aws

개발자 가이드

Amazon Braket

Amazon Braket: 개발자 가이드

Amazon의 상표 및 트레이드 드레스는 Amazon 외 제품 또는 서비스와 함께, Amazon 브랜드 이미지를 떨어뜨리거나 고객에게 혼동을 일으킬 수 있는 방식으로 사용할 수 없습니다. Amazon이 소유하지 않은 기타 모든 상표는 Amazon과 제휴 관계이거나 관련이 있거나 후원 관계와 관계없이 해당 소유자의 자산입니다.

Table of Contents

Amazon Braket이란 무엇입니까?

> **Tip**
>
> 를 사용하여 양자 컴퓨팅의 기초를 알아봅니다 AWS! Amazon Braket Digital Learning Plan에 등록하고 일련의 학습 과정과 디지털 평가를 완료한 후 나만의 디지털 배지를 획득하세요.

Amazon Braket은 연구원, 과학자 및 개발자가 양자 컴퓨팅을 시작하는 데 도움이 AWS 서비스 되는 완전관리형입니다. 양자 컴퓨팅은 양자역학의 법칙을 활용하여 새로운 방식으로 정보를 처리하기 때문에 기존 컴퓨터의 범위를 벗어나는 계산 문제를 해결할 잠재력을 지니고 있습니다.

양자 컴퓨팅 하드웨어에 대한 액세스는 비용이 많이 들고 불편할 수 있습니다. 제한된 액세스로 인해 알고리즘을 실행하고 설계를 최적화하며 기술의 현재 상태를 평가하고 최대 효과를 위해 리소스를 투자할 시기를 계획하기가 어렵습니다. Braket은 이러한 문제를 극복하는 데 도움이 됩니다.

Braket은 다양한 양자 컴퓨팅 기술에 대한 단일 액세스 지점을 제공합니다. Braket을 사용하면 다음을 수행할 수 있습니다.

- 양자 및 하이브리드 알고리즘을 탐구하고 설계합니다.
- 다양한 양자 회로 시뮬레이터에서 알고리즘을 테스트합니다.
- 다양한 유형의 양자 컴퓨터에서 알고리즘을 실행합니다.
- 개념 증명 애플리케이션을 생성합니다.

양자 문제를 정의하고 이를 해결하기 위해 양자 컴퓨터를 프로그래밍하려면 새로운 기술 세트가 필요합니다. 이러한 기술을 익히는 데 도움이 되도록 Braket은 양자 알고리즘을 시뮬레이션하고 실행할 수 있는 다양한 환경을 제공합니다. 요구 사항에 가장 적합한 접근 방식을 찾고 노트북이라고 하는 예제 환경 세트를 빠르게 시작할 수 있습니다.

Braket 개발에는 다음과 같은 세 단계가 있습니다.

- 빌드 - Braket은 간단하게 시작할 수 있는 완전관리형 Jupyter Notebook 환경을 제공합니다. Braket 노트북에는 Amazon Braket SDK를 비롯한 샘플 알고리즘, 리소스 및 개발자 도구가 사전 설치되어 있습니다. Amazon Braket SDK를 사용하면 양자 알고리즘을 빌드한 다음 단 한 줄의 코드만 변경하여 다양한 양자 컴퓨터 및 시뮬레이터에서 이를 테스트하고 실행할 수 있습니다.
- 테스트 - Braket은 완전관리형 고성능 양자 회로 시뮬레이터에 대한 액세스를 제공합니다. 회로를 테스트하고 검증할 수 있습니다. Braket은 모든 기본 소프트웨어 구성 요소와 Amazon Elastic

Compute Cloud(Amazon EC2) 클러스터를 처리하여 고전적인 고성능 컴퓨팅(HPC) 인프라에서 양자 회로를 시뮬레이션하는 부담을 덜어줍니다.

- 실행 - Braket은 다양한 유형의 양자 컴퓨터에 대한 안전한 온디맨드 액세스를 제공합니다. AQT, Rigetti, IonQ IQM및의 게이트 기반 양자 컴퓨터와 QuEra의 아날로그 해밀턴 시뮬레이터에 액세스할 수 있습니다. 또한 선결제 약정이 없으며 개별 공급자를 통해 액세스 권한을 확보할 필요가 없습니다.

양자 컴퓨팅 및 Braket 정보

양자 컴퓨팅은 초기 개발 단계에 있습니다. 현재 범용 내결함성 양자 컴퓨터가 없다는 점을 이해하는 것이 중요합니다. 따라서 특정 유형의 양자 하드웨어는 각 사용 사례에 더 적합하며 다양한 컴퓨팅 하드웨어에 액세스하는 것이 중요합니다. Braket은 타사 공급자를 통해 다양한 하드웨어를 제공합니다.

기존 양자 하드웨어는 노이즈로 인해 오류가 발생하므로 한계가 있습니다. 현재 업계는 '노이즈가 많은 중간 규모 양자(Noisy Intermediate Scale Quantum, NISQ) 시대에 있습니다. NISQ 시대에는 양자 컴퓨팅 디바이스가 너무 많은 노이즈를 발생시켜 Shor의 알고리즘 또는 Grover의 알고리즘과 같은 순수 양자 알고리즘을 유지할 수 없습니다. 더 나은 양자 오류 정정을 사용할 수 있을 때까지, 가장 실용적인 양자 컴퓨팅은 고전(전통) 컴퓨팅 리소스와 양자 컴퓨터를 결합하여 하이브리드 알고리즘을 만드는 것입니다. Braket은 하이브리드 양자 알고리즘 작업을 지원합니다.

하이브리드 양자 알고리즘에서 QPU는 CPU의 공동 프로세서로 사용되므로, 고전적 알고리즘에서 특정 계산 속도가 빨라집니다. 이러한 알고리즘은 계산이 고전 컴퓨터와 양자 컴퓨터 간에 이동하는 반복 처리를 활용합니다. 예를 들어, 화학, 최적화 및 기계 학습에서 양자 컴퓨팅의 현재 애플리케이션은 하이브리드 양자 알고리즘의 일종인 변분 양자 알고리즘을 기반으로 합니다. 변분 양자 알고리즘에서 고전적인 최적화 루틴은 기계 학습 훈련 세트의 오류를 기반으로 신경망의 가중치를 반복적으로 조정하는 것과 동일한 방식으로 파라미터화된 양자 회로의 파라미터를 반복적으로 조정합니다. Braket은 변분 양자 알고리즘을 지원하는 PennyLane 오픈 소스 소프트웨어 라이브러리에 대한 액세스를 제공합니다.

양자 컴퓨팅은 다음과 같은 네 가지 주요 영역에서 계산에 활용되면서 점점 더 주목받고 있습니다.

- 정수론 - 인수분해와 암호학 포함(예: Shor의 알고리즘은 정수론 계산을 위한 기본 양자 방법)
- 최적화 - 제약 조건 만족도, 선형 시스템 해결, 기계 학습 포함
- 오라큘라 컴퓨팅 - 검색, 숨겨진 부분군 및 차수 찾기 포함(예: Grover의 알고리즘은 오라큘라 계산을 위한 기본 양자 방법)
- 시뮬레이션 - 직접 시뮬레이션, 매듭 불변량 및 양자 근사 최적화 알고리즘(Quantum Approximate Optimization Algorithm, QAOA) 애플리케이션 포함

이러한 계산 범주는 금융 서비스, 생명공학, 제조 및 제약 등 다양한 분야에서 활용될 수 있습니다. Braket은 특정 실제 문제 외에도 여러 개념 증명 문제에 이미 적용할 수 있는 기능과 예제 노트북을 제공합니다.

이 섹션의 내용:

- Amazon Braket 작동 방식
- Amazon Braket 용어 및 개념
- 비용 추적 및 저장
- Amazon Braket에 대한 API 참조 및 리포지토리
- Amazon Braket 지원 리전 및 디바이스

Amazon Braket 작동 방식

> **Tip**
>
> 를 사용하여 양자 컴퓨팅의 기초를 알아봅니다 AWS! Amazon Braket Digital Learning Plan에 등록하고 일련의 학습 과정과 디지털 평가를 완료한 후 나만의 디지털 배지를 획득하세요.

Amazon Braket은 온디맨드 회로 시뮬레이터 및 다양한 유형의 양자 처리 장치(QPUs. Amazon Braket에서 디바이스에 대한 원자성 요청은 양자 작업입니다. 게이트 기반 디바이스의 경우이 요청에는 양자 회로(측정 지침 및 샷 수 포함) 및 기타 요청 메타데이터가 포함됩니다. 아날로그 해밀토니안 시뮬레이터의 경우, 양자 작업에는 양자 레지스터의 물리적 레이아웃과 조작장의 시간 및 공간 종속성이 포함됩니다.

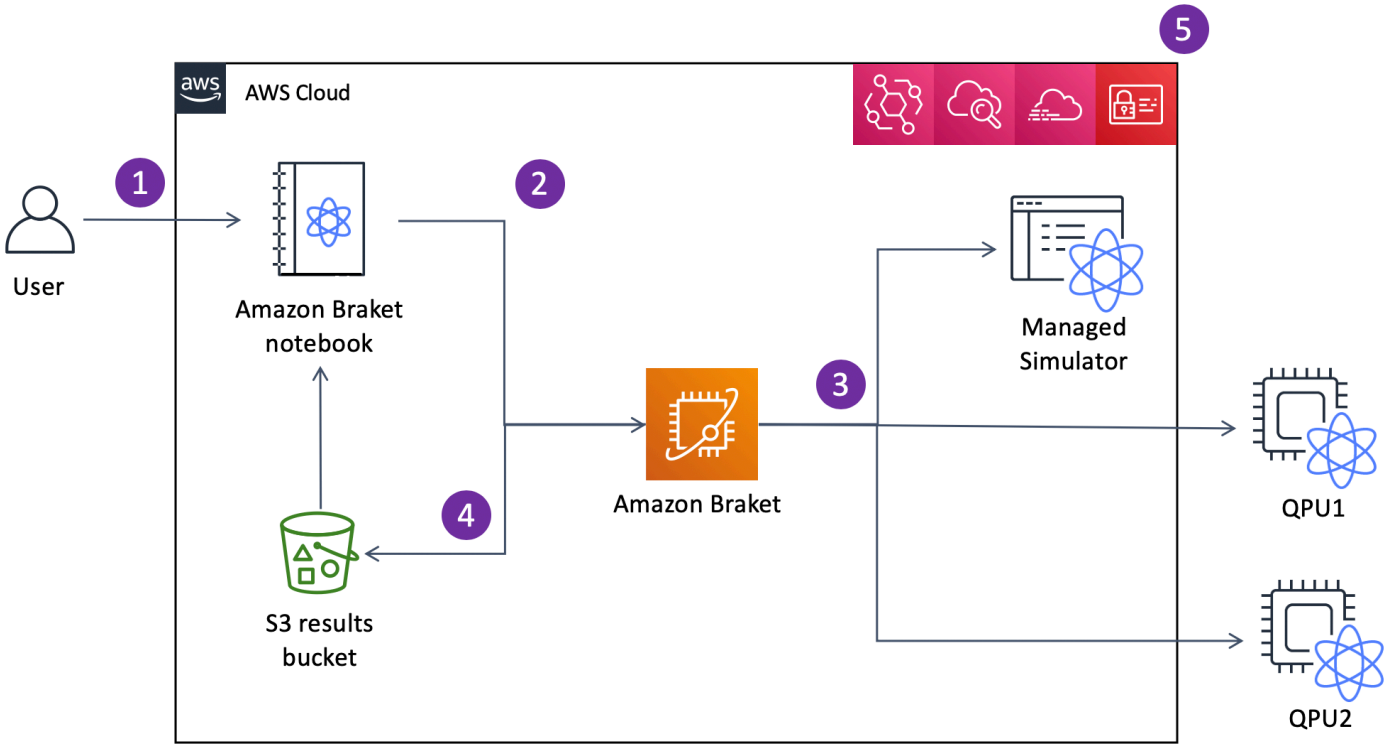
Braket Direct는 양자 컴퓨팅을 탐색하는 방법을 확장하여 연구와 혁신을 AWS가속화하는 프로그램입니다. 다양한 양자 디바이스에서 전용 용량을 예약하고 양자 컴퓨팅 전문가와 직접 소통하며 IonQ, Forte의 최신 트랩 이온 디바이스를 포함한 차세대 기능에 조기에 액세스할 수 있습니다.

이 섹션에서는 Amazon Braket에서 양자 작업을 실행하는 고수준 흐름에 대해 알아봅니다.

이 섹션의 내용:

- Amazon Braket 양자 작업 흐름
- 타사 데이터 처리

Amazon Braket 양자 작업 흐름



Jupyter 노트북을 사용하면 Amazon Braket 콘솔에서 또는 Amazon Braket SDK를 사용하여 양자 작업을 정의, 제출 및 모니터링할 수 있습니다. SDK에서 직접 양자 회로를 빌드할 수 있습니다. 그러나 아날로그 해밀턴 시뮬레이터의 경우 레지스터 레이아웃과 제어 필드(1)를 정의합니다. 양자 작업이 정의되면, 실행할 디바이스를 선택하여 Amazon Braket API(2)에 제출할 수 있습니다. 선택한 디바이스에 따라, 양자 작업은 디바이스가 사용 가능한 상태가 될 때까지 대기열에 추가되고 작업은 구현(3)을 위해 QPU 또는 시뮬레이터로 전송됩니다. Amazon Braket을 사용하면 QPUs, 온디맨드 시뮬레이터, 로컬 시뮬레이터, 임베디드 시뮬레이터 등 지원되는 다양한 양자 디바이스에 액세스할 수 있습니다.

양자 작업을 처리한 후 Amazon Braket은 결과를 Amazon S3 버킷에 반환합니다.이 버킷에서는 데이터가 AWS 계정 (4)에 저장됩니다. 동시에 SDK는 백그라운드에서 결과를 폴링하고 양자 작업 완료 시 Jupyter Notebook에 로드합니다. Amazon Braket 콘솔의 Quantum Tasks 페이지에서 또는 Amazon Braket의 `GetQuantumTask` 작업을 사용하여 양자 작업을 보고 관리할 수도 있습니다 Amazon BraketAPI.

Amazon Braket은 AWS Identity and Access Management 사용자 액세스 관리, 모니터링 AWS CloudTrail 및 로깅과 이벤트 기반 처리(5)를 위해 (IAM), Amazon CloudWatch 및 Amazon EventBridge와 통합됩니다.

타사 데이터 처리

QPU 디바이스에 제출된 양자 작업은 타사 공급자가 운영하는 시설에 있는 양자 컴퓨터에서 처리됩니다. Amazon Braket의 보안 및 타사 처리에 대한 자세한 내용은 Amazon Braket 하드웨어 공급자의 보안을 참조하세요.

Amazon Braket 용어 및 개념

> **Tip**
>
> 를 사용하여 양자 컴퓨팅의 기초를 알아봅니다 AWS! Amazon Braket Digital Learning Plan에 등록하고 일련의 학습 과정과 디지털 평가를 완료한 후 나만의 디지털 배지를 획득하세요.

Braket에서는 다음과 같은 용어와 개념이 사용됩니다.

아날로그 해밀토니안 시뮬레이션

아날로그 해밀토니안 시뮬레이션(Analog Hamiltonian Simulation, AHS)은 다체 시스템의 시간 종속 양자 동역학을 직접 시뮬레이션하기 위한 독특한 양자 컴퓨팅 패러다임입니다. AHS에서는 사용자가 직접 시간 종속 해밀토니안을 지정하고 양자 컴퓨터가 이 해밀토니안에 따른 연속적인 시간 변화를 직접 에뮬레이션하는 방식으로 조정됩니다. AHS 디바이스는 일반적으로 특수 용도 디바이스이며 게이트 기반 디바이스와 같은 범용 양자 컴퓨터가 아닙니다. 이러한 디바이스는 시뮬레이션할 수 있는 해밀토니안 클래스로 제한됩니다. 그러나 이러한 해밀토니안은 디바이스에서 자연스럽게 구현되므로, AHS는 알고리즘을 회로로 공식화하고 게이트 연산을 구현하는 데 필요한 오버헤드가 발생하지 않습니다.

Braket

Braket 서비스는 양자역학의 표준 표기법인 bra-ket 표기법에서 따온 명칭입니다. 1939년 Paul Dirac이 양자 시스템의 상태를 설명하기 위해 도입했으며, Dirac 표기법으로도 알려져 있습니다.

Braket Direct

Braket Direct를 사용하면 원하는 다양한 양자 디바이스에 대한 전용 액세스를 예약하고 양자 컴퓨팅 전문가와 연결하여 워크로드에 대한 지침을 받으며 가용성이 제한된 새로운 양자 디바이스와 같은 차세대 기능에 조기에 액세스할 수 있습니다.

Braket 하이브리드 작업

Amazon Braket에는 하이브리드 알고리즘의 완전관리형 실행을 제공하는 Amazon Braket Hybrid Jobs라는 기능이 있습니다. Braket 하이브리드 작업은 다음과 같은 세 가지 구성 요소로 이루어져 있습니다.

1. 스크립트, Python 모듈 또는 Docker 컨테이너로 제공될 수 있는 알고리즘의 정의입니다.
2. 알고리즘을 실행할 Amazon EC2를 기반으로 하는 하이브리드 작업 인스턴스입니다. 기본값은 ml.m5.xlarge 인스턴스입니다.
3. 알고리즘의 일부인 양자 작업을 실행할 양자 디바이스입니다. 단일 하이브리드 작업에는 일반적으로 많은 양자 작업 모음이 포함됩니다.

디바이스

Amazon Braket에서 디바이스는 양자 작업을 실행할 수 있는 백엔드입니다. 디바이스는 QPU 또는 양자 회로 시뮬레이터일 수 있습니다. 자세한 내용은 Amazon Braket 지원 디바이스를 참조하세요.

오류 완화

오류 완화에는 여러 물리적 회로를 실행하고 측정값을 결합하여 결과를 개선하는 작업이 포함됩니다. 자세한 내용은 오류 완화 기법을 참조하세요.

게이트 기반 양자 컴퓨팅

회로 기반 QC라고도 하는 게이트 기반 양자 컴퓨팅(QC)에서는 계산이 기본 연산(게이트)으로 나뉩니다. 특정 게이트 세트는 범용이므로, 모든 계산을 해당 게이트의 유한 수열로 표현할 수 있습니다. 게이트는 양자 회로의 구성 요소이며 고전적인 디지털 회로의 논리 게이트와 유사합니다.

게이트샷 제한

게이트샷 제한은 샷당 총 게이트 수(모든 게이트 유형의 합계)와 작업당 샷 수를 나타냅니다. 수학적으로 게이트샷 제한은 다음과 같이 표현할 수 있습니다.

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

해밀토니안

물리적 시스템의 양자 동역학은 시스템 구성 요소 간의 상호 작용과 외생적 구동력의 영향에 대한 모든 정보를 인코딩하는 해밀토니안에 의해 결정됩니다. N-큐비트 시스템의 해밀토니안은 일반적으로 고전 컴퓨터에서 복소수의 $2^N \times 2^N$ 행렬로 표현됩니다. 양자 디바이스에서 AHS를 실행하면 이러한 기하급수적 리소스 요구 사항을 피할 수 있습니다.

펄스

펄스는 큐비트로 전송되는 일시적인 물리적 신호입니다. 이는 캐리어 신호를 지지하는 역할을 하고 하드웨어 채널 또는 포트에 바인딩되는 프레임에서 재생되는 파형으로 설명됩니다. 고객은 고주파 사인파 캐리어 신호를 변조하는 아날로그 엔벨로프를 제공하여 자체 펄스를 설계할 수 있습니다. 프레임은 큐비트의 |0# 및 |1#에 대한 에너지 준위 사이의 에너지 분리를 통해 종종 공진 상태로 선택되는 주파수와 위상으로 고유하게 설명됩니다. 따라서 게이트는 미리 결정된 모양과 진폭, 주파수, 기간과 같은 보정된 파라미터를 갖는 펄스로 구현됩니다. 템플릿 파형에 포함되지 않은 사용 사례는 고정된 물리적 주기 시간으로 구분된 값 목록을 제공하여 단일 샘플 분해능에서 지정되는 사용자 지정 파형을 통해 활성화됩니다.

양자 회로

양자 회로는 게이트 기반 양자 컴퓨터에서 계산을 정의하는 명령 세트입니다. 양자 회로는 측정 명령과 함께 qubit 레지스터에 대한 가역적 변환인 양자 게이트 순서로 구성됩니다.

양자 회로 시뮬레이터

양자 회로 시뮬레이터는 고전 컴퓨터에서 실행되고 양자 회로의 측정 결과를 계산하는 컴퓨터 프로그램입니다. 일반 회로의 경우, 양자 시뮬레이션의 리소스 요구 사항은 시뮬레이션할 qubits의 수에 따라 기하급수적으로 증가합니다. Braket은 관리형(Braket API를 통해 액세스) 및 로컬(Amazon Braket SDK의 일부) 양자 회로 시뮬레이터 모두에 대한 액세스를 제공합니다.

양자 컴퓨터

양자 컴퓨터는 중첩 및 얽힘과 같은 양자역학적 현상을 이용하여 계산을 수행하는 물리적 디바이스입니다. 게이트 기반 QC와 같이 양자 컴퓨팅(QC)에는 다양한 패러다임이 있습니다.

양자 처리 장치(Quantum Processing Unit, QPU)

QPU는 양자 작업에서 실행할 수 있는 물리적 양자 컴퓨팅 디바이스입니다. QPU는 게이트 기반 QC와 같은 다양한 QC 패러다임을 기반으로 할 수 있습니다. 자세한 내용은 Amazon Braket 지원 디바이스를 참조하세요.

QPU 네이티브 게이트

QPU 네이티브 게이트를 직접 매핑하여 QPU 제어 시스템에서 펄스를 제어할 수 있습니다. 네이티브 게이트는 추가 컴파일 없이 QPU 디바이스에서 실행할 수 있습니다. QPU 지원 게이트의 부분 집합입니다. Amazon Braket 콘솔의 디바이스 페이지와 Braket SDK를 통해 디바이스의 네이티브 게이트를 찾을 수 있습니다.

QPU 지원 게이트

QPU 지원 게이트는 QPU 디바이스에서 허용되는 게이트입니다. 이러한 게이트는 QPU에서 직접 실행되지 않을 수 있으므로, 네이티브 게이트로 분해해야 할 수 있습니다. Amazon Braket 콘솔의

디바이스 페이지와 Amazon Braket SDK를 통해 디바이스에서 지원되는 게이트를 찾을 수 있습니다.

양자 작업

Braket에서 양자 작업은 디바이스에 대한 원자성 요청입니다. 게이트 기반 QC 디바이스의 경우, 이 요청에는 양자 회로(측정 명령 및 shots 수 포함) 및 기타 요청 메타데이터가 포함됩니다. Amazon Braket SDK를 통해 또는 CreateQuantumTask API 연산을 직접 사용하여 양자 작업을 생성할 수 있습니다. 양자 작업을 생성하면 요청된 디바이스가 사용 가능 상태가 될 때까지 대기열에 추가됩니다. Amazon Braket 콘솔의 양자 작업 페이지에서 또는 GetQuantumTask나 SearchQuantumTasks API 연산을 사용하여 양자 작업을 볼 수 있습니다.

Qubit

양자 컴퓨터의 기본 정보 단위를 qubit(양자 비트)라고 하며, 이는 고전적 컴퓨팅의 비트와 매우 유사합니다. qubit는 초전도 회로 또는 개별 이온 및 원자와 같은 다양한 물리적 구현을 통해 실현될 수 있는 2단계 양자 시스템입니다. 다른 qubit 유형은 광자, 전자 또는 핵 스핀, 혹은 더 이색적인 양자 시스템을 기반으로 합니다.

Queue depth

Queue depth는 특정 디바이스에 대해 대기열에 있는 양자 작업 및 하이브리드 작업 수를 나타냅니다. 디바이스의 양자 작업 및 하이브리드 작업 대기열 수는 Amazon Braket Management Console 또는 Braket Software Development Kit (SDK)를 통해 액세스할 수 있습니다.

1. 작업 대기열 깊이는 일반 우선순위에서 실행되기를 기다리는 양자 작업의 총 수를 나타냅니다.
2. 우선순위 작업 대기열 깊이는 Amazon Braket Hybrid Jobs을 통해 실행되기를 기다리는 제출된 양자 작업의 총 수를 나타냅니다. 하이브리드 작업이 시작되면 이러한 작업은 독립 실행형 작업보다 우선순위가 높아집니다.
3. 하이브리드 작업 대기열 깊이는 현재 디바이스에 대기 중인 하이브리드 작업의 총 수를 나타냅니다. 하이브리드 작업의 일부로 제출된 Quantum tasks에 우선순위가 있으며 Priority Task Queue에서 집계됩니다.

