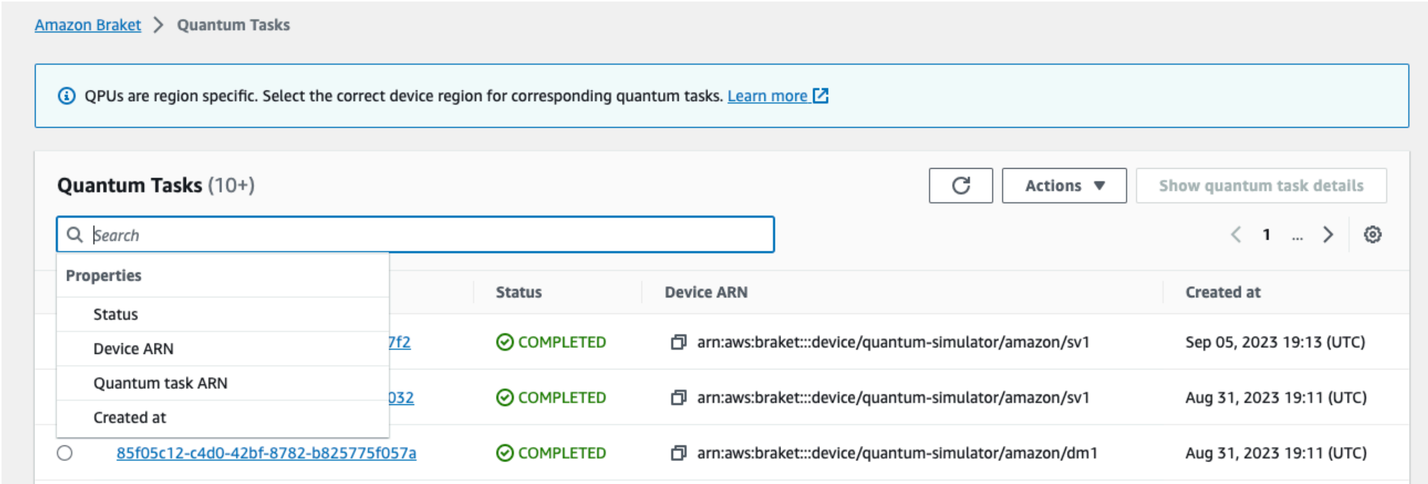
```
from braket.aws import AwsSession, AwsQuantumTask

# restore task with unique arn
task_load = AwsQuantumTask(arn=task_id)
# retrieve the result of the task
result = task_load.result()
```

Monitoring quantum tasks through the Amazon Braket console

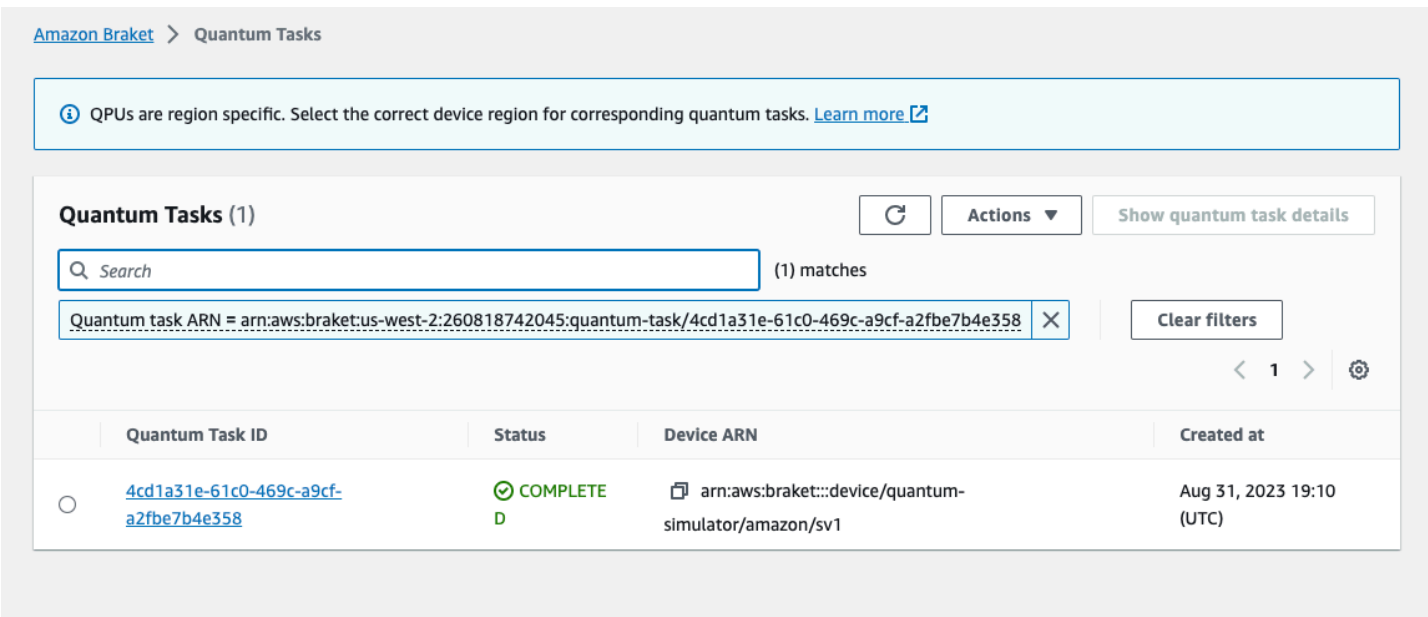
Amazon Braket offers a convenient way of monitoring the quantum task through the [Amazon Braket console](). All submitted quantum tasks are listed in the **Quantum Tasks** field as shown in the following figure. This service is *Region-specific*, which means that you can only view those quantum tasks created in the specific AWS Region.



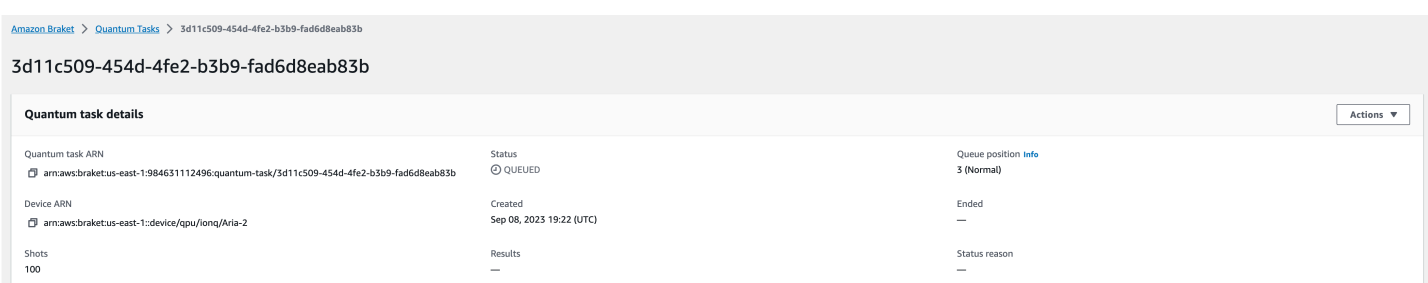
You can search for particular quantum tasks through the navigation bar. The search can be based on Quantum Task ARN (ID), status, device, and creation time. The options appear automatically when you select the navigation bar, as shown in the following example.



