
Amazon Braket

Developer Guide



Amazon Braket: Developer Guide

Copyright © 2020 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

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
Amazon Braket terms and concepts	2
AWS terminology and tips for Amazon Braket	4
How it works	5
Amazon Braket task flow	5
Third-party data processing	6
Supported devices	7
IonQ	9
Rigetti	9
OQC	10
D-Wave	10
Xanadu	10
Local state vector simulator (braket_sv)	11
Local density matrix simulator (braket_dm)	11
State vector simulator (SV1)	11
Density matrix simulator (DM1)	12
Tensor network simulator (TN1)	13
PennyLane's lightning simulators	14
Compare simulators	14
Regions and endpoints	17
Pricing	19
Near real-time cost tracking	19
Best practices for cost savings	20
Quotas	22
Additional quotas and limits	30
When will my task run?	31
Status change notifications in email or SMS	31
QPU availability windows and status	31
Enable Amazon Braket	32
Prerequisites	32
Steps to enable Amazon Braket	32
Enable third-party devices	34
Get started	35
Create an Amazon Braket notebook instance	35
Run your first circuit using the Amazon Braket Python SDK	37
Run your first annealing problem with Ocean	39
Work with Amazon Braket	43
Construct circuits in the SDK	44
Gates and circuits	44
Manual qubit allocation	47
Verbatim compilation	48
Noise simulation	48
Inspecting the circuit	49
Result types	51
Run your circuits with OpenQASM 3.0	53
What is OpenQASM 3.0?	54
When to use OpenQASM 3.0	54
How OpenQASM 3.0 works	54
Prerequisites	54
What OpenQASM features does Braket support?	54
Create and submit an example OpenQASM 3.0 task	57
Support for OpenQASM on different Braket Devices	59
Simulate noise with OpenQASM3	61
Qubit rewiring with OpenQASM3	62

Verbatim Compilation with OpenQASM3	62
The Braket console	63
More resources	63
Submitting tasks to QPUs and simulators	63
Example tasks on Amazon Braket	64
Submitting tasks to a QPU	67
Running a task with the local simulator	68
Task batching	69
Set up SNS notifications (optional)	70
Monitoring and tracking tasks	70
Tracking tasks from the Amazon Braket SDK	70
Advanced logging	72
Monitoring tasks through the Amazon Braket console	74
Working with Boto3	75
Turn on the Amazon Braket Boto3 client	75
Configure AWS CLI profiles for Boto3 and the Amazon Braket SDK	78
Amazon Braket Hybrid Jobs	80
When to use Amazon Braket Hybrid Jobs	80
Run a job with Amazon Braket Hybrid Jobs	81
Create your first job	82
Inputs, outputs, environmental variables, and helper functions	87
Inputs	88
Outputs	88
Environmental variables	89
Helper functions	89
Save job results	89
Save and restart jobs using checkpoints	91
Define the environment for your algorithm script	92
Use hyperparameters	93
Configure the job instance to run your algorithm script	94
Cancel a job	96
Use Amazon Braket Hybrid Jobs to run a QAOA algorithm	97
Accelerate your hybrid workloads with embedded simulators from PennyLane	100
Using lightning.gpu for Quantum Approximate Optimization Algorithm workloads	100
Quantum machine learning and data parallelism	102
Build and debug a job with local mode	105
Bring your own container (BYOC)	105
Configure the default bucket in AwsSession	107
Interact with jobs directly using the API	107
Use PennyLane with Amazon Braket	111
Amazon Braket with PennyLane	111
Hybrid algorithms in Amazon Braket example notebooks	112
Hybrid algorithms with embedded PennyLane simulators	113
Security	114
Shared responsibility for security	114
Data protection	114
Data retention	115
Managing access to Amazon Braket	115
Amazon Braket resources	115
Notebooks and roles	115
About the AmazonBraketFullAccess policy	116
About the AmazonBraketJobsExecutionPolicy policy	119
Restrict user access to certain devices	121
Amazon Braket updates to AWS managed policies	122
Service-linked role	123
Service-linked role permissions for Amazon Braket	123
Resilience	124

Compliance validation	124
Infrastructure Security	125
Third Party Security	125
Troubleshoot	126
Access denied exception	126
A task is failing creation	126
An SDK feature does not work	126
A job fails due to an exceeded quota	127
Something stopped working in your notebook instance	127
Troubleshoot OpenQASM	128
Include statement error	128
Non-contiguous qubits error	128
Mixing physical qubits with virtual qubits error	129
Requesting result types and measuring qubits in the same program error	129
Classical and qubit register limits exceeded error	129
Box not preceded by a verbatim pragma error	129
Verbatim boxes missing native gates error	130
Verbatim boxes missing physical qubits error	130
The verbatim pragma is missing "braket" error	130
Single qubits cannot be indexed error	130
The physical qubits in a two qubit gate are not connected error	131
GetDevice does not return OpenQASM results error	131
Local simulator support warning	132
VPC endpoints (PrivateLink)	133
Considerations for Amazon Braket VPC endpoints	133
Set up Braket and PrivateLink	133
Step 1: Launch an Amazon VPC if needed	133
Step 2: Create an interface VPC endpoint for Braket	134
Step 3: Connect and run Braket tasks through your endpoint	134
More about creating an endpoint	134
Control access with Amazon VPC endpoint policies	135
Tagging resources	136
Using tags	136
More about AWS and tags	136
Supported resources in Amazon Braket	136
Tag restrictions	137
Managing tags in Amazon Braket	137
Add tags	137
View tags	137
Edit tags	138
Remove tags	138
Example of CLI tagging in Amazon Braket	138
Tagging with the Amazon Braket API	138
Monitor with CloudWatch	140
Amazon Braket Metrics and Dimensions	140
Supported Devices	140
Amazon Braket Events with EventBridge	141
Monitor task status with EventBridge	141
Example Amazon Braket EventBridge event	142
Logging with CloudTrail	144
Amazon Braket Information in CloudTrail	144
Understanding Amazon Braket Log File Entries	145
API and SDK Reference	147
Document history	148

What Is Amazon Braket?

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. Amazon Braket helps you overcome these challenges.

Amazon Braket offers a single point of access to a variety of quantum computing technologies. It enables you to:

- 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, Amazon Braket offers different environments to simulate and run your quantum algorithms. You can find the approach that best suits your requirements and you can get started quickly with a set of example environments called *notebooks*.

Amazon Braket development has three aspects — Build, Test, and Run:

Build - Amazon Braket provides fully-managed Jupyter notebook environments that make it easy to get started. Amazon 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 - Amazon Braket provides access to fully-managed, high-performance, quantum circuit simulators. You can test and validate your circuits. Amazon Braket handles all the underlying software components and EC2 clusters to take away the burden of simulating quantum circuits on classical HPC infrastructure.

Run - Amazon Braket provides secure, on-demand access to different types of quantum computers. You have access to gate-based quantum computers from IonQ, OQC, Xanadu, and Rigetti as well as a quantum annealer from D-Wave. You have no upfront commitment, and no need to procure access with individual providers.

About quantum computing and Amazon 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 certain use cases and it is crucial to have access to a variety of computing hardware. Amazon 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. Amazon 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. Amazon 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. Amazon Braket offers capabilities and notebook examples that can already be applied to many proof of concept problems, and certain practical problems.

Amazon Braket terms and concepts

The following terms and concepts are used in Amazon Braket:

Braket

We named the Amazon 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.

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* or *quantum annealing*.

Qubit

The basic unit of information in a quantum computer is called a qubit (quantum bit), in analogy to classical bits. 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.

Gate-based Quantum Computing

In gate-based QC (also called circuit-based QC), computations are broken down into elementary operations (gates). It can be shown that 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* in analogy to the logic gates of classical digital circuits.

Quantum Annealing

Quantum annealing is a form of special purpose quantum computing that tries to utilize quantum fluctuations to find global minima of an objective function. In most approaches, the objective function that is encoded directly in the physical couplings parameters of the qubits. Quantum annealing is mainly used for combinatorial optimization problems (e.g., [QUBO problems](#)), where one has a finite and discrete search space.

Device

In Amazon Braket, a device is a backend that can execute *quantum tasks*. A device can be a *QPU* or a *quantum circuit simulator*. To learn more, see [Amazon Braket supported devices \(p. 7\)](#).

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 grows exponentially with the number of qubits to simulate. Amazon Braket provides access to both managed (accessed through the Braket API) and local (part of the Amazon Braket SDK) quantum circuit simulators.

Quantum Processing Unit (QPU)

A QPU is a physical quantum computing device that can execute a quantum task. QPUs can be based on different QC paradigms, e.g. gate-based QC or quantum annealing. To learn more, see [Amazon Braket supported devices \(p. 7\)](#).

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.

Shots

Since quantum computing is inherently probabilistic, any circuit (or annealing schedule) needs to be evaluated multiple times to get an accurate results. A single circuit execution and measurement is called a *shot*. The number of *shots* (repeated executions) for circuit is chosen based on the desired accuracy for the result. The number of shots can range from 10 to 100,000 shots per task.

Quantum Task

In Amazon 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 Amazon Braket SDK or by using the CreateQuantumTask API operation directly. After you create a task, it will be queued until the requested device becomes available. You can view your quantum tasks on the **Tasks** page of the Amazon Braket console or by using the GetQuantumTask or SearchQuantumTasks API operations.

Braket job

Amazon Braket has a feature called Amazon Braket Hybrid Jobs (or Braket Jobs for short) that provides fully managed executions of hybrid algorithms. A Braket 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 *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 execute the *quantum tasks* that are part of your algorithm. A single job typically contains a collection of many tasks.

QPU supported gates

QPU supported gates are the gates accepted by the QPU device. These gates might not be able to directly run 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 Braket SDK.

QPU native gates

QPU native gates are the gates that 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.

AWS terminology and tips for Amazon Braket

IAM users

An IAM user is an identity that you create in AWS. It represents the person or application that interacts with AWS services and resources. It consists of a name and credentials. By default, when you create a new IAM user in AWS, it has no permissions associated with it. To allow the IAM user to perform specific actions in AWS, such as launching an Amazon EC2 instance or creating an Amazon S3 bucket, you must grant the IAM user the necessary permissions.

- **Best practice:** We recommend that you create an individual IAM user for each person who needs access to AWS. Even if you have multiple employees who require the same level of access, create individual IAM users for each of them. This approach provides additional security by allowing each IAM user to have a unique set of security credentials.

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 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 an IAM 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 task results.

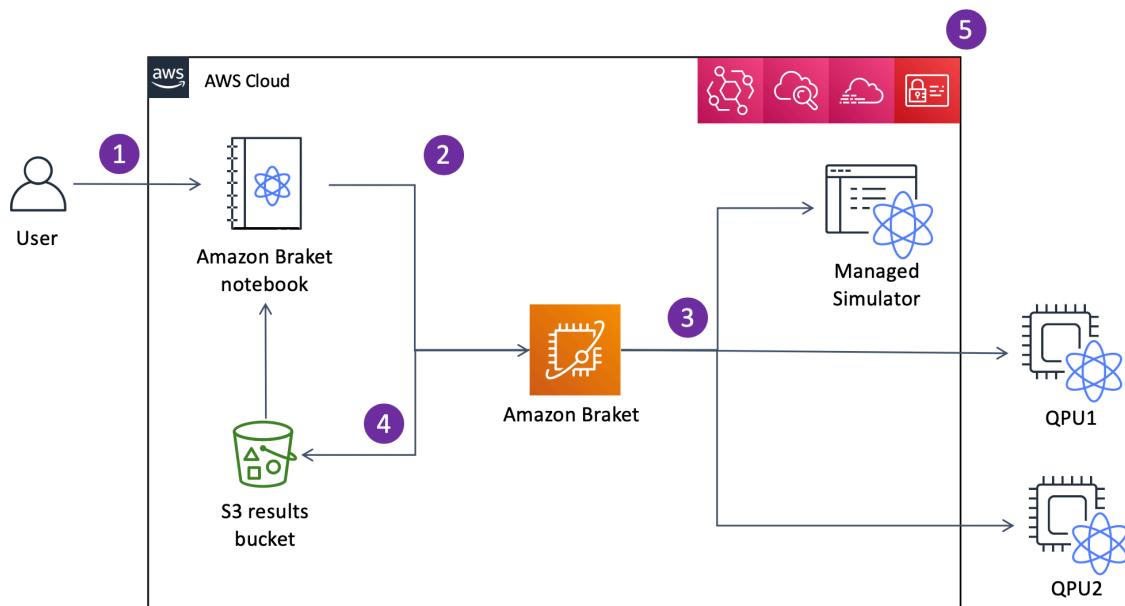
- **Best practice:** You can set permissions to control access to your S3 bucket. For more information, see [Bucket policies and user policies](#) in the Amazon S3 documentation.

How Amazon Braket works

Amazon Braket provides on-demand access to quantum computing devices, including managed circuit simulators and different types of QPUs. In Amazon Braket, the atomic request to a *device* is a task. For *gate-based QC* devices, this includes the quantum circuit (including the measurement instructions and number of shots) and other request metadata. For annealing devices it includes the problem definition, the number of shots, and other optional parameters.

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

Amazon Braket task flow



To make it easy for customers to define, submit, and monitor their tasks, Amazon Braket provides managed Jupyter notebooks (1), which can be accessed and administered through the [Amazon Braket Console](#), that come pre-installed with the [Amazon Braket SDK](#). You can build your quantum circuits directly in the SDK or, for annealing devices, define the annealing problem and parameter. The Amazon Braket SDK also provides a plugin for D-Wave's Ocean tool suite so you can natively program the D-Wave device. After your task is defined, you can choose a device to execute it on and submit it to the Amazon Braket API (2). Depending on the device you chose, the task is queued until the device becomes available and the task is sent to the QPU or simulator for execution (3). Amazon Braket gives you access to five different types of QPUs (D-Wave, IonQ, OQC, Xanadu, Rigetti) and three managed simulators (SV1, DM1, TN1). To learn more, see [Amazon Braket supported devices \(p. 7\)](#).

After your task is processed, 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 task completion. You can also view and manage your tasks on the **Tasks** page in the Amazon Braket console or by using the `GetQuantumTask` operation of the Amazon Braket API.

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

Third-party data processing

Tasks that are submitted to a QPU device, process 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 \(p. 125\)](#).

Amazon Braket supported devices

In Amazon Braket, a device represents a QPU or simulator that you can call to run quantum tasks. A *quantum task* refers to either a set of circuits for gate-based quantum computing or an annealing problem for a quantum annealer device.

Amazon Braket provides access to five QPU devices — from D-Wave, IonQ, Oxford Quantum Circuits, Rigetti, and Xanadu — and six simulator devices. For all devices, you can find further device properties, such as device topology, calibration data, and native gate sets in the Amazon Braket console in the **Devices** tab or by means of the GetDevice API. When constructing a circuit with the simulators, Amazon Braket currently requires that you use contiguous qubits/indices. If you are working with the Amazon Braket SDK, you have access to device properties as shown in the following code example:

```
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 = 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:::device/qpu/d-wave/DW_2000Q_6') #D-
#Wave 2000Q
# device = AwsDevice('arn:aws:braket:::device/qpu/d-wave/Advantage_system4') #D-
#Wave Advantage_system4.1
# device = AwsDevice('arn:aws:braket:us-west-2::device/qpu/d-wave/Advantage_system6') #D-
#Wave Advantage_system6.1
# device = AwsDevice('arn:aws:braket:::device/qpu/ionq/ionQdevice')
#IonQ
# device = AwsDevice('arn:aws:braket:::device/qpu/rigetti/Aspen-11')
#Rigetti Aspen-11
# device = AwsDevice('arn:aws:braket:us-west-1::device/qpu/rigetti/Aspen-M-2')
#Rigetti Aspen M-2
# device = AwsDevice('arn:aws:braket:eu-west-2::device/qpu/oqc/Lucy')
#OQC Lucy
# device = AwsDevice('arn:aws:braket:us-east-1::device/qpu/xanadu/Borealis')
#Xanadu Borealis

# get device properties
device.properties
```

Supported QPUs

- [D-Wave 2000Q \(p. 10\)](#)
- [D-Wave Advantage_system4.1 \(p. 10\)](#)
- [D-Wave Advantage_system6.1 \(p. 10\)](#)

- IonQ (p. 9)
- OQC Lucy (p. 10)
- Rigetti Aspen-11 (p. 9)
- Rigetti Aspen-M-2 (p. 9)
- Xanadu Borealis (p. 10)

Supported simulators

- Local state vector simulator (`braket_sv`) ('Default Simulator') (p. 11)
- Local density matrix simulator (`braket_dm`) (p. 11)
- State vector simulator (SV1) (p. 11)
- Density matrix simulator (DM1) (p. 12)
- Tensor network simulator (TN1) (p. 13)
- PennyLane's Lightning Simulators (p. 14)

Choose the best simulator for your task

- Compare simulators (p. 14)

Note

To view the available AWS Regions for each device, scroll right across the following table.

Amazon Braket devices

Provider	Device Name	Paradigm	Type	Device ARN	Region
D-Wave	DW_2000Q_6	quantum annealer	QPU	arn:aws:braket:::device/qpu/d-wave/DW_2000Q_6	us-west-2
D-Wave	Advantage_system4	quantum annealer	QPU	arn:aws:braket:::device/qpu/d-wave/Advantage_system4	us-west-2
D-Wave	Advantage_system6	quantum annealer	QPU	arn:aws:braket:::device/qpu/d-wave/Advantage_system6	us-west-2
IonQ	ionQdevice	gate-based	QPU	arn:aws:braket:::device/qpu/ionq/ionQdevice	us-east-1
Oxford Quantum Circuits	Lucy	gate-based	QPU	arn:aws:braket:eu-west-2::device/qpu/oqc/Lucy	eu-west-2
Rigetti	Aspen-11	gate-based	QPU	arn:aws:braket:::device/qpu/rigetti/Aspen-11	us-west-1
Rigetti	Aspen M-2	gate-based	QPU	arn:aws:braket:us-west-1::device/qpu/rigetti/Aspen-M-2	us-west-1
Xanadu	Borealis	continuous-variable	QPU	arn:aws:braket:us-east-1::device/qpu/xanadu/Borealis	us-east-1

Provider	Device Name	Paradigm	Type	Device ARN	Region
AWS	braket_sv	gate-based	Simulator	N/A (local simulator in Braket SDK)	N/A
AWS	braket_dm	gate-based	Simulator	N/A (local simulator in Braket SDK)	N/A
AWS	SV1	gate-based	Simulator	arn:aws:braket:::device/quantum-simulator/amazon/sv1	All Regions where Amazon Braket is available.
AWS	DM1	gate-based	Simulator	arn:aws:braket:::device/quantum-simulator/amazon/dm1	All Regions where Amazon Braket is available.
AWS	TN1	gate-based	Simulator	arn:aws:braket:::device/quantum-simulator/amazon/tn1	us-west-2, us-east-1, and eu-west-2

To view additional details about the QPUs you can use with Amazon Braket, see [Amazon Braket Hardware Providers](#).

IonQ

IonQ offers a gate-based QPU based on ion trap technology. IonQ's trapped ion QPUs are built on a chain of trapped $^{171}\text{Yb}^+$ ions that are spatially confined by means of a microfabricated surface electrode trap within a vacuum chamber.

The IonQ device supports the following quantum gates:

```
'x', 'y', 'z', 'rx', 'ry', 'rz', 'h', 'cnot', 's', 'si', 't', 'ti', 'v', 'vi', 'xx', 'yy',
'zz', 'swap', 'i'
```

With verbatim compilation, the IonQ device supports the following native gates:

```
'gpi', 'gpi2', 'ms'
```

These native gates can only be used with verbatim compilation. To learn more about verbatim compilation, see [Verbatim Compilation](#).

Rigetti

Rigetti quantum processors are universal, gate-model machines based on all-tunable superconducting qubits. The Rigetti Aspen-11 system is based on scalable 40-qubit node technology. The Rigetti Aspen-

M-2 system leverages their proprietary multi-chip technology and is assembled from two 40-qubit processors.

The Rigetti devices support 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, the Rigetti devices support the following native gates:

```
'rx', 'rz', 'cz', 'cphaseshift', 'xy'
```

Rigetti superconducting quantum processors can run the 'rx' gate with only the angles of $\pm\pi/2$ or $\pm\pi$.

OQC

OQC quantum processors are universal, gate-model machines, built using scalable Coaxmon technology. The OQC Lucy system is an 8-qubit device with the topology of a ring in which each qubit is connected to its two nearest neighbors.