Queue position

Queue position는 각 디바이스 대기열 내에서 양자 작업 또는 하이브리드 작업의 현재 위치를 나타냅니다. 이는 Braket Software Development Kit (SDK) 또는 Amazon Braket Management Console을 통해 양자 작업 또는 하이브리드 작업에 대해 얻을 수 있습니다.

Shots

양자 컴퓨팅은 본질적으로 확률적이므로 정확한 결과를 얻으려면 모든 회로를 여러 번 평가해야 합니다. 단일 회로 실행 및 측정을 샷이라고 합니다. 회로의 샷 수(반복 실행)는 원하는 결과 정확도에 따라 선택됩니다.

AWS Amazon Braket에 대한 용어 및 팁

IAM 정책

IAM 정책은 AWS 서비스 및 리소스에 대한 권한을 허용하거나 거부하는 문서입니다. IAM 정책을 사용하면 리소스에 대한 사용자의 액세스 수준을 사용자 지정할 수 있습니다. 예를 들어 사용자가 내 모든 Amazon S3 버킷 AWS 계정또는 특정 버킷에만 액세스하도록 허용할 수 있습니다.

- 모범 사례: 권한을 부여할 때는 최소 권한의 보안 원칙을 따릅니다. 이 원칙을 따르면 사용자 또는 역할이 양자 작업을 수행하는 데 필요한 것보다 더 많은 권한을 갖는 것을 방지할 수 있습니다. 예를 들어 직원이 특정 버킷에만 액세스해야 하는 경우 해당 직원에게 AWS 계정의 모든 버킷에 대한 액세스 권한을 부여하는 대신 IAM 정책에 버킷을 지정합니다.

IAM 역할

IAM 역할은 권한에 대한 임시 액세스 권한을 얻기 위해 수임할 수 있는 ID입니다. 사용자, 애플리케이션 또는 서비스가 IAM 역할을 맡기 전에 먼저 해당 역할로 전환할 수 있는 권한이 부여되어야 합니다. 누군가가 IAM 역할을 수임하면, 이전 역할에서 가졌던 모든 이전 권한을 포기하고 새 역할의 권한을 수임하게 됩니다.

- 모범 사례: IAM 역할은 서비스 또는 리소스에 대한 액세스 권한을 장기적으로 부여하기보다는 일시적으로 부여해야 하는 상황에 적합합니다.

Amazon S3 버킷

Amazon Simple Storage Service(Amazon S3)는 버킷에 객체로 데이터를 저장할 수 AWS 서비스 있는 입니다. Amazon S3 버킷은 무제한 스토리지 공간을 제공합니다. Amazon S3 버킷의 최대 객체 크기는 5TB입니다. 이미지, 비디오, 텍스트 파일, 백업 파일, 웹 사이트의 미디어 파일, 아카이브된 문서, Braket 양자 작업 결과 등 모든 유형의 파일 데이터를 Amazon S3 버킷에 업로드할 수 있습니다.

- 모범 사례: S3 버킷에 대한 액세스를 제어하는 권한을 설정할 수 있습니다. 자세한 내용은 Amazon S3 설명서의 버킷 정책을 참조하세요.

비용 추적 및 저장

> **Tip**
>
> 를 사용하여 양자 컴퓨팅의 기초를 알아봅니다 AWS! Amazon Braket Digital Learning Plan에 등록하고 일련의 학습 과정과 디지털 평가를 완료한 후 나만의 디지털 배지를 획득하세요.

Amazon Braket을 사용하면 선결제 약정 없이 온디맨드로 양자 컴퓨팅 리소스에 액세스할 수 있습니다. 사용한 만큼만 지불합니다. 요금에 대한 자세한 내용은 요금 페이지를 참조하세요.

이 섹션의 내용:

- Amazon Braket QPUs 대한 지출 한도 설정
- 근실시간 비용 추적
- 비용 절감 모범 사례

Amazon Braket QPUs 대한 지출 한도 설정

Amazon Braket 지출 한도는 양자 처리 단위(QPUs.

지출 한도 작동 방식: Amazon Braket은 누적 지출을 추적하고 구성된 한도에 대해 모든 작업 생성 요청을 검증합니다. 작업의 예상 비용이 나머지 지출 한도를 초과하는 경우 Amazon Braket은 검증 오류와 함께 작업을 즉시 거부합니다. 선택적으로 지출 한도에 대한 기간을 구성할 수 있습니다. 기간을 구성하면 지정된 기간에만 작업을 제출할 수 있습니다. 기간을 초과하여 제출된 작업은 거부됩니다.

옵트인 설계: 제어를 명시적으로 활성화하지 않는 한 기존 워크플로는 영향을 받지 않습니다. 지출 한도를 삭제하여 모든 제한을 제거할 수 있습니다.

> **Note**
>
> 지출 한도는 온디맨드 및 하이브리드 작업 QPU 작업에만 적용됩니다. 시뮬레이터, 관리형 노트북, 하이브리드 작업 EC2 인스턴스 비용 및 Braket Direct 예약은 제외됩니다. 모든 AWS 서비스에서 포괄적인 비용 관리를 위해를 계속 사용합니다AWS Budgets.

지출 한도 작업 목록

검색

다음 AWS CLI 명령을 사용하면 특정 AWS 리전 및 특정 Braket 디바이스의 지출 한도를 검색하고 나열할 수 있습니다.

```
aws --region {device_region} braket search-spending-limits --filters
 name=deviceArn,operator=EQUAL,values={device_arn}
```

생성

다음 AWS CLI 명령을 사용하면 특정 리전에서 지정된 양자 디바이스에 대한 새 지출 한도를 생성할 수 있습니다. 디바이스에 대한 지출 한도가 이미 있는 경우 요청이 거부됩니다.

```
aws --region {device_region} braket create-spending-limit --device-arn {device_arn}
 --spending-limit {max_spend}
```

업데이트

다음 AWS CLI 명령을 사용하여 기존 지출 한도를 새 최대 지출 값으로 업데이트할 수 있습니다. 현재 지출과 대기 중인 지출의 합계가 이미 요청된 새 최대 지출보다 높으면 요청이 거부됩니다.

```
aws --region {device_region} braket update-spending-limit --spending-limit-arn
 {spending_limit_arn} --spending-limit {new_max_spend}
```

위 예제와 같이 새 최대 지출 대신 또는 추가로 기간을 제공할 수 있습니다.

삭제

다음 AWS CLI 명령을 사용하여 기존 지출 한도를 삭제할 수 있습니다.

```
aws --region {device_region} braket delete-spending-limit --spending-limit-arn
 {spending_limit_arn}
```

위 예제와 같이 새 최대 지출 대신 또는 추가로 기간을 제공할 수 있습니다.

선택 사항이지만 항상 리전 파라미터를 모범 사례로 지정합니다. 디바이스의 리전과 다른 리전에서 실행된 명령은 실패하거나의 경우 잘못된 결과를 `SearchSpendingLimits`반환합니다.

지출 한도를 사용하는 방법에 대한 자세한 예는 예제 노트북을 참조하세요.

작업 검증 작동 방식

AWS 계정이 유효한 `CreateQuantumTask` 요청을 보내면 다음과 같은 게이팅 동작이 적용됩니다. 참고: 남은 예산은 지출 한도와 대기 중인 지출과 현재 지출의 합계 간의 차이입니다. (다음 섹션 참조)

- 사례 1: 작업 디바이스에 대한 지출 한도가 없습니다. 작업이 생성됩니다.
- 사례 2: 대상 디바이스에 대한 지출 한도가 있으며 현재 시간은 지출 한도 기간 이내입니다.
  - 작업의 예상 비용이 나머지 예산보다 낮거나 같으면 CreateQuantumTask가 성공하면 작업이 생성됩니다.
  - 예상 비용이 나머지 예산보다 크면가 `CreateQuantumTask` 실패하고 작업이 생성되지 않습니다.
- 사례 3: 대상 디바이스에 대한 지출 한도가 있으며 현재 시간이 지출 한도 기간을 벗어났습니다. `CreateQuantumTask` 실패하고 작업이 생성되지 않습니다.

남은 예산 계산 방법

남은 예산은 지출 한도와 현재 지출 및 대기 중인 지출의 합계 간의 차이입니다.

지출 한도가 있는 디바이스에 대해 작업이 생성되면 대기 중인 지출이 작업의 예상 비용만큼 증가합니다. 이 이벤트는 다음 표의 첫 번째 행에 나열됩니다. 다음 표에는 작업 진행 상황에 따라 대기 중인 지출과 현재 지출이 어떻게 되는지 나와 있습니다.

이전 양자 작업 상태	새 양자 작업 상태	대기 중인 지출로 변경	현재 지출로 변경
-	CREATED	예상 비용 증가	No change
CREATED	대기됨	No change	No change
임의	RUNNING	No change	No change
임의	취소 중	No change	No change
취소 중	취소됨	예상 비용으로 감소	chnage 없음
임의	FAILED	예상 비용으로 감소	No change
RUNNING	COMPLETED	예상 비용으로 감소	예상 비용 증가(부분적으로 완료된 작업에 맞게 조정됨)

엣지 케이스

Q: 지출 한도를 생성할 때 대기열에 이미 있는 작업이 대기 중인 지출에 포함되나요?

A: 아니요. 이미 생성되었거나, 대기 중이거나, 진행 중인 작업은 새로 생성된 지출 한도의 대기 지출에 포함되지 않습니다.

Q: 지출 한도를 업데이트하여 낮추면 생성되거나 대기열에 있거나 진행 중인 양자 작업이 조기에 종료되나요?

A: 아니요.

Q: 지출 한도의 종료 시간에 도달하면 생성되거나 대기열에 있거나 진행 중인 양자 작업이 조기에 종료됩니까?

A: 아니요. 생성됨, 대기 중 및 기타 진행 중인 작업은 지출 한도 상태와 관계없이 완료할 수 있습니다.

Q: 지출 한도 부족과 지출 한도 0 USD는 어떻게 다릅니까?

A: 지출 한도가 없어 제한 없이 양자 작업을 생성할 수 없습니다. 달러가 0인 지출 한도는 모든 양자 작업을 차단합니다.

Q: 과거 또는 미래의 지출 한도가 0이면 모든 양자 작업 생성이 차단되나요?

A: 예.

Q: 지출 한도를 생성할 때 대기열에 이미 있는 작업의 예상 비용이 해당 작업이 완료된 후 현재 지출에 포함됩니까?

A: 아니요. 지출 한도가 활성 상태인 동안 제출된 작업만 누적 지출에 포함됩니다.

근실시간 비용 추적

Braket SDK는 양자 워크로드에 근실시간 비용 추적을 추가할 수 있는 옵션을 제공합니다. 각 예제 노트북에는 Braket의 양자 처리 장치(QPU) 및 온디맨드 시뮬레이터에 대한 최대 예상 비용을 제공하는 비용 추적 코드가 포함되어 있습니다. 최대 예상 비용은 USD로 표시되며 크레딧 또는 할인은 포함되지 않습니다.

> **Note**
>
> 표시된 요금은 Amazon Braket 시뮬레이터 및 QPU 작업 사용량을 기준으로 한 예상 금액입니다. 표시된 예상 요금은 실제 요금과 다를 수 있습니다. 예상 요금은 할인이나 크레딧을 고려하

지 않으며 Amazon Elastic Compute Cloud(Amazon EC2)와 같은 다른 서비스의 사용에 따라 추가 요금이 발생할 수 있습니다.

SV1에 대한 비용 추적

비용 추적 함수를 사용하는 방법을 보여주기 위해 벨 상태 회로를 구성하고 SV1 시뮬레이터에서 실행하기로 합니다. 먼저 Braket SDK 모듈을 가져오고 벨 상태를 정의한 다음, `Tracker()` 함수를 회로에 추가합니다.

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

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

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

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

노트북을 실행하면 벨 상태 시뮬레이션에 대해 다음과 같은 출력을 기대할 수 있습니다. 추적기 함수에는 전송된 샷 수, 완료된 양자 작업, 실행 기간, 청구된 실행 기간 및 최대 비용이 USD 단위로 표시됩니다. 실행 시간은 시뮬레이션마다 다를 수 있습니다.

```
import datetime

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

tracker.simulator_tasks_cost()
```

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

비용 추적기를 사용하여 최대 비용 설정

비용 추적기를 사용하여 프로그램에 대한 최대 비용을 설정할 수 있습니다. 특정 프로그램에 지출하려는 금액에 대한 최대 임계값을 설정할 수 있습니다. 이렇게 하면 비용 추적기를 사용하여 실행 코드에서 비용 제어 논리를 빌드할 수 있습니다. 다음 예제에서는 Rigetti QPU에서 동일한 회로를 사용하고 비용을 1 USD로 제한합니다. 코드에서 회로를 한 번 반복 실행하는 데 드는 비용은 0.30 USD입니다. 총 비용이 1 USD를 초과할 때까지 반복하도록 논리를 설정했습니다. 따라서 코드 조각은 다음 반복이 1 USD를 초과할 때까지 세 번 실행됩니다. 일반적으로 프로그램은 원하는 최대 비용에 도달할 때까지 계속 반복됩니다. 이 경우에서는 3회 반복입니다.

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

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

> **Note**
>
> 비용 추적기는 실패한 TN1 양자 작업의 기간을 추적하지 않습니다. TN1 시뮬레이션 중에 리허설이 완료되었지만 수축 단계가 실패하면 리허설 요금이 비용 추적기에 표시되지 않습니다.

비용 절감 모범 사례

Amazon Braket 사용에 대한 다음 모범 사례를 고려하세요. 시간을 절약하고 비용을 최소화하며 일반적인 오류를 방지합니다.

시뮬레이터로 확인

- QPU에서 실행하기 전에 시뮬레이터를 사용하여 회로를 확인하면 QPU 사용에 대한 비용을 지불하지 않고도 회로를 미세 조정할 수 있습니다.

- 시뮬레이터에서 회로를 실행한 결과는 QPU에서 회로를 실행한 결과와 동일하지 않을 수 있지만, 시뮬레이터를 사용하여 코딩 오류 또는 구성 문제를 식별할 수 있습니다.

특정 디바이스에 대한 사용자 액세스 제한

- 권한이 없는 사용자가 특정 디바이스에서 양자 작업을 제출하지 못하도록 제한할 수 있습니다. 액세스를 제한하는 권장 방법은 AWS IAM을 사용하는 것입니다. 이렇게 하는 방법에 대한 자세한 내용은 액세스 제한을 참조하세요.
- Amazon Braket 디바이스에 대한 사용자 액세스 권한을 부여하거나 제한하는 방법으로 관리자 계정을 사용하지 않는 것이 좋습니다.

청구 경보 설정

- 청구 금액이 사전 설정된 한도에 도달하면 알림을 받도록 청구 경보를 설정할 수 있습니다. 경보를 설정하는 권장 방법은 다음과 같습니다 AWS Budgets. 사용자 지정 예산을 설정하고 비용 또는 사용량이 예산 금액을 초과할 수 있는 경우 알림을 받을 수 있습니다. 정보는 AWS Budgets에서 확인할 수 있습니다.

샷 수가 적은 TN1 양자 작업 테스트

- 시뮬레이터는 QPU보다 비용이 적게 들지만, 양자 작업이 높은 샷 수로 실행되는 경우 특정 시뮬레이터는 비용이 많이 들 수 있습니다. 적은 shot 수로 TN1 작업을 테스트하는 것이 좋습니다. Shot 수는 SV1 및 로컬 시뮬레이터 작업의 비용에 영향을 주지 않습니다.

양자 작업에 대해 모든 리전 확인

- 콘솔에는 현재에 대한 양자 작업만 표시됩니다 AWS 리전. 제출된 청구 가능한 양자 작업을 찾을 때는 모든 리전을 확인해야 합니다.
- 지원되는 디바이스 설명서 페이지에서 디바이스 및 관련 리전 목록을 볼 수 있습니다.

Amazon Braket에 대한 API 참조 및 리포지토리

> **Tip**
>
> 를 사용하여 양자 컴퓨팅의 기초를 알아봅니다 AWS! Amazon Braket Digital Learning Plan에 등록하고 일련의 학습 과정과 디지털 평가를 완료한 후 나만의 디지털 배지를 획득하세요.

Amazon Braket은 노트북 인스턴스를 생성 및 관리하고 모델을 훈련 및 배포하는 데 사용할 수 있는 API, SDK 및 명령줄 인터페이스를 제공합니다.

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

Amazon Braket Tutorials GitHub 리포지토리에서 코드 예제를 가져올 수도 있습니다.

- Braket 자습서 GitHub

코어 리포지토리

다음은 Braket에 사용되는 키 패키지가 포함된 코어 리포지토리 목록을 표시합니다.

- Braket Python SDK - Braket Python SDK를 사용하여 Python 프로그래밍 언어로 Jupyter 노트북에 코드를 설정합니다. Jupyter 노트북을 설정한 후, Braket 디바이스 및 시뮬레이터에서 코드를 실행할 수 있습니다.

- Braket 스키마 - Braket SDK와 Braket 서비스 간의 계약입니다.
- Braket 기본 시뮬레이터 - Braket에 대한 모든 로컬 양자 시뮬레이터(상태 벡터 및 밀도 행렬)입니다.

플러그인

또한 다양한 디바이스 및 프로그래밍 도구와 함께 사용되는 각종 플러그인이 있습니다. 여기에는 Braket 지원 플러그인뿐만 아니라 아래와 같이 타사에서 지원하는 플러그인이 포함됩니다.

Amazon Braket 지원:

- Amazon Braket 알고리즘 라이브러리 - Python으로 작성된 사전 빌드된 양자 알고리즘의 카탈로그입니다. 그대로 실행하거나 더 복잡한 알고리즘을 빌드하기 위한 출발점으로 활용할 수 있습니다.
- Braket-PennyLane 플러그인 - Braket에서 QML 프레임워크로 PennyLane을 사용합니다.

타사(Braket 팀이 모니터링 및 기여):

- Qiskit-Braket 공급자 - Qiskit SDK를 사용하여 Braket 리소스에 액세스합니다.
- Braket-Julia SDK - (EXPERIMENTAL) Braket SDK의 Julia 네이티브 버전

Amazon Braket 지원 리전 및 디바이스

> **Tip**
>
> 를 사용하여 양자 컴퓨팅의 기초를 알아봅니다 AWS! Amazon Braket Digital Learning Plan에 등록하고 일련의 학습 과정과 디지털 평가를 완료한 후 나만의 디지털 배지를 획득하세요.

Amazon Braket에서 디바이스는 양자 처리 장치(QPU) 또는 양자 작업을 실행하기 위해 호출할 수 있는 시뮬레이터를 나타냅니다. Amazon Braket은 AQT, , IonQ, 및에서 QPU 디바이스에 대한 액세스를 제공합니다IQMQuEraRigetti. 또한는 온디맨드, 로컬 및 임베디드 시뮬레이터에 대한 액세스를 AWS 제공합니다. 임베디드 시뮬레이터에 대한 자세한 내용은 임베디드 시뮬레이터 정보를 참조하세요.

지원되는 양자 하드웨어 공급자에 대한 자세한 내용은 QPU에 양자 작업 제출을 참조하세요. 사용 가능한 시뮬레이터에 대한 자세한 내용은 시뮬레이터에 양자 작업 제출을 참조하세요. 다음 표에는 사용 가능한 디바이스 및 시뮬레이터 목록이 표시됩니다.

제공업체	디바이스 이름	패러다임	Type	디바이스 ARN	리전
AQT	IBEX-Q1	게이트 기반	QPU	arn:aws:braket:eu-north-1::device/qpu/aqt/Ibex-Q1	eu-north-1
IonQ	Aria-1	게이트 기반	QPU	arn:aws:braket:us-east-1::device/qpu/ionq/Aria-1	us-east-1
IonQ	Forte-1	게이트 기반	QPU	arn:aws:braket:us-east-1::device/qpu/ionq/Forte-1	us-east-1
IonQ	Forte-Enterprise-1	게이트 기반	QPU	arn:aws:braket:us-east-1::device/qpu/ionq/Forte-Enterprise-1	us-east-1
IQM	Garnet	게이트 기반	QPU	arn:aws:braket:eu-north-1::device/qpu/iqm/Garnet	eu-north-1
IQM	Emerald	게이트 기반	QPU	arn:aws:braket:eu-north-1::device/qpu/iqm/Emerald	eu-north-1
QuEra	Aquila	아날로그 해밀토니안 시뮬레이션	QPU	arn:aws:braket:us-east-1::device/qpu/quera/Aquila	us-east-1
Rigetti	Ankaa-3	게이트 기반	QPU	arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3	us-west-1
AWS	Braket_sv	게이트 기반	로컬 시뮬레이터	N/A(Braket SDK의 로컬 시뮬레이터)	해당 사항 없음

제공업체	디바이스 이름	패러다임	Type	디바이스 ARN	리전
AWS	Braket_dm	게이트 기반	로컬 시뮬레이터	N/A(Braket SDK의 로컬 시뮬레이터)	해당 사항 없음
AWS	braket_ahs	아날로그 해밀토니안 시뮬레이션	로컬 시뮬레이터	N/A(Braket SDK의 로컬 시뮬레이터)	해당 사항 없음
AWS	SV1	게이트 기반	온디맨드 시뮬레이터	arn:aws:braket:::device/quantum-simulator/amazon/sv1	us-east-1, us-west-1, us-west-2, eu-west-2
AWS	DM1	게이트 기반	온디맨드 시뮬레이터	arn:aws:braket:::device/quantum-simulator/amazon/dm1	us-east-1, us-west-1, us-west-2, eu-west-2
AWS	TN1	게이트 기반	온디맨드 시뮬레이터	arn:aws:braket:::device/quantum-simulator/amazon/tn1	us-east-1, us-west-2 및 eu-west-2

> **Note**
>
> 디바이스 ARNs. 예를 들어 AQT IBEX-Q1 디바이스를 사용할 때 디바이스 ARN에가 포함되어 있는지 확인합니다'Ibex-Q1'.

Amazon Braket에서 사용할 수 있는 QPU에 대한 추가 세부 정보는 Amazon Braket 양자 컴퓨터를 참조하세요.

디바이스 속성

모든 디바이스에 대해 Amazon Braket 콘솔의 디바이스 탭 또는 `GetDevice` API를 통해 디바이스 토폴로지, 보정 데이터 및 네이티브 게이트 세트와 같은 추가 디바이스 속성을 찾을 수 있습니다. Amazon Braket에서는 시뮬레이터로 회로를 구성할 때 연속 큐비트 또는 인덱스를 사용해야 합니다. SDK로 작업할 때 다음 코드 예제에서는 사용 가능한 각 디바이스 및 시뮬레이터의 디바이스 속성에 액세스하는 방법을 보여줍니다.