The following image shows an example of searching for a quantum task based on its unique quantum task ID, which can be obtained by calling `task.id`.



Additionally, seen in the figure below, the status of a quantum task can be monitored while it is in a QUEUED state. Clicking on the quantum task ID shows the details page. This page displays the dynamic queue position for your quantum task relative to the device it will process on.



Quantum tasks submitted as part of a hybrid job will have priority when in queue. Quantum tasks submitted outside of a hybrid job will have normal queuing priority.

Customers wishing to query the Braket SDK, can obtain their quantum task and hybrid job queue positions programmatically. For more information see the When will my task run page.

Tagging Amazon Braket resources

A *tag* is a custom attribute label that you assign or that AWS assigns to an AWS resource. A tag is *metadata* that tells more about your resource. Each tag consists of a *key* and a *value*. Together these are known as *key-value pairs*. For tags that you assign, you define the key and value.

In the Amazon Braket console, you can navigate to a quantum task or a notebook and view the list of tags associated with it. You can add a tag, remove a tag, or modify a tag. You can tag a quantum task or notebook upon creation, and then manage associated tags through the console, AWS CLI, or API.

More about AWS and tags

- For general information on tagging, including naming and usage conventions, see What is Tag Editor? in the *Tagging AWS Resources and Tag Editor* User Guide.
- For information about restrictions on tagging, see Tag naming limits and requirements in the *Tagging AWS Resources and Tag Editor* User Guide.
- For best practices and tagging strategies, see Best Practices for Tagging AWS Resources.
- For a list of services that support using tags, see the *Resource Groups Tagging API Reference*.

The following sections provide more specific information about tags for Amazon Braket.

In this section:

- Using tags
- Supported resources for tagging in Amazon Braket
- Tagging with the Amazon Braket API
- Tagging restrictions
- Managing tags in Amazon Braket
- Example of AWS CLI tagging in Amazon Braket

Using tags

Tags can organize your resources into categories that are useful to you. For example, you can assign a "Department" tag to specify the department that owns this resource.

Each tag has two parts:

- A tag key (for example, *CostCenter*, *Environment*, or *Project*). Tag keys are case sensitive.
- An optional field known as a tag value (for example, *111122223333* or *Production*). Omitting the tag value is the same as using an empty string. Like tag keys, tag values are case sensitive.

Tags help you do the following things:

- **Identify and organize your AWS resources.** Many AWS services support tagging, so you can assign the same tag to resources from different services to indicate that the resources are related.
- **Track your AWS costs.** You activate these tags on the AWS Billing and Cost Management dashboard. AWS uses the tags to categorize your costs and deliver a monthly cost allocation report to you. For more information, see Use cost allocation tags in the *AWS Billing and Cost Management User Guide*.
- **Control access to your AWS resources.** For more information, see Controlling access using tags.

Supported resources for tagging in Amazon Braket

The following resource type in Amazon Braket supports tagging:

- `quantum-task` resource
- **Resource Name:** `AWS::Service::Braket`
- **ARN Regex:** `arn:${Partition}:braket:${Region}:${Account}:quantum-task/${RandomId}`

Note: You can apply and manage tags for your Amazon Braket notebooks in the Amazon Braket console, by using the console to navigate to the notebook resource, although the notebooks actually are Amazon SageMaker AI resources. For more information, see Notebook Instance Metadata in the SageMaker documentation.

Tagging with the Amazon Braket API

- If you're using the Amazon Braket API to set up tags on a resource, call the `TagResource`API.

```
aws braket tag-resource --resource-arn $YOUR_TASK_ARN --tags {\"city\":
\"Seattle\"}
```

- To remove tags from a resource, call the `UntagResource`API.

```
aws braket list-tags-for-resource --resource-arn $YOUR_TASK_ARN
```

- To list all tags that are attached to a particular resource, call the `ListTagsForResource`API.

```
aws braket tag-resource --resource-arn $YOUR_TASK_ARN --tag-keys "[\"city
\",\"state\"]"
```

Tagging restrictions

The following basic restrictions apply to tags on Amazon Braket resources:

- Maximum number of tags that you can assign to a resource: 50
- Maximum key length: 128 Unicode characters
- Maximum value length: 256 Unicode characters
- Valid characters for key and value: `a-z, A-Z, 0-9, space`, and these characters: `_ . : / = + -` and `@`
- Keys and values are case sensitive.
- Don't use `aws` as a prefix for keys; it's reserved for AWS use.

Managing tags in Amazon Braket

You set tags as *properties* on a *resource*. You can view, add, modify, list, and delete tags through the Amazon Braket console, the Amazon Braket API, or the AWS CLI. For more information, see the Amazon Braket API reference.

In this section:

- Adding tags

- Viewing tags
- Editing tags
- Removing tags

Adding tags

You can add tags to taggable resources at the following times:

- **When you create the resource:** Use the console, or include the `Tags` parameter with the `Create` operation in the AWS API.
- **After you create the resource:** Use the console to navigate to the quantum task or notebook resource, or call the `TagResource` operation in the AWS API.

To add tags to a resource when you create it, you also need permission to create a resource of the specified type.

Viewing tags

You can view the tags on any of the taggable resources in Amazon Braket by using the console to navigate to the task or notebook resource, or by calling the AWS `ListTagsForResource` API operation.

You can use the following AWS API command to view tags on a resource:

- **AWS API:** `ListTagsForResource`

Editing tags

You can edit tags by using the console to navigate to the quantum task or notebook resource or you can use the following command to modify the value for a tag attached to a taggable resource. When you specify a tag key that already exists, the value for that key is overwritten:

- **AWS API:** `TagResource`

Removing tags

You can remove tags from a resource by specifying the keys to remove, by using the console to navigate to the quantum task or notebook resource, or when calling the `UntagResource` operation.

- **AWS API:** `UntagResource`

Example of AWS CLI tagging in Amazon Braket

When you are working with the AWS Command Line Interface (AWS CLI) to interact with Amazon Braket, the following code is an example command that demonstrates how to create a tag that applies to a quantum task you create. In this example, the task is being executed on the SV1 quantum simulator with parameter settings specified for the Rigetti quantum processing unit (QPU). It is imporant that inside the example command the tag is specified at the very end, after all of the other required parameters. In this case, the tag has a **Key** of `state` and **Value** of `Washington`. These tags could be used to help categorize or identify this particular quantum task.

```
aws braket create-quantum-task --action /
"{\"braketSchemaHeader\": {\"name\": \"braket.ir.jaqcd.program\", /
    \"version\": \"1\"}, /
    \"instructions\": [{\"angle\": 0.15, \"target\": 0, \"type\": \"rz\"}], /
    \"results\": null, /
    \"basis_rotation_instructions\": null}" /
  --device-arn "arn:aws:braket:::device/quantum-simulator/amazon/sv1" /
  --output-s3-bucket  "my-example-braket-bucket-name" /
  --output-s3-key-prefix "my-example-username"  /
  --shots 100   /
  --device-parameters /
  "{\"braketSchemaHeader\": /
     {\"name\": \"braket.device_schema.rigetti.rigetti_device_parameters\", /
      \"version\": \"1\"}, \"paradigmParameters\": /
       {\"braketSchemaHeader\": /
         {\"name\": \"braket.device_schema.gate_model_parameters\", /
          \"version\": \"1\"}, /
          \"qubitCount\": 2}}" /
          --tags {\"state\":\”Washington\"}
```

This example demonstrates how you can apply tags to your quantum tasks when running them through the AWS CLI, which is helpful for organizing and tracking your Braket resources.