The Lucy device supports the following quantum gates:

```
'ccnot', 'cnot', 'cphaseshift', 'cswap', 'cy', 'cz', 'h', 'i', 'phaseshift', 'rx', 'ry',
'rz', 's', 'si', 'swap', 't', 'ti', 'v', 'vi', 'x', 'y', 'z', 'ecr'
```

With verbatim compilation, the OQC device supports the following native gates:

```
'i', 'rz', 'v', 'x', 'ecr'
```

D-Wave

D-Wave offers quantum annealers based on superconducting qubits. Quantum annealing processors naturally return low-energy solutions. This type of QPU is a specific-purpose machine that's designed to solve quadratic unconstrained optimization (QUBO) problems. This includes optimization and probabilistic sampling problems.

Quantum annealers do not have to meet the strict engineering requirements that universal gate-based machines have to meet. This technology already features approximately 5,000 superconducting qubits compared to fewer than 100 qubits on gate-model quantum computers. Amazon Braket offers access to the superconducting quantum annealers provided by D-Wave Systems. These quantum annealers can be programmed using the high-level, open-source tool suite called Ocean.

For more information, see the [quantum annealing example notebooks](#).

Xanadu

Xanadu builds photonic quantum computers that use continuous variables for quantum computing known as qumodes instead of traditional two-level systems or qubits. In the case of the Borealis QPU, each qumode represents the quantized electromagnetic field of a laser pulse traveling through the device. Gates between two qumodes are implemented by interfering two temporally separated qumodes via programmable beam splitters and delay lines. Instead of the usual one- and two-qubit gates, such as

Hadamard or CNOT, continuous variable quantum computing uses gates such as rotation, displacement, beam-splitting, and squeezing.

Xanadu's photonic quantum computer Borealis is not a universal machine, capable of arbitrary quantum computation, but instead implements a specific protocol known as Gaussian boson sampling (GBS). GBS is a model of photonic quantum computation first introduced by [Hamilton et al.](#) that consists of multi-mode linear optical operations followed by photon-counting measurements. The Borealis device implements GBS with 216 temporally-spaced qumodes. Amazon Braket offers access to Borealis via the open source [Strawberry Fields](#) library for photonic quantum computation.

For more information, see the [Borealis example notebook](#)

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 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 on 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](#).

State vector simulator (SV1)

SV1 is a fully managed, 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 tasks, and a maximum of 100 (50 in us-west-1 and eu-west-2) concurrent 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

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 a fully managed, 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 tasks, and a maximum of 50 concurrent 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 a fully managed, 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 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 for the TN1 simulator.

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 the TN1 simulator

- Avoid all-to-all circuits.
- Test a new circuit or class of circuits with a small number of shots, to learn the circuit's "hardness" for TN1.
- Split large shot simulations over multiple tasks.

PennyLane's lightning simulators

In addition to the Braket simulators, we also support PennyLane's lightning simulators. With PennyLane's lightning simulators, you can leverage advanced gradient computation methods, such as [adjoint differentiation](#), to evaluate gradients faster. The [lightning.qubit simulator](#) is available as a device via 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 Jobs](#) notebook for an example of using [lightning.gpu](#). For more information about embedded simulators, see the [Run a job with Amazon Braket Hybrid Jobs](#).

Compare simulators

This section helps you select the Amazon Braket simulator that's best suited for your task, by describing some concepts, limitations, and use cases.

Choosing between local simulators and managed 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. *Managed simulators* run in the AWS cloud and are designed to scale beyond typical local environments. Managed simulators are optimized for larger circuits, but add some latency overhead per task or batch of tasks. This can imply a trade-off if many 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 a managed 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 the DM1 simulator.

What is a state vector simulator?

The Amazon Braket 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 task increases linearly with the number of gates in the circuit.

What is a density matrix simulator?

The Amazon Braket 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?

The Amazon Braket 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.

The TN1 simulator 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 in order 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.

The SV1 simulator 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 the SV1 simulator, the problem becomes twice as hard.

For many classes of problems, the TN1 simulator can evaluate much larger circuits in realistic time than the SV1 simulator 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 easily.

Limitations of the TN1 simulator

The TN1 simulator can be slower than the SV1 simulator 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 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 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's 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 notebooks

Amazon Braket provides a variety of [example notebooks](#) showing the types of circuits that can either work well for, or challenge, the TN1 and SV1 simulators, such as the quantum Fourier transformation (QFT).

Amazon Braket Regions and endpoints

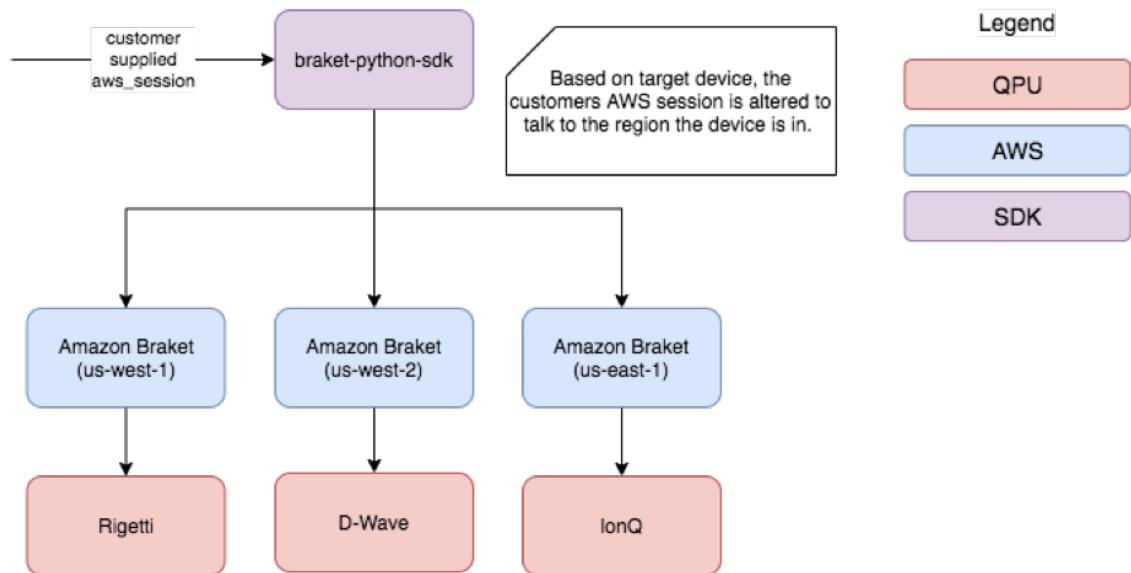
Amazon Braket is available in the following AWS Regions:

Region availability of Amazon Braket

Region Name	Region	Braket Endpoint	QPU
US East (N. Virginia)	us-east-1	braket.us-east-1.amazonaws.com	IonQ
US East (N. Virginia)	us-east-1	braket.us-east-1.amazonaws.com	Xanadu
US West (N. California)	us-west-1	braket.us-west-1.amazonaws.com	Rigetti
US West (Oregon)	us-west-2	braket.us-west-2.amazonaws.com	D-Wave
EU West 2 (London)	eu-west-2	braket.eu-west-2.amazonaws.com	OQC

You can run Amazon Braket from any Region in which it is available, but each QPU is available only in a single Region. Tasks that run on a QPU device can be viewed in the Amazon Braket console in the Region of that device. If you are using the Amazon Braket SDK, you can submit 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, as shown in the following image.

The Amazon Braket SDK automatically routes device requests to the correct region



For general information about how AWS works with Regions and endpoints, see [AWS service endpoints](#) in the *AWS General Reference*.

Amazon Braket pricing

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, please visit our [pricing page](#).

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 managed 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 the SV1 Simulator

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
from braket.annealing import Problem, ProblemType

#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)
```

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, tasks completed, the execution duration, the billed execution duration, and your maximum cost in USD. Your execution time may vary for each simulation.

```
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()
$0.00375
```

Cost tracker on a D-Wave system

You may also run the cost tracker on one of Braket's QPU devices. The following example will show how to run the cost tracking function for a Quadratic Unconstrained Binary Optimization (QUBO) problem on the D-Wave Advantage 6 system.

In your notebook instance, copy over the following example showing the added `Tracker()` function.

```
problem = Problem(ProblemType("QUBO"), {35: 1}, {(35, 36): -1})
device = AwsDevice("arn:aws:braket:us-west-2::device/qpu/d-wave/Advantage_system6")
with Tracker() as tracker:
    result = device.run(problem, shots=1000).result()
```

Our results will show us the number of shots sent and the number of completed tasks for our QUBO problem.

```
tracker.quantum_tasks_statistics()
{'arn:aws:braket:us-west-2::device/qpu/d-wave/Advantage_system6':
 {'shots': 1000, 'tasks': {'COMPLETED': 1}}}

tracker.qpu_tasks_cost()
$0.49
```

Using the cost tracker to set maximum costs

The cost tracker can also be utilized to set maximum costs on a program. You may have a maximum threshold for how much you want to spend on a given program. The cost tracker can be used to build out cost control logic in your execution code. The following example takes a QUBO problem on our D-Wave system and limits the maximum cost to 1 USD. The QUBO problem in our code costs \$0.49 to run one iteration. We have set the logic to repeat the iterations until the next one exceeds 1 USD; hence, the code snippet will run 3 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.

```
problem = Problem(ProblemType("QUBO"), {35: 1}, {(35, 36): -1})
device = AwsDevice("arn:aws:braket:us-west-2::device/qpu/d-wave/Advantage_system6")
with Tracker() as tracker:
    while tracker.qpu_tasks_cost() < 1:
        result = device.run(problem, shots=1000).result()
```

```
tracker.quantum_tasks_statistics()
{'arn:aws:braket:us-west-2::device/qpu/d-wave/Advantage_system6':
 {'shots': 3000, 'tasks': {'COMPLETED': 3}}}

tracker.qpu_tasks_cost()
$1.47
```

Note

The cost tracker will not track duration for failed TN1 simulator 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 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 simulator tasks with low shot counts

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

Check all Regions for tasks

- The console displays tasks only for your current AWS Region. When looking for billable 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 \(p. 7\)](#) documentation page.

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) tasks allows a maximum of 60 per AWS Region.

Resource	Description	Limit	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 additional 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 CreateQuantumTask requests you can send per second in this account per Region.	20	Yes
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	Yes
Burst rate of SearchQuantumTasks requests	The maximum number of additional SearchQuantumTasks	50	No

Resource	Description	Limit	Adjustable
	requests per second (RPS) that you can send in one burst in this account in the current Region.		
Rate of GetQuantumTask requests	The maximum number of GetQuantumTask requests you can send per second in this account per Region.	100	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	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
Rate of GetDevice requests	The maximum number of GetDevice requests you can send per second in this account per Region.	5	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	Yes

Resource	Description	Limit	Adjustable
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
Rate of CreateJob requests	The maximum number of CreateJob requests you can send per second in this account per Region.	1	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 SearchJob requests	The maximum number of SearchJob requests you can send per second in this account per Region.	5	Yes
Burst rate of SearchJob 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
Rate of GetJob requests	The maximum number of GetJob requests you can send per second in this account per Region.	5	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	Yes

Resource	Description	Limit	Adjustable
Burst rate of CancelJob requests	The maximum number of additional CancelJob requests per second (RPS) that you can send in one burst in this account in the current Region.	5	No
Number of concurrent SV1 tasks	The maximum number of concurrent tasks running on the state vector simulator (SV1) in the current Region.	100 (50 in us-west-1 and eu-west-2)	No
Number of concurrent DM1 tasks	The maximum number of concurrent tasks running on the density matrix simulator (DM1) in the current Region.	100 (50 in us-west-1 and eu-west-2)	No
Number of concurrent TN1 tasks	The maximum number of concurrent tasks running on the tensor network simulator (TN1) in the current Region.	10 (5 in eu-west-2)	Yes
Number of concurrent jobs	The maximum number of concurrent jobs in the current Region.	5	Yes

The following are the default classical compute instance quotas for Hybrid Jobs. If you wish to raise these quotas, please contact AWS Support.

Resource	Description	Limits	Adjustable
Maximum number of instances of ml.c4.xlarge for 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
Maximum number of instances of ml.c4.2xlarge for jobs	The maximum number of instances of type ml.c4.2xlarge allowed for all Amazon Braket Hybrid Jobs in this account and region.	5	Yes
Maximum number of instances of ml.c4.4xlarge for jobs	The maximum number of instances of type ml.c4.4xlarge allowed for all Amazon Braket	5	Yes

Resource	Description	Limits	Adjustable
	Hybrid Jobs in this account and region.		
Maximum number of instances of ml.c4.8xlarge for 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
Maximum number of instances of ml.c5.xlarge for 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
Maximum number of instances of ml.c5.2xlarge for 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
Maximum number of instances of ml.c5.4xlarge for 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
Maximum number of instances of ml.c5.9xlarge for 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
Maximum number of instances of ml.c5.18xlarge for 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
Maximum number of instances of ml.c5n.xlarge for 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

Resource	Description	Limits	Adjustable
Maximum number of instances of ml.c5n.2xlarge for 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
Maximum number of instances of ml.c5n.4xlarge for 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
Maximum number of instances of ml.c5n.9xlarge for 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
Maximum number of instances of ml.c5n.18xlarge for jobs	The maximum number of instances of type ml.c5n.18xlarge allowed for all Amazon Braket Hybrid Jobs in this account and region.	0	Yes
Maximum number of instances of ml.g4dn.xlarge for 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
Maximum number of instances of ml.g4dn.2xlarge for jobs	The maximum number of instances of type ml.g4dn.2xlarge allowed for all Amazon Braket Hybrid Jobs in this account and region.	0	Yes
Maximum number of instances of ml.g4dn.4xlarge for 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
Maximum number of instances of ml.g4dn.8xlarge for 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

Resource	Description	Limits	Adjustable
Maximum number of instances of ml.g4dn.12xlarge for 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
Maximum number of instances of ml.g4dn.16xlarge for 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
Maximum number of instances of ml.m4.xlarge for 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
Maximum number of instances of ml.m4.2xlarge for 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
Maximum number of instances of ml.m4.4xlarge for 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
Maximum number of instances of ml.m4.10xlarge for jobs	The maximum number of instances of type ml.m4.10xlarge allowed for all Amazon Braket Hybrid Jobs in this account and region.	0	Yes
Maximum number of instances of ml.m4.16xlarge for jobs	The maximum number of instances of type ml.m4.16xlarge allowed for all Amazon Braket Hybrid Jobs in this account and region.	0	Yes
Maximum number of instances of ml.m5.large for 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

Resource	Description	Limits	Adjustable
Maximum number of instances of ml.m5.xlarge for 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
Maximum number of instances of ml.m5.2xlarge for 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
Maximum number of instances of ml.m5.4xlarge for 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
Maximum number of instances of ml.m5.12xlarge for jobs	The maximum number of instances of type ml.m5.12xlarge allowed for all Amazon Braket Hybrid Jobs in this account and region.	0	Yes
Maximum number of instances of ml.m5.24xlarge for jobs	The maximum number of instances of type ml.m5.24xlarge allowed for all Amazon Braket Hybrid Jobs in this account and region.	0	Yes
Maximum number of instances of ml.p2.xlarge for 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
Maximum number of instances of ml.p2.8xlarge for 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
Maximum number of instances of ml.p2.16xlarge for 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

Resource	Description	Limits	Adjustable
Maximum number of instances of ml.p3.2xlarge for 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
Maximum number of instances of ml.p3.8xlarge for 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
Maximum number of instances of ml.p3.16xlarge for 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
Maximum allowed compute instances for a job	The maximum allowed number of compute instances for a job.	5	Yes

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](#) page in the AWS console.

Note

p3 instances are not available in us-west-1. If your 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 (for example, a circuit or annealing problem) is limited to 3MB in size.
- The maximum number of shots allowed for the SV1 managed simulator, DM1 managed simulator and Rigetti device is 100,000.
- The maximum number of shots allowed for the TN1 managed simulator is 1000.
- For D-Wave, IonQ, and OQC devices, the maximum is 10,000 shots.
- For TN1 and the QPU devices, shots must be > 0.

When will my task run?

When you submit a circuit, Amazon Braket sends it to the device you specify. QPU and simulator tasks are queued and processed in the order they are received. The time required to process your task after you submit it varies depending on the number and complexity of tasks submitted by other Amazon Braket customers and on the availability of the QPU you selected.

Status change notifications in email or SMS

Amazon Braket sends events to Amazon EventBridge when the availability of a QPU changes or when your task's state changes. Follow these steps to receive device and 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 \(p. 141\)](#).

Task completion alerts

You can set up notifications through the Amazon Simple Notification Service (SNS) so that you receive an alert when your Amazon Braket task is complete. Active notifications are useful if you expect a long wait time—for example, when you submit a large task or when you submit a task outside of a device's availability window. If you do not want to wait for the 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 example notebook for setting up notifications](#).

QPU availability windows and status

QPU availability varies among the Quantum Hardware Providers (QHPs).

In the **Devices** page of the Amazon Braket console, you can see the current and upcoming availability windows for each device. You also can view the status of each device.

A device is considered *offline* if it is not available to customers, regardless of availability window. For example, it could be offline due to scheduled maintenance, upgrades, or operational issues.

The **Announcements** page in the Amazon Braket console shows scheduled downtime in advance to all Amazon Braket customers.

Enable Amazon Braket

You can enable Amazon Braket in your account through the [AWS console](#).

Prerequisites

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

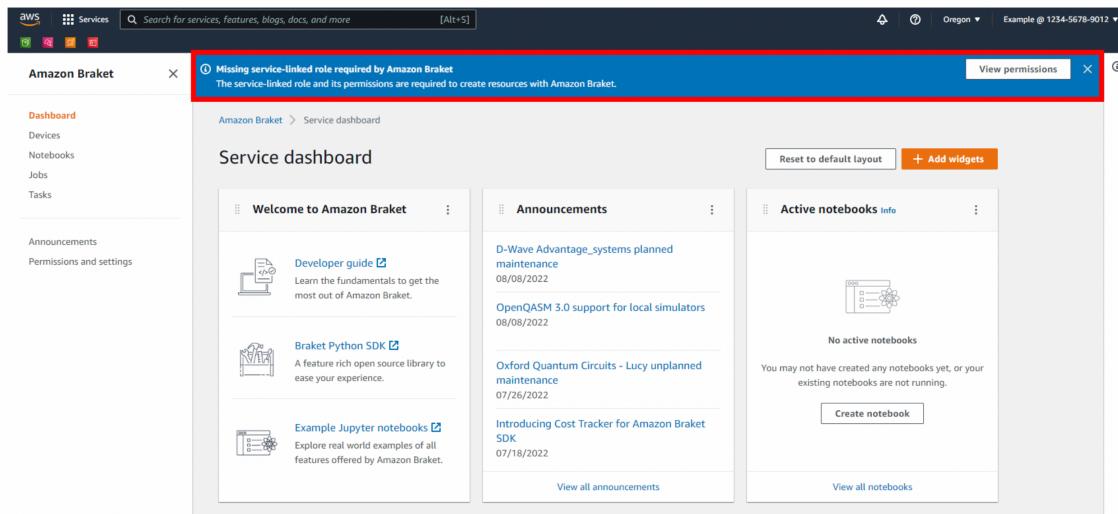
Note

If you are an administrator:

To enable Amazon Braket for other [IAM users](#), for [IAM roles](#), or for an [IAM group](#) in an account, you must grant permissions to each user, role, or group. You can grant these 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

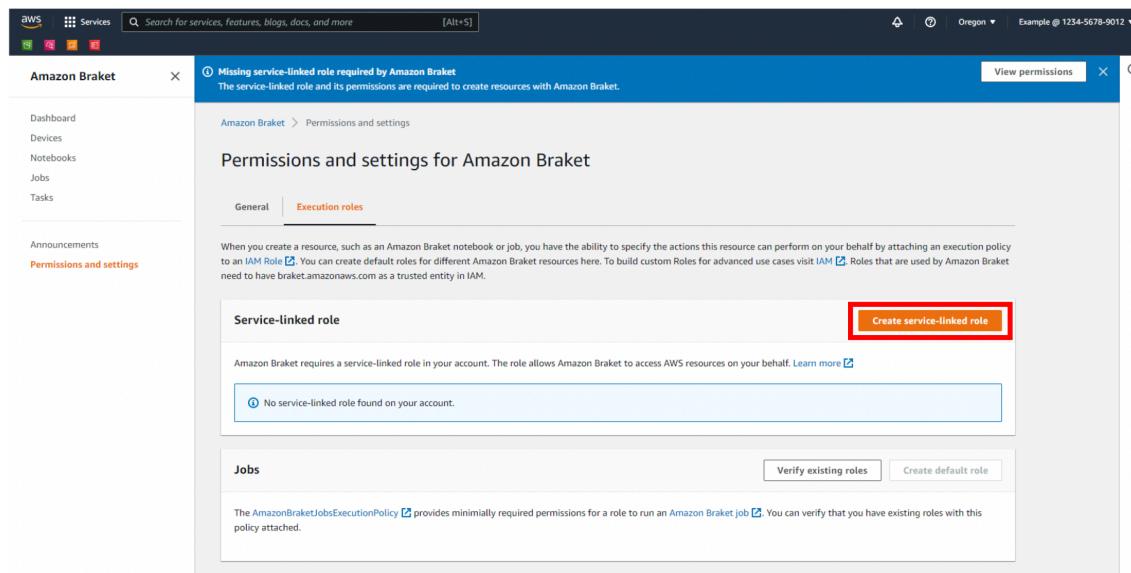
1. Sign in to the [Amazon Braket console](#) with your AWS account.
2. Open the **Amazon Braket** console.
3. When you open the **Amazon Braket** console, you land by default on the **Devices** page. The message boxed in red at the top of the page informs you that you need to create a service-linked role to secure the permissions required to create resources with Amazon Braket.



4. To create a service-linked role for the AWS account, select the **View Permissions** button on the right of the message at the top of the page or choose the **Permissions and settings** tab on the left side of the **Amazon Braket** page and then the **Execution roles** tab on the **Permissions and settings** page. To create this service-linked role, select the **Create service-linked role** button.

Amazon Braket Developer Guide

Steps to enable Amazon Braket



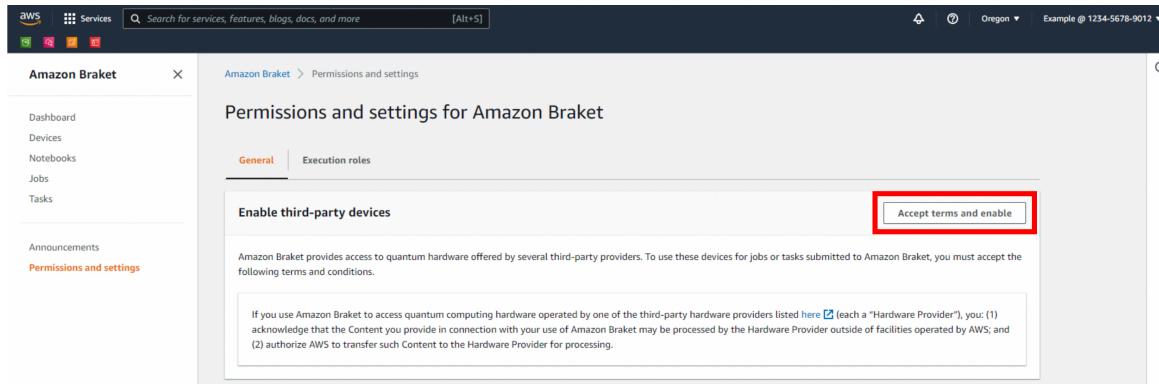
5. In order to use third-party devices, you have to agree to certain conditions regarding data transfer to third parties. For more information, see [Enable third-party devices \(p. 34\)](#).

Note

Systems that do not involve third-party providers, such as Amazon's simulators, can be used without agreeing to the **Enable third-party devices** agreement. But to use third party devices, such as QPU's, this agreement must be signed.

Enable third-party devices

Amazon Braket allows you to run quantum tasks or jobs on any Amazon owned device as long as you have a service-linked role (SLR) in your account. For more information on how to acquire an SLR, see [Steps to enable Amazon Braket \(p. 32\)](#) and see the [Service-linked role \(p. 123\)](#) section to understand what an SLR actually is. To access and enable third-party devices such as QPUs with Amazon Braket, you must agree to certain terms and conditions. The terms and conditions of this agreement are provided on the **General** tab of the **Permissions and settings** page as shown in the following figure. Select the **Accept terms and enable** button to agree to them in order to access third-party devices.



Note

Accepting these terms to enable use of third-party devices only needs to be done **once per account** and only needs to be done if you are accessing third-party hardware. You do not need this to access Amazon Braket's local or managed simulators.

Get started with Amazon Braket

After you have followed the instructions in [Enable Amazon Braket \(p. 32\)](#), you can get started with Amazon Braket.

The steps to get started include:

- [Create an Amazon Braket notebook instance \(p. 35\)](#)
- [Run your first circuit using the Amazon Braket Python SDK \(p. 37\)](#)
- [Run your first annealing problem with Ocean \(p. 39\)](#)

Create an Amazon Braket notebook instance

Amazon Braket provides fully-managed Jupyter notebooks to get you started. The Amazon Braket notebook instances are based on Amazon SageMaker notebook instances. You can [learn more about notebook instances](#). To get started with Braket, follow these steps to create an Amazon Braket notebook instance.

1. Open the [Amazon Braket console](#).
2. Choose **Notebooks** in the left pane, then choose **Create notebook**.
3. In **Notebook instance settings**, enter a **Notebook instance name** using only alphanumeric and hyphen characters.
4. Select the **Notebook instance type**. Choose the smallest type you need. To get started, choose a cost-effective instance type, such as **ml.t3.medium**. The instance types are Amazon SageMaker notebook instances. To learn more, see [Amazon SageMaker pricing](#).
5. In **Permissions and encryption**, select **Create a new role** (a new role with a name that begins with `AmazonBraketServiceSageMakerNotebook` is created).
6. If you want 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.

Amazon Braket Developer Guide

Create an Amazon Braket notebook instance

Screenshot of the AWS Services console showing the creation of an Amazon Braket notebook instance. The 'Notebook instance type' is set to 'ml.t3.medium'. The 'Permissions and encryption' section includes an 'IAM role' dropdown set to 'Use existing role' with 'AmazonBraketServiceSageMakerNotebookRole-20201216T191145' selected. Under 'Root access — optional', the 'Enable' radio button is selected. The 'Encryption key — optional' dropdown is set to 'No custom encryption key'. The 'Network — optional' section contains a red box around the 'Git repository — optional' field, which includes a 'Repository' section with a dropdown for cloning a public git repository and a 'Git repository URL' input field containing 'https://github.com/aws/amazon-braket-sdk-python.git'. The 'Tags — optional' section at the bottom has a note about AWS costs. At the bottom right are 'Cancel' and 'Create notebook instance' buttons.

AWS Services Search for services, features, blogs, docs, and more [Alt+S]

Notebook instance type
Instance types comprise varying combinations of CPU, GPU, memory for building, running your quantum tasks

ml.t3.medium

▶ Additional settings

Permissions and encryption

IAM role
Use existing role

Existing IAM role
AmazonBraketServiceSageMakerNotebookRole-20201216T191145

Root access — *optional*
 Enable - Give users root access to the notebook
 Disable - Don't give users root access to the notebook
Lifecycle configurations always have root access

Encryption key — *optional*
Encrypt your notebook data. Choose an existing KMS key or enter a key ARN.
No custom encryption key

▶ Network — *optional*

▼ Git repository — *optional*

Repository
Jupyter will start with this repository.
Clone a public git repository from url

Git repository URL
Clone a public repository to use for this notebook instance only.
https://github.com/aws/amazon-braket-sdk-python.git

▶ Tags — *optional*
A tag is a label that you assign to an AWS resource. Each tag consists of a key and an optional value. You can use tags to track your AWS costs.

Cancel Create notebook instance

Feedback Looking for language selection? Find it in the new Unified Settings

7. Choose **Create notebook instance**.

It takes several minutes to create the notebook. The notebook is displayed on the **Notebooks** page with a status of **Pending**. When the notebook instance is ready to use, the status changes to **InService**. You may need to refresh the page to display the updated status for the notebook.

Note

You can view and manage your Amazon Braket notebook instances in the Amazon Braket and Amazon SageMaker consoles. Additional Amazon Braket notebook settings are available through the [SageMaker console](#).

If you're working in the Amazon Braket console within AWS, as given previously, the Amazon Braket SDK and plugins are preloaded in the notebooks you just created. If you want to run on your own machine, you can 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` (the latter for use with PennyLane plugins).

Run your first circuit using the Amazon Braket Python SDK

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

The screenshot shows the 'Notebooks' section of the Amazon Braket console. A search bar at the top contains the text 'Name contains: amazon-braket-'. Below it, a table lists one notebook entry:

Name	Instance	Creation time	Status	URL
amazon-braket-first-test	ml.t3.medium	Mar 15, 2021 17:29 (UTC)	InService	amazon-braket-first-test-omsp.notebook.us-east-1.sagemaker.aws

An orange box highlights the URL column for the selected notebook. Below the table, a blue button labeled 'Click here' is visible.

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.

The screenshot shows the Jupyter interface with the 'New' button highlighted. A dropdown menu is open, showing options: 'Notebook', 'Text File', 'Folder', and 'Terminal'. The 'Notebook' option is selected and highlighted with an orange box.

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 Amazon Braket SDK modules and defining a simple 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)
```

