
Amazon Braket

Developer Guide

Amazon Braket: Developer Guide

Copyright ©

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	1
How it works	4
Amazon Braket task flow	4
Third-party data processing	5
Supported devices	6
IonQ	7
Rigetti	7
D-Wave	7
Local Simulator	7
State Vector Simulator (SV1)	7
Regions and endpoints	9
Pricing	10
Quotas	11
Additional quotas and limits	13
When will my task run?	14
Status change notifications via Email or SMS	14
QPU availability windows and status	14
Enable Amazon Braket	15
Prerequisites	15
Steps to enable Amazon Braket	15
Common causes of failure when enabling Amazon Braket	15
Get started	17
Create an Amazon Braket Notebook instance	17
Run your first circuit using the Braket Python SDK	17
Run your first annealing problem with Ocean	20
Work with Amazon Braket	23
Constructing circuits	23
Gates and circuits	23
Manual qubit allocation	26
Inspecting the circuit	27
Result types	28
Submitting tasks to QPUs and simulators	30
Task batching	30
Running tasks on Amazon Braket	31
Submitting tasks to a QPU	34
Running a task with the local simulator	35
Monitoring and tracking tasks	35
Tracking tasks from the Braket SDK	35
Advanced logging	37
Monitoring tasks through the Amazon Braket console	38
Working with Boto3	40
Enable the Amazon Braket Boto3 client	40
Configure AWS CLI profiles with Boto3 and Amazon Braket	42
Security	44
Shared responsibility for security	44
Data protection	44
Managing Access to Amazon Braket	45
Amazon Braket resources	46
Additional permissions required to create a new role	46
Service-linked role	46
Service-linked role permissions for Amazon Braket	47
Resilience	48
Infrastructure Security	48

Third Party Security	48
VPC endpoints (PrivateLink)	50
Considerations for Amazon Braket VPC endpoints	50
Set up Braket and PrivateLink	50
Step 1: Launch an Amazon VPC if needed	50
Step 2: Create an interface VPC endpoint for Braket	51
Step 3: Connect and run Braket tasks through your endpoint	51
More about creating an endpoint	51
Control access with Amazon VPC endpoint policies	52
Tagging resources	53
Using tags	53
Supported resources in Amazon Braket	53
Tag restrictions	54
Managing tags in Amazon Braket	54
Add tags	54
View tags	54
Edit tags	54
Remove tags	55
Example of CLI tagging in Amazon Braket	55
Tagging with the Amazon Braket API	55
Monitor with CloudWatch	56
Amazon Braket Metrics and Dimensions	56
Supported Devices	56
Amazon Braket Events with EventBridge	57
Monitor task status with EventBridge	57
Example Amazon Braket event	58
Logging with CloudTrail	59
Amazon Braket Information in CloudTrail	59
Understanding Amazon Braket Log File Entries	60
Document history	62

What Is Amazon Braket?

Amazon Braket is a fully managed service that helps you get started with quantum computing. Amazon Braket lets researchers and developers alike explore and design quantum algorithms, test them on quantum circuit simulators, and run them on different types of quantum computing technologies.

Quantum computing has the potential to solve computational problems that are beyond the reach of classical computers by harnessing the laws of quantum mechanics to process information in new ways. But defining those problems and programming quantum computers to solve them requires a new set of skills. At the same time, gaining access to quantum computing hardware to run your algorithms and optimize your designs can be expensive and inconvenient. This has made it difficult to evaluate the current state of the technology and plan for when to invest your resources to maximize its potential. Amazon Braket helps overcome these challenges by providing a single access point to different quantum computing technologies that lets developers, researchers, and scientists explore, evaluate, and experiment with quantum computing.

Amazon Braket has three modules, Build, Test, and Run:

Build - Amazon Braket provides fully managed Jupyter notebook environments that make it easy to get started with a few clicks. There is no need to operate your own servers. Amazon Braket notebooks come 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 execute 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 that help you test and validate your circuits with a single line of code. Amazon Braket handles all the underlying software components and EC2 clusters to take away the heavy lifting of simulating quantum circuits on HPC infrastructure.

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

Amazon Braket terms and concepts

The following terms and concepts are used in Amazon Braket:

Braket

We named our 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 \(p. 6\)](#).

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 \(p. 6\)](#).

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.