Monitoring your quantum tasks with EventBridge

Amazon EventBridge monitors status change events in Amazon Braket quantum tasks. Events from Amazon Braket are delivered to EventBridge, almost in real time. You can write rules that indicate which events interest you, including automated actions to take when an event matches a rule. Automatic actions that can be triggered include these:

- Invoking an AWS Lambda function
- Activating an AWS Step Functions state machine
- Notifying an Amazon SNS topic

EventBridge monitors these Amazon Braket status change events:

- The state of qauntum task changes

Amazon Braket guarantees delivery of quantum task status change events. These events are delivered at least once, but possibly out of order.

For more information, see the Events in Amazon EventBridge.

In this section:

- Monitor quantum task status with EventBridge
- Example Amazon Braket EventBridge event

Monitor quantum task status with EventBridge

With EventBridge, you can create rules that define actions to take when Amazon Braket sends notification of a status change regarding a Braket quantum task. For example, you can create a rule that sends you an email message each time the status of a quantum task changes.

1. Log in to AWS using an account that has permissions to use EventBridge and Amazon Braket.
2. Open the Amazon EventBridge console.
3. Using the following values, create an EventBridge rule:
    - For **Rule type**, choose **Rule with an event pattern**.
    - For **Event source**, choose **Other**.

- In the **Event pattern** section, choose **Custom patterns (JSON editor)**, and then paste the following event pattern into the text area:

```
{
  "source": [
    "aws.braket"
  ],
  "detail-type": [
    "Braket Task State Change"
  ]
}
```

  To capture all events from Amazon Braket, exclude the `detail-type` section as shown in the following code:

```
{
  "source": [
    "aws.braket"
  ]
}
```

- For **Target types**, choose **AWS service**, and for **Select a target**, choose a target such as an Amazon SNS topic or AWS Lambda function. The target is triggered when a quantum task state change event is received from Amazon Braket.

  For example, use an Amazon Simple Notification Service (SNS) topic to send an email or text message when an event occurs. To do that, first create an Amazon SNS topic using the Amazon SNS console. To learn more, see Using Amazon SNS for user notifications.

For details about creating rules, see Creating Amazon EventBridge rules that react to events.

Example Amazon Braket EventBridge event

For information on the fields for an Amazon Braket Quantum Task Status Change event, see Events in Amazon EventBridge.

The following attributes appear in the JSON "detail" field.

- **`quantumTaskArn`** (str): The quantum task for which this event was generated.
- **`status`** (Optional[str]): The status to which the quantum task transitioned.

- **deviceArn** (str): The device specified by the user for which this quantum task was created.
- **shots** (int): The number of shots requested by the user.
- **outputS3Bucket** (str): The output bucket specified by the user.
- **outputS3Directory** (str): The output key prefix specified by the user.
- **createdAt** (str): The quantum task creation time as an ISO-8601 string.
- **endedAt** (Optional[str]): The time at which the quantum task reached a terminal state. This field is present only when the quantum task has transitioned to a terminal state.

The following JSON code shows an example of an Amazon Braket Quantum Task Status Change event.


```
{
    "version":"0",
    "id":"6101452d-8caf-062b-6dbc-ceb5421334c5",
    "detail-type":"Braket Task State Change",
    "source":"aws.braket",
    "account":"012345678901",
    "time":"2021-10-28T01:17:45Z",
    "region":"us-east-1",
    "resources":[
        "arn:aws:braket:us-east-1:012345678901:quantum-task/834b21ed-77a7-4b36-a90c-
c776afc9a71e"
    ],
    "detail":{
        "quantumTaskArn":"arn:aws:braket:us-east-1:012345678901:quantum-
task/834b21ed-77a7-4b36-a90c-c776afc9a71e",
        "status":"COMPLETED",
        "deviceArn":"arn:aws:braket:::device/quantum-simulator/amazon/sv1",
        "shots":"100",
        "outputS3Bucket":"amazon-braket-0260a8bc871e",
        "outputS3Directory":"sns-testing/834b21ed-77a7-4b36-a90c-c776afc9a71e",
        "createdAt":"2021-10-28T01:17:42.898Z",
        "eventName":"MODIFY",
        "endedAt":"2021-10-28T01:17:44.735Z"
      }
}
```

Monitoring your metrics with CloudWatch

You can monitor Amazon Braket using Amazon CloudWatch, which collects raw data and processes it into readable, near real-time metrics. You view historical information generated up to 15 months ago or search metrics that have been updated in the last 2 weeks in the Amazon CloudWatch console to gain a better perspective on how Amazon Braket is performing. To learn more, see Using CloudWatch metrics.

> **Note**
>
> You can view the CloudWatch log streams for Amazon Braket notebooks by navigating to the **Notebook detail** page on the Amazon SageMaker AI console. Additional Amazon Braket notebook settings are available through the SageMaker console.

In this section:

- Amazon Braket metrics and dimensions

Amazon Braket metrics and dimensions

Metrics are the fundamental concept in CloudWatch. A metric represents a time-ordered set of data points that are published to CloudWatch. Every metric is characterized by a set of dimensions. To learn more about metrics dimensions in CloudWatch, see CloudWatch dimensions.

Amazon Braket sends the following metric data, specific to Amazon Braket, into the Amazon CloudWatch metrics:

Quantum Task Metrics

Metrics are available if quantum tasks exist. They are displayed under **AWS/Braket/By Device** in the CloudWatch console.

Metric	Description
Count	Number of quantum tasks.
Latency	This metric is emitted when a quantum task has completed. It represents the total time

Metric	Description
	from quantum task initialization to completio n.

Dimensions for Quantum Task Metrics

The quantum task metrics are published with a dimension based on the `deviceArn` parameter, which has the form **arn:aws:braket:::device/xxx**.

Logging your quantum tasks with CloudTrail

Amazon Braket is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in Amazon Braket. CloudTrail captures all API calls for Amazon Braket as events. The calls captured include calls from the Amazon Braket console and code calls to the Amazon Braket operations. If you create a trail, you can enable continuous delivery of CloudTrail events to an Amazon S3 bucket, including events for Amazon Braket. If you do not configure a trail, you can still view the most recent events in the CloudTrail console in **Event history**. Using the information collected by CloudTrail, you can determine the request that was made to Amazon Braket, the IP address from which the request was made, who made the request, when it was made, and additional details.

To learn more about CloudTrail, see the AWS CloudTrail User Guide.

In this section:

- Amazon Braket information in CloudTrail
- Understanding Amazon Braket log file entries

Amazon Braket information in CloudTrail

CloudTrail is enabled on your AWS account when you create the account. When activity occurs in Amazon Braket, that activity is recorded in a CloudTrail event along with other AWS service events in **Event history**. You can view, search, and download recent events in your AWS account. For more information, see Viewing Events with CloudTrail Event History.