Run your circuit on the local simulator

Next, choose the quantum device on which to execute 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).

Here's how 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 you'll find a roughly equal (up to shot noise) distribution of 00 and 11 as measurement outcomes, as expected.

Run your circuit on a managed simulator

Amazon Braket also provides access to a fully-managed, high-performance simulator, SV1, for running larger circuits. SV1 is a 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 \(p. 7\)](#) section and in the AWS console. When running tasks on SV1 (and on TN1 or any QPU), the results of your 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 \(p. 115\)](#).

Note

Fill in your actual, existing bucket name where the following example shows `example-bucket` as your bucket name. Bucket names for Amazon Braket always begin with `amazon-braket-` followed by other identifying characters you add.

```
# get the account ID
aws_account_id = boto3.client("sts").get_caller_identity()["Account"]
# the name of the bucket
my_bucket = "example-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 managed simulator to run your circuit
device = AwsDevice("arn:aws:braket:::device/quantum-simulator/amazon/sv1")

# execute 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 task. Navigate to the **Tasks** tab in the console and your task should be on the top of the list. Alternatively, you can search for your task using the unique task ID or other criteria.

Note

After 90 days, Amazon Braket automatically removes all task IDs and other metadata associated with your 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, just by changing a single line of code. Amazon Braket provides access to QPU devices from D-Wave, IonQ, Oxford Quantum Circuits, Rigetti, and Xanadu. You can find information about the different devices and availability windows in the [Supported Devices \(p. 7\)](#) section, and in the AWS console under the **Devices** tab. The following example shows how to instantiate a Rigetti device.

```
# choose the Rigetti hardware to run your circuit
device = AwsDevice("arn:aws:braket:us-west-1::device/qpu/rigetti/Aspen-M-2")
```

Choose an IonQ device with this code:

```
# choose the Ionq device to run your circuit
device = AwsDevice("arn:aws:braket:::device/qpu/ionq/ionQdevice")
```

D-Wave devices are quantum annealers. They follow a different programming paradigm. The next section explains annealing devices.

When you execute 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 task with 1 day polling timeout
task = device.run(bell, s3_folder, poll_timeout_seconds=24*60*60)
print(task.result().measurement_counts)
```

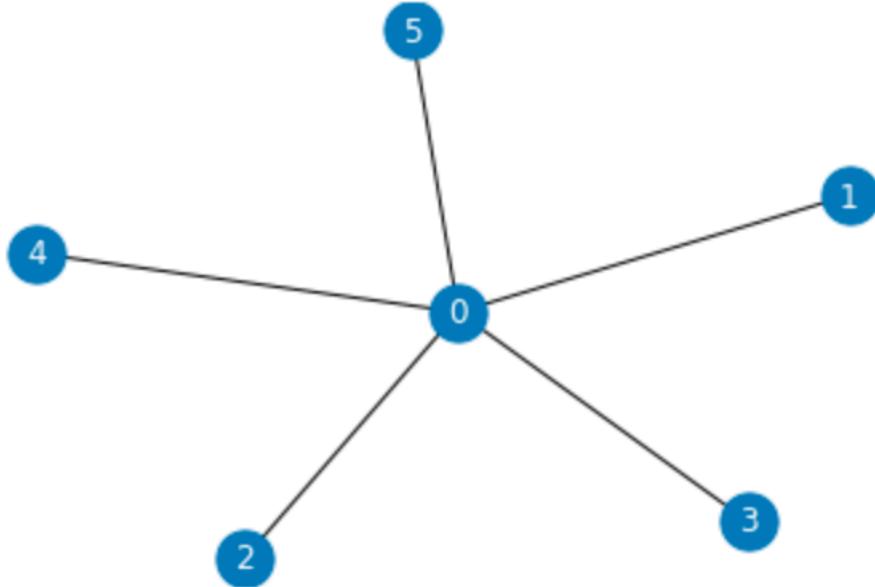
Run your first annealing problem with Ocean

Quantum annealers are special-purpose quantum computers designed to solve combinatorial optimization problems. In particular, quantum annealers solve problems belonging to the class of

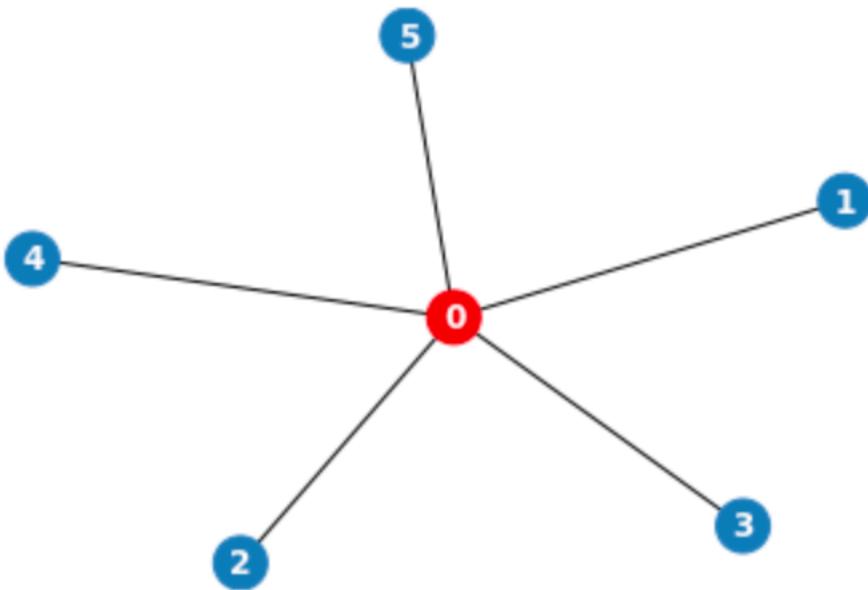
Quadratic Unconstrained Optimization (QUBO). Amazon Braket allows you to program the D-Wave QPUs natively using [D-Wave's Ocean software](#) through the [Braket-Ocean plugin](#). Amazon Braket notebook instances are pre-installed with Ocean and the Braket-Ocean plugin.

Get started with a simple example of solving the [Minimum Vertex Cover \(MVC\) problem](#). Here's the problem definition:

Given an undirected graph with a vertex set and an edge set, a *vertex cover* is a subset of the vertices (nodes) such that each edge in the graph is incident to at least one vertex in the subset. The Minimum Vertex Cover problem seeks to find a cover with a minimum number of vertices in the subset. In other words, in a graph like this:



The goal is to color nodes in red such that every edge touches at least one red node. And you want to do it with as little paint as possible. The optimal solution is to paint the central node red as shown in the figure that follows.



How to solve such a problem with D-Wave's 2000Q QPU

To begin, import the following dependencies and specify an S3 location.

Note

Fill in your actual, existing bucket name where the following example shows example-bucket as your bucket name. Bucket names for Amazon Braket always begin with amazon-braket- followed by other identifying characters you add.

```
# import relevant modules
import boto3
from braket.ocean_plugin import BraketSampler, BraketDWaveSampler
import networkx as nx
import dwave_networkx as dnx
from dwave.system.composites import EmbeddingComposite

# Please enter any S3 bucket starting with 'amazon-braket-' in your account in the code below

my_bucket = f"amazon-braket-Your-Bucket-Name" # the name of the bucket
my_prefix = "Your-Folder-Name" # the name of the folder in the bucket
s3_folder = (my_bucket, my_prefix)
```

Now, set the sampler. Then use EmbeddingComposite to minor-embed a problem automatically into a structured sampler, such as a D-Wave system.

```
# set sampler using BraketSampler
sampler = BraketSampler(s3_folder,'arn:aws:braket:::device/qpu/d-wave/DW_2000Q_6')
# or alternatively using BraketDWaveSampler
sampler = BraketDWaveSampler(s3_folder,'arn:aws:braket:::device/qpu/d-wave/DW_2000Q_6')

# EmbeddingComposite automatically maps the problem to the structure of the solver.
embedded_sampler = EmbeddingComposite(sampler)
```

You can create the graph from the family of random [Erdoes-Renyi graphs](#). Such a graph can be generated using the networkx library. As input, set the desired number of vertices and edges connecting pairs of vertices.

```
# setup Erdos Renyi graph
# 5 nodes
n = 5
# 10 edges
m = 10
# generate graph
graph = nx.gnm_random_graph(n, m, seed=42)
```

Finally, run the problem in D-Wave and print the results.

```
# run the problem on D-Wave using BraketSampler
result = dnx.min_vertex_cover(graph, embedded_sampler, resultFormat="HISTOGRAM")
# or alternatively using BraketDWaveSampler
result = dnx.min_vertex_cover(graph, embedded_sampler, answer_mode="histogram")
print('Result to MVC problem:', result)
print('Size of the vertex cover:', len(result))
```

```
Result to MVC problem: [0, 1, 3, 4]
Size of the vertex cover: 4
```

Notice that the result to the MVC problem is a list containing the vertices [0, 1, 3, 4]. These vertices form a minimum vertex cover such that the subset can reach every edge in the graph.

Work with Amazon Braket

This section shows you how to design quantum circuits and annealing problems, submit these problems as tasks to devices, and monitor the tasks with the Amazon Braket SDK.

The following are the main means of interacting with resources on Amazon Braket.

- The [Amazon Braket Console](#) provides device information and status to help you create, manage, and monitor your resources and 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're 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 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've [installed Boto3](#), an overview of steps for creating a 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:

- [Construct circuits in the SDK \(p. 44\)](#)
- [Run your circuits with OpenQASM 3.0 \(p. 53\)](#)
- [Submitting tasks to QPUs and simulators \(p. 63\)](#)

- [Monitoring and tracking tasks \(p. 70\)](#)
- [Working with Boto3 \(p. 75\)](#)

Construct 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.

In this section:

- [Gates and circuits \(p. 44\)](#)
- [Manual qubit allocation \(p. 47\)](#)
- [Verbatim compilation \(p. 48\)](#)
- [Noise simulation \(p. 48\)](#)
- [Inspecting the circuit \(p. 49\)](#)
- [Result types \(p. 51\)](#)

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-C---
            |
q1 : -H-|-C-
            |
q2 : -H-X-|-
            |
q3 : -H---X-

T : |0| 1 |
```

Example: See all available gates

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

```
import string
from braket.circuits import Gate
# print all available gates in Amazon Braket
```

```
gate_set = [attr for attr in dir(Gate) if attr[0] in string.ascii_uppercase]
print(gate_set)
```

The output from this code lists all of the gates.

```
['CCNot', 'CNot', 'CPhaseShift', 'CPhaseShift00', 'CPhaseShift01', 'CPhaseShift10',
'CSwap', 'CV', 'CY', 'CZ', 'ECR', 'H', 'I', 'ISwap', 'PSwap', 'PhaseShift', 'Rx', 'Ry',
'Rz', 'S', 'Si', 'Swap', 'T', 'Ti', '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, you'd 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.cnot(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(i(XX+YY) theta/4), where theta=0.15 in the examples below
circ.xx(0, 1, 0.15)
# exp(-iXY theta/2)
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)

```

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 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])

```

```
new_instr = instr.copy(target_mapping={0: 5})
```

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.

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

device = AwsDevice("arn:aws:braket:::device/qpu/ionq/ionQdevice")

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

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

```
device = AwsDevice("arn:aws:braket:us-west-1::device/qpu/rigetti/Aspen-M-2")

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

```
Quantum Gates supported by Aspen-M-2:
['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']
```

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.

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.

```
circ = Circuit().h(0).cnot(0, 7) # Indices of actual qubits in the QPU
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](#).

Note

The OQC compiler does not support setting `disable_qubit_rewiring=True`. Setting this flag to True yields the following error: An error occurred (ValidationException) when calling the `CreateQuantumTask` operation: Device arn:aws:braket:eu-west-2::device/qpu/oqc/Lucy does not support disabled qubit rewiring.