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

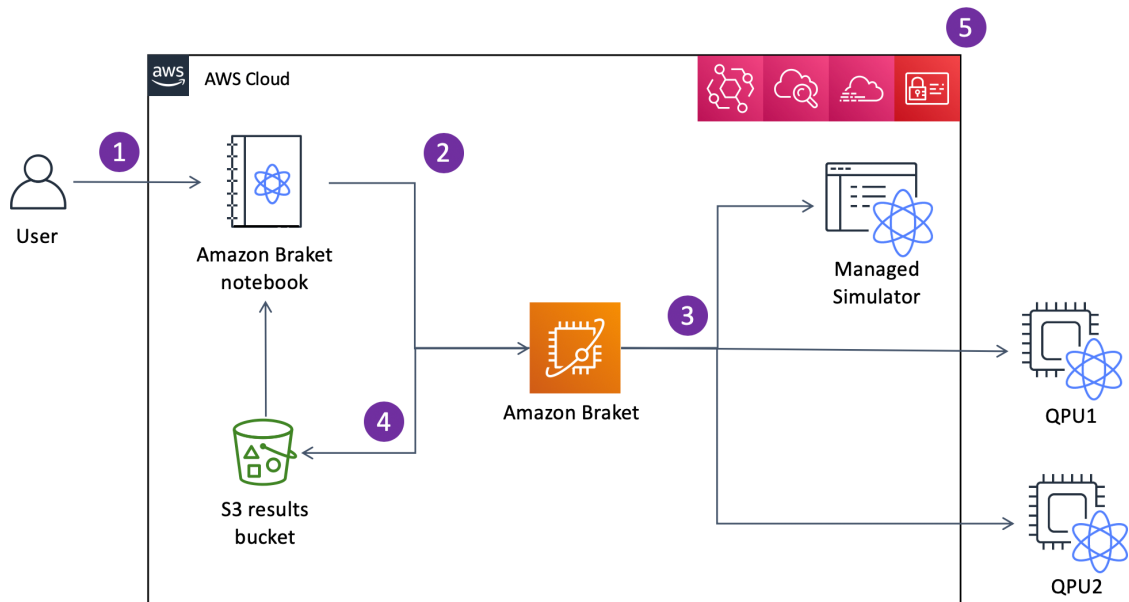
QPU native gates are the gates that can be directly mapped to control pulses by the QPU control system. Native gates can be run on the QPU device without further compilation. Subset of *QPU supported gates*. You can find the native gates of a device on the **Devices** page in the Amazon Braket console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

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) 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 3 different types of QPUs (D-Wave, IonQ, Rigetti) and one managed Simulator, SV1. To learn more, see [Amazon Braket supported devices \(p. 6\)](#).

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.

Of course, 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 \(p. 48\)](#).

Amazon Braket supported devices

In Amazon Braket, a device represents a QPU or simulator that you can call to run quantum tasks; that is, circuits for gate-based tasks and annealing problems for quantum annealers.

Amazon Braket provides access to four QPU devices (from D-Wave, IonQ, and Rigetti) and two 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 through the *GetDevice* API.

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') # V1
# device = LocalSimulator() # Local Simulator
# 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_system1') # D-Wave Advantage_system
# device = AwsDevice('arn:aws:braket:::device/qpu/ionq/ionqdevice') # IonQ
# device = AwsDevice('arn:aws:braket:::device/qpu/rigetti/Aspen-8') # Aspen-8