For an ongoing record of events in your AWS account, including events for Amazon Braket, create a trail. A *trail* enables CloudTrail to deliver log files to an Amazon S3 bucket. By default, when you create a trail in the console, the trail applies to all AWS Regions. The trail logs events from all Regions in the AWS partition and delivers the log files to the Amazon S3 bucket that you specify. Additionally, you can configure other AWS services to further analyze and act upon the event data collected in CloudTrail logs. For more information, see the following:

- Overview for Creating a Trail
- CloudTrail Supported Services and Integrations
- Configuring Amazon SNS Notifications for CloudTrail
- Receiving CloudTrail Log Files from Multiple Regions and Receiving CloudTrail Log Files from Multiple Accounts

All Amazon Braket actions are logged by CloudTrail. For example, calls to the `GetQuantumTask` or `GetDevice` actions generate entries in the CloudTrail log files.

Every event or log entry contains information about who generated the request. The identity information helps you determine the following:

- Whether the request was made with temporary security credentials for a role or federated user.
- Whether the request was made by another AWS service.

For more information, see the CloudTrail userIdentity Element.

Understanding Amazon Braket log file entries

A trail is a configuration that enables delivery of events as log files to an Amazon S3 bucket that you specify. CloudTrail log files contain one or more log entries. An event represents a single request from any source and includes information about the requested action, the date and time of the action, request parameters, and so on. CloudTrail log files are not an ordered stack trace of the public API calls, so they don't appear in any specific order.

The following example is a log entry for the `GetQuantumTask` action, which gets the details of a quantum task.

```
{
  "eventVersion": "1.05",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "foobar",
```

```
    "arn": "foobar",
    "accountId": "foobar",
    "accessKeyId": "foobar",
    "sessionContext": {
      "sessionIssuer": {
        "type": "Role",
        "principalId": "foobar",
        "arn": "foobar",
        "accountId": "foobar",
        "userName": "foobar"
      },
      "webIdFederationData": {},
      "attributes": {
        "mfaAuthenticated": "false",
        "creationDate": "2020-08-07T00:56:57Z"
      }
    }
  },
  "eventTime": "2020-08-07T01:00:08Z",
  "eventSource": "braket.amazonaws.com",
  "eventName": "GetQuantumTask",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "foobar",
  "userAgent": "aws-cli/1.18.110 Python/3.6.10
 Linux/4.9.184-0.1.ac.235.83.329.metal1.x86_64 botocore/1.17.33",
  "requestParameters": {
    "quantumTaskArn": "foobar"
  },
  "responseElements": null,
  "requestID": "20e8000c-29b8-4137-9cbc-af77d1dd12f7",
  "eventID": "4a2fdb22-a73d-414a-b30f-c0797c088f7c",
  "readOnly": true,
  "eventType": "AwsApiCall",
  "recipientAccountId": "foobar"
}
```

The following shows a log entry for the `GetDevice` action, which returns the details of a device event.

```
{
  "eventVersion": "1.05",
  "userIdentity": {
    "type": "AssumedRole",
```

```
    "principalId": "foobar",
    "arn": "foobar",
    "accountId": "foobar",
    "accessKeyId": "foobar",
    "sessionContext": {
      "sessionIssuer": {
        "type": "Role",
        "principalId": "foobar",
        "arn": "foobar",
        "accountId": "foobar",
        "userName": "foobar"
      },
      "webIdFederationData": {},
      "attributes": {
        "mfaAuthenticated": "false",
        "creationDate": "2020-08-07T00:46:29Z"
      }
    }
  },
  "eventTime": "2020-08-07T00:46:32Z",
  "eventSource": "braket.amazonaws.com",
  "eventName": "GetDevice",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "foobar",
  "userAgent": "Boto3/1.14.33 Python/3.7.6 Linux/4.14.158-129.185.amzn2.x86_64 exec-
env/AWS_ECS_FARGATE Botocore/1.17.33",
  "errorCode": "404",
  "requestParameters": {
    "deviceArn": "foobar"
  },
  "responseElements": null,
  "requestID": "c614858b-4dcf-43bd-83c9-bcf9f17f522e",
  "eventID": "9642512a-478b-4e7b-9f34-75ba5a3408eb",
  "readOnly": true,
  "eventType": "AwsApiCall",
  "recipientAccountId": "foobar"
}
```


Advanced logging with Amazon Braket

You can record the whole task-processing process using a logger. These advanced logging techniques allow you to see the background polling and create a record for later debugging.

To use the logger, we recommend changing the `poll_timeout_seconds` and `poll_interval_seconds` parameters, so that a quantum task can be long-running and the quantum task status is logged continuously, with results saved to a file. You can transfer this code to a Python script instead of a Jupyter notebook, so that the script can run as a process in the background.

Configure the logger

First, configure the logger so that all logs are written into a text file automatically, as shown in the following example lines.

```
# import the module
import logging
from datetime import datetime

# set filename for logs
log_file = 'device_logs-'+datetime.strftime(datetime.now(), '%Y%m%d%H%M%S')+'.txt'
print('Task info will be logged in:', log_file)

# create new logger object
logger = logging.getLogger("newLogger")

# configure to log to file device_logs.txt in the appending mode
logger.addHandler(logging.FileHandler(filename=log_file, mode='a'))

# add to file all log messages with level DEBUG or above
logger.setLevel(logging.DEBUG)
```

```
Task info will be logged in: device_logs-20200803203309.txt
```

Create and run the circuit

Now you can create a circuit, submit it to a device to run, and see what happens as shown in this example.

```
# define circuit
circ_log = Circuit().rx(0, 0.15).ry(1, 0.2).rz(2, 0.25).h(3).cnot(control=0,
 target=2).zz(1, 3, 0.15).x(4)
print(circ_log)
# define backend
device = AwsDevice("arn:aws:braket:::device/quantum-simulator/amazon/sv1")
```

```
# define what info to log
logger.info(
    device.run(circ_log, s3_location,
               poll_timeout_seconds=1200, poll_interval_seconds=0.25, logger=logger,
 shots=1000)
    .result().measurement_counts
)
```

Check the log file

You can check what is written into the file by entering the following command.

```
# print logs
! cat {log_file}
```

```
Task arn:aws:braket:us-west-2:123412341234:quantum-
task/5088ec6c-89cf-4338-9750-9f5bb12a0dc4: start polling for completion
Task arn:aws:braket:us-west-2:123412341234:quantum-
task/5088ec6c-89cf-4338-9750-9f5bb12a0dc4: task status CREATED
Task arn:aws:braket:us-west-2:123412341234:quantum-
task/5088ec6c-89cf-4338-9750-9f5bb12a0dc4: task status CREATED
Task arn:aws:braket:us-west-2:123412341234:quantum-
task/5088ec6c-89cf-4338-9750-9f5bb12a0dc4: task status QUEUED
Task arn:aws:braket:us-west-2:123412341234:quantum-
task/5088ec6c-89cf-4338-9750-9f5bb12a0dc4: task status RUNNING
Task arn:aws:braket:us-west-2:123412341234:quantum-
task/5088ec6c-89cf-4338-9750-9f5bb12a0dc4: task status RUNNING
Task arn:aws:braket:us-west-2:123412341234:quantum-
task/5088ec6c-89cf-4338-9750-9f5bb12a0dc4: task status COMPLETED
Counter({'00001': 493, '00011': 493, '01001': 5, '10111': 4, '01011': 3, '10101': 2})
```

Get the ARN from the log file

From the log file output that's returned, as shown in the previous example, you can obtain the ARN information. With the ARN ID, you can retrieve the result of the completed quantum task.

```
# parse log file for arn
with open(log_file) as openfile:
    for line in openfile:
        for part in line.split():
            if "arn:" in part:
                arn = part
```

```
                break
# remove final semicolon in logs
arn = arn[:-1]

# with this arn you can restore again task from unique arn
task_load = AwsQuantumTask(arn=arn, aws_session=AwsSession())

# get results of task
result = task_load.result()
```

Amazon Braket Quotas

The following table lists the service quotas for Amazon Braket. Service quotas, also referred to as limits, are the maximum number of service resources or operations for your AWS account.