Verbatim compilation

When you run a quantum circuit on quantum computers from Rigetti, IonQ, or Oxford Quantum Circuits (OQC), 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 (supported by Rigetti, IonQ, and OQC) 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 you're 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 currently supported on Rigetti, IonQ, and Oxford Quantum Circuits (OQC) 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/Aspen-M-2")
device.properties.action[DeviceActionType.OPENQASM].supportedPragmas
```

There is no additional cost associated with using verbatim compilation. You continue to be charged for tasks executed on Braket QPU devices, notebook instances, and managed 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 OQC and IonQ devices, 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.

```
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 simple circuit with depolarizing noise and a custom Kraus channel.

```
# 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)
```

```
# 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, s3_location)
```

Or

Example 2

```
task = device.run(circ_noise, s3_location)
```

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.

```
# 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)-C-----X-
      |
q1 : -Ry(0.2)--| -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)]))
Instruction('operator': Rx('angle': 0.15, 'qubit_count': 1), 'target':
QubitSet([Qubit(0)]))

MomentsKey(time=0, qubits=QubitSet([Qubit(1)]))
Instruction('operator': Ry('angle': 0.2, 'qubit_count': 1), 'target': QubitSet([Qubit(1)]))

MomentsKey(time=1, qubits=QubitSet([Qubit(0), Qubit(2)]))
Instruction('operator': CNot('qubit_count': 2), 'target': QubitSet([Qubit(0), Qubit(2)]))

MomentsKey(time=1, qubits=QubitSet([Qubit(1), Qubit(3)]))
Instruction('operator': ZZ('angle': 0.15, 'qubit_count': 2), 'target': QubitSet([Qubit(1),
Qubit(3)]))

MomentsKey(time=2, qubits=QubitSet([Qubit(0)]))
Instruction('operator': X('qubit_count': 1), 'target': QubitSet([Qubit(0)]))

```

You can also add gates to a circuit through Moments.

```

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 : ---C-H-

```

T : |0|1|2|

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.

- 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.
- 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 ($\text{mean}([\mathbf{x}-\text{mean}(\mathbf{x})]^2)$) 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	OQC
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

Note

* Rigetti only supports probability result types of up to 40 qubits.

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

```
device = AwsDevice("arn:aws:braket:us-west-1::device/qpu/rigetti/Aspen-M-2")

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

name='Sample' observables=['x', 'y', 'z', 'h', 'i'] minShots=10 maxShots=100000
name='Expectation' observables=['x', 'y', 'z', 'h', 'i'] minShots=10 maxShots=100000
name='Variance' observables=['x', 'y', 'z', 'h', 'i'] minShots=10 maxShots=100000
name='Probability' observables=None minShots=10 maxShots=100000
```

Note

The result types do not apply to D-Wave systems, which is why they are not included in the table above.

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

```
from braket.circuits import 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])
```

Observables

Amazon Braket includes an Observable class, which can be used to specify an observable to be measured.

You can apply at most one unique non-identity observable to each qubit. If you specify two or more different non-identity observables to the same qubit, you see an error. For this purpose, each factor of a tensor product counts as an individual observable, so it is permissible to have multiple tensor products acting on the same qubit, provided that the factor acting on that qubit is the same.

The Observable class includes the following observables.

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

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

# get the tensor product of 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())

# also 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]])))
```

```
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]])
```

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 \(p. 44\)](#) 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 tasks.

- [Create and submit OpenQASM tasks on different Braket devices \(p. 57\)](#)
- [Access the supported operations and result types \(p. 59\)](#)
- [Simulate noise with OpenQASM \(p. 61\)](#)
- [Use verbatim compilation with OpenQASM \(p. 62\)](#)
- [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? \(p. 54\)](#)
- [When to use OpenQASM 3.0 \(p. 54\)](#)
- [How OpenQASM 3.0 works \(p. 54\)](#)
- [Prerequisites \(p. 54\)](#)
- [What OpenQASM features does Braket support? \(p. 54\)](#)
- [Create and submit an example OpenQASM 3.0 task \(p. 57\)](#)
- [Support for OpenQASM on different Braket Devices \(p. 59\)](#)
- [Simulate noise with OpenQASM3 \(p. 61\)](#)
- [Qubit rewiring with OpenQASM3 \(p. 62\)](#)
- [Verbatim Compilation with OpenQASM3 \(p. 62\)](#)
- [The Braket console \(p. 63\)](#)

- [More resources \(p. 63\)](#)

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 simple 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. [.] Braket's 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

Braket's support for OpenQASM 3.0 provides feature parity with the current Intermediate Representation. This means that anything you can do today on hardware devices and managed 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 \(p. 55\)](#)

- [Supported OpenQASM statements \(p. 55\)](#)
- [Braket OpenQASM pragmas \(p. 56\)](#)
- [Advanced feature support for OpenQASM on the Local Simulator \(p. 56\)](#)

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 are used for gate rotation angles:
 - `rx(-0.314) $0;`
- 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]`
- 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`
 - 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 managed 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
- Input
- QASM files

- Subroutines

For examples of each advanced feature, see this [sample notebook](#). For the full OpenQASM specification, see the [OpenQASM website](#).

Create and submit an example OpenQASM 3.0 task

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

In this section:

- [An example OpenQASM 3.0 program \(p. 57\)](#)
- [Use the Python SDK to create OpenQASM 3.0 tasks \(p. 57\)](#)
- [Use Boto3 to create OpenQASM 3.0 tasks \(p. 58\)](#)
- [Use the AWS CLI to create OpenQASM 3.0 tasks \(p. 58\)](#)

An example OpenQASM 3.0 program

To create a OpenQASM 3.0 task, you can start with a simple 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 tasks

You can use the [Amazon Braket Python SDK](#) to submit this program to an Amazon Braket device with the following code.

```
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/Aspen-M-2")
from braket.ir.openqasm import Program

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

# You can also specify an optional s3 bucket location and number of shots,
# if you so choose, when running the program
s3_location = ("amazon-braket-my-bucket", "openqasm-tasks")
my_task = device.run(
    program,
    s3_location,
```

```
    shots=100,  
)
```

Use Boto3 to create OpenQASM 3.0 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 = "amazon-braket-my-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/Aspen-M-2"
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/Aspen-M-2" \
--shots 100 \
--output-s3-bucket "amazon-braket-my-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"
            ],
            ...
        }
    }
}
```

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 Managed 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
}
],
}
...

```

Simulate noise with OpenQASM3

To simulate noise with OpenQASM3, you use *pragma* instructions to add noise operators. For example, to simulate the noisy version of the [GHZ program \(p. 57\)](#) 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>) <qubit>
#pragma braket noise phase_flip(<float>) <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
) <qubit>[, <qubit>]      // maximum of 2 qubits
```

Kraus Operator

In order 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 $2^{2 \times \text{num_qubits}}$ 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 OpenQASM3

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 OpenQASM3

When you run a quantum circuit on quantum computers from Rigetti, OQC, 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'll need to use

native gates within the preserved regions. Currently, IonQ and OQC only support verbatim compilation for the entire circuit, so every instruction in the circuit needs to be enclosed in a verbatim box.

With OpenQASM, you can specify a verbatim pragma around a box of code that is 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`.

```
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 information on verbatim compilation, see the [Verbatim compilation](#) sample notebook.

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 tasks in OpenQASM 3.0 as you had submitting existing tasks.

More 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](#).

Submitting tasks to QPUs and simulators

Amazon Braket provides access to several devices that can run quantum tasks. You can submit tasks individually or you can set up task batching.

QPUs

You can submit 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 task with the task ID, which is introduced in the next section.

- **IonQ** : arn:aws:braket:::device/qpu/ionq/ionQdevice
- **Rigetti Aspen-11** : arn:aws:braket:::device/qpu/rigetti/Aspen-11
- **Rigetti Aspen-M-2** : arn:aws:braket:us-west-1::device/qpu/rigetti/Aspen-M-2
- **D-Wave 2000Q** : arn:aws:braket:::device/qpu/d-wave/DW_2000Q_6
- **D-Wave Advantage_system 4.1** : arn:aws:braket:::device/qpu/d-wave/Advantage_system4
- **D-Wave Advantage_system 6.1** : arn:aws:braket:us-west-2::device/qpu/d-wave/Advantage_system6

- **OQC Lucy** : arn:aws:braket:eu-west-2::device/qpu/oqc/Lucy

Simulators

- **Managed state vector simulator, SV1** : arn:aws:braket:::device/quantum-simulator/amazon/sv1
- **Managed density matrix simulator, DM1** : arn:aws:braket:::device/quantum-simulator/amazon/dm1
- **Managed tensor network simulator, TN1** : arn:aws:braket:::device/quantum-simulator/amazon/tn1
- **The local simulator** : LocalSimulator()

Note

You can cancel tasks in the CREATED state for QPUs and managed simulators. You can cancel tasks in the QUEUED state on a best-effort basis for managed simulators and QPUs. Note that QPU QUEUED tasks are unlikely to be cancelled successfully during QPU availability windows.

In this section:

- [Example tasks on Amazon Braket \(p. 64\)](#)
- [Submitting tasks to a QPU \(p. 67\)](#)
- [Running a task with the local simulator \(p. 68\)](#)
- [Task batching \(p. 69\)](#)
- [Set up SNS notifications \(optional\) \(p. 70\)](#)

Example tasks on Amazon Braket

This section walks through the stages of running an example 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 \(p. 64\)](#)
- [Submit an example task \(p. 65\)](#)
- [Specify shots \(p. 65\)](#)
- [Poll for results \(p. 66\)](#)
- [View the example results \(p. 66\)](#)

Specify the device

First, select and specify the device for your task. This example shows how to choose the simulator, SV1.

```
# choose the managed 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 task

Submit an example task to run on the managed 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 = "amazon-braket-your-s3-bucket-name" # 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 task to run
my_task = device.run(circ, s3_location, shots=1000, poll_timeout_seconds = 100,
    poll_interval_seconds = 10)

# get results of the task
result = my_task.result()
```

The `device.run()` command creates a task through the [CreateQuantumTask API](#). After a short initialization time, the task is queued until capacity exists to run the task on a device. In this case, the device is the managed simulator 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 (for example, a circuit or annealing problem) is limited to 3MB in size.

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`.

The maximum number of shots allowed for the the managed simulator and Rigetti device is 100,000. For D-Wave and IonQ devices, the maximum is 10,000 shots.

Poll for results

When executing `my_task.result()`, the SDK begins polling for a result with the parameters you define upon task creation:

- `poll_timeout_seconds` is the number of seconds to poll the task before it times out when running the task on the managed 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 task is polled. It specifies how often you call the Braket API to get the status when the task is run on the managed 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 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

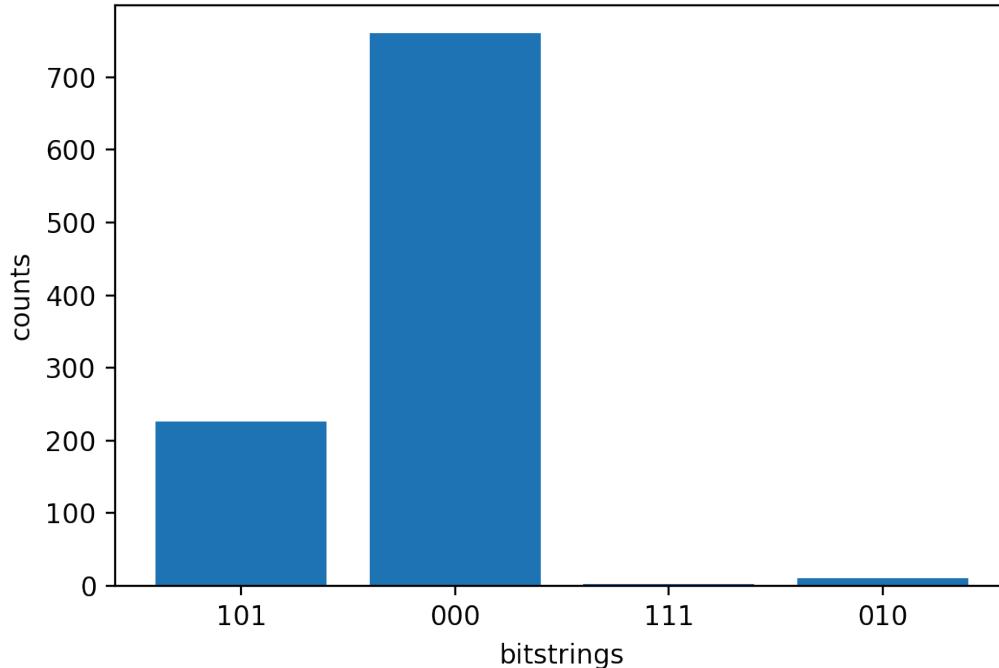
Because you've also specified the `ResultType`, you can view the returned results. The result types appear in the order in which they were added to the circuit.

```
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]
```



Submitting tasks to a QPU

Amazon Braket allows you to run a quantum circuit on a QPU device. The following example shows how to submit a task to the Rigetti or the IonQ device.

Choose the Rigetti 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/Aspen-M-2")

# take a look at the device connectivity graph
device.properties.dict()['paradigm']['connectivity']
```

```
{'fullyConnected': False,
'connectivityGraph': {'0': ['1', '7'],
'1': ['0', '16'],
'2': ['3', '15'],
'3': ['2', '4'],
'4': ['3', '5'],
'5': ['4', '6'],
'6': ['5', '7'],
'7': ['0', '6'],
'11': ['12', '26'],
'12': ['13', '11'],
'13': ['12', '14'],
'14': ['13', '15'],
```

```
'15': ['2', '14', '16'],
'16': ['1', '15', '17'],
'17': ['16'],
'20': ['21', '27'],
'21': ['20', '36'],
'22': ['23', '35'],
'23': ['22', '24'],
'24': ['23', '25'],
'25': ['24', '26'],
'26': ['11', '25', '27'],
'27': ['20', '26'],
'30': ['31', '37'],
'31': ['30', '32'],
'32': ['31', '33'],
'33': ['32', '34'],
'34': ['33', '35'],
'35': ['22', '34', '36'],
'36': ['21', '35', '37'],
'37': ['30', '36']}}
```

The preceding dictionary `connectivityGraph` contains information about the connectivity of the current Rigetti device.

Choose the IonQ device

For the IonQ 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 device
device = AwsDevice("arn:aws:braket:::device/qpu/ionq/ionQdevice")

# take a look at the device connectivity graph
device.properties.dict()['paradigm']['connectivity']

{'fullyConnected': True, 'connectivityGraph': {}}
```

For any QPU device, when you launch a task, remember to specify the S3 bucket in which to store your results. You have the option to adjust the shots (default=1000), the `poll_timeout_seconds` (default = 432000 = 5 days), and the `poll_interval_seconds` (default = 1) when you submit the task, as shown in the following example.

```
my_task = device.run(circ, s3_location, 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 tasks within certain availability windows, but you can still submit tasks any time (24/7) because all corresponding data and metadata are stored reliably in your S3 bucket. As shown in the next section, you can recover your task using `AwsQuantumTask` and your unique task ID.

Running a task with the local simulator

You can send 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 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 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")
```

Task batching

Task batching is available on every Amazon Braket device, except the local simulator. Batching is especially useful for tasks you run on the managed simulators (TN1 or SV1) because they can process multiple tasks in parallel. To help you set up various tasks, [.noloc]` Amazon Braket provides [example notebooks](#).

Batching allows you to launch tasks in parallel. For example, if you wish to make a calculation that requires 10 tasks and the circuits in those tasks are independent of each other, it is a good idea to use batching. That way, you don't have to wait for one task to be complete before another task begins.

The following example shows how to run a batch of 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 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.

About task batching and costs

A few caveats to keep in mind regarding task batching and billing costs:

- By default, task batching retries all time out or fail tasks 3 times.
- A batch of long running 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 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](#)).

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](#).

Set up SNS notifications (optional)

You can set up notifications through the Amazon Simple Notification Service (SNS) so that you receive an alert when your Amazon Braket task is complete. Active notifications are useful if you expect a long wait time; for example, when you submit a large task or when you submit a task outside of a device's availability window. If you do not want to wait for the 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](#).

Monitoring and tracking tasks

After you submit a task, you can keep track of its status through the Amazon Braket SDK and console. When the task completes, Braket saves the results in your specified Amazon S3 location. Completion may take some time, especially for QPU devices, depending on the length of the queue. Status types include:

- CREATED
- RUNNING
- COMPLETED
- FAILED
- CANCELLED

In this section:

- [Tracking tasks from the Amazon Braket SDK \(p. 70\)](#)
- [Advanced logging \(p. 72\)](#)
- [Monitoring tasks through the Amazon Braket console \(p. 74\)](#)

Tracking tasks from the Amazon Braket SDK

The command `device.run(...)` defines a task with a unique 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 task in the background.

Retrieve a result

When you call `task.result()`, the SDK begins polling Amazon Braket to see whether the task is complete. The SDK uses the polling parameters you defined in `.run()`. After the 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: RUNNING  
Status: RUNNING  
Status: COMPLETED
```

Cancel a task

To cancel a task, call the `cancel()` method, as shown in the following example.

```
# cancel 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 task, as shown in the following example.

```
# get the metadata of the 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 task or result

If your kernel dies after you submit the task or if you close your notebook or computer, you can reconstruct the task object with its unique ARN (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()
```

Advanced logging

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 task can be long-running and the 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 task.

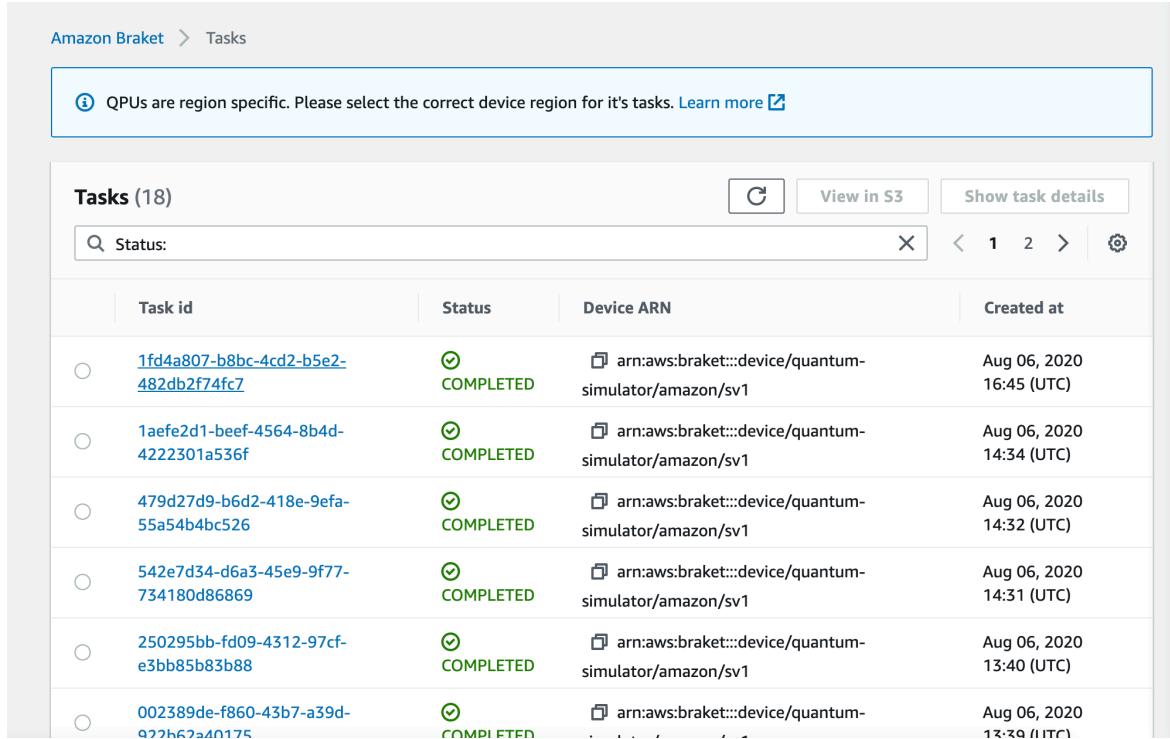
```
# parse log file for arn
with open(log_file) as openfile:
    for line in openfile:
        for part in line.split():
            if "arn:" in part:
                arn = part
                break
# remove final semicolon in logs
arn = arn[:-1]