# get device properties
device.properties
```

Supported QPUs:

- IonQ
- Rigetti Aspen-8
- D-Wave 2000Q
- D-Wave Advantage_system

Supported Simulators:

- Local Simulator ('Default Simulator')
- State Vector Simulator (SV1)

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-east-2
D-Wave	Advantage_system1	quantum annealer	QPU	arn:aws:braket:::device/qpu/d-wave/Advantage_system1	us-east-2

Provider	Device Name	Paradigm	Type	Device ARN	Region
IonQ	ionQdevice	gate-based	QPU	arn:aws:braket:::device/qpu/ionq/ionQdevice	us-east-1
Rigetti	Aspen-8	gate-based	QPU	arn:aws:braket:::device/qpu/rigetti/Aspen-8	us-east-1
AWS	DefaultSimulator	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.

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 171Yb^+ ions, spatially confined via a microfabricated surface electrode trap within a vacuum chamber.

Rigetti

Rigetti quantum processors are universal, gate-model machines based on all-tunable superconducting qubits. The Rigetti Aspen-8 system is based on scalable 32-qubit node technology.

D-Wave

D-Wave offers quantum annealers based on superconducting qubits. Quantum annealing processors naturally return low-energy solutions. This type of QPUs is best suited to solve optimization problems and probabilistic sampling problems.

Local Simulator

The local simulator ("DefaultSimulator") is part of the Amazon Braket SDK and 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.

State Vector Simulator (SV1)

SV1 is a fully managed, high-performance, state vector simulator. It can simulate circuits of up to 34 qubits and has a maximum runtime of 6h. You should expect a 34-qubit, dense, and square (circuit depth

= 34) circuit to take approximately 1-2 hours to complete, depending on the type of gates used and other factors.

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 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

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.

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 can access quantum computing resources on-demand and without upfront commitment, paying only for what you use. To learn more about pricing, please visit our [pricing 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 tasks allows a maximum of 30 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.	60	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 <code>CreateQuantumTask</code> requests	The maximum number of <code>CreateQuantumTask</code> requests you can send per second in this account per Region.	2	Yes
Burst rate of <code>CreateQuantumTask</code> requests	The maximum number of additional <code>CreateQuantumTask</code> requests per second (RPS) that you can send in one burst in this account in the current Region.	20	No
Rate of <code>SearchQuantumTasks</code> requests	The maximum number of <code>SearchQuantumTasks</code> requests you can send per second in this account per Region.	5	Yes

Resource	Description	Limit	Adjustable
Burst rate of <code>SearchQuantumTasks</code> requests	The maximum number of additional <code>SearchQuantumTasks</code> requests per second (RPS) that you can send in one burst in this account in the current Region.	50	No
Rate of <code>GetQuantumTask</code> requests	The maximum number of <code>GetQuantumTask</code> requests you can send per second in this account per Region.	50	Yes
Burst rate of <code>GetQuantumTask</code> requests	The maximum number of additional <code>GetQuantumTask</code> requests per second (RPS) that you can send in one burst in this account in the current Region.	500	No
Rate of <code>CancelQuantumTask</code> requests	The maximum number of <code>CancelQuantumTask</code> requests you can send per second in this account per Region.	2	Yes
Burst rate of <code>CancelQuantumTask</code> requests	The maximum number of additional <code>CancelQuantumTask</code> requests per second (RPS) that you can send in one burst in this account in the current Region.	20	No
Rate of <code>GetDevice</code> requests	The maximum number of <code>GetDevice</code> requests you can send per second in this account per Region.	5	Yes
Burst rate of <code>GetDevice</code> requests	The maximum number of additional <code>GetDevice</code> requests per second (RPS) that you can send in one burst in this account in the current Region.	50	No

Resource	Description	Limit	Adjustable
Rate of SearchDevices requests	The maximum number of SearchDevices requests you can send per second in this account per Region.	5	Yes
Burst rate of SearchDevices requests	The maximum number of additional SearchDevices requests per second (RPS) that you can send in one burst in this account in the current Region.	50	No
Number of concurrent SV1 tasks	The maximum number of concurrent tasks running on the State Vector Simulator (SV1) in the current Region.	10	Yes

Additional quotas and limits

- The Braket quantum task action (for example, a circuit or annealing problem) is limited to 150KB in size.
- The maximum number of shots allowed for the the SV1 managed simulator and Rigetti device is 100,000.
- For D-Wave and IonQ devices, the maximum is 10,000 shots.

When will my task run?

When you submit a circuit in Amazon Braket it is sent to the device you specify. Both QPUs and simulators are queued and processed in the order in which they are received. Your task runs on the QPU when the QPU is available and all other tasks submitted prior to yours have completed. The length of time it takes to process your task after you submit it varies, depending on the number and complexity of tasks that are submitted by other Amazon Braket customers, and on the availability of the QPU you selected.

Status change notifications via Email or SMS

Amazon Braket sends device and task status change events to EventBridge. This will allow you to receive emails or SMS messages when the availability status for a QPU changes or when your task state changes. To get setup for such changes you must first create an SNS topic along with a subscription to email or (as available dependent on region) SMS, see [Getting started with Amazon SNS](#) and [Sending SMS messages](#). Second you need to create a rule in EventBridge that triggers notifications to your SNS topic based on the desired rule, see [Monitoring Amazon Braket with Amazon EventBridge \(p. 57\)](#).

QPU availability windows and status

QPUs have varying availability from the Quantum Hardware Providers. Current and upcoming availability windows for each device are indicated in the **Devices** page of the Amazon Braket console. In addition to availability, status of each device is also indicated. A device is "offline" if it is not available to customers regardless of availability window, for example due to scheduled maintenance, upgrades, or operational issues. Any scheduled downtime is communicated in advance to all Amazon Braket customers through the **Announcements** page in the Amazon Braket console.

Enable Amazon Braket

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

Prerequisites

To enable and run Amazon Braket, you must have an account with permission to initiate Amazon Braket actions. These permissions are included in the **AmazonBraketFullAccess** IAM policy.

If you use an administrator account to enable Amazon Braket for other [IAM users](#) or [IAM roles](#) or 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.

Amazon Braket uses an Amazon S3 bucket to store results from your tasks. Results from all users and roles in the account are stored in the same bucket location, which you specify when you enable Amazon Braket. If you are an administrator, you must give each user or role permission to put files (objects) into the bucket and to get files (objects) from the bucket. To learn more about the permissions necessary to use Amazon Braket, see [Managing access to Amazon Braket \(p. 45\)](#).

When you're setting up, remember to create or select a bucket associated with the AWS account in which you enable Amazon Braket, as shown in the next section.

Steps to enable Amazon Braket

1. Log in to your AWS account and then open the [Amazon Braket console](#).
2. Choose **Go to Braket**.
3. To specify an Amazon S3 bucket to use for your results, do one of the following tasks:
 - a. Choose **Create new bucket** to have Amazon Braket create a bucket in your account named *amazon-braket-uniqueString*.
 - b. Choose **Specify bucket name** to enter a name to include in the bucket name after *amazon-braket*, for example, if you enter "-mycircuits", the resulting bucket name is "amazon-braket-mycircuits".
 - c. Choose **Select an existing bucket** to use an existing bucket from your account. The bucket name must start with *amazon-braket*. You can also specify a folder in the bucket to store your task results.
4. Review the permission policy attached to the service-linked role that Amazon Braket creates in your account.
5. Review the Terms and Conditions for using Amazon Braket, then select the check box to confirm that you have read and accept these terms.
6. Choose **Enable Amazon Braket**.

Common causes of failure when enabling Amazon Braket

Here are some error messages you might see when you enable or run Amazon Braket.

Access denied due to lacking S3 permissions

This error means that the user or role tried to create or manage access to an Amazon S3 bucket, without the required permissions. Be sure that you're signed in with a user or role has the **AmazonBraketFullAccess** policy enabled.

Get started with Amazon Braket

After you have followed the instructions in [Enable Amazon Braket](#), the steps to get started include:

- [Create an Amazon Braket Notebook instance](#) (p. 17)
- [Run your first circuit using the Braket Python SDK](#) (p. 17)
- [Run your first annealing problem with Ocean](#) (p. 20)

Create an Amazon Braket Notebook instance

Amazon Braket provides fully managed Jupyter notebooks to get you started with a few clicks. 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/hyphen characters.
4. Select the **Notebook instance type**. Choose the smallest type you need. To get started, let's choose a cost effective instance type, **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. 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 that you will be able to see and manage your Amazon Braket Notebook instances in both the Amazon Braket and Amazon SageMaker consoles.

Run your first circuit using the 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 (1)' section of the Amazon Braket console. At the top, there is a search bar and a 'Create notebook instance' button. Below the search bar, there is a table with columns: Name, Instance, Creation time, Status, and URL. The table contains one entry: 'amazon-braket-newSDK' with instance type 'ml.t3.medium', creation time 'Aug 14, 2020 17:02 (UTC)', and status 'InService'. The name 'amazon-braket-newSDK' is highlighted with a red box, and a red arrow points to it with the text 'Click here'.

Name	Instance	Creation time	Status	URL
amazon-braket-newSDK	ml.t3.medium	Aug 14, 2020 17:02 (UTC)	InService	amazon-braket-newSDK.notebook.us-west-1.sagemaker.aws

Amazon Braket Notebook instances are pre-installed with the Amazon Braket SDK and all its dependencies. Let's create a new notebook with `conda_braket` kernel.



We will start with a simple “hello world” example. We first construct a circuit that prepares a Bell state, and then we will run that circuit on different devices to obtain the results.

We 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