Some quotas can be increased. For more information, see AWS service quotas.

- Burst rate quotas cannot be increased.
- The maximum rate increase for adjustable quotas (except burst rate, which cannot be adjusted) is 2X the specified default rate limit. For example, a default quota of 60 can be adjusted to a maximum of 120.
- The adjustable quota for concurrent SV1 (DM1) quantum tasks allows a maximum of 60 per AWS Region.
- The maximum allowed number of compute instances for a hybrid job is 1, and the quotas are adjustable.

Resource	Description	Limits	Adjustable
Rate of API requests	The maximum number of requests per second that you can send in this account in the current Region.	140	Yes
Burst rate of API requests	The maximum number of additiona l requests per second (RPS) that you can send in one burst in this account in the current Region.	600	No
Rate of `CreateQuantumTask` requests	The maximum number of `CreateQua`	20 per second	Yes

Resource	Description	Limits	Adjustable
	`ntumTask` requests you can send per second in this account per Region.		
Burst rate of `CreateQuantumTask` requests	The maximum number of additional `CreateQuantumTask` requests per second (RPS) that you can send in one burst in this account in the current Region.	40	No
Rate of `SearchQuantumTasks` requests	The maximum number of `SearchQuantumTasks` requests you can send per second in this account per Region.	5 per second	Yes
Burst rate of `SearchQuantumTasks` requests	The maximum number of additional `SearchQuantumTasks` requests per second (RPS) that you can send in one burst in this account in the current Region.	50	No

Resource	Description	Limits	Adjustable
Rate of `GetQuantumTask` requests	The maximum number of `GetQuantumTask` requests you can send per second in this account per Region.	100 per second	Yes
Burst rate of `GetQuantumTask` requests	The maximum number of additional `GetQuantumTask` requests per second (RPS) that you can send in one burst in this account in the current Region.	500	No
Rate of `CancelQuantumTask` requests	The maximum number of `CancelQuantumTask` requests you can send per second in this account per Region.	2 per second	Yes
Burst rate of `CancelQuantumTask` requests	The maximum number of additional `CancelQuantumTask` requests per second (RPS) that you can send in one burst in this account in the current Region.	20	No

Resource	Description	Limits	Adjustable
Rate of `GetDevice` requests	The maximum number of `GetDevice` requests you can send per second in this account per Region.	5 per second	Yes
Burst rate of `GetDevice` requests	The maximum number of additional `GetDevice` requests per second (RPS) that you can send in one burst in this account in the current Region.	50	No
Rate of `SearchDevices` requests	The maximum number of `SearchDevices` requests you can send per second in this account per Region.	5 per second	Yes
Burst rate of `SearchDevices` requests	The maximum number of additional `SearchDevices` requests per second (RPS) that you can send in one burst in this account in the current Region.	50	No

Resource	Description	Limits	Adjustable
Rate of `CreateJob` requests	The maximum number of `CreateJob` requests you can send per second in this account per Region.	1 per second	Yes
Burst rate of `CreateJob` requests	The maximum number of additional `CreateJob` requests per second (RPS) that you can send in one burst in this account in the current Region.	5	No
Rate of `SearchJobs` requests	The maximum number of `SearchJob` requests you can send per second in this account per Region.	5 per second	Yes
Burst rate of `SearchJobs` requests	The maximum number of additional `SearchJob` requests per second (RPS) that you can send in one burst in this account in the current Region.	50	No

Resource	Description	Limits	Adjustable
Rate of `GetJob` requests	The maximum number of `GetJob` requests you can send per second in this account per Region.	5 per second	Yes
Burst rate of `GetJob` requests	The maximum number of additional `GetJob` requests per second (RPS) that you can send in one burst in this account in the current Region.	25	No
Rate of `CancelJob` requests	The maximum number of `CancelJob` requests you can send per second in this account per Region.	2 per second	Yes
Burst rate of `CancelJob` requests	The maximum number of additiona l `CancelJob` requests per second (RPS) that you can send in one burst in this account in the current Region.	5	No

Resource	Description	Limits	Adjustable
Number of concurrent **SV1** quantum tasks	The maximum number of concurrent quantum tasks running on the state vector simulator (SV1) in the current Region.	100 us-east-1, 50 us-west-1, 100 us-west-2, 50 eu-west-2	No
Number of concurrent **DM1** quantum tasks	The maximum number of concurrent quantum tasks running on the density matrix simulator (DM1) in the current Region.	100 us-east-1, 50 us-west-1, 100 us-west-2, 50 eu-west-2	No
Number of concurrent **TN1** quantum tasks	The maximum number of concurrent quantum tasks running on the tensor network simulator (TN1) in the current Region.	10 us-east-1, 10 us-west-2, 5 eu-west-2,	Yes
Number of concurrent hybrid jobs	The maximum number of concurrent hybrid jobs in the current Region.	3	Yes
Hybrid jobs runtime limit	The maximum amount of time in days that a hybrid job can run.	5	No

The following are the default classical compute instance quotas for Hybrid Jobs. To raise these quotas, contact Support. Additionally, the available regions are specified for each instance.

Resource	Description	Limits	Adjustable	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
Maximum number of instances of ml.c4.xlarge for hybrid jobs	The maximum number of instances of type ml.c4.xlarge allowed for all Amazon Braket Hybrid Jobs in this account and region.	5	Yes	Yes	Yes	Yes	Yes	No
Maximum number of instances of ml.c4.2xlarge for hybrid jobs	The maximum number of instances of type ml.c4.2xlarge allowed for all Amazon Braket	5	Yes	Yes	Yes	Yes	Yes	No

Resource	Description	Limits	Adjustable	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
	Hybrid Jobs in this account and region.							
Maximum number of instances of ml.c4.4xlarge for hybrid jobs	The maximum number of instances of type ml.c4.4xlarge allowed for all Amazon Braket Hybrid Jobs in this account and region.	5	Yes	Yes	Yes	Yes	Yes	No

Resource	Description	Limits	Adjustable	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
Maximum number of instances of ml.c4.8xlarge for hybrid jobs	The maximum number of instances of type ml.c4.8xlarge allowed for all Amazon Braket Hybrid Jobs in this account and region.	5	Yes	Yes	Yes	Yes	No	No

Resource	Description	Limits	Adjustable	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
Maximum number of instances of ml.c5.xlarge for hybrid jobs	The maximum number of instances of type ml.c5.xlarge allowed for all Amazon Braket Hybrid Jobs in this account and region.	5	Yes	Yes	Yes	Yes	Yes	Yes