# 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()
```

Monitoring tasks through the Amazon Braket console

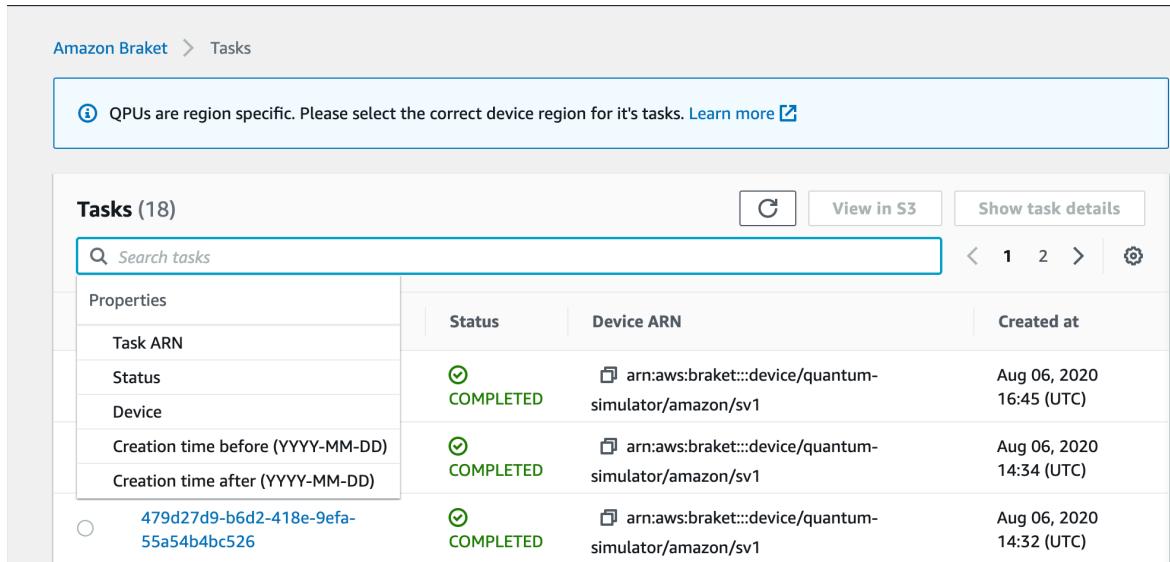
Amazon Braket offers a convenient way of monitoring the task through the [Amazon Braket console](#). All submitted tasks are listed in the **Tasks** field as shown in the following figure. This service is *Region-specific*, which means that you can only view those tasks created in the specific AWS Region.



The screenshot shows the 'Tasks' page in the Amazon Braket console. At the top, there is a message: 'Qpus are region specific. Please select the correct device region for its tasks. [Learn more](#)' with a help icon. Below this, the title 'Tasks (18)' is displayed, followed by a search bar with placeholder 'Status:' and a clear button. To the right of the search bar are buttons for 'View in S3' and 'Show task details'. A navigation bar with arrows and a page number '1 2' is also present. The main table has columns: 'Task id', 'Status', 'Device ARN', and 'Created at'. The data is as follows:

Task id	Status	Device ARN	Created at
1fd4a807-b8bc-4cd2-b5e2-482db2f74fc7	COMPLETED	arn:aws:braket:::device/quantum-simulator/amazon/sv1	Aug 06, 2020 16:45 (UTC)
1aefe2d1-beef-4564-8b4d-4222301a536f	COMPLETED	arn:aws:braket:::device/quantum-simulator/amazon/sv1	Aug 06, 2020 14:34 (UTC)
479d27d9-b6d2-418e-9efa-55a54b4bc526	COMPLETED	arn:aws:braket:::device/quantum-simulator/amazon/sv1	Aug 06, 2020 14:32 (UTC)
542e7d34-d6a3-45e9-9f77-734180d86869	COMPLETED	arn:aws:braket:::device/quantum-simulator/amazon/sv1	Aug 06, 2020 14:31 (UTC)
250295bb-fd09-4312-97cf-e3bb85b83b88	COMPLETED	arn:aws:braket:::device/quantum-simulator/amazon/sv1	Aug 06, 2020 13:40 (UTC)
002389de-f860-43b7-a39d-922b62340175	COMPLETED	arn:aws:braket:::device/quantum-simulator/amazon/sv1	Aug 06, 2020 13:39 (UTC)

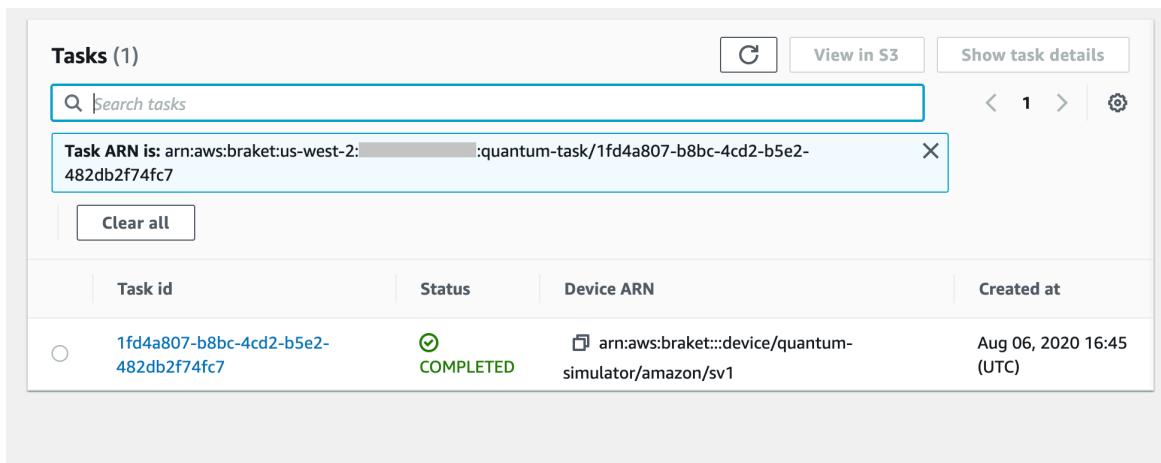
You can search for particular tasks through the navigation bar. The search can be based on Task ARN (ID), status, device, and creation time. The options appear automatically when you select the navigation bar, as shown in the following example.



The screenshot shows the 'Tasks' page in the Amazon Braket console with a search interface. At the top, there is a message: 'Qpus are region specific. Please select the correct device region for its tasks. [Learn more](#)' with a help icon. Below this, the title 'Tasks (18)' is displayed, followed by a search bar with placeholder 'Search tasks' and a clear button. To the right of the search bar are buttons for 'View in S3' and 'Show task details'. A navigation bar with arrows and a page number '1 2' is also present. On the left, there is a sidebar with filter options: 'Properties', 'Task ARN', 'Status', 'Device', 'Creation time before (YYYY-MM-DD)', and 'Creation time after (YYYY-MM-DD)'. The main table has columns: 'Status', 'Device ARN', and 'Created at'. The data is as follows:

Properties	Status	Device ARN	Created at
Task ARN	COMPLETED	arn:aws:braket:::device/quantum-simulator/amazon/sv1	Aug 06, 2020 16:45 (UTC)
Status	COMPLETED	arn:aws:braket:::device/quantum-simulator/amazon/sv1	Aug 06, 2020 14:34 (UTC)
Device	COMPLETED	arn:aws:braket:::device/quantum-simulator/amazon/sv1	Aug 06, 2020 14:32 (UTC)
Creation time before (YYYY-MM-DD)			
Creation time after (YYYY-MM-DD)			
479d27d9-b6d2-418e-9efa-55a54b4bc526	COMPLETED	arn:aws:braket:::device/quantum-simulator/amazon/sv1	Aug 06, 2020 14:32 (UTC)

The following image shows an example of searching for a task based on its unique task ID, which can be obtained by calling `task.id`.



Working with 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 \(p. 75\)](#)
- [Configure AWS CLI profiles for Boto3 and the Amazon Braket SDK \(p. 78\)](#)

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`.

Note

For backwards compatibility with older versions of BraketSchemas, OpenQASM information is omitted from GetDevice API calls. To get this information, the user-agent needs to present a recent version of the BraketSchemas (1.8.0 or later). The Braket SDK automatically reports this for you. If you do not see OpenQASM results in the GetDevice response when using a Braket SDK, you may need to set the `AWS_EXECUTION_ENV` environment variable to configure the user-agent. See the code examples provided in the [GetDevice does not return OpenQASM results error \(p. 131\)](#) topic for how to do this for the AWS CLI, Boto3, and the Go, Java, and JavaScript/TypeScript SDKs.

```
import boto3
```

```
import botocore

client = boto3.client("braket",
    config=botocore.client.Config(user_agent_extra="BraketSchemas/1.8.0"))
```

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 \(p. 76\)](#)
- [Retrieve a device \(p. 76\)](#)
- [Create a quantum task \(p. 76\)](#)
- [Retrieve a quantum task \(p. 77\)](#)
- [Search for quantum tasks \(p. 77\)](#)
- [Cancel quantum task \(p. 78\)](#)

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 response
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 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 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 Amazon 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 \(p. 78\)](#)
- [Step 2: Establish a Boto3 session object \(p. 78\)](#)
- [Step 3: Incorporate the Boto3 session into the Braket AwsSession \(p. 79\)](#)

Step 1: Configure a local AWS CLI profile

It is beyond the scope of this document to explain how to create IAM users and how to configure a non-default profile. For information on these topics, see:

- [Configure an IAM User](#)
- [Establish a CLI profile](#)

To use Amazon Braket, you must provide this IAM 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

```
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, for example, by calling the `device.run(...)` command. All API calls made by that device can leverage the IAM credentials associated with the CLI profile that you previously designated as `profile`.

Amazon Braket Hybrid Jobs User Guide

This user guide provides instructions on Amazon Braket Hybrid Jobs setup and examples of how to create and manage a job. The guide focuses on using Amazon Braket Hybrid Jobs with the [Amazon Braket Python SDK](#) and the [Amazon Braket console](#). A final topic provides a basic introduction to using the Amazon Braket API directly.

In this section:

- [When to use Amazon Braket Hybrid Jobs \(p. 80\)](#)
- [Run a job with Amazon Braket Hybrid Jobs \(p. 81\)](#)
- [Create your first job \(p. 82\)](#)
- [Inputs, outputs, environmental variables, and helper functions \(p. 87\)](#)
- [Save job results \(p. 89\)](#)
- [Save and restart jobs using checkpoints \(p. 91\)](#)
- [Define the environment for your algorithm script \(p. 92\)](#)
- [Use hyperparameters \(p. 93\)](#)
- [Configure the job instance to run your algorithm script \(p. 94\)](#)
- [Cancel a job \(p. 96\)](#)
- [Use Amazon Braket Hybrid Jobs to run a QAOA algorithm \(p. 97\)](#)
- [Accelerate your hybrid workloads with embedded simulators from PennyLane \(p. 100\)](#)
- [Build and debug a job with local mode \(p. 105\)](#)
- [Bring your own container \(BYOC\) \(p. 105\)](#)
- [Configure the default bucket in AwsSession \(p. 107\)](#)
- [Interact with jobs directly using the API \(p. 107\)](#)

When to use Amazon Braket Hybrid Jobs

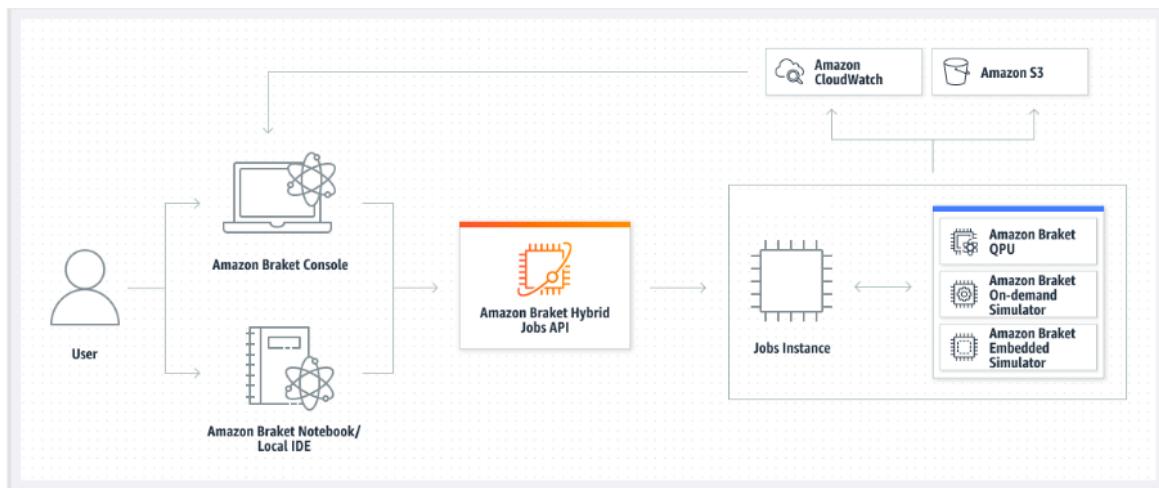
Amazon Braket Hybrid Jobs enables you to easily run hybrid quantum-classical algorithms, such as the Variational Quantum Eigensolver (VQE) and the Quantum Approximate Optimization Algorithm (QAOA), that combine classical compute resources with quantum computing devices to optimize the performance of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

1. **Convenience:** Amazon Braket Hybrid Jobs simplifies setting up and managing your compute environment and keeping it running while your hybrid algorithm runs. You just need to provide your algorithm script and select a quantum device, either a processing unit (QPU) 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.
2. **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.
3. **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.

Run a job with Amazon Braket Hybrid Jobs

To run a 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 \(BYOC\) \(p. 105\)](#).

In either case, next you create a job using the Amazon Braket API, where you provide your algorithm script or container, select the target quantum device the 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, a managed simulator (such as SV1, DM1 or TN1), or the classical job instance itself. With a managed 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 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 a managed or embedded simulator, Amazon Braket starts running the job right away. It spins up the 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 jobs per quantum processing unit (QPU) is restricted, so queues are used to control the number of jobs allowed to run so as not to exceed the limit allowed. If your target device is a QPU, your job first enters the *job queue* of the selected QPU. Once your job has moved up to first position and the device is ready to start a new job, Amazon Braket spins up the job instance needed and runs your job on the device. For the duration of your algorithm, your job has priority access, meaning that tasks from your job run ahead of other tasks queued up on the device. You are only billed when your job starts and not for any wait time in the job queue.

Note

Devices are regional and your 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.

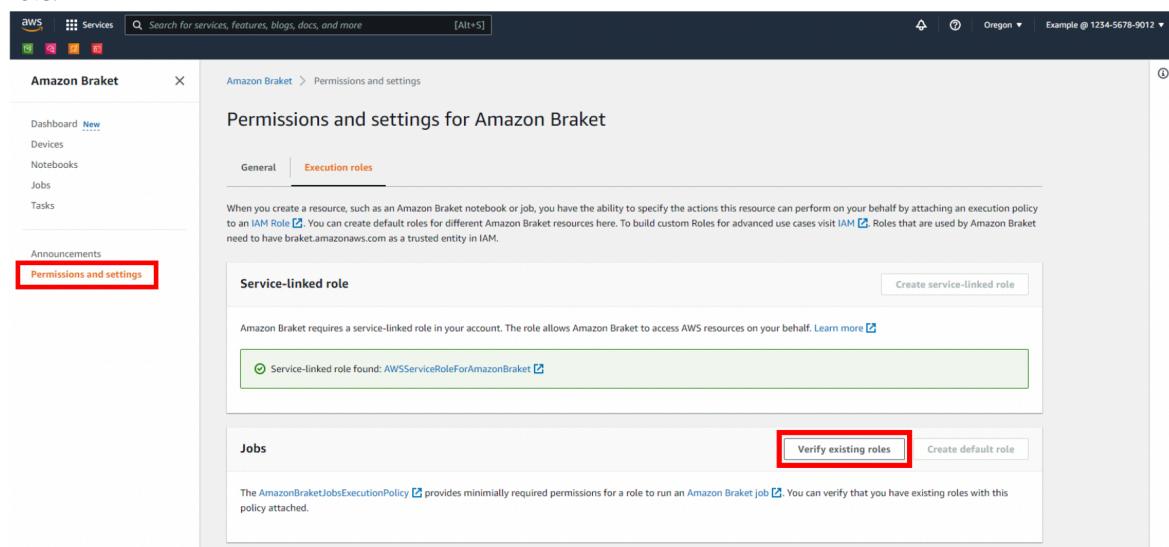
Note

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 (e.g lightning.gpu). If you choose one of the CPU-based embedded simulators (e.g lightning.qubit, or braket:default-simulator), the GPU will not be used and you may incur unnecessary costs.

Create your first job

This section shows you how create a basic first job.

Before you run your first 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 job or guides you through the creation of a default role that can be used to run your job if you do not already have such a role.



To verify that you have roles with sufficient permissions to execute a 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.

Amazon Braket Developer Guide

Create your first job

The screenshot shows the 'Permissions and settings' page for Amazon Braket. The 'Execution roles' tab is selected. In the 'Service-linked role' section, a green box highlights the message 'Service-linked role found: AWSServiceRoleForAmazonBraket'. In the 'Jobs' section, a green box highlights the message 'Roles were found with sufficient permissions to execute jobs.' A red box highlights the 'Show roles' button. Below it, a table lists a single role: 'AmazonBraketJobsExecutionRole' with ARN 'arn:aws:iam:872334998709:role/service-role/AmazonBraketJobsExecutionRole'. A red box highlights the ARN.

If you do not have a role with sufficient permissions to execute a job, you get a message that no such role was found. Select the **Create default role** button to obtain a role with sufficient permissions.

The screenshot shows the 'Permissions and settings' page for Amazon Braket. The 'Execution roles' tab is selected. In the 'Service-linked role' section, a green box highlights the message 'Service-linked role found: AWSServiceRoleForAmazonBraket'. In the 'Jobs' section, a blue box highlights the message 'No roles found with the AmazonBraketJobsExecutionPolicy attached and braket.amazonaws.com as a trusted entity in IAM.' A red box highlights the 'Create default role' button.

If the role was created successfully, you get a message confirming this.

Amazon Braket Developer Guide

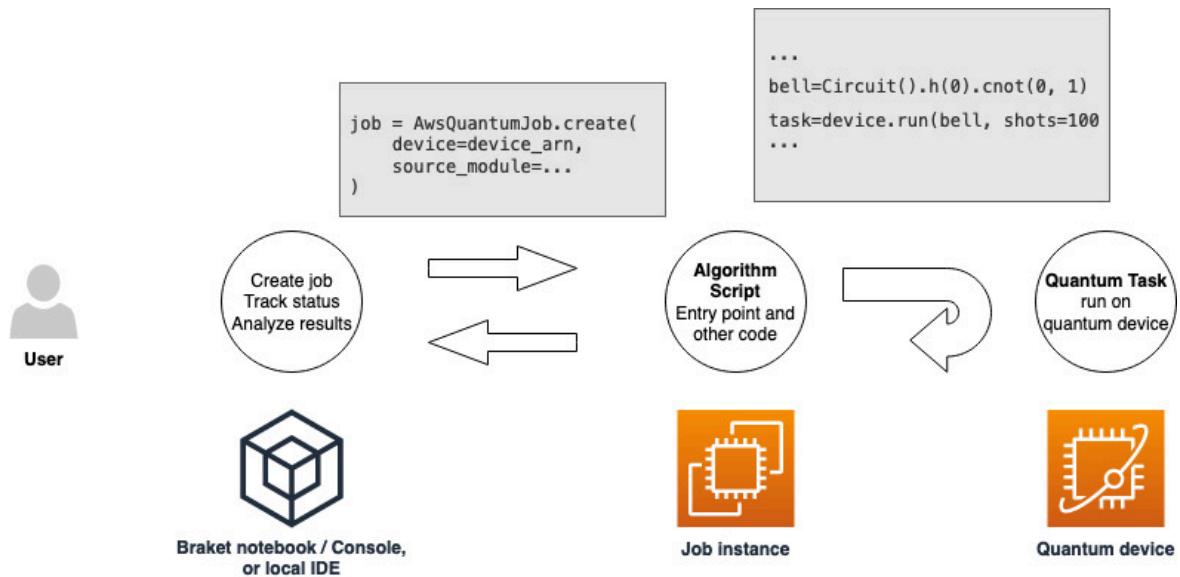
Create your first job

The screenshot shows the 'Permissions and settings' section for Amazon Braket. Under the 'Execution roles' tab, it displays a 'Service-linked role' section with a green success message: 'Service-linked role found: AWSServiceRoleForAmazonBraket'. Below this, under the 'Jobs' section, there is a green success message: 'Created AmazonBraketJobsExecutionRole successfully.'

If you do not have permissions to make this inquiry, you will be denied access. In this case, contact your internal AWS administrator.