```
print(bell)
```

Run your circuit on the local simulator

Next, we 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 up to 25 qubits (depending on your local hardware). Let's instantiate the local simulator,

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

and run our circuit:

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

You should see something like

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

The specific Bell state we have prepared is an equal superposition of $|00\rangle$ and $|11\rangle$, and we 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 [here \(p. 6\)](#) and in the AWS console. When running tasks on SV1 (and any QPU, see below), the results of your task will be stored in an S3 bucket in your account.

For now, we will use the S3 bucket you created in [Enable Amazon Braket \(p. 15\)](#), but you can choose any S3 bucket Amazon Braket has permissions to access. To learn more, see [Managing access to Amazon Braket \(p. 45\)](#).

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_location = (my_bucket, my_prefix)
```

To run a circuit on SV1, you need to provide the location of the S3 bucket chosen above 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_location, 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; 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.

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 IonQ, Rigetti, and D-Wave. You can find information about the different devices and availability windows in the [Supported Devices \(p. 6\)](#) 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:::device/qpu/rigetti/Aspen-8")
```

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 `.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.

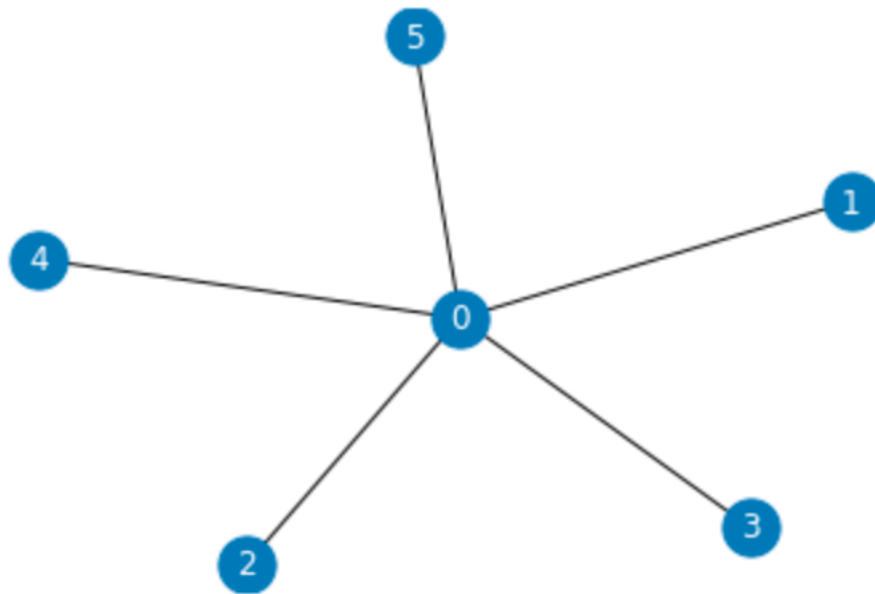
```
# define task with 1 day polling timeout
task = device.run(bell, s3_folder, poll_timeout_seconds=24*60*60)
print(task.result().measurement_counts)
```

Run your first annealing problem with Ocean

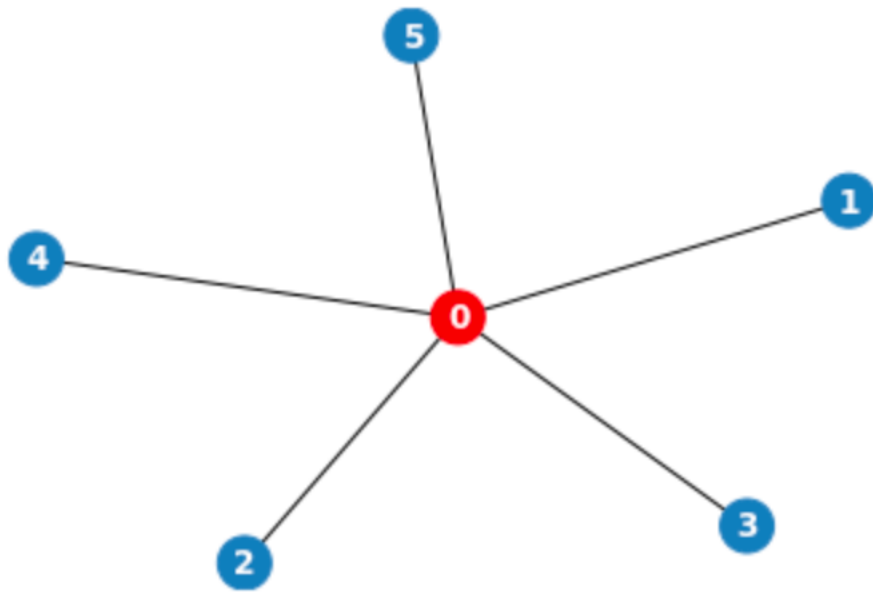
Quantum annealers are special-purpose quantum computers designed to solve combinatorial optimization problems. In particular, quantum annealers solve problems belonging to the class of [Quadratic Unconstrained Optimization \(QUBO\)](#). Amazon Braket allows you to natively program the D-Wave QPUs using [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 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 here 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

# 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_location = (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 [Erdos-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 main ways to interact with resources on Amazon Braket are by means of:

- the Amazon Braket SDK
- the Amazon Braket Console
- and the Amazon Braket API

The Amazon Braket Console helps you manage and monitor your resources and tasks, and it provides device information and status. The examples in this section demonstrate how you can work with the Amazon Braket API directly, using the AWS SDK for Python (Boto3).

In this section:

- [Constructing circuits \(p. 23\)](#)
- [Submitting tasks to QPUs and simulators \(p. 30\)](#)
- [Monitoring and tracking tasks \(p. 35\)](#)
- [Working with Boto3 \(p. 40\)](#)

Constructing circuits

This section provides examples of defining a circuit, viewing available gates, extending a circuit, viewing gates that each device supports, and manually allocating qubits.

Gates and circuits

Quantum gates and circuits are defined in the `braket.circuits` class. You can instantiate a new circuit object by calling `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 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)
```

```
['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']
```