Resource	Description	Limits	Adjustable	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
Maximum number of instances of ml.c5.2xlarge for hybrid jobs	The maximum number of instances of type ml.c5.2xlarge allowed for all Amazon Braket Hybrid Jobs in this account and region.	5	Yes	Yes	Yes	Yes	Yes	Yes

Resource	Description	Limits	Adjustable	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
Maximum number of instances of ml.c5.4xlarge for hybrid jobs	The maximum number of instances of type ml.c5.4xlarge allowed for all Amazon Braket Hybrid Jobs in this account and region.	1	Yes	Yes	Yes	Yes	Yes	Yes

Resource	Description	Limits	Adjustable	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
Maximum number of instances of ml.c5.9xlarge for hybrid jobs	The maximum number of instances of type ml.c5.9xlarge allowed for all Amazon Braket Hybrid Jobs in this account and region.	1	Yes	Yes	Yes	Yes	Yes	Yes

Resource	Description	Limits	Adjustable	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
Maximum number of instances of ml.c5.18xlarge for hybrid jobs	The maximum number of instances of type ml.c5.18xlarge allowed for all Amazon Braket Hybrid Jobs in this account and region.	0	Yes	Yes	Yes	Yes	Yes	Yes

Resource	Description	Limits	Adjustable	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
Maximum number of instances of ml.c5n.xlarge for hybrid jobs	The maximum number of instances of type ml.c5n.xlarge allowed for all Amazon Braket Hybrid Jobs in this account and region.	0	Yes	Yes	Yes	Yes	No	No

Resource	Description	Limits	Adjustable	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
Maximum number of instances of ml.c5n.2xlarge for hybrid jobs	The maximum number of instances of type ml.c5n.2xlarge allowed for all Amazon Braket Hybrid Jobs in this account and region.	0	Yes	Yes	Yes	Yes	No	No

Resource	Description	Limits	Adjustable	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
Maximum number of instances of ml.c5n.4xlarge for hybrid jobs	The maximum number of instances of type ml.c5n.4xlarge allowed for all Amazon Braket Hybrid Jobs in this account and region.	0	Yes	Yes	Yes	Yes	No	No

Resource	Description	Limits	Adjustable	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
Maximum number of instances of ml.c5n.9xlarge for hybrid jobs	The maximum number of instances of type ml.c5n.9xlarge allowed for all Amazon Braket Hybrid Jobs in this account and region.	0	Yes	Yes	Yes	Yes	No	No

Resource	Description	Limits	Adjustable	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
Maximum number of instances of ml.c5n.18 xlarge for hybrid jobs	The maximum number of instances of type ml.c5n.18 xlarge allowed for all Amazon Braket Hybrid Jobs in this account and region.	0	Yes	Yes	Yes	Yes	No	No

Resource	Description	Limits	Adjustable	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
Maximum number of instances of ml.g4dn.xlarge for hybrid jobs	The maximum number of instances of type ml.g4dn.xlarge allowed for all Amazon Braket Hybrid Jobs in this account and region.	0	Yes	Yes	Yes	Yes	Yes	Yes

Resource	Description	Limits	Adjustable	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
Maximum number of instances of ml.g4dn.2 xlarge for hybrid jobs	The maximum number of instances of type ml.g4dn.2 xlarge allowed for all Amazon Braket Hybrid Jobs in this account and region.	0	Yes	Yes	Yes	Yes	Yes	Yes

Resource	Description	Limits	Adjustable	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
Maximum number of instances of ml.g4dn.4xlarge for hybrid jobs	The maximum number of instances of type ml.g4dn.4xlarge allowed for all Amazon Braket Hybrid Jobs in this account and region.	0	Yes	Yes	Yes	Yes	Yes	Yes

Resource	Description	Limits	Adjustable	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
Maximum number of instances of ml.g4dn.8xlarge for hybrid jobs	The maximum number of instances of type ml.g4dn.8xlarge allowed for all Amazon Braket Hybrid Jobs in this account and region.	0	Yes	Yes	Yes	Yes	Yes	Yes

Resource	Description	Limits	Adjustable	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
Maximum number of instances of ml.g4dn.12xlarge for hybrid jobs	The maximum number of instances of type ml.g4dn.12xlarge allowed for all Amazon Braket Hybrid Jobs in this account and region.	0	Yes	Yes	Yes	Yes	Yes	Yes

Resource	Description	Limits	Adjustable	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
Maximum number of instances of ml.g4dn.16xlarge for hybrid jobs	The maximum number of instances of type ml.g4dn.16xlarge allowed for all Amazon Braket Hybrid Jobs in this account and region.	0	Yes	Yes	Yes	Yes	Yes	Yes

Resource	Description	Limits	Adjustable	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
Maximum number of instances of ml.m4.xlarge for hybrid jobs	The maximum number of instances of type ml.m4.xlarge allowed for all Amazon Braket Hybrid Jobs in this account and region.	5	Yes	Yes	Yes	Yes	Yes	No

Resource	Description	Limits	Adjustable	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
Maximum number of instances of ml.m4.2xlarge for hybrid jobs	The maximum number of instances of type ml.m4.2xlarge allowed for all Amazon Braket Hybrid Jobs in this account and region.	5	Yes	Yes	Yes	Yes	Yes	No

Resource	Description	Limits	Adjustable	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
Maximum number of instances of ml.m4.4xlarge for hybrid jobs	The maximum number of instances of type ml.m4.4xlarge allowed for all Amazon Braket Hybrid Jobs in this account and region.	2	Yes	Yes	Yes	Yes	Yes	No

Resource	Description	Limits	Adjustable	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
Maximum number of instances of ml.m4.10 large for hybrid jobs	The maximum number of instances of type ml.m4.10 large allowed for all Amazon Braket Hybrid Jobs in this account and region.	0	Yes	Yes	Yes	Yes	Yes	No

Resource	Description	Limits	Adjustable	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
Maximum number of instances of ml.m4.16large for hybrid jobs	The maximum number of instances of type ml.m4.16large allowed for all Amazon Braket Hybrid Jobs in this account and region.	0	Yes	Yes	Yes	Yes	Yes	No

Resource	Description	Limits	Adjustable	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
Maximum number of instances of ml.m5.large for hybrid jobs	The maximum number of instances of type ml.m5.large allowed for all Amazon Braket Hybrid Jobs in this account and region.	5	Yes	Yes	Yes	Yes	Yes	Yes

Resource	Description	Limits	Adjustable	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
Maximum number of instances of ml.m5.xlarge for hybrid jobs	The maximum number of instances of type ml.m5.xlarge allowed for all Amazon Braket Hybrid Jobs in this account and region.	5	Yes	Yes	Yes	Yes	Yes	Yes

Resource	Description	Limits	Adjustable	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
Maximum number of instances of ml.m5.2xlarge for hybrid jobs	The maximum number of instances of type ml.m5.2xlarge allowed for all Amazon Braket Hybrid Jobs in this account and region.	5	Yes	Yes	Yes	Yes	Yes	Yes