The screenshot shows the 'Permissions management for Amazon Braket' section. It displays a 'Jobs' section with a red warning message: 'Amazon Braket jobs require the roles with managed policy AmazonBraketJobsExecutionPolicy attached, which provides minimally required permissions to an Amazon Braket job.' Below this, a red-bordered box contains an 'AccessDenied' error message: 'User: arn:aws:sts::012345678912:assumed-role/SampleRoleName/username is not authorized to perform: iam>ListAttachedRolePolicies on resource: role AmazonBraketJobsExecutionRole with an explicit deny'.

Once you have a role with permissions to run a job, you are ready to proceed. The key piece of your first Braket 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 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 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 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 \(p. 37\)](#)

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 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 job. The Amazon Braket SDK autoselects this AWS Region. For example, a job in us-east-1 can use IonQ, SV1, DM1, and TN1 devices; but not D-Wave or Rigetti devices.

If you choose a quantum computer instead of a simulator, Braket schedules your jobs to run all of their tasks with priority access.

```
from braket.aws import AwsQuantumJob

job = AwsQuantumJob.create(
    "arn:aws:braket:::device/quantum-simulator/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(
    "arn:aws:braket:::device/quantum-simulator/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/braket-job-
default-1631915042705
.....
.
.
.

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
```

Alternatively, you can access the log output from Amazon CloudWatch. 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 last part of your job-`arn` in the name. In the example above, this is `braket-job-default-1631915042705/algo-1-1631915190`.

Amazon Braket Developer Guide

Inputs, outputs, environmental variables, and helper functions

The screenshot shows the AWS CloudWatch Log groups interface. The left sidebar has 'Logs' selected under 'Log groups'. The main area shows a list of log events for a specific job. The first event is timestamped 2021-11-10T17:01:01.993-07:00 and contains the message: "There are older events to load. [Load more](#)". Subsequent events show various AWS API calls related to the Braket job, such as 'aws-amazon-braket-sdk-python-staging' and 'aws-amazon-braket-sdk-python-staging-3f885a942c09911b104eede05328733f34779fa6/test/unit_tests/braket/circuits/test_gates.py'.

You can also view the status of the job in the console. Select the **Jobs** page and **Settings**.

The screenshot shows the AWS Amazon Braket Jobs Settings page. The 'Jobs' tab is selected in the left sidebar. The main area displays the 'Details' section for a job named 'JobTest-autograd-1636588595'. It shows the Job ARN (arn:aws:braket:west-1:645855316487:job/JobTest-autograd-1636588595), Device (arn:aws:braket::device/quantum-simulator/amazon/sv1), and Status reason (—). To the right, there are sections for 'Event times' (Created at: Wed, 10 Nov 2021 23:56:41 GMT; Started at: Wed, 10 Nov 2021 23:56:47 GMT; Ended at: Thu, 11 Nov 2021 00:03:21 GMT) and 'Stopping conditions' (Max runtime (seconds): 432000). Below these, the 'Source code and instance configuration' section lists the entry point (qa0a_source.qa0a_algorithm_script:start_here), container image (292282985366.dkr.ecr.us-west-1.amazonaws.com/tensorflow-jobs:2.4.1-cpu-py37-ubuntu18.04), script source (s3://amazon-braket-us-west-1-645855316487/jobs/JobTest-autograd-1636588595/script/source.tar.gz), instance type (ml.m5.large), checkpoint data path (s3://amazon-braket-us-west-1-645855316487/jobs/JobTest-autograd-1636588595/checkpoints), and volume size (GB) (30).

Your 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>` directory. You can configure the S3 locations where these artifacts are stored by specifying a different `code_location` when the 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>` 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`.

Inputs, outputs, environmental variables, and helper functions

In addition to the file or files that makes up your complete algorithm script, your job can have additional inputs and outputs. When your job starts, Amazon Braket copies inputs provided as part of the job

creation into the container that runs the algorithm script. When the 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 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 \(p. 88\)](#)
- [Outputs \(p. 88\)](#)
- [Environmental variables \(p. 89\)](#)
- [Helper functions \(p. 89\)](#)

Inputs

Input data: You can provide input data using `input_data`. Specify that `input_data` is a keyword in the `AwsQuantumJob.create` function in the SDK. This data is copied to the container filesystem at the location given by the environment variable "AMZN_BRAKET_INPUT_DIR". For an example, see the [QAOA with Amazon Braket Hybrid Jobs and PennyLane](#) Jupyter notebook.

Note

When the input data is large (>1GB), there will be a long wait time before the 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 job request, and, finally, the job request is submitted to Braket service.

Hyperparameters: If you pass in `hyperparameters`, they are available under the environment variable "AMZN_BRAKET_HP_FILE". Hyperparameters are passed directly to the API (not via S3) when creating a job.

Checkpoints: To specify a job-arn whose checkpoint you want to use in a new job, use the `copy_checkpoints_from_job` command. This command copies over the checkpoint data to the `checkpoint_configs3Uri` of the new 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 job will not be used in the new job.

Outputs

Tasks: Task results are stored in the S3 location specified in `output_data_config`. If you don't specify this value, it defaults to `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 you don't specify this value, it defaults to `s3://amazon-braket-<region>-<accountID>/jobs/<job-name>/data`. We provide the SDK helper function `save_job_result` that allows you can store results conveniently and in the correct format 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 job is running. For an example of how to use algorithm metrics, see [Use Amazon Braket Hybrid Jobs to run a QAOA algorithm_ \(p. 97\)](#).

Environmental variables

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

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

Helper functions

Amazon Braket provides several helper functions to simplify the interactions with container inputs and outputs. The following example demonstrates how to use them.

```
save_job_result() # helper function to save your job results
save_job_checkpoint() # helper function to save checkpoints
load_job_checkpoint() # helper function to load a previously saved job checkpoints
```

Save job results

You can save the results generated by the algorithm script so that they are available from the job object in the 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. To save the results of the 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 job, it outputs the identifier and the job-arn, followed by the state of the job every 10 seconds until the job is COMPLETED, after which it shows you the results of the bell circuit. For example:

```
arn:aws:braket:us-west-2:111122223333:job/braket-job-default-1234567890123
INITIALIZED
RUNNING
...
RUNNING
RUNNING
COMPLETED
{'measurement_counts': [{'11': 53, '00': 47}, {'00': 51, '11': 49}]}

---


```

Save and restart jobs using checkpoints

You can save intermediate iterations of your 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 job completed!!!!")
```

When you run the job, it creates the file `<jobname>-checkpoint-1.json` in your job artifacts in the checkpoints directory with a default /opt/jobs/checkpoints path. The job script remains unchanged unless you want to change this default path.

If you want to load a job from a checkpoint generated by a previous job, the algorithm script uses `from braket.jobs import load_job_checkpoint`. The logic to load in your algorithm script is:

```
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 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="arn:aws:braket:<region>:<account-id>:job/
<previous_job_name>", #ADD
```

Define the environment for your algorithm script

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

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

The following table provides details about the containers and the libraries with which they come.

Amazon Braket containers

Type	PennyLane- TensorFlow:2.4.1-gpu- py37-ubuntu18.04	PennyLane-PyTorch1.9.1- gpu-py38-ubuntu20.04	Braket-Base:1.0.0-cpu-py37- ubuntu18.04
Base	111122223333.dkr.ecr.us- east-1.amazonaws.com/ amazon-braket- tensorflow-jobs:2.4.1-gpu- py37-cu110-ubuntu18.04	111122223333.dkr.ecr.us- west-2.amazonaws.com/ amazon-braket-pytorch- jobs:1.9.1-gpu-py38- cu111-ubuntu20.04	ubuntu:18.04
Inherited Libraries	<ul style="list-style-type: none">• awscli• numpy• pandas• scipy	<ul style="list-style-type: none">• awscli• numpy• pandas• scipy	
Additional Libraries	<ul style="list-style-type: none">• amazon-braket-default- simulator• amazon-braket-ocean- plugin• amazon-braket- pennylane-plugin• amazon-braket-schemas• amazon-braket-sdk• boto3, dwave-ocean-sdk• ipykernel• keras• matplotlib• networkx• openbabel• PennyLane• PennyLane-Qchem• protobuf• psi4• rsa• PennyLane-Lightning- gpu	<ul style="list-style-type: none">• amazon-braket-default- simulator• amazon-braket-ocean- plugin• amazon-braket- pennylane-plugin• amazon-braket-schemas• amazon-braket-sdk• boto3, dwave-ocean-sdk• ipykernel• keras• matplotlib• networkx• openbabel• PennyLane• PennyLane-Qchem• protobuf• psi4• rsa• PennyLane-Lightning- gpu	<ul style="list-style-type: none">• amazon-braket-default- simulator• amazon-braket-ocean- plugin• amazon-braket-pennylane- plugin• amazon-braket-schemas• amazon-braket-sdk• awscli• boto3• dwave-ocean-sdk• ipykernel• matplotlib• networkx• numpy• openbabel• pandas• PennyLane• protobuf• psi4

Type	PennyLane-TensorFlow:2.4.1-gpu-py37-ubuntu18.04	PennyLane-PyTorch1.9.1-gpu-py38-ubuntu20.04	Braket-Base:1.0.0-cpu-py37-ubuntu18.04
	• cuQuantum	• cuQuantum	• rsa • scipy

You can view and access the open source container definitions at [aws/amazon-braket-containers](https://aws.amazon.com/amazon-braket-containers). Choose the container that best matches your use case. The container must be in the AWS Region from which you invoke your job. You specify the container image when you create a job by adding one of the following three arguments to your `create(...)` call in the 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:1.0-cpu-py37-ubuntu18.04"`
- **Tensorflow image** `image_uri="292282985366.dkr.ecr.us-east-1.amazonaws.com/amazon-braket-tensorflow-jobs:2.4.1-gpu-py37-cu110-ubuntu18.04"`
- **PyTorch image** `image_uri="292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-pytorch-jobs:1.9.1-gpu-py38-cu111-ubuntu20.04"`

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_tf = retrieve_image(Framework.PL_TENSORFLOW, "us-west-2")
image_uri_pytorch = retrieve_image(Framework.PL_PYTORCH, "us-west-2")
```

Use hyperparameters

You can define hyperparameters needed by your algorithm, such as the learning rate or the depth of your Ansatz, when you create a job. Hyperparameter values are used to tune an algorithm for optimal performance. This is currently a manual process in Amazon Braket. You specify the hyperparameter values that you want to test when searching for the optimal set of values. You load the hyperparameters in your algorithm script with the following code.

```
import json

with open(hp_file, "r") as f:
    hyperparams = json.load(f)
```

You pass in the hyperparameter during job creation by adding the following code to your job script.

```
job = AwsQuantumJob.create(
    ...
    hyperparameters={
        "param-1": "first parameter",
        "param-2": "second parameter",
        ...
    },
)
```

Configure the job instance to run your algorithm 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 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 will be able to additionally request more than one instance in the `InstanceConfig` by specifying the `instanceCount` and setting it >1 . 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 job using the data parallel feature. See the following example notebook for more details on how-to see [this Braket example](#).

The following three tables list the available instance types and specs for standard, compute optimized, and accelerated computing instances.

Note

To view the default classical compute instance quotas for Hybrid Jobs, see [this page \(p. 22\)](#).

Standard Instances	vCPU	Memory
<code>ml.m5.large</code> (default)	2	8 GiB
<code>ml.m5.xlarge</code>	4	16 GiB
<code>ml.m5.2xlarge</code>	8	32 GiB
<code>ml.m5.4xlarge</code>	16	64 GiB
<code>ml.m5.12xlarge</code>	48	192 GiB
<code>ml.m5.24xlarge</code>	96	384 GiB
<code>ml.m4.xlarge</code>	4	16 GiB
<code>ml.m4.2xlarge</code>	8	32 GiB
<code>ml.m4.4xlarge</code>	16	64 GiB

Standard Instances	vCPU	Memory
ml.m4.10xlarge	40	256 GiB

Compute Optimized Instances	vCPU	Memory
ml.c4.xlarge	4	7.5 GiB
ml.c4.2xlarge	8	15 GiB
ml.c4.4xlarge	16	30 GiB
ml.c4.8xlarge	36	192 GiB
ml.c5.xlarge	4	8 GiB
ml.c5.2xlarge	8	16 GiB
ml.c5.4xlarge	16	32 GiB
ml.c5.9xlarge	36	72 GiB
ml.c5.18xlarge	72	144 GiB
ml.c5n.xlarge	4	10.5 GiB
ml.c5n.2xlarge	8	21 GiB
ml.c5n.4xlarge	16	42 GiB
ml.c5n.9xlarge	36	96 GiB
ml.c5n.18xlarge	72	192 GiB

Accelerated Compting Instances	vCPU	Memory
ml.p2.xlarge	4	61 GiB
ml.p2.8xlarge	32	488 GiB
ml.p2.16xlarge	64	732 GiB
ml.p3.2xlarge	8	61 GiB
ml.p3.8xlarge	32	244 GiB
ml.p3.16xlarge	64	488 GiB
ml.g4dn.xlarge	4	16 GiB
ml.g4dn.2xlarge	8	32 GiB
ml.g4dn.4xlarge	16	64 GiB
ml.g4dn.8xlarge	32	128 GiB
ml.g4dn.12xlarge	48	192 GiB

Accelerated Compting Instances	vCPU	Memory
ml.g4dn.16xlarge	64	256 GiB

Note

p3 instances are not available in us-west-1. If your 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.

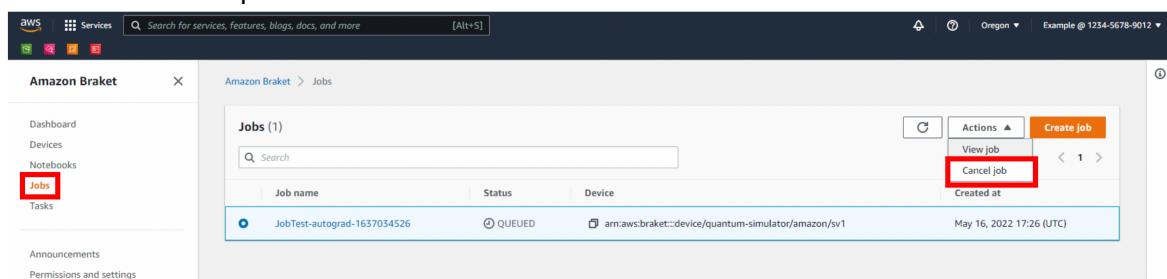
```
from braket.jobs.config import InstanceConfig

job = AwsQuantumJob.create(
    ...
    instance_config=InstanceConfig(
        instanceType="ml.p3.8xlarge",
        volumeSizeInGb=50,
    ),
    ...
),
```

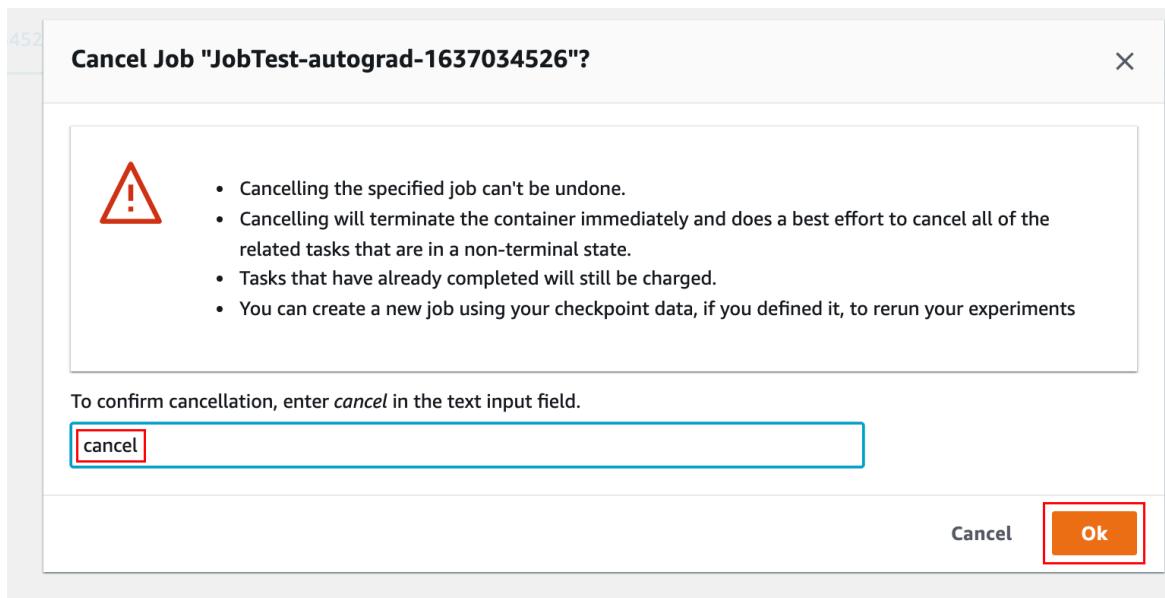
Cancel a job

You may need to cancel a job in a non-terminal state. This can be done either in the console or with code.

To cancel your job in the console, select the job to cancel from the **Jobs** page. Then select **Cancel 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 job using code from the Braket Python SDK, use the `job_arn` to identify the 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 job container immediately and does a best effort to cancel all of the related tasks that are still in a non-terminal state.

Use Amazon Braket Hybrid Jobs to run a QAOA algorithm

In this section you use what you have learned to write an actual hybrid program using PennyLane. You use the algorithm script to address a Quantum Approximate Optimization Algorithm (QAOA) problem. It creates a cost function corresponding to a classical Max Cut optimization problem, specifies a parametrized quantum circuit, and uses a simple 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 it is considered a best practice to provide the problem specification through a dedicated channel in the input data configuration.

```
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,
        )

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

The job script is fairly similar to the previous scripts, except that it also tracks and prints metrics and logs produced from the algorithm script, and downloads the results to your local directory.

```
import boto3
import time

from braket.aws import AwsQuantumJob, AwsSession
from braket.jobs.image_uris import Framework, retrieve_image
from braket.jobs.metrics_datadefinitions import MetricType

device_arn = "arn:aws:braket:::device/quantum-simulator/amazon/sv1"

hyperparameters = {
    # Number of tasks per iteration = 2 * (num_nodes + num_edges) * p + 1
    "p": "2",
    "seed": "1967",
    # Maximum number of simultaneous tasks allowed
    "max_parallel": "10",
    # Number of total optimization iterations, including those from previous checkpoint (if any)
    "num_iterations": "5",
    # Step size / learning rate for gradient descent
    "stepsize": "0.1",
    # Shots for each circuit execution
    "shots": "1000",
}

# Use either the TensorFlow or PyTorch container for PennyLane
region = AwsSession().region
image_uri = retrieve_image(Framework.PL_TENSORFLOW, region)
# image_uri = retrieve_image(Framework.PL_PYTORCH, region)
```

```
start_time = time.time()

job = AwsQuantumJob.create(
    image_uri=image_uri,
    entry_point="qaoa_source.algorithm_script:start_here",
    device=device_arn,
    source_module="qaoa_source",
    hyperparameters=hyperparameters,
)

print(job.arn)
while job.state() not in AwsQuantumJob.TERMINAL_STATES:
    print(job.state())
    time.sleep(10)

end_time = time.time()
print(job.state())

print(end_time - start_time)

print(job.metadata())

print(job.result())

# Metrics may not show up immediately, so wait for 120 seconds
time.sleep(120)
print(job.metrics())
# Print out logs from CloudWatch
print(job.logs())
# Download outputs to local directory
job.download_result()
```

Accelerate your hybrid workloads with embedded simulators from PennyLane

Let's look at how you can use embedded simulators from PennyLane on Amazon Braket Hybrid Jobs to run hybrid workloads. PennyLane's GPU-based simulator, `lightning.gpu`, uses the [Nvidia cuQuantum library](#) to accelerate circuit simulations. The GPU simulator is pre-configured in all of 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 Quantum Approximate Optimization Algorithm 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, let's compare the two PennyLane state vector simulators `lightning.qubit` (which is CPU-based) and `lightning.gpu` (which is GPU-based). You'll need to provide the simulators with some custom gate decompositions in order to compute various gradients.

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

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

```
# Specify device that the 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 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 CPU based simulator (`lightning.qubit`), the GPU **will not** be used. Make sure to use the GPU based simulator if you wish to target the GPU!

First, you can create two 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's 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's \$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 let's make the problem harder, and try solving a Max-Cut problem on a 24-vertex graph, which will translate to 24 qubits. Run the jobs again on the same two instances and compare the cost.