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

device = AwsDevice('arn:aws:braket:::device/quantum-simulator/amazon/sv1')
 # SV1
# device = LocalSimulator()
 # Local State Vector Simulator
# device = LocalSimulator("default")
 # Local State Vector Simulator
# device = LocalSimulator(backend="default")
 # Local State Vector Simulator
# device = LocalSimulator(backend="braket_sv")
 # Local State Vector Simulator
# device = LocalSimulator(backend="braket_dm")
 # Local Density Matrix Simulator
# device = LocalSimulator(backend="braket_ahs")
 # Local Analog Hamiltonian Simulation
# device = AwsDevice('arn:aws:braket:::device/quantum-simulator/amazon/tn1')
 # TN1
# device = AwsDevice('arn:aws:braket:::device/quantum-simulator/amazon/dm1')
 # DM1
# device = AwsDevice('arn:aws:braket:eu-north-1::device/qpu/aqt/Ibex-Q1')
 # AQT IBEX-Q1
# device = AwsDevice('arn:aws:braket:us-east-1::device/qpu/ionq/Aria-1')
 # IonQ Aria-1
# device = AwsDevice('arn:aws:braket:us-east-1::device/qpu/ionq/Forte-1')
 # IonQ Forte-1
# device = AwsDevice('arn:aws:braket:us-east-1::device/qpu/ionq/Forte-Enterprise-1')
 # IonQ Forte-Enterprise-1
# device = AwsDevice('arn:aws:braket:eu-north-1::device/qpu/iqm/Garnet')
 # IQM Garnet
```

```
# device = AwsDevice('arn:aws:braket:eu-north-1::device/qpu/iqm/Emerald')
 # IQM Emerald
# device = AwsDevice('arn:aws:braket:us-east-1::device/qpu/quera/Aquila')
 # QuEra Aquila
# device = AwsDevice('arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3')
 # Rigetti Ankaa-3

# Get device properties
device.properties
```

Amazon Braket의 리전 및 엔드포인트

리전 및 엔드포인트의 전체 목록은 AWS 일반 참조를 참조하세요.

QPU 디바이스에서 실행되는 양자 작업은 해당 디바이스의 리전에 있는 Amazon Braket 콘솔에서 볼 수 있습니다. Amazon Braket SDK를 사용하는 경우 작업 중인 리전에 관계없이 QPU 디바이스에 양자 작업을 제출할 수 있습니다. SDK는 지정된 QPU에 대해 리전에 대한 세션을 자동으로 생성합니다.

Amazon Braket은 AWS 리전다음에서 사용할 수 있습니다.

리전 이름	리전	Braket 엔드포인트
미국 동부(버지니아 북부)	us-east-1	braket.us-east-1.amazonaws.com(IPv4 전용) braket.us-east-1.api.aws(듀얼 스택)
미국 서부(캘리포니아 북부)	us-west-1	braket.us-west-1.amazonaws.com(IPv4 전용) braket.us-west-1.api.aws(듀얼 스택)
US West 2(오레곤)	us-west-2	braket.us-west-2.amazonaws.com(IPv4 전용) braket.us-west-2.api.aws(듀얼 스택)

리전 이름	리전	Braket 엔드포인트
EU North 1(스톡홀름)	eu-north-1	braket.eu-north-1.amazonaws.com(IPv4 전용) braket.eu-north-1.api.aws(듀얼 스택)
EU West 2(런던)	eu-west-2	braket.eu-west-2.amazonaws.com(IPv4 전용) braket.eu-west-2.api.aws(듀얼 스택)

> **Note**
>
> Amazon Braket SDK는 IPv6 전용 네트워크를 지원하지 않습니다.

Amazon Braket 시작하기

> **Tip**
>
> 를 사용하여 양자 컴퓨팅의 기초를 알아봅니다 AWS! Amazon Braket Digital Learning Plan에 등록하고 일련의 학습 과정과 디지털 평가를 완료한 후 나만의 디지털 배지를 획득하세요.

Amazon Braket 활성화의 지침을 따른 후 Amazon Braket을 시작할 수 있습니다.

시작하는 단계는 다음과 같습니다.

- Amazon Braket 활성화
- Amazon Braket 노트북 인스턴스 생성
- 를 사용하여 Braket 노트북 인스턴스 생성 CloudFormation

Amazon Braket 활성화

> **Tip**
>
> 를 사용하여 양자 컴퓨팅의 기초를 알아봅니다 AWS! Amazon Braket Digital Learning Plan에 등록하고 일련의 학습 과정과 디지털 평가를 완료한 후 나만의 디지털 배지를 획득하세요.

AWS 콘솔을 통해 계정에서 Amazon Braket을 활성화할 수 있습니다.

이 섹션의 내용:

- 사전 조건
- Amazon Braket을 활성화하는 단계

사전 조건

Amazon Braket을 활성화하고 실행하려면 Amazon Braket 작업을 시작할 권한이 있는 사용자 또는 역할이 있어야 합니다. 이러한 권한은 AmazonBraketFullAccess IAM 정책(arn:aws:iam::aws:policy/AmazonBraketFullAccess)에 포함됩니다.

> **Note**
>
> 관리자인 경우:
> 다른 사용자에게 Amazon Braket에 대한 액세스 권한을 부여하려면 AmazonBraketFullAccess 정책을 연결하거나 생성한 사용자 지정 정책을 연결하여 사용자에게 권한을 부여합니다. Amazon Braket을 사용하는 데 필요한 권한에 대한 자세한 내용은 Amazon Braket에 대한 액세스 관리를 참조하세요.

Amazon Braket을 활성화하는 단계

1. 를 사용하여 Amazon Braket 콘솔에 로그인합니다 AWS 계정.
2. Amazon Braket 콘솔을 엽니다.
3. Braket 랜딩 페이지에서 시작하기를 클릭하여 서비스 대시보드 페이지로 이동합니다. 서비스 대시보드 상단의 알림은 다음 세 단계를 안내합니다.
   a. 서비스 연결 역할(SLR) 생성
   b. 타사 양자 컴퓨터에 대한 액세스 활성화
   c. 새 Jupyter Notebook 인스턴스 생성

타사 양자 디바이스를 사용하려면 자신과 해당 디바이스 간의 데이터 전송 AWS과 관련된 특정 조건에 동의해야 합니다. 이 계약의 이용 약관은 Amazon Braket 콘솔의 권한 및 설정 페이지에 있는 일반 탭에 나와 있습니다.

> **Note**
>
> Braket 로컬 시뮬레이터 또는 온디맨드 시뮬레이터와 같이 타사와 관련이 없는 양자 디바이스는 타사 디바이스 활성화 계약에 동의하지 않아도 사용할 수 있습니다.
> 타사 하드웨어에 액세스하는 경우, 타사 디바이스 사용을 허용하기 위한 이 약관 동의는 계정당 한 번만 수행하면 됩니다.

Amazon Braket 노트북 인스턴스 생성

> **Tip**
>
> 를 사용하여 양자 컴퓨팅의 기초를 알아봅니다 AWS! Amazon Braket Digital Learning Plan에 등록하고 일련의 학습 과정과 디지털 평가를 완료한 후 나만의 디지털 배지를 획득하세요.

Amazon Braket은 시작하는 데 도움이 되는 완전관리형 Jupyter Notebook을 제공합니다. Amazon Braket 노트북 인스턴스는 Amazon SageMaker AI 노트북 인스턴스를 기반으로 합니다. 다음 단계에서는 신규 및 기존 고객을 위한 새 노트북 인스턴스를 생성하는 방법을 간략하게 설명합니다.

신규 Amazon Braket 고객:

1. Amazon Braket 콘솔을 열고 왼쪽 창의 대시보드 페이지로 이동합니다.
2. 대시보드 페이지 중앙의 Amazon Braket 시작 모달에서 시작하기를 클릭합니다. 노트북 이름을 제공하여 기본 Jupyter Notebook을 생성합니다.
   a. 노트북을 생성하는 데 몇 분 정도 걸릴 수 있습니다.
   b. 노트북 페이지에 노트북이 보류 중 상태로 나열됩니다.
   c. 노트북 인스턴스를 사용할 준비가 되면 상태가 InService로 변경됩니다.
   d. 페이지를 새로 고쳐 노트북의 업데이트된 상태를 표시합니다.

기존 Amazon Braket 고객:

1. Amazon Braket 콘솔을 열고 왼쪽 창에서 노트북을 선택합니다.
2. 노트북 인스턴스 생성을 선택합니다.
   a. 노트북이 없는 경우 표준 설정을 선택하여 기본 Jupyter Notebook을 생성합니다.
3. 영숫자와 하이픈 문자만 사용하여 노트북 인스턴스 이름을 입력하고 원하는 시각적 모드를 선택합니다.
4. 노트북의 노트북 비활성 관리자를 활성화하거나 비활성화합니다.
   a. 활성화한 경우, 노트북을 초기화하기 전에 원하는 유휴 기간을 선택합니다. 노트북을 초기화하면 컴퓨팅 요금이 더 이상 발생하지 않지만, 스토리지 요금은 계속 부과됩니다.
   b. 노트북 인스턴스의 유휴 시간을 확인하려면 명령 모음으로 이동하여 Braket 탭을 선택한 다음 비활성 관리자 탭을 선택합니다.

> **Note**
>
> 작업을 저장하려면 SageMaker AI 노트북 인스턴스를 Git 리포지토리와 통합합니다. 또는 노트북 인스턴스가 다시 시작될 때 덮어쓰기 되지 않도록 `/Braket Algorithms` 및 `/Braket Examples` 폴더 밖으로 작업을 옮깁니다.

5. (선택 사항) 고급 설정을 사용하면 액세스 권한, 추가 구성 및 네트워크 액세스 설정을 사용하여 노트북을 생성할 수 있습니다.
   a. 노트북 구성에서 인스턴스 유형을 선택합니다.
      i. 비용 효율적인 표준 인스턴스 유형인 ml.t3.medium이 기본적으로 선택됩니다. 인스턴스 요금에 대한 자세한 내용은 Amazon SageMaker AI 요금을 참조하세요.
   b. 공개 Github 리포지토리를 노트북 인스턴스와 연결하려면 Git 리포지토리 드롭다운을 클릭하고 리포지토리 드롭다운 메뉴에서 URL에서 공개 git 리포지토리 복제를 선택합니다. Git 리포지토리 URL 텍스트 표시줄에 리포지토리의 URL을 입력합니다.
   c. 액세스 권한에서 선택적 IAM 역할, 루트 액세스 및 암호화 키를 구성합니다.
   d. 네트워크 액세스에서 Jupyter Notebook 인스턴스에 대한 사용자 지정 네트워크 및 액세스 설정을 구성합니다.
6. 설정을 검토하고 노트북 인스턴스를 식별하는 태그를 설정합니다. 시작을 클릭합니다.

> **Note**
>
> Amazon Braket 및 Amazon SageMaker AI 콘솔에서 Amazon Braket 노트북 인스턴스를 보고 관리합니다. SageMaker 콘솔을 통해 추가 Amazon Braket 노트북 설정을 사용할 수 있습니다.

Amazon Braket SDK 내의 AWS Amazon Braket 콘솔에서 작업하는 경우 플러그인은 생성한 노트북에 미리 로드됩니다. 자체 시스템에서 실행하려면 명령 `pip install amazon-braket-sdk`를 실행하거나 PennyLane 플러그인에 대한 명령 `pip install amazon-braket-pennylane-plugin`을 실행할 때 SDK 및 플러그인을 설치합니다.

를 사용하여 Braket 노트북 인스턴스 생성 CloudFormation

> **Tip**
>
> 를 사용하여 양자 컴퓨팅의 기초를 알아봅니다 AWS! Amazon Braket Digital Learning Plan에 등록하고 일련의 학습 과정과 디지털 평가를 완료한 후 나만의 디지털 배지를 획득하세요.

CloudFormation 를 사용하여 Amazon Braket 노트북 인스턴스를 관리할 수 있습니다. Braket 노트북 인스턴스는 Amazon SageMaker AI를 기반으로 빌드됩니다. CloudFormation을 사용하면 의도된 구성을 설명하는 템플릿 파일로 노트북 인스턴스를 프로비저닝할 수 있습니다. 템플릿 파일은 JSON 또는 YAML 형식으로 작성됩니다. 체계적이고 반복 가능한 방식으로 인스턴스를 생성, 업데이트 및 삭제할 수 있습니다. 이는 AWS 계정에서 여러 Braket 노트북 인스턴스를 관리할 때 유용할 수 있습니다.

Braket 노트북에 대한 CloudFormation 템플릿을 생성한 후를 사용하여 리소스를 CloudFormation 배포합니다. 자세한 내용은 CloudFormation 사용 설명서의 CloudFormation 콘솔에서 스택 생성을 참조하세요.

CloudFormation을 사용하여 Braket 노트북 인스턴스를 생성하려면 다음 세 단계를 수행합니다.

1. SageMaker AI 수명 주기 구성 스크립트를 생성합니다.
2. SageMaker AI가 수임할 AWS Identity and Access Management (IAM) 역할을 생성합니다.
3. 접두사 **`amazon-braket-`**이 붙은 SageMaker AI 노트북 인스턴스을 생성합니다.

생성하는 모든 Braket 노트북에 수명 주기 구성을 재사용할 수 있습니다. 동일한 실행 권한을 할당하는 Braket 노트북에 대해 IAM 역할을 재사용할 수도 있습니다.

이 섹션의 내용:

1단계: SageMaker AI 수명 주기 구성 생성

다음 템플릿을 사용하여 SageMaker AI 수명 주기 구성 스크립트를 생성합니다. 이 스크립트는 Braket에 맞게 SageMaker AI 노트북 인스턴스를 사용자 지정합니다. 수명 주기 CloudFormation 리소스의 구

성 옵션은 CloudFormation 사용자 가이드의 AWS::SageMaker::NotebookInstanceLifecycleConfig를 참조하세요.

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

            exit 0
```

2단계: Amazon SageMaker AI에서 수임하는 IAM 역할 생성

Braket 노트북 인스턴스를 사용하면 SageMaker AI가 사용자를 대신하여 연산을 수행합니다. 예를 들어, 지원되는 디바이스에서 회로를 사용하여 Braket 노트북을 실행한다고 가정해 보겠습니다. 노트북 인스턴스 내에서 SageMaker AI가 사용자 대신 Braket에서 연산을 실행합니다. 노트북 실행 역할은 SageMaker AI가 사용자를 대신하여 실행할 수 있는 정확한 연산을 정의합니다. 자세한 내용은 Amazon SageMaker AI 개발자 안내서의 SageMaker AI 역할을 참조하세요.

다음 예제를 사용하여 필요한 권한이 있는 Braket 노트북 실행 역할을 생성합니다. 필요에 따라 정책을 수정할 수 있습니다.

> **Note**
>
> 역할에 접두사 `braketnotebookcdk-"`가 붙은 Amazon S3 버킷의 `s3:ListBucket` 및 `s3:GetObject` 연산에 대한 권한이 있는지 확인합니다. Braket 노트북 설치 스크립트를 복사하려면 수명 주기 구성 스크립트에 이러한 권한이 필요합니다.

```
ExecutionRole:
    Type: "AWS::IAM::Role"
```

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

3단계: 접두사 **amazon-braket-**이 붙은 SageMaker AI 노트북 인스턴스 생성

SageMaker AI 수명 주기 스크립트와 1단계 및 2단계에서 생성한 IAM 역할을 사용하여 SageMaker AI 노트북 인스턴스를 생성합니다. 노트북 인스턴스는 Braket에 맞게 사용자 지정되며 Amazon Braket 콘솔을 사용하여 액세스할 수 있습니다. 이 CloudFormation 리소스의 구성 옵션에 대한 자세한 내용은 CloudFormation 사용자 가이드의 AWS::SageMaker::NotebookInstance를 참조하세요.

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

Amazon Braket을 사용하여 양자 작업 빌드

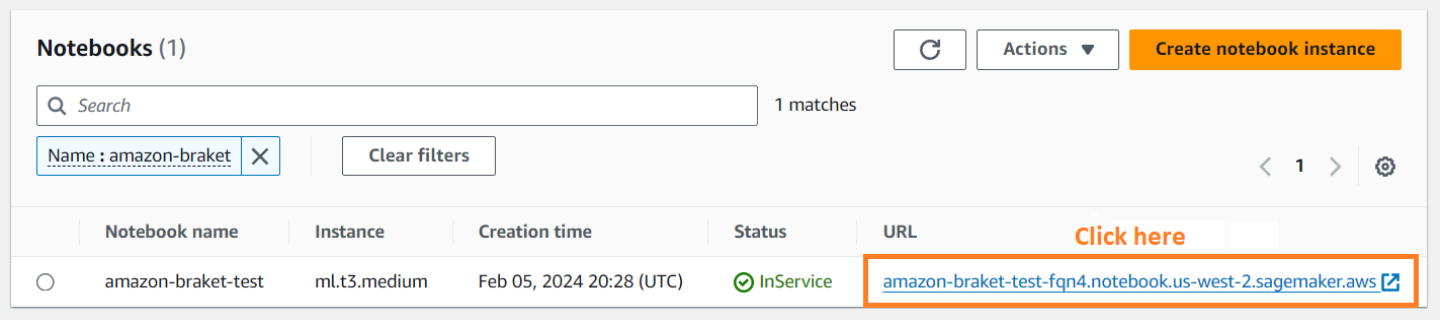
Braket은 간단하게 시작할 수 있는 완전관리형 Jupyter Notebook 환경을 제공합니다. Braket 노트북에는 Amazon Braket SDK를 비롯한 샘플 알고리즘, 리소스 및 개발자 도구가 사전 설치되어 있습니다. Amazon Braket SDK를 사용하면 양자 알고리즘을 빌드한 다음 단 한 줄의 코드만 변경하여 다양한 양자 컴퓨터 및 시뮬레이터에서 이를 테스트하고 실행할 수 있습니다.

이 섹션의 내용:

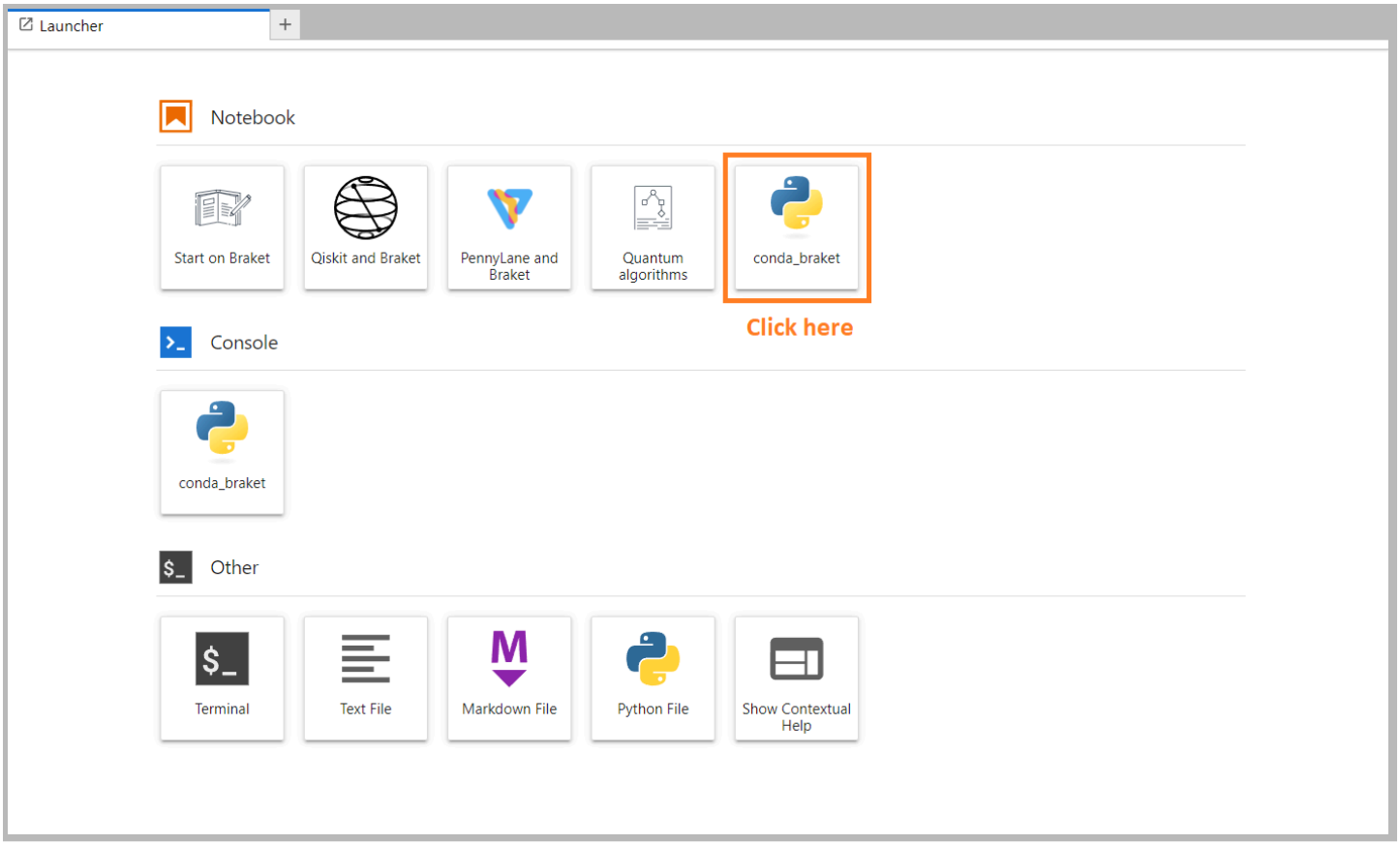
- 첫 번째 회로 빌드
- 전문가 조언 받기
- OpenQASM 3.0으로 회로 실행
- 실험 기능 살펴보기
- Amazon Braket의 펄스 제어
- 아날로그 해밀토니안 시뮬레이션
- AWS Boto3 작업

첫 번째 회로 빌드

노트북 인스턴스가 시작되고 나면, 방금 생성한 노트북을 선택하여 표준 Jupyter 인터페이스로 인스턴스를 엽니다.



Amazon Braket 노트북 인스턴스에는 Amazon Braket SDK 및 모든 종속성이 사전 설치되어 있습니다. 먼저 `conda_braket` 커널을 사용하여 새 노트북을 생성합니다.



간단한 “Hello, world!”로 시작할 수 있습니다. 예. 먼저 벨 상태를 준비하는 회로를 구성한 다음, 다른 디바이스에서 해당 회로를 실행하여 결과를 얻습니다.

먼저 Amazon Braket SDK 모듈을 가져오고 simpleBRAKETlong; SDK 모듈을 정의한 다음 기본 벨 상태 회로를 정의하여' 시작'을 가져와서 시작합니다.

```
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)
```

다음 명령을 사용하여 회로를 시각화할 수 있습니다.

```
print(bell)
```

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

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

로컬 시뮬레이터에서 회로 실행

다음으로, 회로를 실행할 양자 디바이스를 선택합니다. Amazon Braket SDK는 신속한 프로토타이핑 및 테스트를 위한 로컬 시뮬레이터와 함께 제공됩니다. 최대 25qubits(로컬 하드웨어에 따라 다름)까지 지원되는 더 작은 회로에는 로컬 시뮬레이터를 사용하는 것이 좋습니다.

로컬 시뮬레이터를 인스턴스화하려면 다음과 같이 한 다음,

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

회로를 실행합니다.

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

다음과 같은 결과가 표시되어야 합니다.

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

준비한 특정 벨 상태는 예상대로 |00# 및 |11#의 동등한 중첩이며 측정 결과로서 00 및 11의 분포가 거의 동등(최대 shot 노이즈)합니다.

온디맨드 시뮬레이터에서 회로 실행

또한 Amazon Braket은 대규모 회로 실행을 위한 온디맨드 고성능 시뮬레이터 SV1에 대한 액세스를 제공합니다. SV1은 최대 34qubits의 양자 회로 시뮬레이션을 허용하는 온디맨드 상태 벡터 시뮬레이터입니다. SV1에 대한 자세한 내용은 지원되는 디바이스 섹션 및 AWS 콘솔에서 확인할 수 있습니다. SV1(및 TN1 또는 QPU)에서 양자 작업을 실행할 때 양자 작업의 결과는 계정의 S3 버킷에 저장됩니다. 버킷을 지정하지 않으면 Braket SDK가 기본 버킷 `amazon-braket-{region}-{accountID}`를 생성합니다. 자세한 내용은 Amazon Braket에 대한 액세스 관리를 참조하세요.

> **Note**
>
> 다음 예제가 amazon-braket-s3-demo-bucket을 버킷 이름으로 표시하는 실제 기존 버킷 이름을 입력합니다. Amazon Braket의 버킷 이름은 항상 amazon-braket-으로 시작하며 그 뒤에 추가한 다른 식별 문자가 옵니다. S3 버킷을 설정하는 방법에 대한 자세한 내용은 Amazon S3 시작하기를 참조하세요.

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

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

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

SV1에서 회로를 실행하려면 이전에 선택한 S3 버킷의 위치를 .run() 직접 호출에서 위치 인수로 제공해야 합니다.

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

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

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

Amazon Braket 콘솔은 양자 작업에 대한 추가 정보를 제공합니다. 콘솔의 양자 작업 탭으로 이동하면 양자 작업이 목록 상단에 있어야 합니다. 아니면, 고유한 양자 작업 ID 또는 기타 기준을 사용하여 양자 작업을 검색할 수 있습니다.

> **Note**
>
> 90일이 지나면 Amazon Braket은 양자 작업과 연결된 모든 양자 작업 ID 및 기타 메타데이터를 자동으로 제거합니다. 자세한 내용은 데이터 보존을 참조하세요.

QPU에서 실행

Amazon Braket을 사용하면 단 한 줄의 코드만 변경하여 물리적 양자 컴퓨터에서 이전 양자 회로 예제를 실행할 수 있습니다. Amazon Braket은 다양한 Quantum 처리 장치(QPU) 디바이스에 대한 액세스를 제공합니다. 지원되는 디바이스 섹션과 AWS 콘솔의 디바이스 탭에서 다양한 디바이스 및 가용성 기간에 대한 정보를 찾을 수 있습니다. 다음 예제에서는 IQM 디바이스를 인스턴스화하는 방법을 보여줍니다.