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 that gates are appended in place, and the example that follows adds all of the gates listed above to the same circuit.

```
circ = Circuit()
# toffoli gate with q0, q1 the control qubits and q2 the target.
circ.ccnnot(0, 1, 2)
# cnot gate
circ.cnot(0, 1)
# controlled Z rotation with angle 0.15
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)
# X rotation with angle 0.15
circ.rx(0, 0.15)
# Y rotation with angle 0.15
circ.ry(0, 0.15)
# Z rotation with angle 0.15
circ.rz(0, 0.15)
# Hadamard gates applied to q0, q1, q2
circ.h(range(3))
# identity gates applied to q0, q1, q2
circ.i([0, 1, 2])
# iswap gate, iswap = [[1,0,0,0],[0,0,1j,0],[0,1j,0,0],[0,0,0,1]]
circ.iswap(0, 1)
# pswap gate, PSWAP(phi) = [[1,0,0,0],[0,0,exp(1j*phi),0],[0,exp(1j*phi),0,0],[0,0,0,1]]
circ.pswap(0, 1, 0.15)
```

```
# X gate applied to q1, q2
circ.x([1, 2])
# Y gate applied to q1, q2
circ.y([1, 2])
# Z gate applied to q1, q2
circ.z([1, 2])
# S gate applied to q0, q1, q2
circ.s([0, 1, 2])
# conjugate transpose of S gate applied to q0, q1
circ.si([0, 1])
# T gate applied to q0, q1
circ.t([0, 1])
# conjugate transpose of T gate applied to q0, q1
circ.ti([0, 1])
# square root of not gate applied to q0, q1, q2
circ.v([0, 1, 2])
# conjugate transpose of square root of not gate applied to q0, q1, q2
circ.vi([0, 1, 2])
#  $\exp(-iXX\theta/2)$ , where  $\theta=0.15$  in the example 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:::device/qpu/rigetti/Aspen-8")

# 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-8:
['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 to be able to run on quantum hardware. This compilation happens automatically when you submit a circuit to Amazon Braket.

Manual qubit allocation

When you run a quantum circuit on quantum computers from Rigetti, you can optionally use manual qubit allocation to get control over 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 thus can obtain more accurate results.

Here's an example of 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](#).

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 that **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 example that follows.

```
# 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 three, 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`. Types of results that a circuit can return are as follows:

- **Amplitude** — returns the amplitude of specified quantum states in the output wave function. It is available on 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.
- **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 ($\text{meanx} - \text{x.mean}(\wedge 2)$) of 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	Rigetti	IonQ
Amplitude	Y	Y	N	N

Expectation	Y	Y	Y	Y
Probability	Y	Y	Y	Y
State vector	Y	N	N	N
Sample	Y	Y	Y	Y
Variance	Y	Y	Y	Y

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

```
device = AwsDevice("arn:aws:braket:::device/qpu/rigetti/Aspen-8")
# 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
```