Resource	Description	Limits	Adjustable	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
Maximum number of instances of ml.m5.4xlarge for hybrid jobs	The maximum number of instances of type ml.m5.4xlarge allowed for all Amazon Braket Hybrid Jobs in this account and region.	5	Yes	Yes	Yes	Yes	Yes	Yes

Resource	Description	Limits	Adjustable	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
Maximum number of instances of ml.m5.12large for hybrid jobs	The maximum number of instances of type ml.m5.12large allowed for all Amazon Braket Hybrid Jobs in this account and region.	0	Yes	Yes	Yes	Yes	Yes	Yes

Resource	Description	Limits	Adjustable	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
Maximum number of instances of ml.m5.24large for hybrid jobs	The maximum number of instances of type ml.m5.24large allowed for all Amazon Braket Hybrid Jobs in this account and region.	0	Yes	Yes	Yes	Yes	Yes	Yes

Resource	Description	Limits	Adjustable	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
Maximum number of instances of ml.p2.xlarge for hybrid jobs	The maximum number of instances of type ml.p2.xlarge allowed for all Amazon Braket Hybrid Jobs in this account and region.	0	Yes	Yes	No	Yes	No	No

Resource	Description	Limits	Adjustable	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
Maximum number of instances of ml.p2.8xlarge for hybrid jobs	The maximum number of instances of type ml.p2.8xlarge allowed for all Amazon Braket Hybrid Jobs in this account and region.	0	Yes	Yes	No	Yes	No	No

Resource	Description	Limits	Adjustable	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
Maximum number of instances of ml.p2.16xlarge for hybrid jobs	The maximum number of instances of type ml.p2.16xlarge allowed for all Amazon Braket Hybrid Jobs in this account and region.	0	Yes	Yes	No	Yes	No	No

Resource	Description	Limits	Adjustable	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
Maximum number of instances of ml.p3.2xlarge for hybrid jobs	The maximum number of instances of type ml.p3.2xlarge allowed for all Amazon Braket Hybrid Jobs in this account and region.	0	Yes	Yes	No	Yes	No	No

Resource	Description	Limits	Adjustable	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
Maximum number of instances of ml.p4d.24xlarge for hybrid jobs	The maximum number of instances of type ml.p4d.24xlarge allowed for all Amazon Braket Hybrid Jobs in this account and region.	0	Yes	Yes	No	Yes	No	No

Resource	Description	Limits	Adjustable	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
Maximum number of instances of ml.p3dn.24xlarge for hybrid jobs	The maximum number of instances of type ml.p3dn.24xlarge allowed for all Amazon Braket Hybrid Jobs in this account and region.	0	Yes	Yes	No	Yes	No	No

Resource	Description	Limits	Adjustable	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
Maximum number of instances of ml.p3.8xlarge for hybrid jobs	The maximum number of instances of type ml.p3.8xlarge allowed for all Amazon Braket Hybrid Jobs in this account and region.	0	Yes	Yes	No	Yes	Yes	No

Resource	Description	Limits	Adjustable	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
Maximum number of instances of ml.p3.16xlarge for hybrid jobs	The maximum number of instances of type ml.p3.16xlarge allowed for all Amazon Braket Hybrid Jobs in this account and region.	0	Yes	Yes	No	Yes	Yes	No

Requesting limit updates

If you receive a ServiceQuotaExceeded exception for an instance type and do not have sufficient instances available for it, you may request a limit increase from the [Service Quotas](https://console.aws.amazon.com/servicequotas/home) page in the AWS console and search for Amazon Braket under AWS Services.

> **Note**
>
> If your hybrid job is unable to provision requested ML compute capacity, use another region. In addition, if you do not see an instance in the table, it is not available for Hybrid Jobs.

Additional quotas and limits

- The Amazon Braket quantum task action is limited to 3MB in size.
- For SV1, the maximum running duration is 3 hours for circuits up to 31 qubits, and 11 hours for circuits over 31 qubits.
- The maximum number of shots per task allowed for SV1, DM1, and Rigetti devices is 50,000.
- The maximum number of shots per task allowed for TN1 is 1000.
- For all IonQ's devices: When using an on-demand model, there is a 1 Million gateshot limit, and a minimum of 2500 shots for Error mitigation tasks. For a direct reservation, there is no gateshot limit, and a minimum of 500 shots for Error mitigation tasks.
- For QuEra's Aquila device, the maximum is 1,000 shots per task.
- For IQM's Garnet and Emerald devices, the maximum is 20,000 shots per task.
- For TN1 and the QPU devices, shots per task must be > 0.

Document history for the Amazon Braket Developer Guide

The following table describes the documentation releases for Amazon Braket.

- **Latest API Reference update:** August 14, 2025
- **Latest documentation update:** September 2, 2025

Change	Description	Date
New Braket provided CUDA-Q container	Added support for a provided CUDA-Q hybrid jobs container . For more information, see Define the environment for your algorithm script.	September 2, 2025
New local device emulator feature	Added support for a Local quantum device emulator tool to emulate your verbatim programs before submitting them to quantum devices.	August 25, 2025
Moved Pennylane and CUDA-Q pages under Build section	Moved Pennylane and CUDA-Q pages to appear under the Build section in the table of contents.	August 15, 2025
New `ProgramSet` feature	Added support for program sets, an operation to run multiple quantum circuits in a single quantum task.	August 14, 2025
New device IQM Emerald	Added support for the IQM Emerald device. An 54-quibit device with a square (Crystal) lattice topology.	July 21, 2025

Updated the AmazonBraketServiceRolePolicy policy	AmazonBraketServiceRolePolicy now only provides **s3:*** and **logs:*** actions to the **aws:PrincipalAccount**. This restricts access to only the requester's buckets and log groups.	July 11, 2025
New Experimental Capability feature: Dynamic circuits	Mid-circuit measurement and feed-forward operations are available as Experimental Capabilities, see Access to dynamic circuits on IQM devices.	June 26, 2025
Updated the AmazonBraketFullAccess policy	AmazonBraketFullAccess now includes **pricing:GetProducts** to display hardware costs on the console.	April 14, 2025
New device IonQ Forte-Enterprise-1	Added support for the IonQ Forte-Enterprise-1 device. An 36 quibit device that utilizes trapped ion technology.	March 17, 2025
Improved S3 conditions permissions	To improve security, `AmazonBraketFullAccess` now only provides `s3:*` actions to the `aws:PrincipalAccount` . This restricts access to the requester's own buckets only.	March 7, 2025
New device Rigetti Ankaa-3	Added support for the Rigetti Ankaa-3 device. An 84 quibit device that utilizes scalable multi-chip technology.	January 14, 2025

Rigetti Ankaa-2 device retirement	Removed support for the Rigetti Ankaa-2 device.	January 14, 2025
Support for IPv6 traffic	Amazon Braket now supports IPv6 traffic using the dualstack endpoint `braket.{region}.api.aws` .	December 12, 2024
Support for [NVIDIA's CUDA-Q on Amazon Braket]()	Customers can now run quantum programs using NVIDIA's CUDA-Q developer framework on Amazon Braket.	December 6, 2024
IonQ Forte-1 device is readily available	IonQ Forte-1 device is no longer reservation-only and now readily availble to our customers.	November 22, 2024
Rigetti Aspen-M-3 device retirement	Removed support for the Rigetti Aspen-M-3 device.	September 27, 2024
IonQ Harmony device retirement	Removed support for the IonQ Harmony device.	August 29, 2024
New device Rigetti Ankaa-2	Added support for the Rigetti Ankaa-2 device. An 84 quibit device that utilizes scalable multi-chip technology.	August 26, 2024
Developer guide reorganiz ation	The new developer guide takes the existing Build, Test, Run customer journey and guides users along this path with Amazon Braket.	August 23, 2024
OQC Lucy device retirement	Removed support for the OQC Lucy device.	June 28, 2024