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

또는 이 코드가 있는 IonQ 디바이스를 선택합니다.

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

디바이스를 선택한 후 워크로드를 실행하기 전에, 다음 코드로 디바이스 대기열 깊이를 쿼리하여 양자 작업 또는 하이브리드 작업 수를 확인할 수 있습니다. 또한 고객은 Amazon Braket Management Console의 디바이스 페이지에서 디바이스별 대기열 깊이를 볼 수 있습니다.

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

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

작업을 실행하면 Amazon Braket SDK가 결과를 폴링합니다(기본 시간 초과는 5일). 다음 예제와 같이, `.run()` 명령에서 `poll_timeout_seconds` 파라미터를 수정하여 이 기본값을 변경할 수 있습니다. 폴링 시간 초과가 너무 짧으면 QPU를 사용할 수 없고 로컬 시간 초과 오류가 반환되는 경우와 같이 폴링 시간 내에 결과가 반환되지 않을 수 있다는 점을 유의하세요. `task.result()` 함수를 직접적으로 호출하여 폴링을 다시 시작할 수 있습니다.

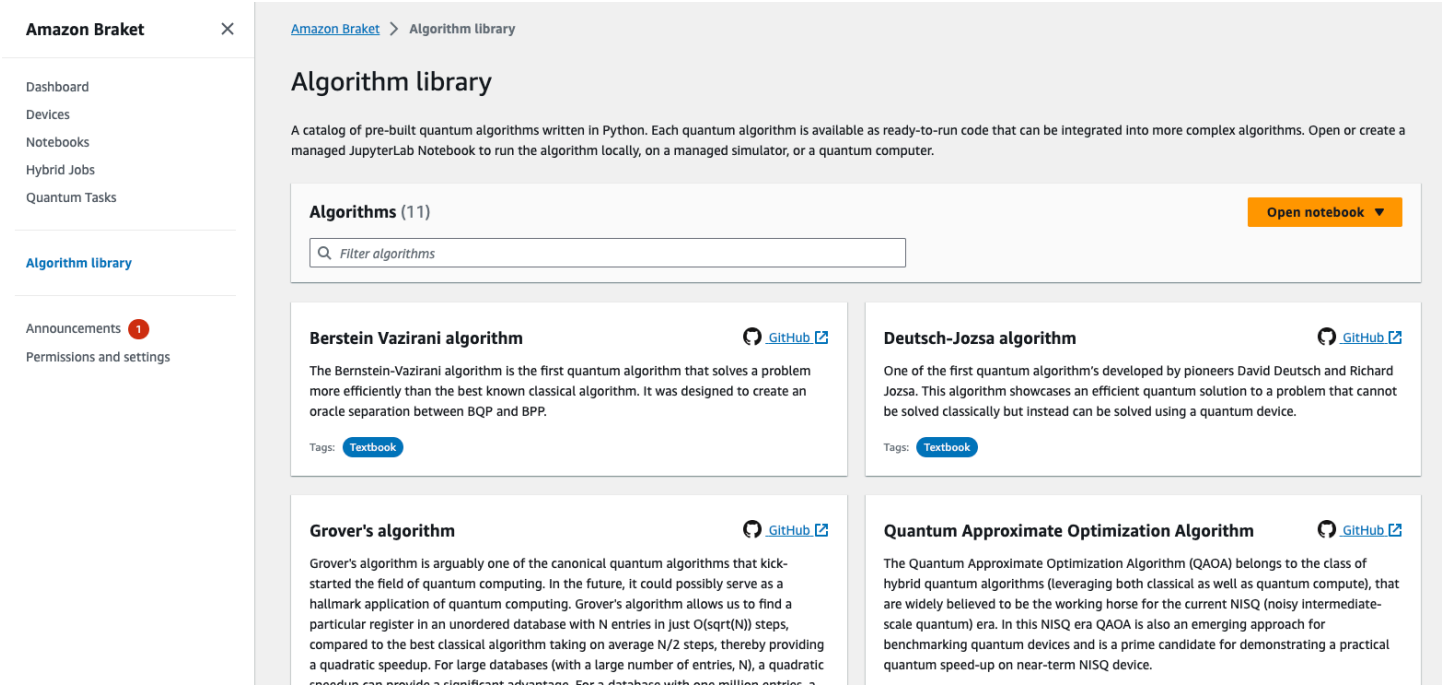
```
# Define quantum task with 1 day polling timeout
task = device.run(bell, s3_folder, poll_timeout_seconds=24*60*60)
print(task.result().measurement_counts)
```

또한 양자 작업 또는 하이브리드 작업을 제출한 후 queue_position() 함수를 직접적으로 호출하여 대기열 위치를 확인할 수 있습니다.

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

첫 번째 양자 알고리즘 빌드

Amazon Braket 알고리즘 라이브러리는 Python으로 작성된 사전 빌드된 양자 알고리즘의 카탈로그입니다. 이러한 알고리즘을 그대로 실행하거나 더 복잡한 알고리즘을 빌드하기 위한 출발점으로 활용할 수 있습니다. Braket 콘솔에서 알고리즘 라이브러리에 액세스할 수 있습니다. 자세한 내용은 Braket Github 알고리즘 라이브러리를 참조하세요.



Braket 콘솔은 알고리즘 라이브러리에서 사용 가능한 각 알고리즘에 대한 설명을 제공합니다. GitHub 링크를 선택하여 각 알고리즘의 세부 정보를 보거나, 노트북 열기를 선택하여 열거나 사용 가능한 모든 알고리즘이 포함된 노트북을 생성합니다. 노트북 옵션을 선택하면 노트북의 루트 폴더에서 Braket 알고리즘 라이브러리를 찾을 수 있습니다.

SDK에서 회로 구성

이 섹션에서는 회로 정의, 사용 가능한 게이트 보기, 회로 확장, 각 디바이스가 지원하는 게이트 보기의 예제를 제공합니다. 또한 qubits를 수동으로 할당하는 방법, 컴파일러가 정의된 대로 정확하게 회로를

실행하도록 지시하는 방법, 노이즈가 있는 시뮬레이터를 사용하여 노이즈가 있는 회로를 빌드하는 방법에 대한 지침도 포함되어 있습니다.

또한 특정 QPU를 사용하여 다양한 게이트에 대해 Braket에서 펄스 수준으로 작업할 수도 있습니다. 자세한 내용은 Amazon Braket의 펄스 제어를 참조하세요.

이 섹션의 내용:

- 게이트 및 회로
- 프로그램 세트
- 부분 측정
- 수동 qubit 할당
- 축어적 컴파일
- 노이즈 시뮬레이션

게이트 및 회로

양자 게이트 및 회로는 Amazon Braket Python SDK의 `braket.circuits` 클래스에 정의됩니다. SDK에서 `Circuit()`을 직접적으로 호출하여 새 회로 객체를 인스턴스화할 수 있습니다.

예: 회로 정의

이 예제는 표준, 단일 큐비트 아다마르 게이트 및 2큐비트 CNOT 게이트로 구성된 4qubits(q0, q1, q2 및 q3으로 레이블 지정)의 샘플 회로를 정의하는 것으로 시작합니다. 다음 예제와 같이 `print` 함수를 직접적으로 호출하여 이 회로를 시각화할 수 있습니다.

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

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

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

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

예: 파라미터화된 회로 정의

이 예제에서는 자유 파라미터에 따라 달라지는 게이트가 있는 회로를 정의합니다. 이러한 파라미터의 값을 지정하여 새 회로를 생성하거나, 회로를 제출할 때 특정 디바이스에서 양자 작업으로 실행할 수 있습니다.

```
from braket.circuits import Circuit, FreeParameter

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

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

다음과 같이 단일 `float`(모든 자유 파라미터가 취하는 값) 또는 각 파라미터의 값을 지정하는 키워드 인수를 회로에 제공하여 파라미터화된 회로에서 파라미터화되지 않은 새 회로를 생성할 수 있습니다.

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

`my_circuit`은 수정되지 않으므로 이를 사용하여 고정된 파라미터 값으로 많은 새 회로를 인스턴스화할 수 있습니다.

예: 회로의 게이트 수정

다음 예제에서는 제어 및 전원 한정자를 사용하는 게이트가 있는 회로를 정의합니다. 이러한 수정 사항을 사용하여 제어된 Ry 게이트와 같은 새 게이트를 생성할 수 있습니다.

```
from braket.circuits import Circuit

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

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

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

print(my_circuit)
```

게이트 한정자는 로컬 시뮬레이터에서만 지원됩니다.

예: 사용 가능한 모든 게이트 보기

다음 예제에서는 Amazon Braket에서 사용 가능한 모든 게이트를 보는 방법을 보여줍니다.

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

이 코드의 출력에는 모든 게이트가 나열됩니다.

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

이러한 게이트는 해당 유형의 회로에 대한 메서드를 직접적으로 호출하여 회로에 추가할 수 있습니다. 예를 들어 `circ.h(0)`를 직접적으로 호출하여 첫 번째 qubit에 아다마르 게이트를 추가합니다.

> **Note**
>
> 게이트가 제자리에 추가되며, 다음 예제에서는 이전 예제에 나열된 모든 게이트를 동일한 회로에 추가합니다.

```
circ = Circuit()
# toffoli gate with q0, q1 the control qubits and q2 the target.
circ.ccnot(0, 1, 2)
# cnot gate
circ.cnot(0, 1)
```

```
# controlled-phase gate that phases the |11> state, cphaseshift(phi) =
 diag((1,1,1,exp(1j*phi))), where phi=0.15 in the examples below
circ.cphaseshift(0, 1, 0.15)
# controlled-phase gate that phases the |00> state, cphaseshift00(phi) =
 diag([exp(1j*phi),1,1,1])
circ.cphaseshift00(0, 1, 0.15)
# controlled-phase gate that phases the |01> state, cphaseshift01(phi) =
 diag([1,exp(1j*phi),1,1])
circ.cphaseshift01(0, 1, 0.15)
# controlled-phase gate that phases the |10> state, cphaseshift10(phi) =
 diag([1,1,exp(1j*phi),1])
circ.cphaseshift10(0, 1, 0.15)
# controlled swap gate
circ.cswap(0, 1, 2)
# swap gate
circ.swap(0,1)
# phaseshift(phi)= diag([1,exp(1j*phi)])
circ.phaseshift(0,0.15)
# controlled Y gate
circ.cy(0, 1)
# controlled phase gate
circ.cz(0, 1)
# Echoed cross-resonance gate applied to q0, q1
circ = Circuit().ecr(0,1)
# X rotation with angle 0.15
circ.rx(0, 0.15)
# Y rotation with angle 0.15
circ.ry(0, 0.15)
# Z rotation with angle 0.15
circ.rz(0, 0.15)
# Hadamard gates applied to q0, q1, q2
circ.h(range(3))
# identity gates applied to q0, q1, q2
circ.i([0, 1, 2])
# iswap gate, iswap = [[1,0,0,0],[0,0,1j,0],[0,1j,0,0],[0,0,0,1]]
circ.iswap(0, 1)
# pswap gate, PSWAP(phi) = [[1,0,0,0],[0,0,exp(1j*phi),0],[0,exp(1j*phi),0,0],
[0,0,0,1]]
circ.pswap(0, 1, 0.15)
# X gate applied to q1, q2
circ.x([1, 2])
# Y gate applied to q1, q2
circ.y([1, 2])
# Z gate applied to q1, q2
```

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

사전 정의된 게이트 세트 외에도, 자체 정의된 단위 게이트를 회로에 적용할 수도 있습니다. 이는 `targets` 파라미터에 의해 정의된 qubits에 적용되는 단일 큐비트 게이트(다음 소스 코드에 표시됨) 또는 다중 큐비트 게이트일 수 있습니다.

```
import numpy as np

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

예: 기존 회로 확장

명령을 추가하여 기존 회로를 확장할 수 있습니다. `Instruction`은 양자 디바이스에서 수행할 양자 작업을 설명하는 양자 지시어입니다. `Instruction` 연산자에는 유형 `Gate`의 객체만 포함됩니다.

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

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

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

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

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

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

예: 각 디바이스가 지원하는 게이트 보기

시뮬레이터는 Braket SDK의 모든 게이트를 지원하지만, QPU 디바이스는 더 작은 부분 집합을 지원합니다. 디바이스 속성에서 디바이스의 지원되는 게이트를 확인할 수 있습니다. 다음은 IonQ 디바이스의 예제입니다.

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

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

# Get device name
device_name = device.name
# Show supportedQuantumOperations (supported gates for a device)
device_operations = device.properties.dict()['action']['braket.ir.openqasm.program']
['supportedOperations']
```

```
print('Quantum Gates supported by {}:\n {}'.format(device_name, device_operations))
```

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

지원되는 게이트는 양자 하드웨어에서 실행되기 전에 네이티브 게이트로 컴파일해야 할 수 있습니다. 회로를 제출하면 Amazon Braket이 이 컴파일을 자동으로 수행합니다.

예: 디바이스에서 지원하는 네이티브 게이트의 충실도를 프로그래밍 방식으로 검색

Braket 콘솔의 디바이스 페이지에서 충실도 정보를 볼 수 있습니다. 프로그래밍 방식으로 동일한 정보에 액세스하는 것이 도움이 되는 경우도 있습니다. 다음 코드는 QPU의 두 게이트 간에 두 qubit 게이트 충실도를 추출하는 방법을 보여줍니다.

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

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

# Specify the qubits
a=10
b=11
edge_properties_entry =
 device.properties.standardized.twoQubitProperties['10-11'].twoQubitGateFidelity
gate_name = edge_properties_entry[0].gateName
fidelity = edge_properties_entry[0].fidelity
print(f"Fidelity of the {gate_name} gate between qubits {a} and {b}: {fidelity}")
```

프로그램 세트

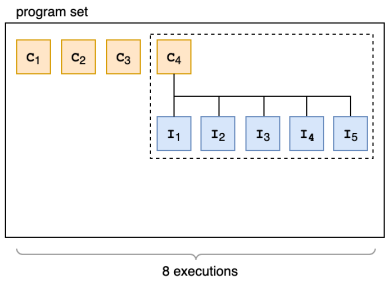
프로그램 세트는 단일 양자 작업에서 여러 양자 회로를 효율적으로 실행합니다. 해당 작업에서는 최대 100개의 양자 회로 또는 최대 100개의 서로 다른 파라미터 세트가 있는 단일 파라메트릭 회로를 제출할 수 있습니다. 이 연산은 후속 회로 실행 사이의 시간을 최소화하고 양자 작업 처리 오버헤드를 줄여줍니다. 현재 프로그램 세트는 Amazon Braket Local Simulator 및 AQT, IQM 및 Rigetti 디바이스에서 지원됩니다.

ProgramSet 정의

다음 첫 번째 코드 예제에서는 파라미터화된 회로와 파라미터가 없는 회로를 모두 사용하여 `ProgramSet`를 빌드하는 방법을 보여줍니다.

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

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

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

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

# Creating the program set
program_set_1 = ProgramSet([
    circ1,
    circ2,
    circ3,
    circuit_binding,
])
```

이 프로그램 세트에는 `circ1`, `circ2`, `circ3`, `circuit_binding`의 네 가지 고유한 프로그램이 포함되어 있습니다. `circuit_binding` 프로그램은 5개의 서로 다른 파라미터 바인딩으로 실행되어 5개의 실행 파일을 생성합니다. 다른 3개의 파라미터 없는 프로그램은 각각 하나의 실행 파일을 생성합니다. 그러면 다음 이미지와 같이 총 8개의 실행 파일이 생성됩니다.



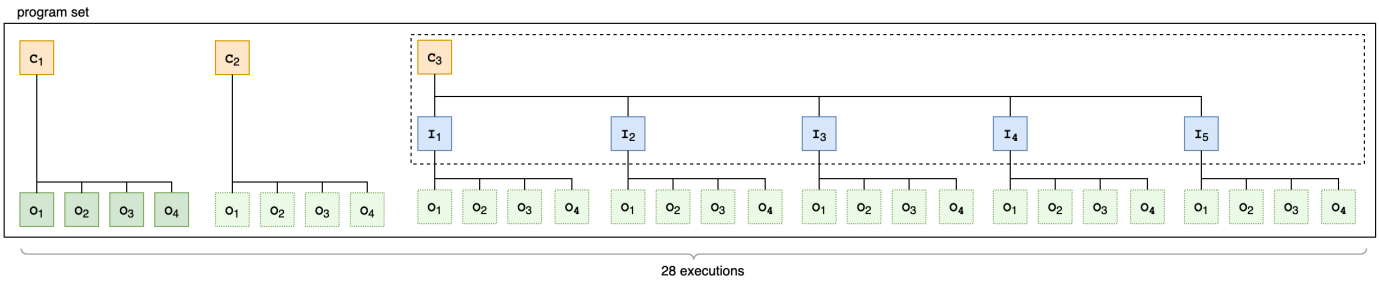

```
ProgramSet(
    [c1,
     c2,
     c3,
     CircuitBinding(c4, [i1, i2, i3, i4, i5])]
)
```

다음 두 번째 코드 예제에서는 `product()` 메서드를 사용하여 프로그램 세트의 각 실행 파일에 동일한 관찰 가능 항목 세트를 연결하는 방법을 보여줍니다.

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

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

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

파라미터 없는 프로그램의 경우 각 관찰 가능 항목은 각 회로에 대해 측정됩니다. 파라메트릭 프로그램의 경우 다음 이미지에서와 같이 각 관찰 가능 항목은 각 입력 세트에 대해 측정됩니다.

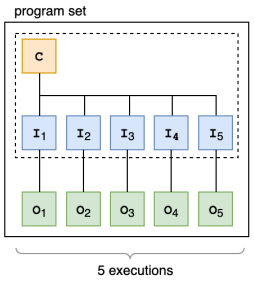
```
ProgramSet.product(
    circuits=[c1, c2, CircuitBinding(c3, [i1, i2, i3, i4, i5])],
    observables=[o1, o2, o3, o4]
)
```



다음 세 번째 코드 예제에서는 `zip()` 메서드를 사용하여 개별 관찰 가능 항목을 `ProgramSet`의 특정 파라미터 세트와 페어링하는 방법을 보여줍니다.

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

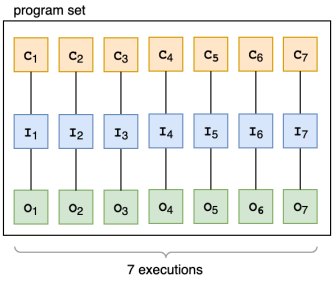
```
)
```




```
ProgramSet.zip(
    circuits=CicruitBinding(
        circuit=c,
        input_sets=[i1, i2, i3, i4, i5]
    ),
    observables=[o1, o2, o3, o4, o5]
)
```

`CircuitBinding()` 대신, 관찰 가능 항목의 목록을 회로 및 입력 세트 목록과 함께 직접 압축할 수 있습니다.

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




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

프로그램 세트에 대한 자세한 내용과 예제는 amazon-braket-examples Github의 프로그램 세트 폴더를 참조하세요.

디바이스에서 프로그램 세트 검사 및 실행

프로그램 세트의 실행 파일 수는 고유한 파라미터 바인딩 회로 수와 같습니다. 다음 코드 예제를 사용하여 회로 실행 파일 및 샷의 총 수를 계산합니다.

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

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

> **Note**
>
> 프로그램 세트를 사용하면 프로그램 세트의 모든 회로에서 총 샷 수를 기준으로 단일 작업당 요금과 샷당 요금을 지불하게 됩니다.

프로그램 세트를 실행하려면 다음 코드 예제를 사용합니다.

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

Rigetti 디바이스를 사용하는 경우 작업이 부분적으로 완료되고 부분적으로 대기열에 있는 동안 프로그램 세트가 RUNNING 상태로 유지될 수 있습니다. 결과를 더 빨리 얻으려면 프로그램 세트를 하이브리드 작업으로 제출하는 것을 고려하세요.

결과 분석

다음 코드를 실행하여 ProgramSet에서 실행 파일의 결과를 분석하고 측정합니다.

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

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

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

부분 측정

양자 회로의 모든 큐비트를 측정하는 대신 부분 측정을 사용하여 개별 큐비트 또는 큐비트의 부분 집합을 측정합니다.

> **Note**
>
> 중간 회로 측정 및 피드포워드 연산과 같은 추가 기능은 실험 기능으로 사용할 수 있습니다. IQM 디바이스의 동적 회로에 대한 액세스를 참조하세요.

예: 큐비트의 부분 집합 측정

다음 코드 예제는 벨 상태 회로에서 큐비트 0만 측정하여 부분 측정을 보여줍니다.

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

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

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

# Run the circuit
task = device.run(circuit, shots=10)

# Get the results
result = task.result()

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

수동 qubit 할당

Rigetti의 양자 컴퓨터에서 양자 회로를 실행할 때 선택적으로 수동 qubit 할당을 사용하여 알고리즘에 어떤 qubits를 사용할지 제어할 수 있습니다. Amazon Braket 콘솔과 Amazon Braket SDK를 사용하면 선택한 QPU 디바이스의 최신 보정 데이터를 검사할 수 있으므로, 실험에 가장 적합한 qubits를 선택할 수 있습니다.

수동 qubit 할당을 사용하면 보다 정확하게 회로를 실행하고 개별 qubit 속성을 조사할 수 있습니다. 연구원과 고급 사용자는 최신 디바이스 보정 데이터를 기반으로 회로 설계를 최적화하고 더 정확한 결과를 얻을 수 있습니다.

다음 예제에서는 명시적으로 qubits를 할당하는 방법을 보여줍니다.

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

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

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

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

자세한 내용은 GitHub의 Amazon Braket 예제, 또는 특히 이 노트북: QPU 디바이스에서 큐비트 할당을 참조하세요.

축어적 컴파일

게이트 기반 양자 컴퓨터에서 양자 회로를 실행할 때 컴파일러가 수정 없이 정의된 대로 정확하게 회로를 실행하도록 지시할 수 있습니다. 축어적 컴파일을 사용하여 전체 회로를 지정된 대로 정확하게 보존하거나 특정 부분만 보존(Rigetti에서만 지원)하도록 지정할 수 있습니다. 하드웨어 벤치마킹 또는 오류 완화 프로토콜을 위한 알고리즘을 개발할 때는 하드웨어에서 실행 중인 게이트 및 회로 레이아웃을 정확하게 지정할 수 있는 옵션이 필요합니다. 축어적 컴파일을 사용하면 특정 최적화 단계를 꺼서 컴파일 프로세스를 직접 제어할 수 있으므로, 회로가 설계된 대로 정확하게 실행됩니다.

축어적 컴파일은 AQT, IQM, 및 Rigetti 디바이스에서 지원되며 IonQ네이티브 게이트를 사용해야 합니다. 축어적 컴파일을 사용하는 경우 디바이스의 토폴로지를 확인하여 게이트가 연결된 qubits에서 직접적으로 호출되고 회로가 하드웨어에서 지원되는 네이티브 게이트를 사용하는지 확인하는 것이 좋습니다. 다음 예제에서는 디바이스에서 지원하는 네이티브 게이트 목록에 프로그래밍 방식으로 액세스하는 방법을 보여줍니다.

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

Rigetti의 경우, 축어적 컴파일과 함께 사용할 수 있도록 `disableQubitRewiring=True`를 설정하여 qubit 재배선을 꺼야 합니다. 컴파일에서 축어적 상자를 사용할 때 `disableQubitRewiring=False`가 설정된 경우 양자 회로는 검증에 실패하고 실행되지 않습니다.

축어적 컴파일이 회로에 대해 활성화되어 있고 이를 지원하지 않는 QPU에서 실행되는 경우, 지원되지 않는 연산으로 인해 작업이 실패했음을 나타내는 오류가 생성됩니다. 더 많은 양자 하드웨어가 기본적

으로 컴파일러 함수를 지원함에 따라 이 기능은 이러한 디바이스를 포함하도록 확장될 예정입니다. 축어적 컴파일을 지원하는 디바이스에는 다음 코드로 쿼리할 때 지원되는 연산으로 포함됩니다.

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

축어적 컴파일 사용과 관련된 추가 비용은 없습니다. Amazon Braket 요금 페이지에 지정된 현재 요금을 기준으로 Braket QPU 디바이스, 노트북 인스턴스 및 온디맨드 시뮬레이터에서 실행되는 양자 작업에 대해서는 계속 요금이 부과됩니다. 자세한 내용은 축어적 컴파일 예제 노트북을 참조하세요.

> **Note**
>
> OpenQASM을 사용하여 AQT 및 IonQ 디바이스에 대한 회로를 쓰고 회로를 물리적 큐비트에 직접 매핑하려는 경우 OpenQASM에서 `disableQubitRewiring` 플래그를 무시`#pragma braket verbatim`하므로를 사용해야 합니다.

노이즈 시뮬레이션

로컬 노이즈 시뮬레이터를 인스턴스화하려면 다음과 같이 백엔드를 변경할 수 있습니다.

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

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

다음과 같은 두 가지 방법으로 노이즈가 있는 회로를 빌드할 수 있습니다.

1. 노이즈가 있는 회로를 상향식으로 빌드합니다.
2. 기존의 노이즈 없는 회로를 가져와 전체적으로 노이즈를 주입합니다.

다음 예제에서는 탈분극 노이즈가 있는 기본 회로와 사용자 지정 Kraus 채널을 사용하는 접근 방식을 보여줍니다.