To call a `ResultType`, append it to a circuit. For 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])
```

Note: You can apply an observable to a qubit *only once*. If you specify two or more observables to the same qubit, you will see an error.

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

Submitting tasks to QPUs and simulators

Amazon Braket provides access to several devices that can execute quantum tasks.

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. The results of the task can be retrieved with the task ID, which is introduced in the next section.

- **IonQ**: `arn:aws:braket:::device/qpu/ionq/ionQdevice`
- **Rigetti**: `arn:aws:braket:::device/qpu/rigetti/Aspen-8`
- **D-Wave 2000Q**: `arn:aws:braket:::device/qpu/d-wave/DW_2000Q_6`
- **D-Wave Advantage_system**: `arn:aws:braket:::device/qpu/d-wave/Advantage_system1`

Simulators

- **Cloud-based, managed simulator**: `arn:aws:braket:::device/quantum-simulator/amazon/sv1`
- **The local simulator**: `LocalSimulator()`

Note that cloud-based simulators do not support cancellation of tasks.

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 simulator, SV1, because it can process multiple tasks in parallel. To help you set up various tasks, 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 more specifically about batching, see [Quantum task batching](#).

Running 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, it is recommended to start by running the circuit on a simulator, such as SV1.

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 execute 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 S3 location specified in the call. The positional argument `s3_location` is required for all devices except the local simulator.

Note that the Braket quantum task action (for example, a circuit or annealing problem) is limited to 150KB 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.
- 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.

Polling 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]]
```

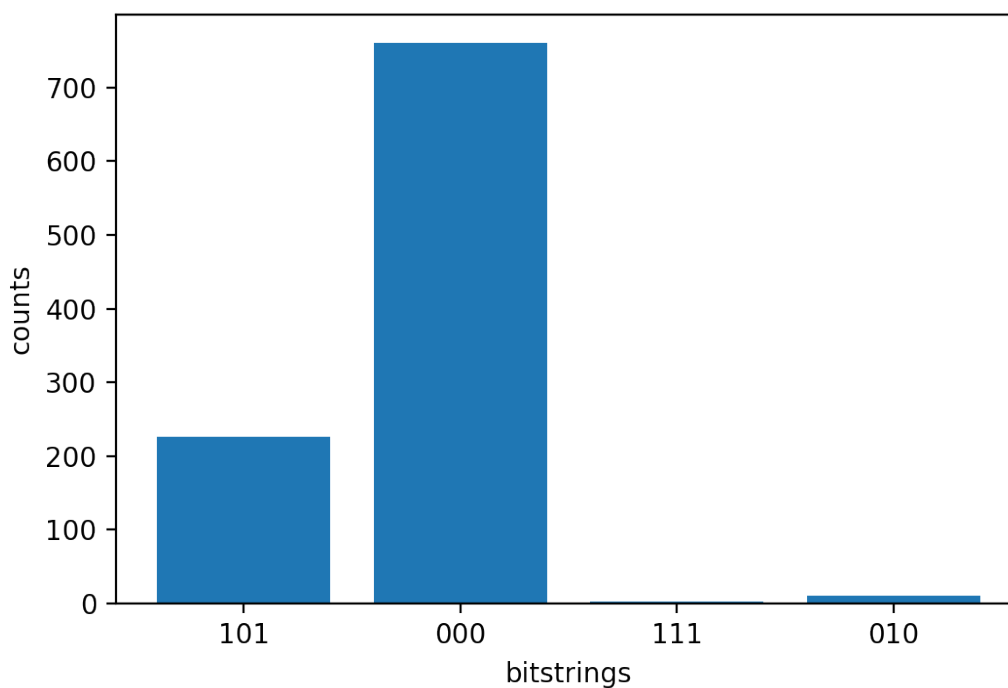
```
...  
[0 0 0]  
[0 0 0]  
[0 0 0]]  
Counts for collapsed states:  
Counter({'000': 761, '101': 226, '010': 10, '111': 3})  
Probabilities for collapsed states:  
{'101': 0.226, '000': 0.761, '111': 0.003, '010': 0.01}
```

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.

First, here's how to 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:::device/qpu/rigetti/Aspen-8")

# 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 dictionary `connectivityGraph` shown above contains information about the connectivity of the current Rigetti device.

Here's how to choose the IonQ device

For the IonQ device, as shown below, the `connectivityGraph` is empty, 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. The following example shows how.

```
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: Both QPU devices have limited capacity. You can expect longer wait times when capacity is reached.

Amazon Braket can execute QPU tasks within certain availability windows. Still, you can 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 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.

```
# import the LocalSimulator module  
from braket.devices import LocalSimulator  
  
device = LocalSimulator()  
my_task = device.run(circ, shots=1000)
```

Monitoring and tracking tasks

After a task is submitted, 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 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`

Tracking tasks from the 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 that `task = device.run()` is an asynchronous operation, which means that you can keep working while the system processes your task in the background.

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: QUEUED
Status: QUEUED
Status: QUEUED
Status: QUEUED
Status: QUEUED
Status: QUEUED
Status: QUEUED
Status: QUEUED
Status: RUNNING
Status: RUNNING
Status: COMPLETED
```

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
```

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
```

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.

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
```

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
)
```