Note

You'll see that the time to run this 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 roughly an hour, while for the p3.2xlarge instance it's roughly 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 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, allowing you to experiment with quantum computing even for problems which 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 leverage 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 GPU-based simulators like PennyLane's `lightning.gpu` to improve the performance over CPU-based simulators.

To create a 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 pre-configured in the Braket PyTorch and TensorFlow containers. The `dist` module tells our algorithm script the total number of GPUs for the training (`world_size`), and 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 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 executes the saving and printing command, the log would be flooded with the repeated information, and the results would be overwriting 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 `m1.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 job creation, Amazon Braket SDK automatically insert the two hyperparameters for you. For Amazon Braket API users, you need to include these two hyperparameters.

With the instance and data parallelism configured, you can now submit your job. There are 8 GPUs in a `m1.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='m1.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 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 job may throw errors, or each GPU may repeatedly process the same data slice, which is inefficient.

Let's 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 `m1.p3.16xlarge` instance used in this example cost \$0.4692 per minute. Without data parallelism, it takes the simulator about 45 minutes to train the model for 1 epoch (i.e., 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!

Build and debug a job with local mode

If 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 job. Instead, you run jobs locally on your Braket Notebook instance or on some other 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 QPU while in local mode.

To use local mode, modify `AwsQuantumJob` to `LocalQuantumJob` wherever it occurs. For instance, to run the example from [Create your first job \(p. 82\)](#), edit the job script 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 [here](#). In addition, not all parameters are supported in local mode.

Bring your own container (BYOC)

Amazon Braket Hybrid Jobs provides three pre-built containers for running code in different environments that are described in the [Define the environment for your algorithm script \(p. 92\)](#) topic. If one of these containers supports your use case, you only have to provide your algorithm script when you create a job. Minor missing dependencies can be added from your algorithm script using pip.

Note

The [Strawberry Fields plugin](#) is not natively included in the Amazon Braket hybrid jobs containers. If you wish to run a hybrid workload using the Xanadu device, you have to BYOC with the Strawberry Fields dependencies included.

If none of these containers support your use case, or if you wish to expand on them, Amazon Braket Hybrid Jobs supports running jobs with your own custom Docker container image. This BYOC capability enables you to run code in an environment that has software installed that you specify. The container

starts running a specified entrypoint script which you may then use to access custom user code and data. For examples, see the [Amazon Braket Hybrid Jobs](#) tutorials.

To run Amazon Braket Hybrid Jobs in your own container, follow these steps:

- 1. Set up prerequisites:** To build and upload your custom container, you must have Docker installed. Amazon Braket notebooks come pre-installed with Docker.
- 2. Create your container:** Create a new directory with a Dockerfile to install and set up the software and the environment your script needs to run. An example file can be downloaded by following the instructions in the [Define the environment for your algorithm script \(p. 92\)](#) topic. You may need to add additional packages to support your container. This file may have more software than you need, in which case you may speed up the build process by removing the unnecessary components. Note that the sagemaker-training component is necessary and should not be removed. You must specify the initial script to run when your container starts with the environment variable ENV SAGEMAKER_PROGRAM your_file.py, where your_file.py is a file that exists in the directory /opt/ml/code inside of your container. When creating a job, the user is able to specify an Amazon S3 location with code and specify an entry point to run. Braket Jobs creates environment variables for your script to find user input code to run. The Amazon S3 location and entry point to the user code are exposed to the initial script through the environment variables AMZN_BRAKET_SCRIPT_S3_URI and AMZN_BRAKET_SCRIPT_ENTRY_POINT respectively. The user input may be compressed and a compression type can be accessed through the variable AMZN_BRAKET_SCRIPT_COMPRESSION_TYPE. It is the responsibility of your script, your_file.py, to use this information to run the user code. An example script which runs user code on user data is braket_container.py in the examples you can download as described in the [Define the environment for your algorithm script \(p. 92\)](#) topic.
- 3. Create an Amazon ECR repository:** The repository you create must be private. If you specify a public image, the Amazon Braket Hybrid Jobs client throws an error when it attempts to validate the image location. Also, the AWS Region of the container image must match the AWS Region where you run the job. To set up your repository, access the [Amazon ECR console](#). Then, select **Create repository**, choose **private** for visibility setting, and give your new repository a name. For a full user guide on setting up a repository, see [Getting started with Amazon ECR using the AWS Management Console](#). Please NOTE that the standard role created for Jobs only has permissions to download container images beginning with the prefix "amazon-braket-" and that you will need to add inline permissions to your job role if you wish to use container images with a different name (see current policy restrictions [here](#)). If you plan to run jobs from an account that is different than the account you used to create the repository, you must allow the account used to run the jobs to pull the images you have created from the account used to create them. The actions that you must allow on your repository are ecr:DescribeImages and ecr:BatchGetImage. To add these permissions to the image from the Amazon ECR console, select your image and choose **permissions** in the menu. You can add a rule to allow these two actions for the account you wish to allow to run jobs with your container.
- 4. Build the image, and push it to the repository:** You must authenticate Docker to upload to Amazon ECR. You can retrieve an authentication token and authenticate your Docker client to your registry with the AWS CLI using the following code.

```
aws ecr get-login-password --region ${your_region} | docker login --username AWS --password-stdin ${aws_account_id}.dkr.ecr.${your_region}.amazonaws.com
```

Now you should be able to build your image and push it to your repository.

```
cd ${your_docker_directory}
docker build -t ${your_job_container} .
docker tag ${your_job_container}:latest ${aws_account_id}.dkr.ecr.
${your_region}.amazonaws.com/${your_job_container}:latest
docker push ${aws_account_id}.dkr.ecr.${your_region}.amazonaws.com/
${your_job_container}:latest
```

To create a job with your own container, call `AwsQuantumJob.create` with the argument `image_uri` specified. You can either use a QPU, a managed simulator or simply run your code locally on the classical processor available with Amazon Braket Hybrid jobs. To run your code on the classical processor, specify the `instanceType` and the `instanceCount` you wish to use by updating the `InstanceConfig`. Note that if you specify an `instance_count > 1`, you will need to make sure that your code has the ability to 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

The device arn allows you to track the simulator you used as 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 are not using the Amazon Braket SDK to create tasks, you should pass the value of the environmental variable `AMZN_BRAKET_JOB_TOKEN` to the `jobToken` parameter in the `CreateQuantumTask` request. Failing to do so causes the tasks to not get priority and to be billed as regular stand-alone tasks.

Configure the default bucket in AwsSession

Providing your own `AwsSession` gives you greater flexibility, for example, in the location of your default bucket. By default, an `AwsSession` has a default bucket location of `f"amazon-braket-{id}-{region}"`. But you can override that default when creating an `AwsSession`. Users can optionally pass in an `AwsSession` object into `AwsQuantumJob.create` with the parameter name `aws_session`, as shown in the following code example.

```
aws_session = AwsSession(default_bucket="other-default-bucket")  
  
# then you can use that AwsSession when creating a job  
job = AwsQuantumJob.create(  
    ...  
    aws_session=aws_session  
)
```

Interact with 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 you interact with Amazon Braket Hybrid Jobs using the [Amazon Braket Python SDK](#). It offers convenient defaults and protections that ensure your job runs successfully.

This topic covers the basics of using the API, but if you choose this path, be prepared for a lot of iteration and work to get your job to run.

To use the API, you must access your account with a role that has the following permissions:

1. AmazonBraketFullAccess managed policy
2. The following inline policy

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": "s3:CreateBucket",
            "Resource": "arn:aws:s3:::amazon-braket-*",
            "Effect": "Allow"
        },
        {
            "Action": "s3:GetObject",
            "Resource": "arn:aws:s3:::braket-*",
            "Effect": "Allow"
        },
        {
            "Action": [
                "s3:GetObject",
                "s3:PutObject",
                "s3>ListBucket"
            ],
            "Resource": "arn:aws:s3:::amazon-braket-*",
            "Effect": "Allow"
        },
        {
            "Action": [
                "logs>CreateLogStream",
                "logs>DescribeLogStreams",
                "logs>PutLogEvents",
                "logs>CreateLogGroup",
                "logs>GetQueryResults",
                "logs>GetLogEvents",
                "logs>StartQuery",
                "logs>StopQuery"
            ],
            "Resource": "arn:aws:logs:*::*:log-group:*",
            "Effect": "Allow"
        },
        {
            "Action": "iam>PassRole",
            "Resource": "arn:aws:iam::*:role/*",
            "Effect": "Allow"
        }
    ]
}
```

The `CreateJob` API requires that you specify all the required parameters for the job. To use Python, compress your algorithm script files, such as an `input.tar.gz` file, and run the following script. Update the **bold** parts of the code to match your account information and entry point that specify the path, file, and method where your job starts.

```
from braket.aws import AwsDevice, AwsSession
import boto3
from botocore.session import get_session
from datetime import datetime

s3_client = boto3.client("s3")

creds = get_session().get_credentials()
```

```

session = boto3.Session(aws_access_key_id=creds.access_key,
    aws_secret_access_key=creds.secret_key, # grab a session with the right region
    aws_session_token=creds.token, region_name="us-west-2")
client = session.client('braket', endpoint_url = "https://kpg9e8yzsg.execute-api.us-
west-2.amazonaws.com/V4") # set gamma endpoint for Braket client

project_name = "job-test"
job_name = project_name + "-" + datetime.strftime(datetime.now(), "%Y%m%d%H%M%S")
bucket = "*amazon-braket-Your-Bucket-Name*"
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::*AccountID*:role/AmazonBraketJobsPreviewRole (https://us-west-2.console.aws.amazon.com/iam/home?#/roles/AmazonBraketJobsPreviewRole)",
    algorithmSpecification={
        "scriptModeConfig": {
            "entryPoint": "*alpha_test_job.fancy_module.my_file:start_here*",
            "containerImage": {"uri": "111122223333.dkr.ecr.us-west-2.amazonaws.com/
sagemaker-byoc-test:latest"}
            "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/Aspen-M-2"
            ]
        }
    },
    hyperParameters={
        "user": "*AccountRole*",
        "bucket": f"s3://{bucket}"
    }
)

```

```
    },
    stoppingCondition={
        "maxRuntimeInSeconds": 1200,
        "maximumTaskLimit": 10
    },
)
```

Once you create your job, you can access the job details through the `GetJob` API or the console. To get the 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 job, call the `CancelJob` API with the `JobArn`.

```
cancelJob = client.cancel_job(jobArn=job["jobArn"])
```

You can specify checkpoints as part of the `createJob` 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`.

Use PennyLane with Amazon Braket

Hybrid algorithms are algorithms that contain classical and quantum instructions. The classical instructions are executed on classical hardware (an EC2 instance or your laptop), and the quantum instructions are executed either on a simulator or on a quantum computer. We recommend that you run hybrid algorithms using the hybrid jobs feature. For more guidance, see [When to use Amazon Braket Jobs \(p. 80\)](#).

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 much easier and 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 ([QPE 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.

Amazon Braket with PennyLane

Amazon Braket provides support for [PennyLane](#), an open-source software framework built around the concept of *quantum differentiable programming*. This framework allows you 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 fast, easy 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 this command:

```
import pennylane as qml
```

Tutorial notebooks help you get started quickly. Alternatively, you can use PennyLane on Amazon Braket from any 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 this 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
s3 = ("my-bucket", "my-prefix")
sv1 = qml.device("braket.aws.qubit", device_arn="arn:aws:braket:::device/quantum-simulator/
amazon/sv1", s3_destination_folder=s3, 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)
```

You can find information and PennyLane tutorial examples in the [Amazon Braket examples repository](#).

The Amazon Braket PennyLane plugin enables you to switch between Amazon Braket QPU and 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. It can be installed from the [PennyLane Plugin GitHub repository](#).

To find out more, visit the [PennyLane Documentation](#).

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 any of these other [example notebooks](#).

Hybrid algorithms with embedded PennyLane simulators

Amazon Braket Hybrid Jobs now comes with five high performance CPU- and GPU-based simulators from [PennyLane](#). This family of simulators can be embedded directly within your jobs container and includes the fast state-vector lightning.qubit simulator, the lightning.gpu simulator accelerated using NVIDIA's [cuQuantum library](#), and others. These 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 managed simulator such as SV1, or directly using the embedded simulator from PennyLane.

The embedded simulator is already available with the Hybrid jobs container, you simply need to specify the device (e.g device="local:pennylane/lightning.gpu") in the Braket SDK as shown below. To use the lightning.gpu simulator, you also need to specify a GPU instance in the InstanceConfig as shown in the code snippet below:

```
job = AwsQuantumJob.create(  
    device="local:pennylane/lightning.gpu",  
    source_module="algorithm_script.py",  
    instance_config=InstanceConfig(instanceType="ml.p3.8xlarge")  
)
```

and refer to the device in your algorithm code using the environment variable AMZN_BRAKET_DEVICE_ARN:

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

Refer to the [example notebook](#) to get started with using a PennyLane embedded simulator with hybrid jobs.

Security in Amazon Braket

This chapter helps you understand how to apply the shared responsibility model when using Amazon Braket. It shows you how to configure Amazon 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 Amazon Braket resources.

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from a data center and network architecture that is built to meet the requirements of the most security-sensitive organizations. You are responsible for other factors, including the sensitivity of your data, your company's requirements, and applicable laws and regulations.

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

For more information about data privacy, see the [Data Privacy FAQ](#).

For data protection purposes, we recommend that you protect AWS account credentials and set up individual user accounts with 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 recommend TLS 1.2 or later.
- Set up API and user activity logging with AWS CloudTrail.
- 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 personal data that is stored in Amazon S3.
- If you require FIPS 140-2 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 \(FIPS\) 140-2](#).

We strongly recommend that you never put sensitive identifying information, such as your customer account numbers, into free-form fields such as a **Name** field. This includes when you work with Braket or other AWS services using the console, API, CLI, or AWS SDKs. Any data that you enter into Braket or other services might get picked up for inclusion in diagnostic logs. When you provide a URL to an external server, 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 task IDs and other metadata associated with your 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 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 IAM users and roles. You can grant (or deny) the required permissions to any IAM user or role in your account. To do so, attach the appropriate Amazon Braket policy to that user or role in the account, as given in this chapter.

As a prerequisite, you must [enable Amazon Braket](#). To enable Amazon 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 S3 buckets.

In this section:

- [Amazon Braket resources \(p. 115\)](#)
- [Notebooks and roles \(p. 115\)](#)
- [About the AmazonBraketFullAccess policy \(p. 116\)](#)
- [About the AmazonBraketJobsExecutionPolicy policy \(p. 119\)](#)
- [Restrict user access to certain devices \(p. 121\)](#)
- [Amazon Braket updates to AWS managed policies \(p. 122\)](#)

Amazon Braket resources

Amazon Braket creates one type of resource, which is the *quantum-task* resource. Here is the form of the ARN for that resource type:

- **Resource Name:** AWS::Service::Braket
- **ARN Regex:** arn:\${Partition}:braket:\${Region}:\${Account}:quantum-task/\${RandomId}

Notebooks and roles

Notebooks are another type of resource that Amazon Braket utilizes on your behalf. A notebook is an Amazon SageMaker resource, which Braket is able to share. The notebooks require a specific IAM role to function: a 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.

```
{  
    "Version": "2012-10-17",
```

```
"Statement": [
    {
        "Effect": "Allow",
        "Action": "iam:CreateRole",
        "Resource": "arn:aws:iam::*:role/service-role/
AmazonBraketServiceSageMakerNotebookRole-*"
    },
    {
        "Effect": "Allow",
        "Action": "iam:CreatePolicy",
        "Resource": "arn:aws:iam::*:policy/service-role/
AmazonBraketServiceSageMakerNotebookAccess-*"
    },
    {
        "Effect": "Allow",
        "Action": "iam:AttachRolePolicy",
        "Resource": "arn:aws:iam::*:role/service-role/
AmazonBraketServiceSageMakerNotebookRole-*",
        "Condition": {
            "StringLike": {
                "iam:PolicyARN": [
                    "arn:aws:iam::aws:policy/AmazonBraketFullAccess",
                    "arn:aws:iam::*:policy/service-role/
AmazonBraketServiceSageMakerNotebookAccess-*"
                ]
            }
        }
    }
]
```

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.

About the AmazonBraketFullAccess policy

The **AmazonBraketFullAccess** policy grants permissions for Amazon Braket operations, including permissions for these tasks:

- **Amazon Elastic Container Registry** – to read and download container images to be used for 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 as well as 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. Maintain usage log files for your account** – to create, store, and view logging information about Amazon Braket usage in your account. Query metrics on jobs log groups. Encompass the proper Braket path and allowing 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 to put objects into and get objects from any bucket in your account whose name begins with *amazon-braket-*. These permissions are required for Amazon Braket to put files containing results from processed 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 Notebook** – to create and manage SageMaker Notebook instances scoped to the resource from "arn:aws:sagemaker:::notebook-instance/amazon-braket-"

Policy contents

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:PutObject",
                "s3>ListBucket",
                "s3>CreateBucket",
                "s3:PutBucketPublicAccessBlock",
                "s3:PutBucketPolicy"
            ],
            "Resource": "arn:aws:s3:::amazon-braket-*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3>ListAllMyBuckets"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "ecr:GetDownloadUrlForLayer",
                "ecr:BatchGetImage",
                "ecr:BatchCheckLayerAvailability"
            ],
            "Resource": "arn:aws:ecr:*::repository/amazon-braket*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "ecr:GetAuthorizationToken"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "logs:Describe*",
                "logs:Get*",
                "logs>List*",
                "logs:StartQuery",
                "logs:StopQuery",
                "logs:TestMetricFilter",
                "logs:FilterLogEvents"
            ],
            "Resource": "arn:aws:logs::*:log-group:/aws;braket*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "iam>ListRoles",
                "iam>ListRolePolicies",
                "iam:GetRole",
                "iam:GetUser"
            ],
            "Resource": "*"
        }
    ]
}
```

```
        "iam:GetRolePolicy",
        "iam>ListAttachedRolePolicies"
    ],
    "Resource": "*"
},
{
    "Effect": "Allow",
    "Action": [
        "sagemaker>ListNotebookInstances"
    ],
    "Resource": "*"
},
{
    "Effect": "Allow",
    "Action": [
        "sagemaker>CreatePresignedNotebookInstanceUrl",
        "sagemaker>CreateNotebookInstance",
        "sagemaker>DeleteNotebookInstance",
        "sagemaker>DescribeNotebookInstance",
        "sagemaker>StartNotebookInstance",
        "sagemaker>StopNotebookInstance",
        "sagemaker>UpdateNotebookInstance",
        "sagemaker>ListTags",
        "sagemaker>AddTags",
        "sagemaker>DeleteTags"
    ],
    "Resource": "arn:aws:sagemaker:*:*:notebook-instance/amazon-braket-*"
},
{
    "Effect": "Allow",
    "Action": [
        "sagemaker>DescribeNotebookInstanceLifecycleConfig",
        "sagemaker>CreateNotebookInstanceLifecycleConfig",
        "sagemaker>DeleteNotebookInstanceLifecycleConfig",
        "sagemaker>ListNotebookInstanceLifecycleConfigs",
        "sagemaker>UpdateNotebookInstanceLifecycleConfig"
    ],
    "Resource": "arn:aws:sagemaker:*:*:notebook-instance-lifecycle-config/amazon-
braket-*"
},
{
    "Effect": "Allow",
    "Action": "braket:*",
    "Resource": "*"
},
{
    "Effect": "Allow",
    "Action": "iam>CreateServiceLinkedRole",
    "Resource": "arn:aws:iam::*:role/aws-service-role/braket.amazonaws.com/
AWSServiceRoleForAmazonBraket*",
    "Condition": {
        "StringEquals": {
            "iam:AWSServiceName": "braket.amazonaws.com"
        }
    }
},
{
    "Effect": "Allow",
    "Action": [
        "iam>PassRole"
    ],
    "Resource": "arn:aws:iam::*:role/service-role/
AmazonBraketServiceSageMakerNotebookRole*",
    "Condition": {
        "StringLike": {
            "iam:PassedToService": [

```

```
        "sagemaker.amazonaws.com"
    ]
}
},
{
    "Effect": "Allow",
    "Action": [
        "iam:PassRole"
    ],
    "Resource": "arn:aws:iam::*:role/service-role/AmazonBraketJobsExecutionRole*",
    "Condition": {
        "StringLike": {
            "iam:PassedToService": [
                "braket.amazonaws.com"
            ]
        }
    }
},
{
    "Effect": "Allow",
    "Action": [
        "logs:GetQueryResults"
    ],
    "Resource": [
        "arn:aws:logs:*:log-group:)"
    ]
},
{
    "Effect": "Allow",
    "Action": [
        "logs:PutLogEvents",
        "logs>CreateLogStream",
        "logs>CreateLogGroup"
    ],
    "Resource": "arn:aws:logs:*log-group:/aws;braket*"
},
{
    "Effect": "Allow",
    "Action": "cloudwatch:PutMetricData",
    "Resource": "*",
    "Condition": {
        "StringEquals": {
            "cloudwatch:namespace": "/aws;braket"
        }
    }
}
]
```

About the AmazonBraketJobsExecutionPolicy policy

The **AmazonBraketJobsExecutionPolicy** policy grants permissions for execution roles used in Amazon Braket Hybrid Jobs

- **Amazon Elastic Container Registry** - permissions to read and download container images to be used for 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. Maintain usage log files for your account**
 - Create, store, and view logging information about Amazon Braket usage in your account. Query metrics on jobs log groups. Encompass the proper Braket path and allowing 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 Amazon Braket to put files containing results from processed 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*`.