```
import scipy.stats
import numpy as np
```

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

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

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

```
from braket.circuits import Noise

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

다음 두 예제에서와 같이 회로 실행은 이전과 동일한 사용자 환경입니다.

예제 1.

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

또는

예제 2.

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

추가 예제는 Braket 입문용 노이즈 시뮬레이터 예제를 참조하세요.

회로 검사

Amazon Braket의 양자 회로에는 `Moments`라고 하는 의사 시간 개념이 있습니다. 각 qubit는 `Moment`당 단일 게이트를 경험할 수 있습니다. `Moments`의 목적은 회로와 게이트를 더 쉽게 처리하고 시간 구조를 제공하는 것입니다.

> **Note**
>
> 모멘트는 일반적으로 QPU에서 게이트가 실행되는 실제 시간과 일치하지 않습니다.

회로의 깊이는 해당 회로의 총 모멘트 수에 따라 결정됩니다. 다음 예제와 같이 메서드 `circuit.depth`를 직접적으로 호출하는 회로 깊이를 볼 수 있습니다.

```
from braket.circuits import Circuit

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

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

위 회로의 총 회로 깊이는 3입니다(모멘트 `0`, `1` 및 `2`로 표시됨). 각 모멘트의 게이트 연산을 확인할 수 있습니다.

`Moments`는 키-값 쌍의 딕셔너리 역할을 합니다.

- 키는 의사 시간과 qubit 정보가 포함된 `MomentsKey()`입니다.
- 값은 `Instructions()` 유형으로 할당됩니다.

```
moments = circ.moments
```

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

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

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

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

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

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

Moments를 통해 회로에 게이트를 추가할 수도 있습니다.

```
from braket.circuits import Instruction, Gate

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

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

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

결과 유형 목록

Amazon Braket은 ResultType을 사용하여 회로를 측정할 때 다양한 유형의 결과를 반환할 수 있습니다. 회로는 다음과 같은 유형의 결과를 반환할 수 있습니다.

- AdjointGradient는 제공된 관찰 가능 항목의 기대값의 기울기(벡터 미분)를 반환합니다. 이 관찰 가능 항목은 수반 미분법을 사용하여 지정된 파라미터와 관련하여 제공된 대상에 작용합니다. 샷이 0인 경우에만 이 방법을 사용할 수 있습니다.
- Amplitude는 출력 파동 함수에서 지정된 양자 상태의 진폭을 반환합니다. SV1 및 로컬 시뮬레이터에서만 사용할 수 있습니다.
- Expectation은 지정된 관찰 가능 항목의 기대값을 반환하며,이 값은 이 장 뒷부분에 소개되는 Observable 클래스로 지정할 수 있습니다. 관찰 가능 항목을 측정하는 데 사용되는 대상 qubits를 지정해야 하며, 지정된 대상의 수는 관찰 가능 항목이 작용하는 qubits의 수와 같아야 합니다. 대상이 지정되지 않은 경우 관찰 가능 항목은 1qubit에서만 작동해야 하며 모든 qubits에 병렬로 적용됩니다.
- Probability는 계산 기저 상태를 측정할 확률을 반환합니다. 대상이 지정되지 않은 경우 Probability는 모든 기저 상태를 측정할 확률을 반환합니다. 대상이 지정되면 지정된 qubits에 있는 기저 벡터의 주변 확률만 반환됩니다. 관리형 시뮬레이터 및 QPU는 최대 15큐비트로 제한되며 로컬 시뮬레이터는 시스템의 메모리 크기로 제한됩니다.
- Reduced density matrix는 qubits의 시스템에서 지정된 대상 qubits의 하위 시스템에 대한 밀도 행렬을 반환합니다. 이 결과 유형의 크기를 제한하기 위해 Braket은 대상 qubits의 수를 최대 8개로 제한합니다.
- StateVector는 전체 상태 벡터를 반환합니다. 로컬 시뮬레이터에서 사용할 수 있습니다.
- Sample은 지정된 대상 qubit 세트 및 관찰 가능 항목의 측정 수를 반환합니다. 대상이 지정되지 않은 경우 관찰 가능 항목은 1qubit에서만 작동해야 하며 모든 qubits에 병렬로 적용됩니다. 대상이 지정된 경우 지정된 대상의 수는 관찰 가능 항목이 작용하는 qubits의 수와 같아야 합니다.
- Variance는 지정된 대상 qubit 세트 및 관찰 가능 항목의 분산(mean($[x-mean(x)]^2$))을 요청된 결과 유형으로 반환합니다. 대상이 지정되지 않은 경우 관찰 가능 항목은 1qubit에서만 작동해야 하

며 모든 qubits에 병렬로 적용됩니다. 그렇지 않으면 지정된 대상의 수는 관찰 가능 항목이 적용될 수 있는 qubits의 수와 같아야 합니다.

다양한 공급자에 대해 지원되는 결과 유형:

	로컬 sim	SV1	DM1	TN1	AQT	IonQ	IQM	Rigetti
수반 기울기	N	Y	N	N	N	N	N	N
진폭	Y	Y	N	N	N	N	N	N
기대값	Y	Y	Y	Y	Y	Y	Y	Y
확률	Y	Y	Y	N	Y	Y	Y	Y
축약 밀도 행렬	Y	N	Y	N	N	N	N	N
상태 벡터	Y	N	N	N	N	N	N	N
샘플	Y	Y	Y	Y	Y	Y	Y	Y
분산	Y	Y	Y	Y	Y	Y	Y	Y

다음 예제와 같이 디바이스 속성을 검토하여 지원되는 결과 유형을 확인할 수 있습니다.

```
from braket.aws import AwsDevice

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

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

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

`ResultType`을 직접적으로 호출하려면 다음 예제와 같이 회로에 추가합니다.

```
from braket.circuits import Circuit, Observable

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

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

> **Note**
>
> 다양한 양자 디바이스는 다양한 형식으로 결과를 제공합니다. 예를 들어 Rigetti 디바이스는 측정값을 반환하는 반면 IonQ 디바이스는 확률을 제공합니다. Amazon Braket SDK는 모든 결과에 대한 측정 속성을 제공합니다. 그러나 확률을 반환하는 디바이스의 경우, 샷당 측정값을 사용할 수 없으므로 이러한 측정값은 사후 계산되며 확률을 기반으로 합니다. 결과가 사후 계산되었는지 확인하려면 결과 객체에서 `measurements_copied_from_device`를 확인합니다. 이 연산은 Amazon Braket SDK GitHub 리포지토리의 gate_model_quantum_task_result.py 파일에 자세히 설명되어 있습니다.

관찰 가능 항목

Amazon Braket의 `Observable` 클래스를 사용하면 특정 관찰 가능 항목을 측정할 수 있습니다.

각 qubit에 하나의 고유한 비동일성 관찰 가능 항목만 적용할 수 있습니다. 동일한 qubit에 두 개 이상의 서로 다른 비동일성 관찰 가능 항목을 지정하는 경우 오류가 발생합니다. 이를 위해 텐서 곱의 각 인자는 개별 관찰 가능 항목으로 간주됩니다. 다시 말해, 해당 qubit에 작용하는 인자가 동일하게 유지되는 한, 동일한 qubit에 대해 여러 텐서 곱을 가질 수 있습니다.

관찰 가능 항목의 규모를 조정하고 다른 관찰 가능 항목을 추가할 수 있습니다(규모 조정 여부에 관계 없음). 그러면 AdjointGradient 결과 유형에 사용할 수 있는 Sum이 생성됩니다.

Observable 클래스에는 다음과 같은 관찰 가능 항목이 포함됩니다.

```
import numpy as np

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

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

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

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

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

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

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

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

파라미터

회로는 자유 파라미터를 포함할 수 있습니다. 이러한 자유 파라미터는 한 번만 구성하면 여러 번 실행할 수 있으며 기울기를 계산하는 데 사용할 수 있습니다.

각 자유 파라미터는 다음에 사용되는 문자열 인코딩 이름을 사용합니다.

- 파라미터 값 설정
- 사용할 파라미터 식별

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

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

수반 기울기

SV1 디바이스는 다항 해밀토니안을 포함한 관찰 가능 항목 기대값의 수반 기울기를 계산합니다. 파라미터를 미분하려면 이름(문자열 형식)을 지정하거나 직접 참조로 지정합니다.

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

device = AwsDevice(Devices.Amazon.SV1)

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

고정된 파라미터 값을 파라미터화된 회로에 인수로 전달하면 자유 파라미터가 제거됩니다. 자유 파라미터가 더 이상 존재하지 않으므로, 이 회로를 `AdjointGradient`로 실행하면 오류가 생성됩니다. 다음 코드 예제에서는 올바른 사용법과 잘못된 사용법을 보여줍니다.

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

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

전문가 조언 받기

Braket 관리 콘솔에서 직접 양자 컴퓨팅 전문가와 소통하면서 워크로드에 대한 추가 지침을 받아보세요.

Braket Direct를 통해 전문가 조언 옵션을 찾아보려면 Braket 콘솔을 열고 왼쪽 창에서 Braket Direct를 선택한 다음 전문가 조언 섹션으로 이동합니다. 다음과 같은 전문가 조언 옵션을 사용할 수 있습니다.

- Braket 상담 시간: Braket 상담 시간은 1:1 세션으로, 선착순이며 매월 진행됩니다. 이용 가능한 각 상담 시간 간격은 30분이며 무료입니다. Braket 전문가와의 상담을 통해 use-case-to-device 적합성을 살펴보고 알고리즘에 최적화된 Braket 활용 옵션을 모색하며 Amazon Braket Hybrid Jobs, Braket Pulse 또는 아날로그 해밀토니안 시뮬레이션 같은 특정 Braket 기능을 사용하는 방법에 대한 권장 사항을 얻어 아이디어 구상부터 실행까지의 과정을 더 빠르게 진행할 수 있습니다.
  - Braket 상담 시간에 등록하려면 등록을 선택하고 연락처 정보, 워크로드 세부 정보 및 원하는 논의 주제를 입력합니다.
  - 그러면 다음 가능한 시간대에 대한 일정 초대가 이메일로 발송됩니다.

  > **Note**
  >
  > 긴급한 문제 또는 빠른 문제 해결 질문의 경우 AWS Support에 문의하는 것이 좋습니다. 긴급하지 않은 질문의 경우 AWS re:Post 포럼 또는 Quantum Computing Stack Exchange를 사용하여 이전에 답변된 질문을 찾아보고 새 질문을 할 수도 있습니다.

- 양자 하드웨어 공급자 제품: IonQ, QuEra 및 Rigetti는 각각 AWS Marketplace를 통해 전문 서비스 제품을 제공합니다.
  - 제품을 찾아보려면 연결을 선택하고 목록을 찾습니다.
  - 의 전문 서비스 상품에 대한 자세한 내용은 전문 서비스 제품을 AWS Marketplace참조하세요.
- Amazon Quantum Solutions Lab(QSL): QSL은 양자 컴퓨팅 전문가들로 구성된 공동 연구 및 전문 서비스 팀으로, 양자 컴퓨팅을 효과적으로 탐구하고 이 기술의 현재 성능을 평가하는 데 도움을 줄 수 있습니다.
  - QSL에 연락하려면 연결을 선택하고 연락처 정보와 사용 사례 세부 정보를 입력합니다.

- QSL 팀에서 다음 단계에 대한 안내를 이메일로 보내드립니다.

OpenQASM 3.0으로 회로 실행

Amazon Braket은 이제 게이트 기반 양자 디바이스 및 시뮬레이터에 대해 OpenQASM 3.0을 지원합니다. 이 사용자 가이드는 Braket에서 지원하는 OpenQASM 3.0의 부분 집합에 대한 정보를 제공합니다. Braket 고객은 이제 SDK를 사용하여 Braket 회로를 제출하거나 Amazon Braket API 및 Amazon Braket Python SDK를 사용하여 모든 게이트 기반 디바이스에 OpenQASM 3.0 문자열을 직접 제공할 수 있습니다.

이 가이드의 주제에서는 다음 양자 작업을 완료하는 방법에 대한 다양한 예제를 안내합니다.

- 다양한 Braket 디바이스에서 OpenQASM 양자 작업 생성 및 제출
- 지원되는 연산 및 결과 유형에 액세스
- OpenQASM으로 노이즈 시뮬레이션
- OpenQASM으로 축어적 컴파일 사용
- OpenQASM 문제 해결

또한 이 가이드에서는 Braket의 OpenQASM 3.0으로 구현할 수 있는 특정 하드웨어별 기능과 추가 리소스에 대한 링크를 소개합니다.

이 섹션의 내용:

- OpenQASM 3.0이란 무엇입니까?
- OpenQASM 3.0을 사용해야 하는 시기
- OpenQASM 3.0 작동 방식
- 사전 조건
- Braket은 어떤 OpenQASM 기능을 지원하나요?
- 예제 OpenQASM 3.0 양자 작업 생성 및 제출
- 다양한 Braket 디바이스에서의 OpenQASM에 대한 지원
- OpenQASM 3.0으로 노이즈 시뮬레이션
- OpenQASM 3.0을 사용한 Qubit 재배선
- OpenQASM 3.0을 사용한 축어적 컴파일
- Braket 콘솔

OpenQASM 3.0이란 무엇입니까?

Open Quantum Assembly Language(OpenQASM)는 양자 명령을 위한 중간 표현입니다. OpenQASM은 오픈 소스 프레임워크이며 게이트 기반 디바이스의 양자 프로그램 사양에 널리 사용됩니다. OpenQASM을 통해 사용자는 양자 계산의 구성 요소를 형성하는 양자 게이트 및 측정 연산을 프로그래밍할 수 있습니다. OpenQASM의 이전 버전(2.0)은 여러 양자 프로그래밍 라이브러리에서 기본 프로그램을 설명하는 데 사용되었습니다.

새 버전의 OpenQASM(3.0)은 이전 버전을 확장하여 최종 사용자 인터페이스와 하드웨어 설명 언어 간의 격차를 해소하기 위해 펄스 수준 제어, 게이트 타이밍, 고전적 제어 흐름과 같은 더 많은 기능을 포함합니다. 현재 버전 3.0에 대한 세부 정보 및 사양은 GitHub OpenQASM 3.x Live Specification에서 확인할 수 있습니다. OpenQASM의 향후 개발은 OpenQASM 3.0 기술 운영 위원회에서 관리하며,이 위원회 AWS 는 IBM, Microsoft 및 University of Innsbruck과 함께 회원입니다.

OpenQASM 3.0을 사용해야 하는 시기

OpenQASM은 아키텍처에 구애받지 않는 저수준 제어 기능을 통해 양자 프로그램을 지정하는 표현적 프레임워크를 제공하므로, 여러 게이트 기반 디바이스에 걸쳐 표현 수단으로 적합합니다. OpenQASM에 대한 Braket 지원은 게이트 기반 양자 알고리즘 개발을 위한 일관된 접근 방식으로의 채택을 촉진하여 사용자가 여러 프레임워크에서 라이브러리를 학습하고 유지할 필요성을 줄여줍니다.

OpenQASM 3.0에 기존 프로그램 라이브러리가 있는 경우 이러한 회로를 완전히 다시 쓰는 대신 Braket에서 사용하도록 조정할 수 있습니다. 또한 연구원과 개발자는 OpenQASM 기반 알고리즘 개발을 지원하는 타사 라이브러리를 점점 더 많이 활용할 수 있게 되는 이점을 누릴 수 있습니다.

OpenQASM 3.0 작동 방식

Braket의 OpenQASM 3.0 지원은 현재 중간 표현과 기능 패리티를 제공합니다. 즉, Braket API를 사용하여 하드웨어 디바이스 및 온디맨드 시뮬레이터에서 현재 수행할 수 있는 모든 작업을 Braket을 통해 OpenQASM으로 수행할 수 있습니다. 현재 Braket의 디바이스에 회로가 제공되는 방식과 유사한 방식으로 모든 게이트 기반 디바이스에 OpenQASM 문자열을 직접 제공하여 OpenQASM 3.0 프로그램을 실행할 수 있습니다. Braket 사용자는 OpenQASM 3.0을 지원하는 타사 라이브러리를 통합할 수도 있습니다. 이 가이드의 나머지 부분에서는 Braket에서 사용할 OpenQASM 표현을 개발하는 방법을 자세히 설명합니다.

사전 조건

Amazon Braket에서 OpenQASM 3.0을 사용하려면 Amazon Braket Python Schema 버전 v1.8.0과 Amazon Braket Python SDK 버전 v1.17.0 이상이 있어야 합니다.

Amazon Braket을 처음 사용하는 경우 Amazon Braket을 활성화해야 합니다. 지침은 Amazon Braket 활성화를 참조하세요.

Braket은 어떤 OpenQASM 기능을 지원하나요?

다음 섹션에는 Braket에서 지원하는 OpenQASM 3.0 데이터 유형, 문 및 pragma 명령이 나열되어 있습니다.

이 섹션의 내용:

- 지원되는 OpenQASM 데이터 유형
- 지원되는 OpenQASM 문
- Braket OpenQASM pragma
- 로컬 시뮬레이터에서의 OpenQASM에 대한 고급 기능 지원
- OpenPulse에서 지원되는 연산 및 문법

지원되는 OpenQASM 데이터 유형

Amazon Braket은 다음과 같은 OpenQASM 데이터 유형을 지원합니다.

- (가상 및 물리적) 큐비트 인덱스에는 음이 아닌 정수가 사용됩니다.
  - `cnot q[0], q[1];`
  - `h $0;`
- 게이트 회전 각도에는 부동 소수점 숫자 또는 상수를 사용할 수 있습니다.
  - `rx(-0.314) $0;`
  - `rx(pi/4) $0;`

> **Note**
>
> pi는 OpenQASM의 기본 제공 상수이며 파라미터 이름으로 사용할 수 없습니다.

- 복소수 배열(허수부에 대한 OpenQASM `im` 표기법 사용)은 일반적인 에르미트 관찰 가능 항목을 정의하기 위한 결과 유형 pragma와 단위 pragma에서 허용됩니다.
  - `#pragma braket unitary [[0, -1im], [1im, 0]] q[0]`
  - `#pragma braket result expectation hermitian([[0, -1im], [1im, 0]]) q[0]`

지원되는 OpenQASM 문

Amazon Braket은 다음과 같은 OpenQASM 문을 지원합니다.

- `Header: OPENQASM 3;`
- 클래식 비트 선언:
  - `bit b1;`(동등하게, `creg b1;`)
  - `bit[10] b2;`(동등하게, `creg b2[10];`)
- 큐비트 선언:
  - `qubit b1;`(동등하게, `qreg b1;`)
  - `qubit[10] b2;`(동등하게, `qreg b2[10];`)
- 배열 내 인덱싱: `q[0]`
- 입력: `input float alpha;`
- 물리적 qubits의 사양: `$0`
- 디바이스에서 지원되는 게이트 및 연산:
  - `h $0;`
  - `iswap q[0], q[1];`

> **Note**
>
> 디바이스의 지원되는 게이트는 OpenQASM 작업에 대한 디바이스 속성에서 확인할 수 있습니다. 이러한 게이트를 사용하는 데 게이트 정의가 필요하지 않습니다.

- 축어적 상자 문. 현재, 상자 기간 표기법은 지원되지 않습니다. 축어적 상자에는 네이티브 게이트와 물리적 qubits가 필요합니다.

```
#pragma braket verbatim
```

```
box{
    rx(0.314) $0;
}
```

- qubits 또는 전체 qubit 레지스터에 대한 측정 및 측정 할당입니다.
  - `measure $0;`
  - `measure q;`
  - `measure q[0];`
  - `b = measure q;`
  - `measure q # b;`
- 장벽 문은 장벽 경계에서 게이트 재정렬 및 최적화를 방지하여 회로 컴파일 및 실행을 명시적으로 제어할 수 있습니다. 또한 실행 중에 엄격한 시간 순서를 적용하여 후속 작업이 시작되기 전에 모든 작업이 완료되도록 합니다.
  - `barrier;`
  - `barrier q[0], q[1];`
  - `barrier $3, $6;`

Braket OpenQASM pragma

Amazon Braket은 다음과 같은 OpenQASM pragma 명령을 지원합니다.

- 노이즈 pragma
  - `#pragma braket noise bit_flip(0.2) q[0]`
  - `#pragma braket noise phase_flip(0.1) q[0]`
  - `#pragma braket noise pauli_channel`
- 측어적 pragma
  - `#pragma braket verbatim`
- 결과 유형 pragma
  - 기저 불변 결과 유형:
    - 상태 벡터: `#pragma braket result state_vector`
    - 밀도 행렬: `#pragma braket result density_matrix`
  - 기울기 계산 pragma:

    - 수반 기울기: `#pragma braket result adjoint_gradient expectation(2.2 * x[0] @ x[1]) all`
- Z 기저 결과 유형:
    - 진폭: `#pragma braket result amplitude "01"`
    - 확률: `#pragma braket result probability q[0], q[1]`
- 기저 회전 결과 유형
    - 기대값: `#pragma braket result expectation x(q[0]) @ y([q1])`
    - 분산: `#pragma braket result variance hermitian([[0, -1im], [1im, 0]]) $0`
    - 샘플: `#pragma braket result sample h($1)`

> **Note**
>
> OpenQASM 3.0은 OpenQASM 2.0과 역호환되므로 2.0을 사용하여 작성된 프로그램은 Braket에서 실행할 수 있습니다. 그러나 Braket에서 지원하는 OpenQASM 3.0의 기능은 `qreg` vs `creg` 및 `qubit` vs `bit`와 같이 약간의 구문 차이가 있습니다. 측정 구문에도 차이가 있으므로 올바른 구문으로 뒷받침되어야 합니다.

로컬 시뮬레이터에서의 OpenQASM에 대한 고급 기능 지원

`LocalSimulator`는 Braket의 QPU 또는 온디맨드 시뮬레이터의 일부로 제공되지 않는 고급 OpenQASM 기능을 지원합니다. 다음 기능 목록은 `LocalSimulator`에서만 지원됩니다.

- 게이트 한정자
- OpenQASM 기본 제공 게이트
- 고전적 변수
- 고전적 연산
- 사용자 지정 게이트
- 고전적 제어
- QASM 파일
- 서브루틴

각 고급 기능의 예제는 이 샘플 노트북을 참조하세요. 전체 OpenQASM 사양은 OpenQASM 웹 사이트를 참조하세요.

OpenPulse에서 지원되는 연산 및 문법

지원되는 OpenPulse 데이터 유형

Cal 블록:

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

Defcal 블록:

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

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

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

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

프레임:

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

파형:

```
// prebuilt
```

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

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

사용자 지정 게이트 보정 예제:

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

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

defcal my_cz $1, $0 {
    barrier q0_q1_cz_frame, q0_rf_frame;
    play(q0_q1_cz_frame, wf1);
    delay[300ns] q0_rf_frame
    shift_phase(q0_rf_frame, 4.366186381749424);
    delay[300ns] q0_rf_frame;
    shift_phase(q0_rf_frame.phase, 5.916747563126659);
    barrier q0_q1_cz_frame, q0_rf_frame;
    shift_phase(q0_q1_cz_frame, 2.183093190874712);
}

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

임의 펄스 예제:

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

```
    ro[0] = capture_v0(r0_measure);
    ro[1] = capture_v0(r1_measure);
}
```

예제 OpenQASM 3.0 양자 작업 생성 및 제출

Amazon Braket Python SDK, Boto3 또는를 사용하여 Amazon Braket 디바이스 AWS CLI 에 OpenQASM 3.0 양자 작업을 제출할 수 있습니다.

이 섹션의 내용:

- OpenQASM 3.0 프로그램 예제
- Python SDK를 사용하여 OpenQASM 3.0 양자 작업 생성
- Boto3을 사용하여 OpenQASM 3.0 양자 작업 생성
- AWS CLI 를 사용하여 OpenQASM 3.0 작업 생성

OpenQASM 3.0 프로그램 예제

OpenQASM 3.0 작업을 생성하려면, 다음 예제와 같이 GHZ 상태를 준비하는 기본 OpenQASM 3.0 프로그램(ghz.qasm)으로 시작할 수 있습니다.

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

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

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

c = measure q;
```

Python SDK를 사용하여 OpenQASM 3.0 양자 작업 생성

Amazon Braket Python SDK를 사용하여 다음 코드로 이 프로그램을 Amazon Braket 디바이스에 제출할 수 있습니다. 예제 Amazon S3 버킷 위치 “amzn-s3-demo-bucket”을 자체 Amazon S3 버킷 이름으로 바꿔야 합니다.