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})
```

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 view only those tasks created in the specific AWS Region.

Amazon Braket Developer Guide

Monitoring tasks through the Amazon Braket console

Amazon Braket > Tasks

QPU's are region specific. Please select the correct device region for it's tasks. [Learn more](#)

Tasks (18) View in S3 Show task details

Search Status: X < 1 2 > ⚙️

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-927b62a40175	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 next example.

Amazon Braket > Tasks

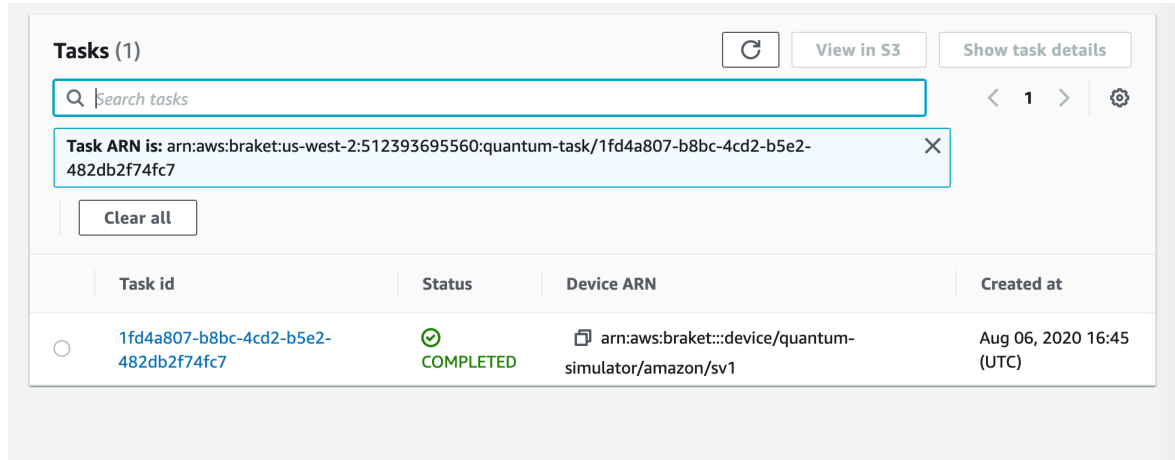
QPU's are region specific. Please select the correct device region for it's tasks. [Learn more](#)

Tasks (18) View in S3 Show task details

Search tasks < 1 2 > ⚙️

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)	COMPLETED	arn:aws:braket:::device/quantum-simulator/amazon/sv1	Aug 06, 2020 14:32 (UTC)
Creation time after (YYYY-MM-DD)	COMPLETED	arn:aws:braket:::device/quantum-simulator/amazon/sv1	Aug 06, 2020 14:32 (UTC)
479d27d9-b6d2-418e-9efa-55a54b4bc526	COMPLETED	arn:aws:braket:::device/quantum-simulator/amazon/sv1	Aug 06, 2020 14:32 (UTC)

Here is an example of searching for a task based on its unique task ID, which can be obtained by calling `task.id`.



The screenshot shows the Amazon Braket console interface. At the top, there's a 'Tasks (1)' header with a refresh button, 'View in S3', and 'Show task details'. Below this is a search bar labeled 'search tasks'. A tooltip is displayed over the search bar, showing 'Task ARN is: arn:aws:braket:us-west-2:512393695560:quantum-task/1fd4a807-b8bc-4cd2-b5e2-482db2f74fc7' with a close button. Below the tooltip is a 'Clear all' button. The main part of the console is a table with the following columns: Task id, Status, Device ARN, and Created at. There is one row in the table representing a completed task.

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)

Working with Boto3

Boto3 is the Amazon Web Services (AWS) SDK for Python. It enables Python developers to create, configure, and manage AWS services, such as Amazon Braket. Boto3 provides an easy-to-use, object-oriented API, as well as low-level access to Amazon Braket. Getting started with Boto3 is easy, but it requires a few steps.

Follow the instructions in the [Boto3 Quickstart guide](#) to learn how to install and configure Boto3.

Enable the Amazon Braket Boto3 client

To use Boto3 with Amazon Braket, you must first import it and specify a client that you'll be using to connect to the `braket` API, as shown in the example that follows.

```
import boto3

# Create the braket client
braket = boto3.client('braket')
```

Now that you have a `braket` client, you can make requests and process responses from the service. You can get more detail on request and response data in the [API Reference](#).

The following examples show more about working with devices and sessions. Examples include these tasks:

- Search for devices
- Retrieve devices
- Create a quantum task
- Retrieve a quantum task
- Search for quantum tasks
- Cancel quantum tasks

Example

- `search_devices(**kwargs)`

Searches 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'])
```

- `get_device(deviceArn)`

Retrieves 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']}")
```

Example

- `create_quantum_task(**kwargs)`

Creates 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")
```

- `get_quantum_task(quantumTaskArn)`

Retrieves 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'])
```

Example

- `search_quantum_tasks(**kwargs)`

Searches 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(quantumTaskArn)`

Cancels 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 with Boto3 and Amazon Braket

The Amazon Braket SDK relies upon the default CLI credentials, unless you explicitly specify otherwise. This default is recommended when you run on a managed Amazon Braket notebook, because you must provide an IAM Role with sufficient permissions to launch the notebook instance. If you run your code locally, on an EC2 instance, for example, you can establish named AWS CLI profiles. Each profile can have 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 Braket so that API calls are made with the permissions from that profile.

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 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')
```

Substitute a value of `region` 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 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`.

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

Managing access to Amazon Braket

This section describes the permissions that are required to run Amazon Braket. You can grant the required permissions to any IAM user or role in your account by attaching the appropriate Amazon Braket policy to that user or role in the account. Specifically, to enable Amazon Braket, select (or sign in as) a user or role that has administrator (admin) permissions or is assigned the **AmazonBraketFullAccess** policy.

The **AmazonBraketFullAccess** policy grants permissions for all Amazon Braket operations, as well as permissions for these tasks:

- **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.
- **Keep AWS CloudTrail logs** – all *describe*, *get*, and *list* actions, as well as starting and stopping queries, testing metrics filters, and filter 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** – 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.
- **Maintain usage log files for your account** – Create, store, and view logging information about Amazon Braket usage in your account.