```
"Version": "2012-10-17",
"Statement": [
{
  "Effect": "Allow",
  "Action": [
    "s3:GetObject",
    "s3:PutObject",
    "s3>ListBucket",
    "s3>CreateBucket",
    "s3:PutBucketPublicAccessBlock",
    "s3:PutBucketPolicy"
  ],
  "Resource": "arn:aws:s3:::amazon-braket-*"
},
{
  "Effect": "Allow",
  "Action": [
    "ecr:GetDownloadUrlForLayer",
    "ecr:BatchGetImage",
    "ecr:BatchCheckLayerAvailability"
  ],
  "Resource": "arn:aws:ecr:*::repository/amazon-braket*"
},
{
  "Effect": "Allow",
  "Action": [
    "ecr:GetAuthorizationToken"
  ],
  "Resource": "*"
},
{
  "Effect": "Allow",
  "Action": [
    "braket:CancelJob",
    "braket:CancelQuantumTask",
    "braket>CreateJob",
    "braket>CreateQuantumTask",
    "braket:GetDevice",
    "braket:GetJob",
    "braket:GetQuantumTask",
    "braket:SearchDevices",
    "braket:SearchJobs",
    "braket:SearchQuantumTasks",
    "braket>ListTagsForResource",
    "braket:TagResource",
    "braket:UntagResource"
  ],
  "Resource": "*"
},
{
  "Effect": "Allow",
  "Action": [
    "iam:PassRole"
  ],
  "Resource": "arn:aws:iam::*:role/service-role/AmazonBraketJobsExecutionRole*",
  "Condition": {
```

```

    "StringLike": {
      "iam:PassedToService": [
        "braket.amazonaws.com"
      ]
    }
  },
  {
    "Effect": "Allow",
    "Action": [
      "iam>ListRoles"
    ],
    "Resource": "arn:aws:iam::*:role/*"
  },
  {
    "Effect": "Allow",
    "Action": [
      "logs:GetQueryResults"
    ],
    "Resource": [
      "arn:aws:logs:*:log-group:)"
    ],
  },
  {
    "Effect": "Allow",
    "Action": [
      "logs>PutLogEvents",
      "logs>CreateLogStream",
      "logs>CreateLogGroup",
      "logs>GetLogEvents",
      "logs>DescribeLogStreams",
      "logs>StartQuery",
      "logs>StopQuery"
    ],
    "Resource": "arn:aws:logs:*log-group:/aws/braket"
  },
  {
    "Effect": "Allow",
    "Action": "cloudwatch>PutMetricData",
    "Resource": "*",
    "Condition": {
      "StringEquals": {
        "cloudwatch:namespace": "/aws/braket"
      }
    }
  }
]
}

```

Restrict user access to certain devices

To restrict access for certain users to certain Amazon Braket devices, you can add a *deny permissions* policy to a specific IAM role.

The following example restricts access to all QPUs for the AWS account 012345678901.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Action": [
        "braket>CreateQuantumTask",

```

```

        "braket:CancelQuantumTask",
        "braket:GetQuantumTask",
        "braket:SearchQuantumTasks",
        "braket:GetDevice",
        "braket:SearchDevices"
    ],
    "Resource": [
        "arn:aws:braket:*::device/qpu/*"
    ]
}
]
}
}

```

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 Amazon 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>:<account id>:device/qpu/<provider>/<device_id>`
- `arn:aws:braket:<region>:<account id>: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-2:012345678901:device/qpu/*`
- Equivalently, to select all QPUs in the us-west-2 region ONLY, because devices are a service resource, not a customer resource: `arn:aws:braket:us-west-2:*:device/qpu/*``
- To restrict access to all managed simulator devices: `arn:aws:braket:*:012345678901:device/quantum-simulator/*``
- To restrict access to the IonQ device in us-east-1 region: `arn:aws:braket:us-east-1:012345678901:device/ionq/ionQdevice`
- To restrict access to devices from a certain provider, for example, to D-Wave QPU devices: `arn:aws:braket:*:012345678901:device/qpu/d-wave/*``
- To restrict access to TN1 device: `arn:aws:braket:*:012345678901:device/quantum-simulator/amazon/tn1``

Amazon Braket updates to AWS managed policies

The following table provides details about updates to AWS managed policies for Amazon Braket since this service began tracking these changes.

Change	Description	Date
AmazonBraketFullAccess (p. 116) - 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
AmazonBraketFullAccess (p. 116) - Full access policy for Amazon Braket	Amazon Braket adjusted iam:PassRole permissions for AmazonBraketFullAccess to include the service-role/ path.	November 29, 2021

Change	Description	Date
AmazonBraketJobsExecutionPolicy (p. 119) - Jobs execution policy for Amazon Braket Hybrid Jobs	Amazon Braket updated the jobs execution role ARN to include the service-role/ path.	November 29, 2021
Amazon Braket started tracking changes	Amazon Braket started tracking changes for its AWS managed policies.	November 29, 2021

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.

Service-linked role permissions for Amazon Braket

Amazon Braket uses the **AWSServiceRoleForAmazonBraket** service-linked role that trusts the `braket.amazonaws.com` entity to assume the role.

You must configure permissions to allow an IAM entity (such as a user, group, or role) to create, edit, or delete a service-linked role. For more information, see [Service-Linked Role Permissions](#) in the *IAM User Guide*.

The service-linked role in Amazon Braket is granted the following permissions by default:

- **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.

The following policy is attached to the **AWSServiceRoleForAmazonBraket** service-linked role:

```
{"Version": "2012-10-17",
 "Statement": [
     {"Effect": "Allow",
      "Action": [
          "s3:GetObject",
          "s3:PutObject",
          "s3>ListBucket"
      ],
      ...
  }
]
```

```
        "Resource": "arn:aws:s3:::amazon-braket*"
    },
    {"Effect": "Allow",
        "Action": [
            "logs:Describe*",
            "logs:Get*",
            "logs>List*",
            "logs:StartQuery",
            "logs:StopQuery",
            "logs:TestMetricFilter",
            "logs:FilterLogEvents"
        ],
        "Resource": "arn:aws:logs:*::log-group:/aws;braket/*"
    },
    {"Effect": "Allow",
        "Action": "braket:*",
        "Resource": "*"
    },
    {"Effect": "Allow",
        "Action": "iam>CreateServiceLinkedRole",
        "Resource": "arn:aws:iam::*:role/aws-service-role/braket.amazonaws.com/
AWSServiceRoleForAmazonBraket*",
        "Condition": {"StringEquals": {"iam:AWSServiceName": "braket.amazonaws.com"
        }
    }
}
]
```

Resilience in Amazon Braket

The AWS global infrastructure is built around AWS Regions and Availability Zones.

Each Region provides multiple *availability zones* that are physically separated and isolated. These availability zones (AZs) are connected through low-latency, high-throughput, and highly redundant networking. As a result, availability zones are more highly available, fault tolerant, and scalable than traditional single- or multiple-datacenter infrastructures.

You can design and operate applications and databases that fail over between AZs automatically, without interruption.

For more information about AWS Regions and availability zones, see [AWS Global Infrastructure](#).

Compliance validation for Amazon Braket

Your compliance responsibility when using Amazon Braket 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 and Compliance Quick Start Guides](#) – These deployment guides discuss architectural considerations and provide steps for deploying security- and compliance-focused baseline environments on AWS.
- [AWS Compliance Resources](#) – This collection of workbooks and guides might apply to your industry and location.

For a list of AWS services in scope of specific compliance programs, see [AWS Services in Scope by Compliance Program](#). For general information, see [AWS Compliance Programs](#).

Infrastructure Security in Amazon Braket

As a managed service, Amazon Braket is protected by the AWS global network security procedures that are described in the [Amazon Web Services: Overview of Security Processes](#) whitepaper.

For access to Amazon Braket through the network, you make calls to published AWS APIs. Clients must support Transport Layer Security (TLS) 1.2 or later. Clients also must support cipher suites with perfect forward secrecy (PFS) such as Ephemeral Diffie-Hellman (DHE) or Elliptic Curve Ephemeral Diffie-Hellman (ECDHE). Most modern systems such as Java 7 and later support these modes.

Additionally, requests must be signed 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.

Security of Amazon Braket Hardware Providers

QPUs on Amazon Braket are hosted by third-party hardware providers. When you run your 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.

Troubleshoot

Solve common problems you might find when working with Amazon Braket.

Topics

In this section:

- [Access denied exception \(p. 126\)](#)
- [A task is failing creation \(p. 126\)](#)
- [An SDK feature does not work \(p. 126\)](#)
- [A job fails due to an exceeded quota \(p. 127\)](#)
- [Something stopped working in your notebook instance \(p. 127\)](#)
- [Troubleshoot OpenQASM \(p. 128\)](#)

Access denied exception

If you receive an **AccessDeniedException**, as shown in the following image, when enabling or using Braket, then you are most likely attempting to enable or use Braket in a region where your restricted role does not have access.



In such cases, you should 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.

A task is failing creation

If you receive an error along the lines of "An error occurred (ValidationException) when calling the CreateQuantumTask operation: Caller doesn't have access to amazon-braket-..." make sure you are referring to an existing s3_folder since we do not auto create new Amazon S3 buckets and prefixes for you.

If you are accessing the API directly and getting an error like "Failed to create quantum task: Caller doesn't have access to s3://MY_BUCKET" make sure you are not including s3:// in the Amazon S3 bucket path.

An SDK feature does not work

Make sure your SDK (and schemas) are up-to-date. From the notebook or your python editor run

```
!pip install --upgrade amazon-braket-sdk
```

Make sure your SDK and schemas are up-to-date. To update the SDK from the notebook or your python editor run the following:

```
pip install --upgrade amazon-braket-schemas
```

If you are accessing Amazon Braket from your own client make sure your AWS region is set to one supported by Amazon Braket.

A job fails due to an exceeded quota

My job fails with ServiceQuotaExceededException

A job running tasks against the Amazon Braket simulators can fail to be created if you exceed the concurrent task limit for the simulator device you are targeting. The service limits are explained in the [Quotas \(p. 22\)](#) topic. If you are running multiple jobs from your account, which run concurrent tasks against a simulator device, you could encounter this error.

To see the number of concurrent 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 tasks against a device using Amazon CloudWatch metrics: [Braket > By Device](#).

To avoid running into these errors, you can either:

1. Request a service quota increase for the number of concurrent tasks for the simulator device. This is only applicable to the SV1 device.
2. Handle `ServiceQuotaExceeded` exceptions in your code and retry.

Something stopped working in your 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.

Troubleshoot OpenQASM

This section provides troubleshooting pointers that might be useful when encountering errors using OpenQASM 3.0.

In this section:

- [Include statement error \(p. 128\)](#)
- [Non-contiguous qubits error \(p. 128\)](#)
- [Mixing physical qubits with virtual qubits error \(p. 129\)](#)
- [Requesting result types and measuring qubits in the same program error \(p. 129\)](#)
- [Classical and qubit register limits exceeded error \(p. 129\)](#)
- [Box not preceded by a verbatim pragma error \(p. 129\)](#)
- [Verbatim boxes missing native gates error \(p. 130\)](#)
- [Verbatim boxes missing physical qubits error \(p. 130\)](#)
- [The verbatim pragma is missing "braket" error \(p. 130\)](#)
- [Single qubits cannot be indexed error \(p. 130\)](#)
- [The physical qubits in a two qubit gate are not connected error \(p. 131\)](#)
- [GetDevice does not return OpenQASM results error \(p. 131\)](#)
- [Local simulator support warning \(p. 132\)](#)

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 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

GetDevice does not return OpenQASM results error

If you do not see OpenQASM results in the GetDevice response when using a Braket SDK, you may need to set `AWS_EXECUTION_ENV` environment variable to configure user-agent. See the code examples provided below for how to do this for the Go and Java SDKs.

To set `AWS_EXECUTION_ENV` environment variable to configure user-agent when using the AWS CLI:

```
% export AWS_EXECUTION_ENV="aws-cli BraketSchemas/1.8.0"  
# Or for single execution  
% AWS_EXECUTION_ENV="aws-cli BraketSchemas/1.8.0" aws braket <cmd> [options]
```

To set `AWS_EXECUTION_ENV` environment variable to configure user-agent when using Boto3:

```
import boto3  
import botocore  
  
client = boto3.client("braket",  
    config=botocore.client.Config(user_agent_extra="BraketSchemas/1.8.0"))
```

To set `AWS_EXECUTION_ENV` environment variable to configure user-agent when using the JavaScript/TypeScript (SDK v2):

```
import Braket from 'aws-sdk/clients;braket';  
const client = new Braket({ region: 'us-west-2', credentials: AWS_CREDENTIALS,  
    customUserAgent: 'BraketSchemas/1.8.0' });
```

To set `AWS_EXECUTION_ENV` environment variable to configure user-agent when using the JavaScript/TypeScript (SDK v3):

```
import { Braket } from '@aws-sdk/client-braket';  
const client = new Braket({ region: 'us-west-2', credentials: AWS_CREDENTIALS,  
    customUserAgent: 'BraketSchemas/1.8.0' });
```

To set `AWS_EXECUTION_ENV` environment variable to configure user-agent when using the Go SDK:

```
os.Getenv("AWS_EXECUTION_ENV", "BraketGo BraketSchemas/1.8.0")  
mySession := session.Must(session.NewSession())
```

```
svc := braket.New(mySession)
```

To set AWS_EXECUTION_ENV environment variable to configure user-agent when using the Java SDK:

```
ClientConfiguration config = new ClientConfiguration();
config.setUserAgentSuffix("BraketSchemas/1.8.0");
BraketClient braketClient =
    BraketClientBuilder.standard().withClientConfiguration(config).build();
```

Local simulator support warning

The LocalSimulator supports advanced features in OpenQASM that may not be available on QPUs or managed 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 managed simulators.

For more information on supported OpenQASM features, [click here \(p. 56\)](#).

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 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 [Interface VPC endpoints \(AWS PrivateLink\)](#) in the *Amazon VPC User Guide*.

Considerations for Amazon Braket VPC endpoints

Before you set up an interface VPC endpoint for Braket, ensure that you review [Interface endpoint properties and limitations](#) 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 [Controlling access to services with VPC endpoints](#) 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 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 [Creating an interface 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`

Note: If you enable private DNS for the endpoint, you can make API requests to Braket using its default DNS name for the Region, for example, `braket.us-east-1.amazonaws.com`.

For more information, see [Accessing a service through an interface endpoint](#) in the [Amazon VPC User Guide](#).

Step 3: Connect and run Braket 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 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**. 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*.

More 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 [Creating an Interface 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 [Controlling access to services with VPC endpoints](#) 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](#)
- [Controlling Access to Services with VPC Endpoints](#)

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 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 task or notebook upon creation, and then manage associated tags through the console, AWS CLI, or API.

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](#) in the [IAM User Guide](#).

More about AWS and tags

- For general information on tagging, including naming and usage conventions, see [Tagging AWS Resources](#) in the [AWS General Reference](#).
- For information about restrictions on tagging, see [Tag naming limits and requirements](#) in the [AWS General Reference](#).
- For best practices and tagging strategies, see [Tagging best practices](#) and [AWS Tagging Strategies](#).
- 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.

Supported resources 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 resources. For more information, see [Notebook Instance Metadata](#) in the SageMaker documentation.

Tag 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](#).

Add 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 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.

View 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`**

Edit tags

You can edit tags by using the console to navigate to the 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](#)

Remove tags

You can remove tags from a resource by specifying the keys to remove, by using the console to navigate to the task or notebook resource, or when calling the [UntagResource](#) operation.

- **AWS API:** [UntagResource](#)

Example of CLI tagging in Amazon Braket

If you're working with the AWS CLI, here is an example command showing how to create a tag that applies to a quantum task you create for the SV1 simulator with parameter settings of the Rigetti QPU. Notice that the tag is specified at the end of the example command. In this case, **Key** is given the value **state** and **Value** is given the value Washington.

```
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"}
```

Tagging with the Amazon Braket API

- If you're using the Amazon Braket API to set up tags on a resource, call the [TagResourceAPI](#).

```
aws braket tag-resource --resource-arn $YOUR_TASK_ARN --tags {"city":  
    "Seattle"}
```

- To remove tags from a resource, call the [UntagResourceAPI](#).

```
aws braket list-tags-for-resource --resource-arn $YOUR_TASK_ARN
```

- To list all tags that are attached to a particular resource, call the [ListTagsForResourceAPI](#).

```
aws braket tag-resource --resource-arn $YOUR_TASK_ARN --tag-keys "[\"city\", \"state\"]"
```

Monitoring Amazon Braket with Amazon 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](#).

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:

Task Metrics

Metrics are available if tasks exist. They are displayed under **AWS/Braket/By Device** in the CloudWatch console.

Metric	Description
Count	Number of tasks.
Latency	This metric is emitted when a task has completed. It represents the total time from task initialization to completion.

Dimensions for Task Metrics

The task metrics are published with a dimension based on the `deviceArn` parameter, which has the form `arn:aws:braket:::device/xxx`.

Supported Devices

For a list of supported devices and device ARNs, see [Braket devices](#).

Note

You can view the CloudWatch log streams for Amazon Braket notebooks by navigating to the **Notebook detail** page on the Amazon SageMaker console. Additional Amazon Braket notebook settings are available through the [SageMaker console](#).

Events and automated actions for Amazon Braket with Amazon EventBridge

Amazon EventBridge monitors status change events in Amazon Braket tasks. Events from Amazon Braket are delivered to EventBridge, almost in real time. You can write simple 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 task changes

Amazon Braket guarantees delivery of task status change events. These events are delivered at least once, but possibly out of order.

For more information, see the [Events and Event Patterns in EventBridge](#).

In this section:

- [Monitor task status with EventBridge \(p. 141\)](#)
- [Example Amazon Braket EventBridge event \(p. 142\)](#)

Monitor 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 task. For example, you can create a rule that sends you an email message each time the status of a 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 at <https://console.aws.amazon.com/events/>.
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 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 Task Status Change event, see [Events and Event Patterns in EventBridge](#).

The following attributes appear in the JSON "detail" field.

- quantumTaskArn** (str): The task for which this event was generated.
- status** (Optional[str]): The status to which the task transitioned.
- deviceArn** (str): The device specified by the user for which this 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 task creation time as an ISO-8601 string.
- endedAt** (Optional[str]): The time at which the task reached a terminal state. This field is present only when the task has transitioned to a terminal state.

The following JSON code shows an example of an Amazon Braket 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"  
    }  
}
```

Amazon Braket API logging 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 API 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](#).

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 root or AWS Identity and Access Management (IAM) user credentials.
- 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": "2020-08-07T00:56:57Z"  
            }  
        }  
    },  
    "eventTime": "2020-08-07T01:00:08Z",  
    "eventSource": "braket.amazonaws.com",  
    "eventName": "GetDevice",  
    "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": {  
        "deviceArn": "foobar"  
    },  
    "responseElements": null,  
    "requestID": "20e8000c-29b8-4137-9cbc-af77d1dd12f7",  
    "eventID": "4a2fdb22-a73d-414a-b30f-c0797c088f7c",  
    "readOnly": true,  
    "eventType": "AwsApiCall",  
    "recipientAccountId": "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"  
}
```

API & SDK Reference Guide for Amazon Braket

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 Go](#)
- [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](#)

Document history

The following table describes the documentation for this release of Amazon Braket.

- **API version:** April 28, 2022
- **Latest API Reference update:** April 28, 2022
- **Latest documentation update:** September 13, 2022

Change	Description	Date
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 screenshots 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 suport 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 simplification 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-based 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 <code>servicerole</code> / path. For information on policy updates, see the Amazon Braket updates to AWS managed policies (p. 122) 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 <code>braket_dm</code>	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 <i>quantum-task</i> 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