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

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

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

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

Boto3을 사용하여 OpenQASM 3.0 양자 작업 생성

다음 예제와 같이 AWS Python SDK for Braket(Boto3)을 사용하여 OpenQASM 3.0 문자열을 이용해 양자 작업을 생성할 수도 있습니다. 다음 코드 조각은 위에서와 같이 GHZ 상태를 준비하는 ghz.qasm을 참조합니다.

```
import boto3
import json

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

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

action = {
    "braketSchemaHeader": {
        "name": "braket.ir.openqasm.program",
        "version": "1"
    },
    "source": source
}
```

```
device_parameters = {}
device_arn = "arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3"
shots = 100

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

AWS CLI 를 사용하여 OpenQASM 3.0 작업 생성

다음 예제와 같이 AWS Command Line Interface (CLI)를 사용하여 OpenQASM 3.0 프로그램을 제출할 수도 있습니다.

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

다양한 Braket 디바이스에서의 OpenQASM에 대한 지원

OpenQASM 3.0을 지원하는 디바이스의 경우 `action` 필드는 Rigetti 및 IonQ 디바이스에 대한 다음 예제와 같이 `GetDevice` 응답을 통해 새 작업을 지원합니다.

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

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

펄스 제어를 지원하는 디바이스의 경우 `pulse` 필드가 `GetDevice` 응답에 표시됩니다. 다음 예제에서는 Rigetti 디바이스에 대한 이 `pulse` 필드를 보여줍니다.

```
// Rigetti
{
```

```
  "pulse": {
    "braketSchemaHeader": {
      "name": "braket.device_schema.pulse.pulse_device_action_properties",
      "version": "1"
    },
    "supportedQhpTemplateWaveforms": {
      "constant": {
        "functionName": "constant",
        "arguments": [
          {
            "name": "length",
            "type": "float",
            "optional": false
          },
          {
            "name": "iq",
            "type": "complex",
            "optional": false
          }
        ]
      },
      ...
    },
    "ports": {
      "q0_ff": {
        "portId": "q0_ff",
        "direction": "tx",
        "portType": "ff",
        "dt": 1e-9,
        "centerFrequencies": [
          375000000
        ]
      },
      ...
    },
    "supportedFunctions": {
      "shift_phase": {
        "functionName": "shift_phase",
        "arguments": [
          {
            "name": "frame",
            "type": "frame",
            "optional": false
          },
```

```
          {
            "name": "phase",
            "type": "float",
            "optional": false
          }
        ]
      },
     ...
    },
    "frames": {
      "q0_q1_cphase_frame": {
        "frameId": "q0_q1_cphase_frame",
        "portId": "q0_ff",
        "frequency": 462475694.24460185,
        "centerFrequency": 375000000,
        "phase": 0,
        "associatedGate": "cphase",
        "qubitMappings": [
          0,
          1
        ]
      },
      ...
    },
    "supportsLocalPulseElements": false,
    "supportsDynamicFrames": false,
    "supportsNonNativeGatesWithPulses": false,
    "validationParameters": {
      "MAX_SCALE": 4,
      "MAX_AMPLITUDE": 1,
      "PERMITTED_FREQUENCY_DIFFERENCE": 400000000
    }
  }
}
```

앞서 언급된 필드는 다음을 상세히 설명합니다.

포트:

QPU에 선언된 사전 제작 외부(extern) 디바이스 포트와 지정된 포트의 관련 속성을 설명합니다. 이 구조에 나열된 모든 포트는 사용자가 제출한 OpenQASM 3.0 프로그램 내에서 유효한 식별자로 사전 선언됩니다. 포트의 추가 속성은 다음과 같습니다.

- 포트 ID(portId)
  - OpenQASM 3.0에서 식별자로 선언된 포트 이름입니다.
- 방향(direction)
  - 포트의 방향입니다. 드라이브 포트는 펄스(direction “tx”)를 전송하는 반면, 측정 포트는 펄스(direction “rx”)를 수신합니다.
- 포트 유형(portType)
  - 이 포트가 담당하는 작업의 유형(예: 드라이브, 캡처 또는 ff - fast-flux)입니다.
- Dt(dt)
  - 지정된 포트에서 단일 샘플 시간 단계를 나타내는 초 단위 시간입니다.
- 큐비트 매핑(qubitMappings)
  - 지정된 포트와 연결된 큐비트입니다.
- 중심 주파수(centerFrequencies)
  - 포트에서 사전 선언되거나 사용자 정의된 모든 프레임에 대한 연관 중심 주파수의 목록입니다. 자세한 내용은 프레임을 참조하세요.
- QHP 특정 속성(qhpSpecificProperties)
  - QHP 전용 포트에 대한 기존 속성을 자세히 설명하는 선택적 맵입니다.

프레임:

QPU에서 선언된 사전 제작 외부 프레임과 해당 프레임에 대한 관련 속성을 설명합니다. 이 구조에 나열된 모든 프레임은 사용자가 제출한 `OpenQASM 3.0` 프로그램 내에서 유효한 식별자로 사전 선언됩니다. 프레임의 추가 속성은 다음과 같습니다.

- 프레임 ID(frameId)
  - OpenQASM 3.0에서 식별자로 선언된 프레임 이름입니다.
- 포트 ID(portId)
  - 프레임에 연결된 하드웨어 포트입니다.
- 주파수(frequency)
  - 프레임의 기본 초기 주파수입니다.
- 중심 주파수(centerFrequency)
  - 프레임의 주파수 대역폭 중심입니다. 일반적으로 프레임은 중심 주파수 주변의 특정 대역폭으로만 조정할 수 있습니다. 따라서 주파수 조정은 중심 주파수의 지정된 델타 범위 내에서 이루어져야 합니다. 대역폭 값은 검증 파라미터에서 확인할 수 있습니다.

- 위상(phase)
  - 프레임의 기본 초기 위상입니다.
- 연결된 게이트(associatedGate)
  - 지정된 프레임과 연결된 게이트입니다.
- 큐비트 매핑(qubitMappings)
  - 지정된 프레임과 연결된 큐비트입니다.
- QHP 특정 속성(qhpSpecificProperties)
  - QHP 전용 프레임에 대한 기존 속성을 자세히 설명하는 선택적 맵입니다.

SupportsDynamicFrames:

OpenPulse `newframe` 함수를 통해 프레임을 `cal` 또는 `defcal` 블록에서 선언할 수 있는지 여부를 설명합니다. false인 경우 프레임 구조에 나열된 프레임만 프로그램 내에서 사용할 수 있습니다.

SupportedFunctions:

지정된 함수에 대해 연결된 인수, 인수 유형 및 반환 유형 외에도 디바이스에 대해 지원되는 OpenPulse 함수를 설명합니다. OpenPulse 함수의 사용 예제를 보려면 OpenPulse 사양을 참조하세요. 현재 Braket은 다음을 지원합니다.

- shift_phase
  - 프레임의 위상을 지정된 값만큼 이동합니다.
- set_phase
  - 프레임의 위상을 지정된 값으로 설정합니다.
- swap_phases
  - 두 프레임 간에 위상을 바꿉니다.
- shift_frequency
  - 프레임의 주파수를 지정된 값만큼 이동합니다.
- set_frequency
  - 프레임의 주파수를 지정된 값으로 설정합니다.
- play
  - 파형을 예약합니다.
- capture_v0

- 캡처 프레임의 값을 비트 레지스터로 반환합니다.

SupportedQhpTemplateWaveforms:

디바이스에서 사용할 수 있는 사전 빌드된 파형 함수와 관련 인수 및 유형을 설명합니다. 기본적으로 Braket Pulse는 모든 디바이스에서 다음과 같은 사전 빌드된 파형 루틴을 제공합니다.

상수

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

τ는 파형의 길이이고 iq는 복소수입니다.

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

가우시안

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

τ는 파형의 길이이고, σ는 가우시안의 너비이며 A는 진폭입니다. ZaE를 True로 설정하면 가우시안은 파형의 시작과 끝에서 0이 되도록 오프셋 및 재조정되며 최대값에서 A에 도달합니다.

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

DRAG 가우시안

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

τ는 파형의 길이, σ는 가우시안의 너비, β는 자유 파라미터, A는 진폭입니다. ZaE를 True로 설정하면 DRAG(Derivative Removal by Adiabatic Gate) 가우시안은 파형의 시작과 끝에서 0이 되도록 오프셋 및 재조정되며 실수부는 최대값에서 A에 도달합니다. DRAG 파형에 대한 자세한 내용은 백서 약한 비선형 큐비트에서 누출 제거를 위한 단순 펄스를 참조하세요.

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

Erf 제곱

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

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

여기서 L은 길이이고 W는 파형의 너비이며 σ는 엣지의 상승 및 하강 속도를 정의하고 t_1=(L-W)/2 및 $t_2$2=(L+W)/2, A는 진폭입니다. ZaE를 True로 설정하면 가우시안은 파형의 시작과 끝에서 0이 되도록 오프셋 및 재조정되며 최대값에서 A에 도달합니다. 다음 방정식은 파형의 재조정된 버전입니다.

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

여기서 a=erf(W/2σ) 및 b=erf(-t_1/σ)/2+erf(t $_2$/σ)/2 입니다.

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

SupportsLocalPulseElements:

포트, 프레임 및 파형과 같은 펄스 요소를 defcal 블록에서 로컬로 정의할 수 있는지 여부를 설명합니다. 값이 false인 경우 요소를 cal 블록에서 정의해야 합니다.

SupportsNonNativeGatesWithPulses:

비네이티브 게이트를 펄스 프로그램과 함께 사용할 수 있는지 여부를 설명합니다. 예를 들어 사용된 큐비트에 대해 defcal을 통해 게이트를 먼저 정의하지 않으면 프로그램의 H 게이트와 같은 비네이티브 게이트를 사용할 수 없습니다. 디바이스 기능에서 네이티브 게이트 nativeGateSet 키 목록을 확인할 수 있습니다.

ValidationParameters:

다음을 포함하여 펄스 요소 검증 경계를 설명합니다.

- 파형(임의 및 사전 빌드됨)의 최대 규모/최대 진폭 값
- 제공된 중심 주파수의 최대 주파수 대역폭을 Hz 단위로 표시
- 초 단위의 최소 펄스 길이/지속 시간
- 초 단위의 최대 펄스 길이/지속 시간

OpenQASM으로 지원되는 연산, 결과 및 결과 유형

각 디바이스에서 지원하는 OpenQASM 3.0 기능을 확인하려면 디바이스 기능 출력의 `action` 필드에서 `braket.ir.openqasm.program` 키를 참조하세요. 예를 들어 다음은 Braket 상태 벡터 시뮬레이터 SV1에 사용할 수 있는 지원되는 연산 및 결과 유형입니다.

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

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

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

OpenQASM 3.0으로 노이즈 시뮬레이션

OpenQASM3으로 노이즈를 시뮬레이션하려면 pragma 명령을 사용하여 노이즈 연산자를 추가합니다. 예를 들어 이전에 제공된 GHZ 프로그램의 노이즈 있는 버전을 시뮬레이션하려면 다음 OpenQASM 프로그램을 제출할 수 있습니다.

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

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

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

c = measure q;
```

지원되는 모든 pragma 노이즈 연산자에 대한 사양은 다음 목록에 나와 있습니다.

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

Kraus 연산자

Kraus 연산자를 생성하려면 행렬 목록을 반복하여 행렬의 각 요소를 복소수 표현식으로 출력하면 됩니다.

Kraus 연산자를 사용할 때는 다음 사항에 유의하세요.

- qubits의 수는 2를 초과할 수 없습니다. 스키마의 현재 정의는 이 제한을 설정합니다.
- 인수 목록의 길이는 8의 배수여야 합니다. 즉, 2x2 행렬로만 구성되어야 합니다.
- 총 길이는 2^{2*num_qubits} 행렬을 초과하지 않습니다. 즉, 1qubit의 경우 행렬이 4개이고 2qubits의 경우 행렬이 16개입니다.
- 제공된 모든 행렬은 완전 양성 트레이스 보존(Completely Positive Trace Preserving, CPTP)입니다.
- Kraus 연산자와 그 전치 켤레의 곱은 단위 행렬에 합산되어야 합니다.

OpenQASM 3.0을 사용한 Qubit 재배선

Amazon Braket은 Rigetti 디바이스의 OpenQASM 내에서 물리적 qubit 표기법을 지원합니다(자세한 내용은 이 페이지 참조). 나이브 재배선 전략과 함께 물리적 qubits를 사용하는 경우 qubits가 선택한 디바이스에 연결되어 있는지 확인합니다. 또는 qubit 레지스터를 대신 사용하는 경우 Rigetti 디바이스에서 PARTIAL 재배선 전략이 기본적으로 활성화됩니다.

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

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

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

OpenQASM 3.0을 사용한 축어적 컴파일

Rigetti 및 IonQ와 같은 공급자가 제공하는 양자 컴퓨터에서 양자 회로를 실행할 때 컴파일러가 수정 없이 정의된 대로 정확하게 회로를 실행하도록 지시할 수 있습니다. 이 기능을 축어적 컴파일이라고 합니다. Rigetti 디바이스를 사용하면 전체 회로 또는 특정 부분만 보존되는 항목을 정확하게 지정할 수 있습니다. 회로의 특정 부분만 보존하려면 보존된 리전 내에서 네이티브 게이트를 사용해야 합니다. 현재, IonQ는 전체 회로에 대해 축어적 컴파일만 지원하므로 회로의 모든 명령을 축어적 상자에 포함시켜야 합니다.

OpenQASM을 사용하면 코드 상자 주위에 축어적 pragma를 명시적으로 지정할 수 있습니다. 그러면 코드 상자는 하드웨어의 저수준 컴파일 루틴에 의해 변경되지 않고 최적화되지 않은 상태로 유지됩니다. 다음 코드 예제에서는 `#pragma braket verbatim` 지시어를 사용하여 이를 달성하는 방법을 보여줍니다.

```
OPENQASM 3;

bit[2] c;

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

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

예제 및 모범 사례를 포함하여 축어적 컴파일 프로세스에 대한 자세한 내용은 amazon-braket-examples github 리포지토리에서 제공되는 축어적 컴파일 샘플 노트북을 참조하세요.

Braket 콘솔

OpenQASM 3.0 작업은 Amazon Braket 콘솔에서 사용하고 관리할 수 있습니다. 콘솔에서 OpenQASM 3.0으로 양자 작업을 제출하는 경험은 기존 양자 작업을 제출할 때와 동일합니다.

추가 리소스

OpenQASM은 모든 Amazon Braket 리전에서 사용할 수 있습니다.

Amazon Braket에서 OpenQASM을 시작하기 위한 예제 노트북은 Braket 자습서 GitHub를 참조하세요.

OpenQASM 3.0을 사용한 기울기 계산

Amazon Braket은 `shots=0` (정확) 모드에서 실행할 때 온디맨드 시뮬레이터와 로컬 시뮬레이터 모두에서 기울기 계산을 지원합니다. 이는 수반 미분법을 사용하여 달성됩니다. 계산하려는 기울기를 지정하려면 다음 예제의 코드에 설명된 대로 적절한 pragma를 제공할 수 있습니다.

```
OPENQASM 3.0;
input float alpha;
```

```
bit[2] b;
qubit[2] q;

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

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

모든 개별 파라미터를 명시적으로 나열하는 대신, pragma 내에서 `all` 키워드를 지정할 수도 있습니다. 이렇게 하면 나열된 모든 `input` 파라미터에 대한 기울기가 계산되므로, 파라미터 수가 매우 클 때 편리한 옵션이 될 수 있습니다. 이 경우 pragma는 다음 예제의 코드와 유사합니다.

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

Amazon Braket의 OpenQASM 3.0 구현에서는 개별 연산자, 텐서 곱, 에르미트 관찰 가능 항목, `Sum` 관찰 가능 항목 등 모든 관찰 가능 항목 유형이 지원됩니다. 기울기를 계산할 때 사용할 특정 연산자는 `expectation()` 함수 내에 래핑되어야 하며, 관찰 가능 항목의 각 항이 작용하는 큐비트는 명시적으로 지정되어야 합니다.

OpenQASM 3.0을 사용한 특정 큐비트 측정

Amazon Braket에서 제공하는 로컬 상태 벡터 시뮬레이터 및 로컬 밀도 행렬 시뮬레이터는 회로 큐비트의 부분 집합을 선택적으로 측정할 수 있는 OpenQASM 프로그램 제출을 지원합니다. 흔히 부분 측정이라고 하는 이 기능을 사용하면 보다 표적화되고 효율적인 양자 계산이 가능합니다. 예를 들어 다음 코드 조각에서는 2큐비트 회로를 생성하고 첫 번째 큐비트만 측정하도록 선택할 수 있으며, 두 번째 큐비트를 측정하지 않은 상태로 둘 수 있습니다.

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

이 예제에서는 q[0]과 q[1]이라는 2개의 큐비트가 있는 양자 회로가 있지만 첫 번째 큐비트의 상태만 측정하는 데 관심이 있습니다. 이는 큐비트[0]의 상태를 측정하고 그 결과를 고전적 비트 b[0]으로 저장하는 라인 b[0] = measure q[0]에 의해 달성됩니다. 이 부분 측정 시나리오를 실행하려면 Amazon Braket에서 제공하는 로컬 상태 벡터 시뮬레이터에서 다음 코드를 실행할 수 있습니다.

```
from braket.devices import LocalSimulator

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

작업 속성에서 requiresAllQubitsMeasurement 필드를 검사하여 디바이스가 부분 측정을 지원하는지 확인할 수 있습니다. False인 경우 부분 측정이 지원됩니다.

```
from braket.devices import Devices

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

여기서 requiresAllQubitsMeasurement은 False이며, 이는 모든 큐비트를 측정해야 하는 것은 아님을 나타냅니다.

실험 기능 살펴보기

실험 기능을 사용하면 가용성이 제한되고 새로운 소프트웨어 기능이 도입된 하드웨어에 액세스할 수 있습니다. 이러한 기능은 표준 사양을 초과하는 디바이스 성능에 영향을 미칠 수 있습니다. Amazon Braket SDK를 통해 작업별로 실험 소프트웨어 기능을 자동으로 활성화할 수 있습니다.

실험 기능을 사용하려면 양자 작업을 생성할 때 experimental_capabilities 파라미터를 지정합니다. 이 파라미터를 로 설정"ALL"하여 해당 작업에 사용 가능한 모든 실험 기능을 활성화합니다. 다음 예제에서는 디바이스에서 회로를 실행할 때 실험 기능을 활성화하는 방법을 보여줍니다.

```
from braket.aws import AwsDevice

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

task = device.run(
   circuit,
```

```
    shots=1000,
    experimental_capabilities="ALL"
)
```

> **Note**
>
> 이러한 기능은 실험적이며 예고 없이 변경될 수 있습니다. 디바이스 성능은 게시된 사양과 다를 수 있으며 결과는 표준 작업과 다를 수 있습니다. 각 작업에 대해 실험 기능을 명시적으로 활성화해야 합니다. 이 파라미터가 없는 작업은 표준 디바이스 기능만 사용합니다.

이 섹션의 내용:

QuEra Aquila에서 로컬 디튜닝에 액세스

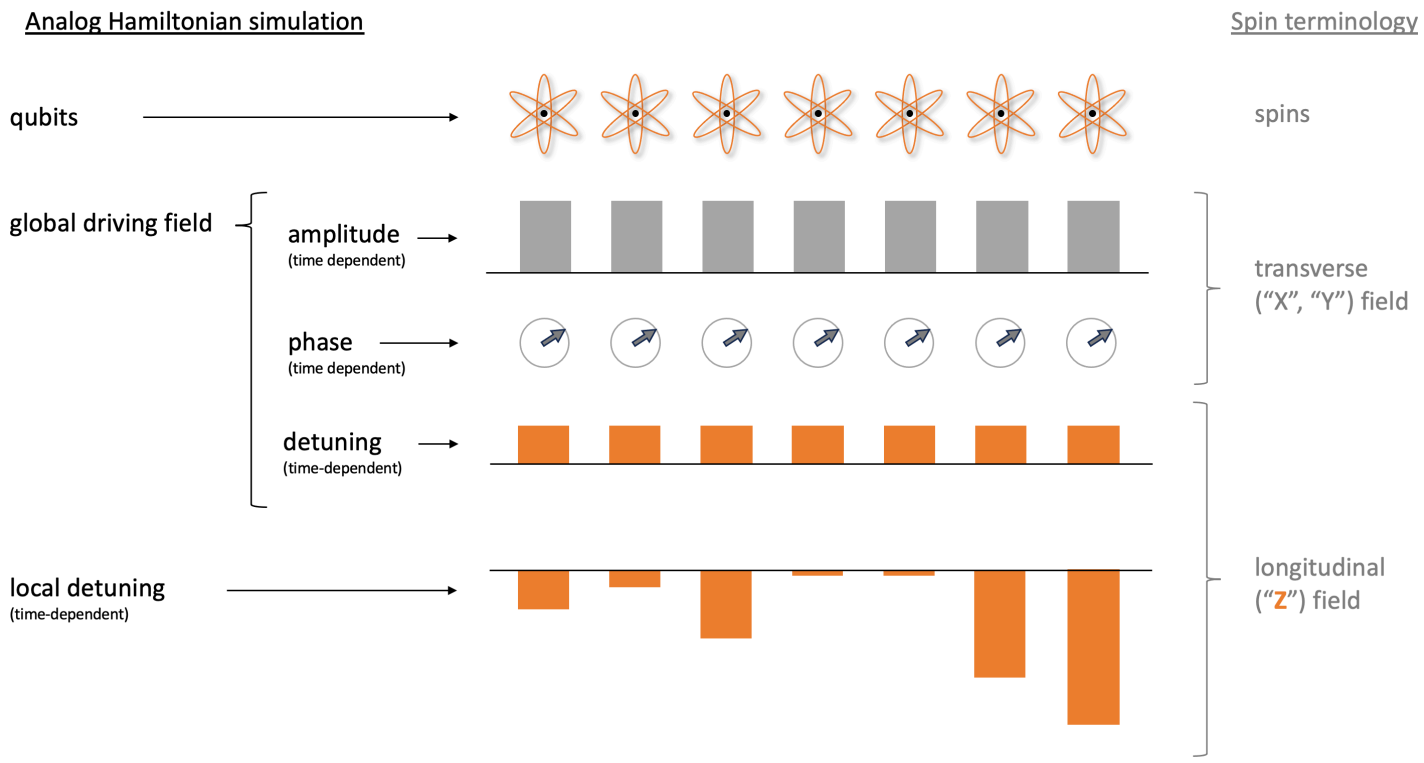
로컬 디튜닝(Local Detuning, LD)은 사용자 지정 가능한 공간 패턴이 있는 새로운 시간 종속 제어 장입니다. LD장은 사용자 지정 가능한 공간 패턴에 따라 큐비트에 영향을 미치며, 균일한 구동장과 Rydberg-Rydberg 상호 작용이 생성할 수 있는 것 이상으로 서로 다른 큐비트에 대해 서로 다른 해밀토니안을 구현합니다.

제약 조건:

로컬 디튜닝장의 공간 패턴은 각 AHS 프로그램에 대해 사용자 지정할 수 있지만, 프로그램이 진행되는 동안 일정합니다. 로컬 디튜닝장의 시계열은 0에서 시작하고 끝나야 하며 모든 값은 0보다 작거나 같아야 합니다. 또한 로컬 디튜닝장의 파라미터는 수치적 제약 조건에 의해 제한되며, 이는 특정 디바이스 속성 섹션 - `aquila_device.properties.paradigm.rydberg.rydbergLocal`의 Braket SDK를 통해 볼 수 있습니다.