The **AmazonBraketFullAccess** policy artifact is shown in this example code:

```
{
  "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"
          }
        }
      }
    ]
  }
}
```

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}`

Additional permissions required to create a new role

When you are creating a notebook in the Braket console, you'll see an option to create a new role. The **AmazonBraketFullAccess** managed policy doesn't provide the ability to create roles. If you have the **AmazonBraketFullAccess** role enabled, but without the proper additional permissions, you will see an error. The additional IAM permissions you must have to create a role when you are launching a notebook are listed in the following code artifact.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iam:CreateRole",
        "iam:CreatePolicy",
        "iam:AttachRolePolicy",
        "iam:UpdateRolePolicy"
      ],
      "Resource": "arn:aws:iam::586707540009:role/service-role/AmazonBraketServiceSageMakerNotebookRole-*"
    }
  ]
}
```

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 service-linked role named with the prefix **AWSServiceRoleForAmazonBraket**.

The **AWSServiceRoleForAmazonBraket** service-linked role trusts the following services to assume the role:

- Amazon Braket

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 and 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](#).

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

QPU's 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.

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 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*.

Using tags

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.

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.

- 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}`

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 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:** Include the `Tags` parameter with the `Create` operation in the [AWS API](#).
- **After you create the resource:** 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, 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 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, 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 [TagResource API](#).

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

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

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

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

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

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 up to 15 months, 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 Braket specific metric data to 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 and represents the total time from 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](#).

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

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

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.

- Log in to AWS using an account that has permissions to use EventBridge and Amazon Braket.
- Open the Amazon EventBridge console at <https://console.aws.amazon.com/events/>.
- Choose **Create rule**.
- Enter a **Name** for the rule, and, optionally, a description.
- Under **Define pattern** choose **Event pattern**.
- Under **Event matching pattern**, choose **Custom pattern**.
- In the **Event pattern** box, add the following pattern and then choose **Save**.

```
{
  "source": [
    "aws.braket"
  ],
  "detail-type": [
    "Braket Task State Change"
  ]
}
```

- In the **Select event bus** section, choose the event bus to use. If you have not created a custom event bus, choose **AWS default event bus**.

Confirm that **Enable the rule on the selected event bus** is toggled on.

- Under **Select targets**, choose the target action to take 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](#).

- Optionally, choose **Add target** to specify an additional target action for the event rule.
- Choose **Create**.

To capture all events from Amazon Braket, exclude the `detail-type` section as shown in the following code:

```
{
  "source": [
    "aws.braket"
  ]
}
```

Example Amazon Braket event

The following example shows a task status change event:

```
{
  "version": "0",
  "id": "foobar",
  "detail-type": "Braket Task State Change",
  "source": "aws.braket",
  "account": "foobar",
  "time": "2020-08-06T05:10:45Z",
  "region": "us-east-1",
  "resources": [
    "foobar"
  ],
  "detail": {
    "quantumTaskArn": "foobar",
    "created": "foobar",
    "irType": "GA-MODEL",
    "shots": "100",
    "resultsS3ObjectKey": "Aug2020/sanity/24de4823-9688-4b7d-b916-32b547ab6454",
    "resultsS3Bucket": "braket-load-tests-013039061202",
    "modified": "foobar",
    "backendArn": "foobar",
    "status": "COMPLETED"
  }
}
```

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.metall1.x86_64 botocore/1.17.33",
  "requestParameters": {
    "quantumTaskArn": "foobar"
  },
  "responseElements": null,
  "requestID": "20e8000c-29b8-4137-9cbc-af77d1dd12f7",
  "eventID": "4a2fdb22-a73d-414a-b30f-c0797c088f7c",
  "readOnly": true,
  "eventType": "AwsApiCall",
  "recipientAccountId": "foobar"
}
```

The following shows a log entry for the `GetDevice` action, which returns the details of a device event.

```
{
  "eventVersion": "1.05",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "foobar",
    "arn": "foobar",
    "accountId": "foobar",
    "accessKeyId": "foobar",
```



```
    "sessionContext": {
      "sessionIssuer": {
        "type": "Role",
        "principalId": "foobar",
        "arn": "foobar",
        "accountId": "foobar",
        "userName": "foobar"
      },
      "webIdFederationData": {},
      "attributes": {
        "mfaAuthenticated": "false",
        "creationDate": "2020-08-07T00:46:29Z"
      }
    },
    "eventTime": "2020-08-07T00:46:32Z",
    "eventSource": "braket.amazonaws.com",
    "eventName": "GetDevice",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "foobar",
    "userAgent": "Boto3/1.14.33 Python/3.7.6 Linux/4.14.158-129.185.amzn2.x86_64 exec-env/
AWS_ECS_FARGATE Botocore/1.17.33",
    "errorCode": "404",
    "requestParameters": {
      "deviceArn": "foobar"
    },
    "responseElements": null,
    "requestID": "c614858b-4dcf-43bd-83c9-bcf9f17f522e",
    "eventID": "9642512a-478b-4e7b-9f34-75ba5a3408eb",
    "readOnly": true,
    "eventType": "AwsApiCall",
    "recipientAccountId": "foobar"
  }
}
```

Document history

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

- **API version:** 2019-09-01
- **Latest API Reference update:** October 30, 2020
- **Latest documentation update:** November 24, 2020

Change	Description	Date
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