New device IQM Garnet and region Europe North 1	Added support for the IQM Garnet device. A 20-qubit device with a square lattice topology. Expanded Braket supported regions to Europe North 1 (Stockholm).	May 22, 2024
Local detuning released	Experimental capabilities now include the local detuning feature of QuEra's Aquila QPU.	April 11, 2024
Notebook inactivity manager released	When creating a notebook instance, enable the inactivit y manager and set an idle duration time to automatic ally reset the Braket notebook instance.	March 27, 2024
Table of contents rework	Reorganized the Amazon Braket table of contents to abide by the AWS style guide requirements and improve the flow of content for customer experience.	December 12, 2023
Braket direct released	Added support for Braket direct features, including: • Working with reservations • Getting Expert advice • Explore Experimental Capabilities	November 27, 2023

Updated Create an Amazon Braket notebook instance	Updated the documentation to add information to create a notebook instance for new and existing Amazon Braket customers.	November 27, 2023
Updated Bring your own container (BYOC)	Updated the documentation to add information about when to BYOC, the recipe to BYOC, and running Braket Hybrid Jobs on the container.	October 18, 2023
Hybrid jobs decorator released	Added Running your local code as a hybrid job page. Contains examples: • Create a hybrid job from local Python code • Install additional Python packages and source code • Save and load data into a hybrid job instance • Best practices for hybrid job decorators	October 16, 2023
Added Queue visibility	Updated the Developer's Guide documentation to include queue depth and queue position. Updated the API documentation to reflect new API changes for queue visibility.	September 25, 2023

Standardize naming in documentation	Updated the documentation to change any instances of "job" to "hybrid job" and "task" to "quantum task"	September 11, 2023
New device IonQ Aria 2	Added support for the IonQ Aria 2 device	September 8, 2023
Updated Native Gates	Updated the documentation to add information about programmatic access to native gates from Rigetti.	August 16, 2023
Xanadu departure	Updated the documentation to remove all Xanadu devices	June 2, 2023
New device IonQ Aria	Added support for the IonQ Aria device	May 16, 2023
Retired Rigetti device	Discontinued support for Rigetti Aspen-M-2	May 2, 2023
Updated **AmazonBraketFullAccess** policy information	Updated the script that defines the contents of the **AmazonBraketFullAccess** policy to include the **servicequotas:GetServiceQuota** and **cloudwatch:GetMetricData** actions as well as information about limitations with respect to quotas.	April 19, 2023
Guided Journeys launch	Changed the documentation to reflect the more up to date and simplified method for Braket onboarding.	April 5, 2023

New device Rigetti Aspen-M-3	Added support for the Rigetti Aspen-M-3 device	January 17, 2023
New adjoint gradient feature	Added information about the adjoint gradient feature offered by SV1	December 7, 2022
New algorithm library feature	Added information about the Braket algorithm library, which provides a catalog of pre-built quantum algorithms	November 28, 2022
D-Wave departure	Updated the documentation to accommodate the removal of all D-Wave devices	November 17, 2022
New device QuEra Aquila	Added support for the QuEra Aquila device	October 31, 2022
Support for Braket Pulse	Added support for Braket Pulse, which allows for pulse control to be used on Rigetti and OQC devices	October 20, 2022
Support for IonQ native gates	Added support for the native gate set offered by the IonQ device	September 13, 2022
New instance quotas	Updated the default classical compute instance quotas associated with Hybrid Jobs	August 22, 2022
New service dashboard	Updated console screensho ts to include the service dashboard	August 17, 2022
New device Rigetti Aspen-M-2	Added support for the Rigetti Aspen-M-2 device	August 12, 2022

New OpenQASM features	Added OpenQASM features support for the local simulators (braket_sv and braket_dm)	August 4, 2022
New cost tracking procedures	Added how to get near-real time maximum cost estimates for simulators and hardware workloads	July 18, 2022
New Xanadu Borealis device	Added support for the Xanadu Borealis device	June 2, 2022
New onboarding simplific ation procedures	Added information on how the new and simplified onboarding procedures work	May 16, 2022
New device D-Wave Advantage_system6.1	Added support for the D-Wave Advantage_system6.1 device	May 12, 2022
Support for embedded simulators	Added how to run embedded simulations with hybrid jobs and how to use the PennyLane lightning simulator	May 4, 2022
AmazonBraketFullAccess - Full access policy for Amazon Braket	Added s3:ListAllMyBuckets permissions to allow users to view and inspect the buckets created and used for Amazon Braket	March 31, 2022
Support for OpenQASM	Added OpenQASM 3.0 support for gate-base d quantum devices and simulators	March 7, 2022

New Quantum Hardware Provider, Oxford Quantum Circuits and new region, eu-west-2	Added support for OQC and eu-west-2	February 28, 2022
New Rigetti device	Added support for Rigetti Aspen M-1	February 15, 2022
New resource limits	Increased the maximum number of concurrent DM1 and SV1 tasks from 55 to 100	January 5, 2022
New Rigetti device	Added support for Rigetti Aspen-11	December 20, 2021
Retired Rigetti device	Discontinued support for Rigetti Aspen-10 device	December 20, 2021
New result type	Reduced density matrix result type supported by local density matrix simulator and DM1 devices	December 20, 2021
Updated policy description	Amazon Braket updated the role ARN to include the `servicerole/` path. For information on policy updates, see the Amazon Braket updates to AWS managed policies table.	November 29, 2021
Amazon Braket Jobs	User guide for Amazon Braket Hybrid Jobs and API added	November 29, 2021
New Rigetti device	Added support for Rigetti Aspen-10	November 20, 2021

Retired D-Wave device	Discontinued support for D-Wave QPU, Advantage _system1	November 4, 2021
New D-Wave device	Added support for an additional D-Wave QPU, Advantage_system4	October 5, 2021
New noise simulators	Added support for a Density matrix simulator (DM1), which can simulate circuits of up to 17 qubits and a local noise simulator *braket_dm*	May 25, 2021
PennyLane support	Added support for PennyLane on Amazon Braket	December 8, 2020
New simulator	Added support for a Tensor Network Simulator (TN1), which allows larger circuits	December 8, 2020
Task batching	Braket supports customer task batching	November 24, 2020
Manual qubit allocation	Braket supports manual qubit allocation on the Rigetti device	November 24, 2020
Adjustable quotas	Braket supports self-service adjustable quotas for your task resources	October 30, 2020
Support for PrivateLink	You can set up private VPC endpoints for your Braket jobs	October 30, 2020
Support for tags	Braket supports API-based tags for the *quantum-task* resource	October 30, 2020

New D-Wave device	Added support for an additional D-Wave QPU, Advantage_system1	September 29, 2020
Initial release	Initial release of the Amazon Braket documentation	August 12, 2020