제한 사항:

로컬 디튜닝장을 사용하는 양자 프로그램을 실행할 때(해밀토니안에서 크기가 상수 0으로 설정된 경우에도) 디바이스는 Aquila 속성의 성능 섹션에 나열된 T2 시간보다 더 빠른 결어긋남을 경험합니다. 불필요한 경우 AHS 프로그램의 해밀토니안에서 로컬 디튜닝장을 생략하는 것이 좋습니다.



예제:

1. 스핀 시스템에서 비균일 종방향 자기장의 효과 시뮬레이션

   구동장의 진폭과 위상은 스핀에 대한 횡방향 자기장과 동일한 효과를 큐비트에 미치지만, 구동장의 디튜닝과 로컬 디튜닝의 합은 스핀에 대한 종방향 자기장과 동일한 효과를 큐비트에 발생시킵니다. 로컬 디튜닝장을 공간적으로 제어하면 더 복잡한 스핀 시스템을 시뮬레이션할 수 있습니다.

2. 비평형 초기 상태 준비

   예제 노트북 Rydberg 원자를 이용한 격자 게이지 이론 시뮬레이션은 9원자 선형 배열의 중심 원자가 Z2 정렬 상으로 시스템을 어닐링할 때 여기되는 것을 억제하는 방법을 보여줍니다. 준비 단계 후에는 로컬 디튜닝장이 감소하고 AHS 프로그램은 이 특정 비평형 상태에서 시작하여 시스템의 시간 변화를 계속 시뮬레이션합니다.

3. 가중치 기반 최적화 문제 해결

   예제 노트북 Maximum weight independent set(MWIS)는 Aquila에서 MWIS 문제를 해결하는 방법을 보여줍니다. 로컬 디튜닝장은 단위 디스크 그래프의 노드에 대한 가중치를 정의하는 데 사용되며, 엣지는 Rybderg 봉쇄 효과로 구현됩니다. 균일 바닥 상태에서 시작하여 로컬 디튜닝장을 점진적으로 증가시키면 시스템이 MWIS 해밀토니안의 바닥 상태로 전환되어 문제에 대한 해결책을 찾을 수 있습니다.

QuEra Aquila에서 높은 기하형상에 액세스

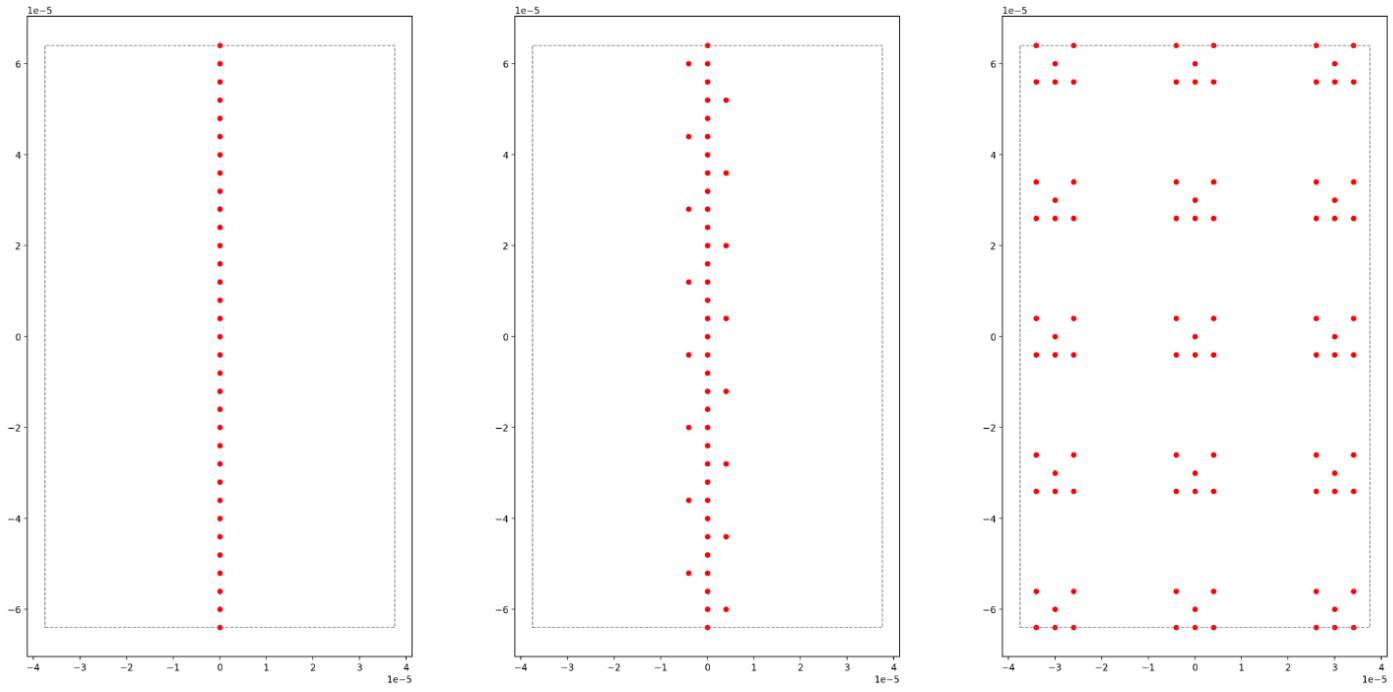
높은 기하형상 기능을 사용하면 높이가 증가된 기하형상을 지정할 수 있습니다. 이 기능을 사용하면 AHS 프로그램의 원자 배열이 Aquila의 일반 기능을 넘어 y 방향으로 추가 길이를 확장할 수 있습니다.

제약 조건:

높은 기하형상의 최대 높이는 0.000128m(128um)입니다.

제한 사항:

이 실험 기능이 계정에 대해 활성화되면 디바이스 속성 페이지와 `GetDevice` 직접 호출에 표시되는 기능은 계속해서 높이에 대한 일반 하한을 반영합니다. AHS 프로그램이 일반 기능을 넘어서는 원자 배열을 사용하는 경우 채우기 오류가 증가할 것으로 예상됩니다. 작업 결과의 `pre_sequence` 부분에서 예상치 못한 0의 수가 증가하는 현상을 발견하게 되며, 이는 결과적으로 완벽하게 초기화된 배열을 얻을 가능성을 낮춥니다. 이 효과는 원자가 많은 행에서 가장 강력합니다.



예:

1. 더 큰 1차원 및 준-1차원 배열

   원자 사슬 및 사다리형 배열은 더 높은 원자 번호로 확장할 수 있습니다. y와 병렬로 긴 방향을 지정하면 이러한 모델의 더 긴 인스턴스를 프로그래밍할 수 있습니다.
2. 작은 기하형상을 가진 작업의 실행을 다중화할 수 있는 더 많은 공간

예제 노트북 Parallel quantum tasks on Aquila는 하나의 원자 배열에 해당 기하형상의 다중화된 복사본을 배치하여 사용 가능한 영역을 최대한 활용하는 방법을 보여줍니다. 사용 가능한 영역이 많을수록 더 많은 사본을 배치할 수 있습니다.

QuEra Aquila에서 좁은 기하형상에 액세스

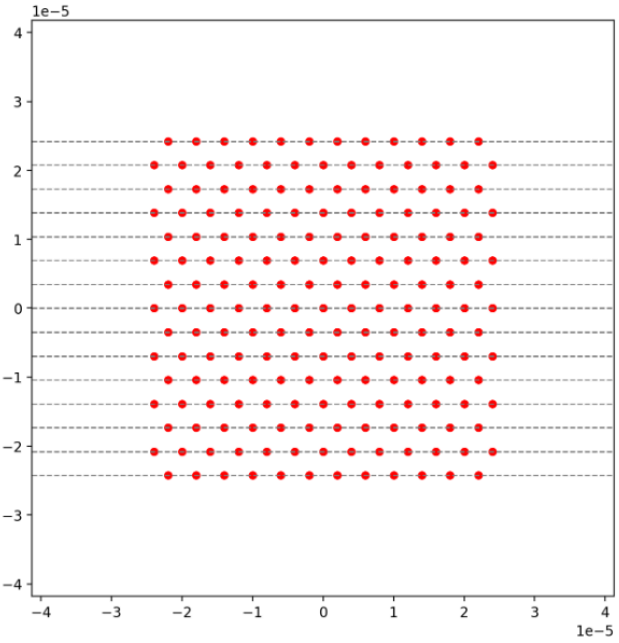
좁은 기하형상 기능을 사용하면 이웃하는 행 사이의 간격이 더 짧은 기하형상을 지정할 수 있습니다. AHS 프로그램에서 원자는 최소 수직 간격으로 구분된 행으로 배열됩니다. 두 원자 위치의 y 좌표는 0(동일한 행)이거나 최소 행 간격(다른 행)보다 크게 차이나야 합니다. 좁은 기하형상 기능을 사용하면 최소 행 간격이 축소되어 더 좁은 원자 배열을 생성할 수 있습니다. 이 확장은 원자 간 최소 유클리드 거리 요구 사항을 변경하지 않지만, 멀리 떨어진 원자들이 서로 더 가까운 이웃하는 행을 차지하는 격자 생성을 허용합니다. 대표적인 예로는 삼각형 격자가 있습니다.

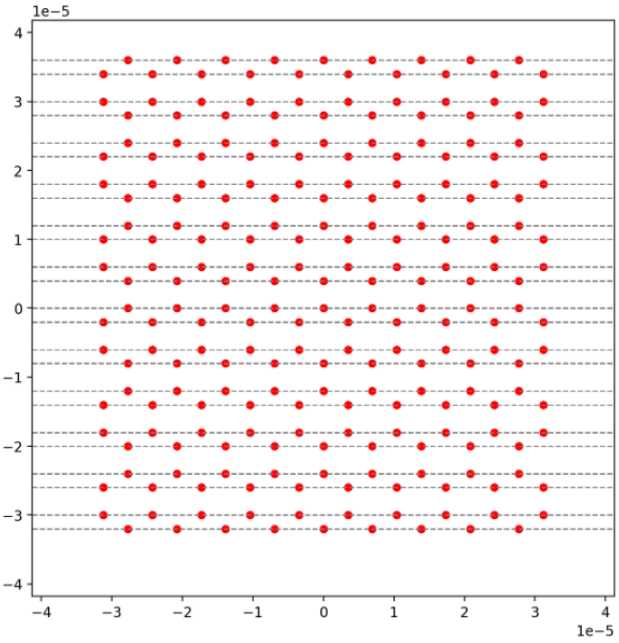
제약 조건:

좁은 기하형상의 최소 행 간격은 0.000002m(2um)입니다.

제한 사항:

이 실험 기능이 계정에 대해 활성화되면 디바이스 속성 페이지와 `GetDevice` 직접 호출에 표시되는 기능은 계속해서 높이에 대한 일반 하한을 반영합니다. AHS 프로그램이 일반 기능을 넘어서는 원자 배열을 사용하는 경우 채우기 오류가 증가할 것으로 예상됩니다. 고객은 작업 결과의 `pre_sequence` 부분에서 예상치 못한 0의 수가 증가하는 현상을 발견하게 되며, 이는 결과적으로 완벽하게 초기화된 배열을 얻을 가능성을 낮춥니다. 이 효과는 원자가 많은 행에서 가장 강력합니다.





예:

1. 작은 격자 상수가 있는 비직교 격자

   행 간격이 더 좁을수록 일부 원자에 가장 가까운 이웃이 대각선 방향에 위치하는 격자를 생성할 수 있습니다. 대표적인 예로는 삼각형, 육각형, 카고메 격자 및 일부 준결정이 있습니다.

2. 튜닝 가능한 격자 계열

   AHS 프로그램에서 상호 작용은 원자 쌍 간의 거리를 조정하여 튜닝됩니다. 행 간격이 더 좁을수록 원자 구조를 정의하는 각도와 거리가 최소 행 간격 제약 조건의 제한을 덜 받으므로, 서로 상대적으로 다른 원자 쌍의 상호 작용을 더 자유롭게 조정할 수 있습니다. 대표적인 예로는 결합 길이가 다른 Shastry-Sutherland 격자 계열이 있습니다.

IQM 디바이스의 동적 회로

IQM 디바이스의 동적 회로를 사용하면 MCM(중간 회로 측정) 및 피드포워드 연산을 수행할 수 있습니다. 이러한 기능을 통해 양자 연구원과 개발자는 조건부 논리 및 큐비트 재사용 기능을 갖춘 고급 양자 알고리즘을 구현할 수 있습니다. 이 실험 기능은 리소스 효율성이 향상된 양자 알고리즘을 탐구하고 양자 오류 완화 및 오류 정정 방식을 연구하는 데 도움이 됩니다.

주요 명령:

- `measure_ff`: 피드포워드 제어에 대한 측정을 구현하여 큐비트를 측정하고 피드백 키를 사용하여 결과를 저장합니다.
- `cc_prx`: 지정된 피드백 키와 연결된 결과가 |1# 상태를 측정하는 경우에만 적용되는 고전적 제어 회전을 구현합니다.

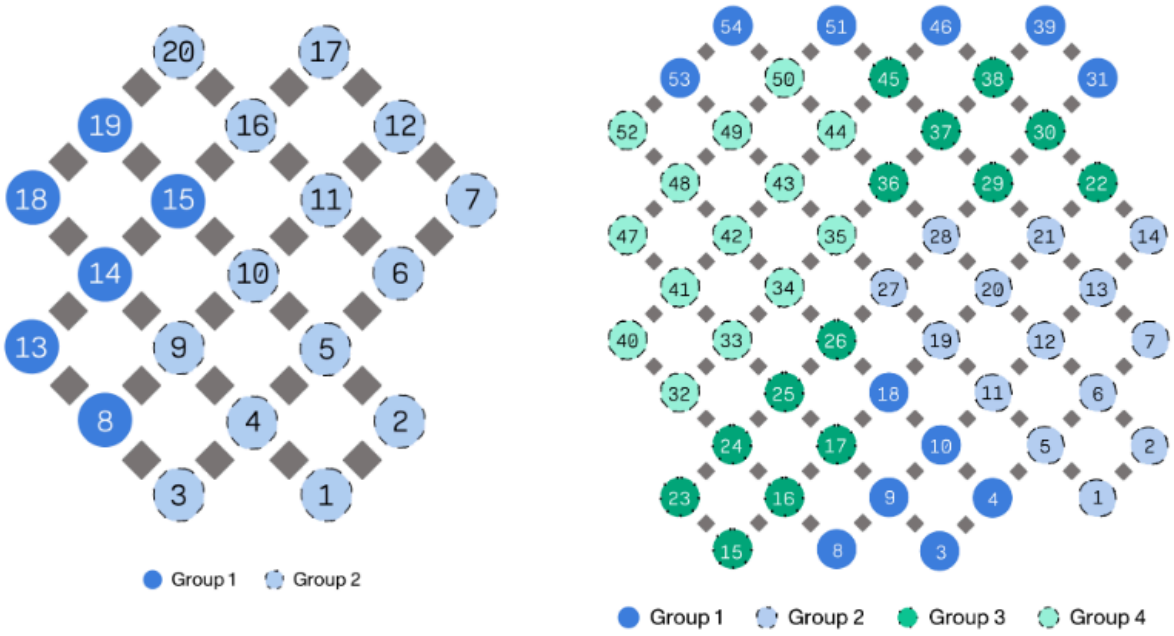
Amazon Braket은 OpenQASM, Amazon Braket SDK 및 Amazon Braket Qiskit Provider를 통해 동적 회로를 지원합니다.

제약 조건:

1. `measure_ff` 명령의 피드백 키는 고유해야 합니다.
2. `cc_prx`는 동일한 피드백 키로 `measure_ff` 이후에 발생해야 합니다.
3. 단일 회로에서 큐비트의 피드포워드는 자체적으로 또는 다른 큐비트로만 제어할 수 있습니다. 서로 다른 회로에서는 서로 다른 제어 쌍을 가질 수 있습니다.
   a. 예를 들어 큐비트 1이 큐비트 2로 제어되는 경우 동일한 회로에서 큐비트 3으로 제어할 수 없습니다. 큐비트 1과 큐비트 2 사이에 제어가 적용되는 횟수에는 제약이 없습니다. 큐비트 2에서 활성 초기화를 수행하지 않는 한, 큐비트 2는 큐비트 3(또는 큐비트 1)으로 제어할 수 있습니다.
4. 제어는 동일한 그룹 내의 큐비트에만 적용할 수 있습니다. IQM Garnet 및 Emerald 디바이스의 큐비트 그룹은 다음 이미지에 있습니다.
5. 이러한 기능이 있는 프로그램은 축어적 프로그램으로 제출해야 합니다. 축어적 프로그램에 대한 자세한 내용은 OpenQASM 3.0으로 축어적 컴파일을 참조하세요.

제한 사항:

MCM은 프로그램에서 피드포워드 제어에만 사용할 수 있습니다. MCM 결과(0 또는 1)는 작업 결과의 일부로 반환되지 않습니다.



이러한 이미지는 두 IQM 디바이스 모두에 대한 큐비트 그룹을 표시합니다. Garnet 20큐비트 디바이스에는 2개의 큐비트 그룹이 포함되어 있고, Emerald 54큐비트 디바이스에는 4개의 큐비트 그룹이 포함되어 있습니다.

예:

1. 활성 초기화를 통한 큐비트 재사용

   조건부 초기화 연산을 지원하는 MCM은 단일 회로 실행 내에서 큐비트 재사용을 가능하게 합니다. 따라서 회로 깊이 요구 사항이 줄어들고 양자 디바이스 리소스 사용률이 향상됩니다.

2. 활성 비트 플립 보호

   동적 회로는 비트 플립 오류를 감지하고 측정 결과에 따라 정정 연산을 적용합니다. 이 구현은 양자 오류 감지 실험 역할을 합니다.

3. 순간이동 실험

   상태 순간이동은 로컬 양자 연산과 MCM의 고전적 정보를 사용하여 큐비트 상태를 전송합니다. 게이트 순간이동은 직접 양자 연산 없이 큐비트 간에 게이트를 구현합니다. 이러한 실험은 양자 오류 정정, 측정 기반 양자 컴퓨팅, 양자 통신이라는 세 가지 핵심 영역에서 기본적인 서브루틴을 보여줍니다.

4. 개방 양자 시스템 시뮬레이션

동적 회로는 데이터 큐비트 및 환경 얽힘과 환경 측정을 통해 양자 시스템의 노이즈를 모델링합니다. 이 접근 방식은 특정 큐비트를 사용하여 데이터 및 환경 요소를 나타냅니다. 노이즈 채널은 환경에 적용되는 게이트 및 측정을 통해 설계될 수 있습니다.

동적 회로 사용에 대한 자세한 내용은 Amazon Braket 노트북 리포지토리의 추가 예제를 참조하세요.

Amazon Braket의 펄스 제어

펄스는 양자 컴퓨터의 큐비트를 제어하는 아날로그 신호입니다. Amazon Braket의 특정 디바이스를 사용하면 펄스 제어 기능에 액세스하여 펄스를 사용해 회로를 제출할 수 있습니다. OpenQASM 3.0을 사용하여 Braket SDK를 통해 또는 Braket API를 통해 직접 펄스 제어에 액세스할 수 있습니다. 먼저 Braket의 펄스 제어에 대한 몇 가지 주요 개념을 소개합니다.

이 섹션의 내용:

- 프레임
- 포트
- 파형
- Hello Pulse 작업
- 펄스를 사용하여 네이티브 게이트에 액세스

프레임

프레임은 양자 프로그램 내에서 시계 역할과 위상 역할을 모두 수행하는 소프트웨어 추상화입니다. 시계 시간은 각 사용량과 주파수로 정의된 상태 저장 캐리어 신호에 따라 증가합니다. 신호를 큐비트로 전송할 때 프레임은 큐비트의 캐리어 주파수, 위상 오프셋 및 파형 엔벨로프가 방출되는 시간을 결정합니다. Braket Pulse에서 프레임 구성은 디바이스, 주파수 및 위상에 따라 달라집니다. 디바이스에 따라 사전 정의된 프레임을 선택하거나 포트를 제공하여 새 프레임을 인스턴스화할 수 있습니다.

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

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

# Create a custom frame
```

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

포트

포트는 큐비트를 제어하는 모든 입력/출력 하드웨어 구성 요소를 나타내는 소프트웨어 추상화입니다. 이를 통해 하드웨어 공급업체는 사용자가 큐비트를 조작하고 관찰하기 위해 상호 작용할 수 있는 인터페이스를 제공할 수 있습니다. 포트는 커넥터의 이름을 나타내는 단일 문자열로 특성화됩니다. 또한 이 문자열은 파형을 얼마나 세밀하게 정의할 수 있는지를 지정하는 최소 시간 증분을 표시합니다.

```
from braket.pulse import Port

Port0 = Port("channel_0", dt=1e-9)
```

파형

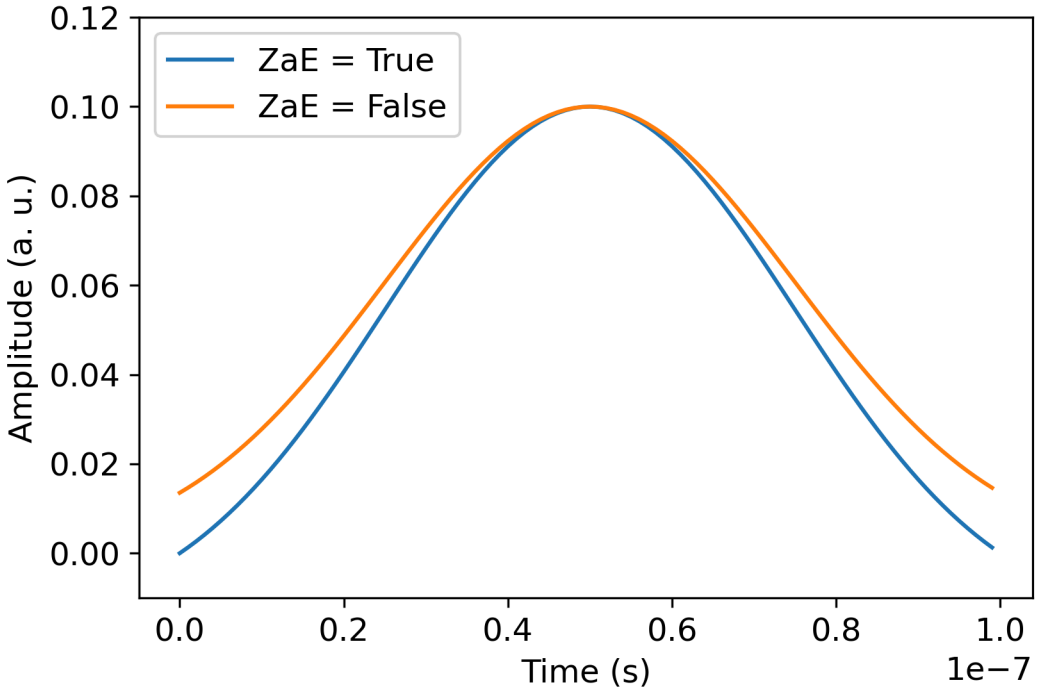
파형은 출력 포트에서 신호를 방출하거나 입력 포트를 통해 신호를 캡처하는 데 사용할 수 있는 시간 종속 엔벨로프입니다. 복소수 목록을 통해 또는 하드웨어 공급자로부터 목록을 생성하는 파형 템플릿을 사용하여 직접 파형을 지정할 수 있습니다.

```
from braket.pulse import ArbitraryWaveform, ConstantWaveform
import numpy as np

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

Braket Pulse는 상수 파형, 가우시안 파형, DRAG(Derivative Removal by Adiabatic Gate) 파형을 포함한 표준 파형 라이브러리를 제공합니다. 다음 예제와 같이 `sample` 함수를 통해 파형 데이터를 검색하여 파형의 모양을 그릴 수 있습니다.

```
from braket.pulse import GaussianWaveform
import numpy as np
import matplotlib.pyplot as plt

zero_at_edge1 = GaussianWaveform(1e-7, 25e-9, 0.1, True)
# or zero_at_edge1 = GaussianWaveform(1e-7, 25e-9, 0.1)
zero_at_edge2 = GaussianWaveform(1e-7, 25e-9, 0.1, False)

times_1 = np.arange(0, zero_at_edge1.length, drive_frame.port.dt)
times_2 = np.arange(0, zero_at_edge2.length, drive_frame.port.dt)
```

```
plt.plot(times_1, zero_at_edge1.sample(drive_frame.port.dt))
plt.plot(times_2, zero_at_edge2.sample(drive_frame.port.dt))
```



위의 이미지는 `GaussianWaveform`에서 생성된 가우시안 파형을 보여줍니다. 펄스 길이 100ns, 너비 25ns, 진폭 0.1(임의 단위)을 선택했습니다. 파형은 펄스 창 중앙에 위치합니다. `GaussianWaveform`은 부울 인수`zero_at_edges`(범례의 ZaE)를 허용합니다. `True`로 설정하면 이 인수는 t=0 및 t=`length`의 점이 0이 되도록 가우시안 파형을 오프셋하고 최대값이 `amplitude` 인수에 해당하도록 진폭의 크기를 재조정합니다.

Hello Pulse 작업

이 섹션에서는 Rigetti 디바이스에서 펄스를 사용하여 직접 단일 큐비트 게이트를 특성화하고 구성하는 방법을 알아봅니다. 큐비트에 전자기장을 가하면 Rabi 진동이 발생하여 큐비트가 0 상태와 1 상태 사이를 전환합니다. 펄스의 보정된 길이와 위상을 사용하면 Rabi 진동이 단일 큐비트 게이트를 계산할 수 있습니다. 여기서는 보다 복잡한 펄스 시퀀스를 빌드하는 데 사용되는 기본 블록인 pi/2 펄스를 측정하기 위한 최적의 펄스 길이를 결정합니다.

먼저 펄스 시퀀스를 빌드하려면 `PulseSequence` 클래스를 가져옵니다.

```
from braket.aws import AwsDevice
```

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

import numpy as np
```

그런 다음 QPU의 Amazon Resource Name(ARN)을 사용하여 새 Braket 디바이스를 인스턴스화합니다. 다음 코드 블록은 Rigetti Ankaa-3을 사용합니다.

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

다음 펄스 시퀀스에는 파형 재생과 큐비트 측정이라는 두 가지 구성 요소가 포함됩니다. 펄스 시퀀스는 일반적으로 프레임에 적용할 수 있습니다. 장벽 및 지연과 같은 몇 가지 예외를 제외하고, 큐비트에 적용할 수 있습니다. 펄스 시퀀스를 구성하기 전에 사용 가능한 프레임을 검색해야 합니다. 드라이브 프레임은 Rabi 진동에 대한 펄스를 적용하는 데 사용되며, 판독 프레임은 큐비트 상태를 측정하는 데 사용됩니다. 이 예제에서는 큐비트 25의 프레임을 사용합니다.

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

이제 드라이브 프레임에서 재생할 파형을 생성합니다. 목표는 다양한 펄스 길이에 대한 큐비트의 동작을 특성화하는 것입니다. 매번 길이가 다른 파형을 재생합니다. 매번 새 파형을 인스턴스화하는 대신, Braket에서 지원하는 펄스 시퀀스 내 `FreeParameter`를 사용합니다. 자유 파라미터로 파형과 펄스 시퀀스를 한 번 생성한 다음, 다른 입력 값으로 동일한 펄스 시퀀스를 실행할 수 있습니다.

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

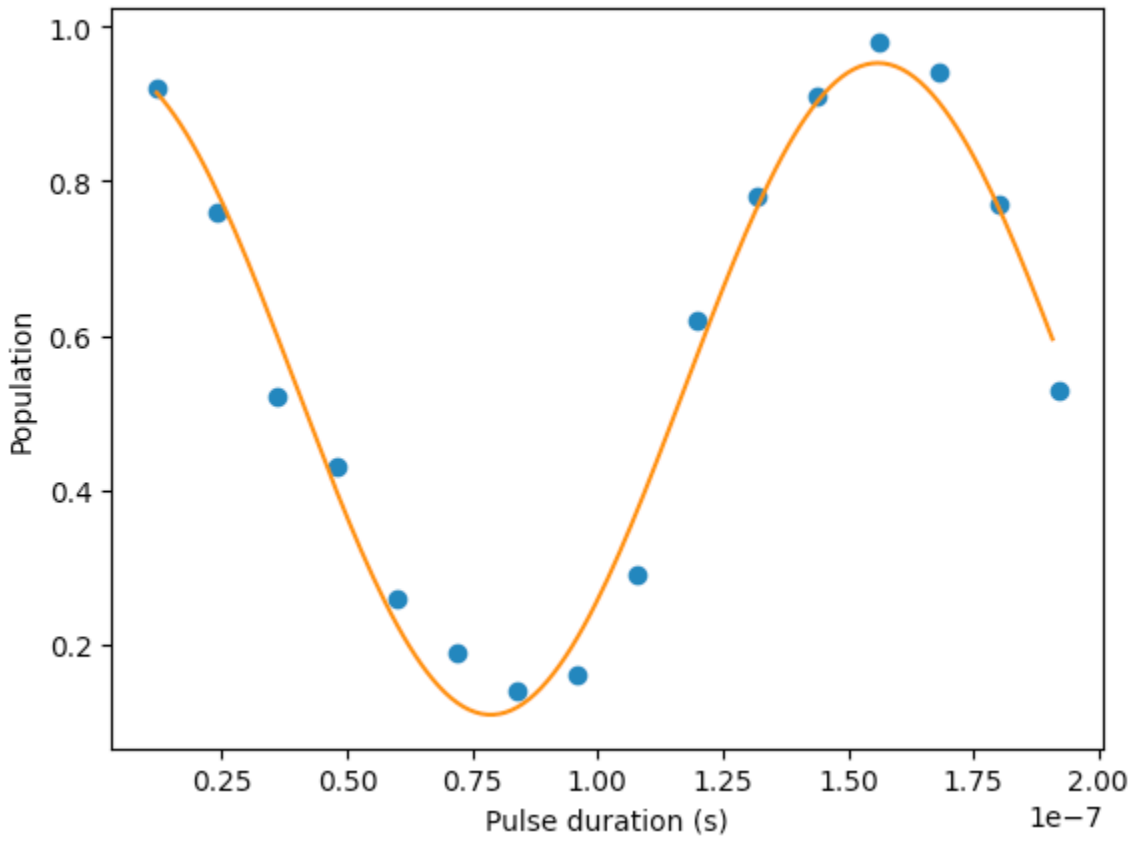
마지막으로, 펄스 시퀀스로 결합합니다. 펄스 시퀀스에서 `play`는 드라이브 프레임에서 지정된 파형을 재생하고 `capture_v0`은 판독 프레임에서 상태를 측정합니다.

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

다양한 펄스 길이를 스캔하여 QPU에 제출합니다. QPU에서 펄스 시퀀스를 실행하기 전에 자유 파라미터의 값을 바인딩합니다.

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

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

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

큐비트 측정의 통계는 0 상태와 1 상태 사이에서 진동하는 큐비트의 진동 역학을 보여줍니다. 측정 데이터에서 Rabi 주파수를 추출하고 펄스 길이를 미세 조정하여 특정 1큐비트 게이트를 구현할 수 있습니다. 예를 들어 아래 그림의 데이터에서 주기성은 약 154ns입니다. 따라서 pi/2 회전 게이트는 길이 =38.5ns인 펄스 시퀀스에 해당합니다.



OpenPulse를 사용한 Hello Pulse

OpenPulse는 일반 양자 디바이스의 펄스 수준 제어를 지정하는 언어이며 OpenQASM 3.0 사양의 일부입니다. Amazon Braket은 OpenQASM 3.0 표현을 사용하여 펄스를 직접 프로그래밍할 수 있도록 OpenPulse를 지원합니다.

Braket은 네이티브 명령에서 펄스를 표현하기 위한 기본 중간 표현으로 OpenPulse를 사용합니다. OpenPulse는 `defcal`("define calibration"의 줄임말) 선언 형태로 명령 보정 추가를 지원합니다. 이러한 선언을 사용하면 저수준 제어 문법 내에서 게이트 명령의 구현을 지정할 수 있습니다.

다음 명령을 사용하여 Braket `PulseSequence`의 OpenPulse 프로그램을 볼 수 있습니다.

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

OpenPulse 프로그램을 직접 구성할 수도 있습니다.

```
from braket.ir.openqasm import Program

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

스크립트로 `Program` 객체를 생성합니다. 그런 다음 프로그램을 QPU에 제출합니다.

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

program = Program(source=openpulse_script)

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

펄스를 사용하여 네이티브 게이트에 액세스

연구원은 특정 QPU에서 지원하는 네이티브 게이트가 어떻게 펄스로 구현되는지 정확히 알아야 하는 경우가 많습니다. 펄스 시퀀스는 하드웨어 공급자에 의해 주의 깊게 보정되지만, 이러한 시퀀스에 대한 액세스를 통해 연구원은 더 나은 게이트를 설계하거나 특정 게이트의 펄스를 확장하여 제로 노이즈 외삽과 같은 오류 완화를 위한 프로토콜을 탐구할 기회를 얻습니다.

Amazon Braket은 Rigetti의 네이티브 게이트에 대한 프로그래밍 방식 액세스를 지원합니다.

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

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

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

> **Note**
>
> 하드웨어 공급자는 종종 하루에 두 번 이상 QPU를 주기적으로 보정합니다. Braket SDK를 사용하면 최신 게이트 보정을 얻을 수 있습니다.

```
device.refresh_gate_calibrations()
```

RX 또는 XY 게이트와 같은 지정된 네이티브 게이트를 검색하려면 `Gate` 객체와 관심 대상 큐비트를 전달해야 합니다. 예를 들어 qubit 0에 적용된 RX(π/2)의 펄스 구현을 검사할 수 있습니다.

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

pulse_sequence_rx_pi_2_q0 = calibrations.pulse_sequences[rx_pi_2_q0]
```

`filter` 함수를 사용하여 필터링된 보정 세트를 생성할 수 있습니다. 게이트 목록 또는 `QubitSet` 목록을 전달합니다. 다음 코드는 RX(π/2) 및 qubit 0에 대한 모든 보정을 포함하는 두 세트를 생성합니다.

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

```
q0_calibrations = calibrations.filter(qubits=QubitSet([0]))
```

이제 사용자 지정 보정 세트를 연결하여 네이티브 게이트의 작업을 제공하거나 수정할 수 있습니다. 예를 들어 다음 회로를 고려합니다.

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

PulseSequence 객체의 딕셔너리를 gate_definitions 키워드 인수에 전달하여 qubit 0의 rx 게이트에 대한 사용자 지정 게이트 보정으로 실행할 수 있습니다. GateCalibrations 객체의 속성 pulse_sequences에서 딕셔너리를 구성할 수 있습니다. 지정되지 않은 모든 게이트는 양자 하드웨어 공급자의 펄스 보정으로 대체됩니다.

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

아날로그 해밀토니안 시뮬레이션

아날로그 해밀토니안 시뮬레이션(AHS)은 기존 양자 회로 모델과는 현저히 다른 양자 컴퓨팅의 새로운 패러다임입니다. 각 회로가 한 번에 몇 개의 큐비트에서만 작동하는 게이트 시퀀스를 대신합니다. AHS 프로그램은 해당 해밀토니안의 시간 종속 및 공간 종속 파라미터로 정의됩니다. 시스템의 해밀토니안은 에너지 준위와 외부 힘의 영향을 인코딩하며, 이는 상태의 시간 변화를 함께 제어합니다. N-큐비트 시스템의 경우 해밀토니안은 복소수의 $2^N X 2^N$ 정방 행렬로 표현할 수 있습니다.

AHS를 수행할 수 있는 양자 디바이스는 내부 제어 파라미터를 신중하게 튜닝하여 사용자 지정 해밀토니안에서 양자 시스템의 시간 변화를 밀접하게 근사화하도록 설계되었습니다. 결맞음 구동장의 진폭 조정 및 파라미터 디튜닝을 예로 들 수 있습니다. AHS 패러다임은 응집 물질 물리학 또는 양자 화학과 같이 상호 작용하는 입자가 많은 양자 시스템의 정적 및 동적 특성을 시뮬레이션하는 데 매우 적합합니다. QuEra의 Aquila 디바이스와 같이 특수 제작된 양자 처리 장치(QPU)는 AHS의 장점을 사용하고 기존의 디지털 양자 컴퓨팅 접근 방식을 넘어 혁신적인 방식으로 문제를 해결하도록 개발되었습니다.

이 섹션의 내용:

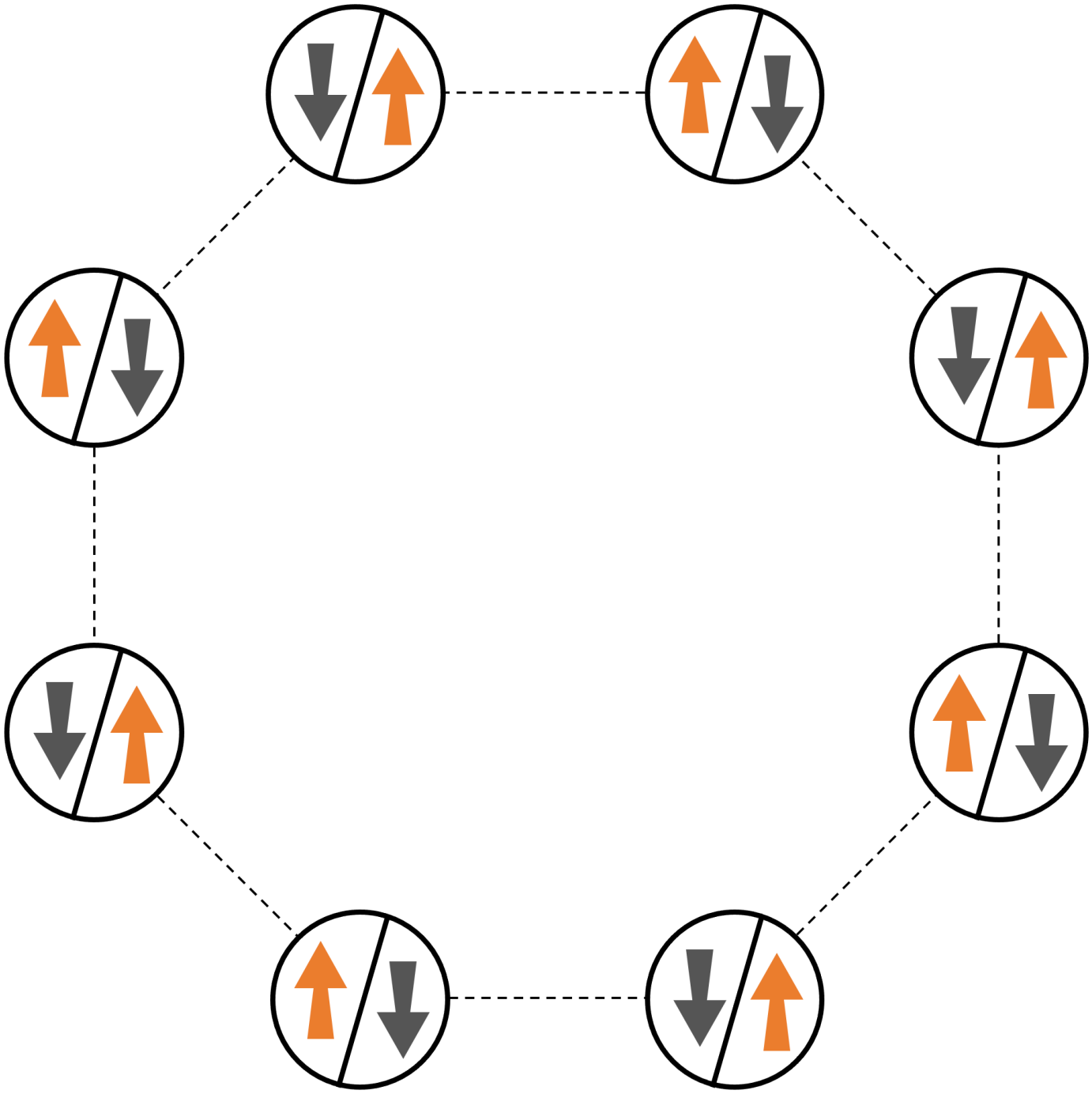
Hello AHS: 첫 번째 아날로그 해밀토니안 시뮬레이션 실행

이 섹션에서는 첫 번째 아날로그 해밀토니안 시뮬레이션을 실행하는 방법에 대한 정보를 제공합니다.

이 섹션의 내용:

상호 작용 스핀 체인

상호 작용하는 입자가 많은 시스템의 표준 예제로, 8개의 스핀으로 이루어진 고리를 생각해 보겠습니다(각각 “up” |↑# 및 “down” |↓# 상태일 수 있음). 이 모델 시스템은 비록 작지만, 이미 자연적으로 발생하는 자성 물질의 몇 가지 흥미로운 현상을 보여줍니다. 이 예제에서는 연속적인 스핀이 반대 방향을 가리키는 소위 반강자성 정렬을 준비하는 방법을 보여줍니다.



배열

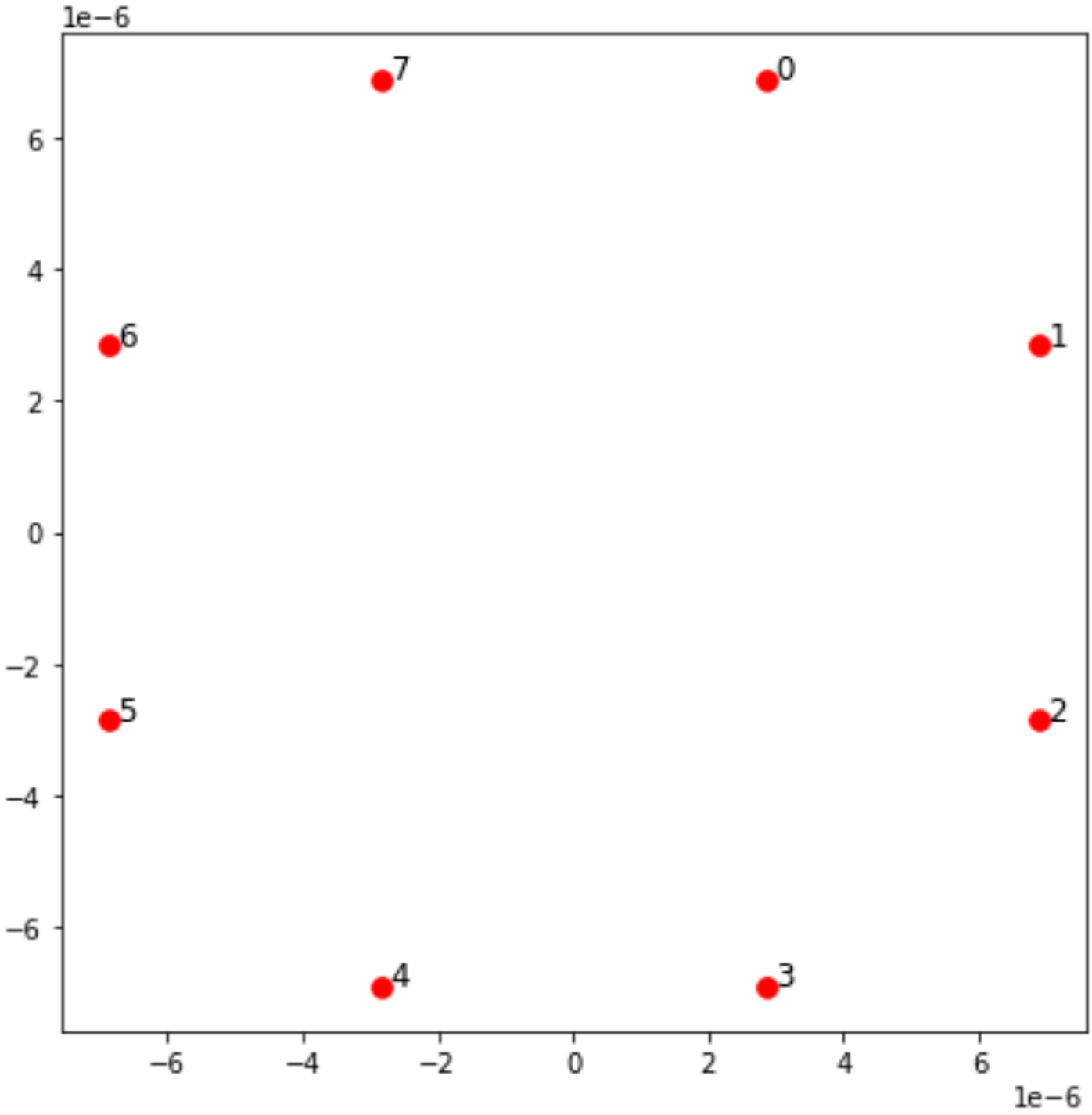
각 스핀을 나타내기 위해 하나의 중성 원자를 사용하며, “up” 및 “down” 스핀 상태는 각각 원자의 여기된 Rydberg 상태와 바닥 상태로 인코딩됩니다. 먼저 2차원 배열을 생성합니다. 다음 코드를 사용하여 위의 스핀 고리를 프로그래밍할 수 있습니다.

사전 조건: Braket SDK를 pip 설치해야 합니다. (Braket 호스팅 노트북 인스턴스를 사용하는 경우 이 SDK는 노트북과 함께 사전 설치되어 제공됩니다.) 플롯을 재현하려면 shell 명령 `pip install matplotlib`을 사용하여 matplotlib을 별도로 설치해야 합니다.

```
from braket.ahs.atom_arrangement import AtomArrangement
import numpy as np
import matplotlib.pyplot as plt  # Required for plotting

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

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

또한 다음으로 플롯할 수 있습니다.

```
fig, ax = plt.subplots(1, 1, figsize=(7, 7))
xs, ys = [register.coordinate_list(dim) for dim in (0, 1)]
ax.plot(xs, ys, 'r.', ms=15)

for idx, (x, y) in enumerate(zip(xs, ys)):
    ax.text(x, y, f" {idx}", fontsize=12)

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



상호 작용

반강자성 위상을 준비하려면 이웃하는 스핀 간의 상호 작용을 유도해야 합니다. 이를 위해 van der Waals 상호 작용을 사용합니다. 이 상호 작용은 기본적으로 중성 원자 디바이스(예: QuEra의 Aquila 디바이스)에 의해 구현됩니다. 스핀 표현을 사용하면 이 상호 작용에 대한 해밀토니안 항을 모든 스핀 쌍 (j,k)에 대한 합으로 표현할 수 있습니다.

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

여기서 nj=|↑$_j$##↑$_j$|는 스핀 j가 “up” 상태에 있는 경우에만 1, 그렇지 않으면 0의 값을 취하는 연산자입니다. 강도는 $V_{j,k}=C_6/(d_{j,k})^6$이며, 여기서 C_6은 고정 계수이고 $d_{j,k}$는 스핀 j와 k 사이의 유클리드 거리입니다. 이 상호 작용 항의 즉각적인 효과는 스핀 j와 스핀 k가 “up”인 모든 상태의 에너지가 ($V_{j,k}$ 만큼) 증가한다는 것입니다. AHS 프로그램의 나머지 부분을 신중하게 설계함으로써, 이 상호 작용은 이웃한 스핀이 모두 “up” 상태에 있는 것을 방지합니다. 이는 일반적으로 "Rydberg 봉쇄"로 알려진 효과입니다.

구동장

AHS 프로그램을 시작할 때 모든 스핀은 (기본적으로) “down” 상태에서 시작되며 소위 강자성 위상에 있습니다. 반강자성 위상을 준비한다는 목표를 염두에 두고, 이 상태에서 "up" 상태가 선호되는 다체 상태로 스핀을 원활하게 전환하는 시간 종속 결맞음 구동장을 지정합니다. 해당 해밀토니안은 다음과 같이 쓸 수 있습니다.

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

여기서 Ω(t),ϕ(t),Δ(t)는 모든 스핀에 균일하게 영향을 미치는 구동장의 시간 종속 전역 진폭(Rabi 주파수라고도 함), 위상 및 디튜닝입니다. 여기서 $S_{-,k}$=|↓$_k$##↑$_k$|and $S_{+,k}=(S_{-,k})^{\dagger}$=|↑$_k$##↓$_k$|은 각각 스핀 k의 하강 및 상승 연산자이고, n_k=|↑$_k$##↑$_k$|는 이전과 동일한 연산자입니다. 구동장의 Ω 부분은 모든 스핀의 "down" 상태와 "up" 상태를 동시에 일관되게 결합하고 Δ 부분은 "up" 상태에 대한 에너지 보상을 제어합니다.

강자성 위상에서 반강자성 위상으로의 원활한 전환을 프로그래밍하기 위해 다음 코드로 구동장을 지정합니다.

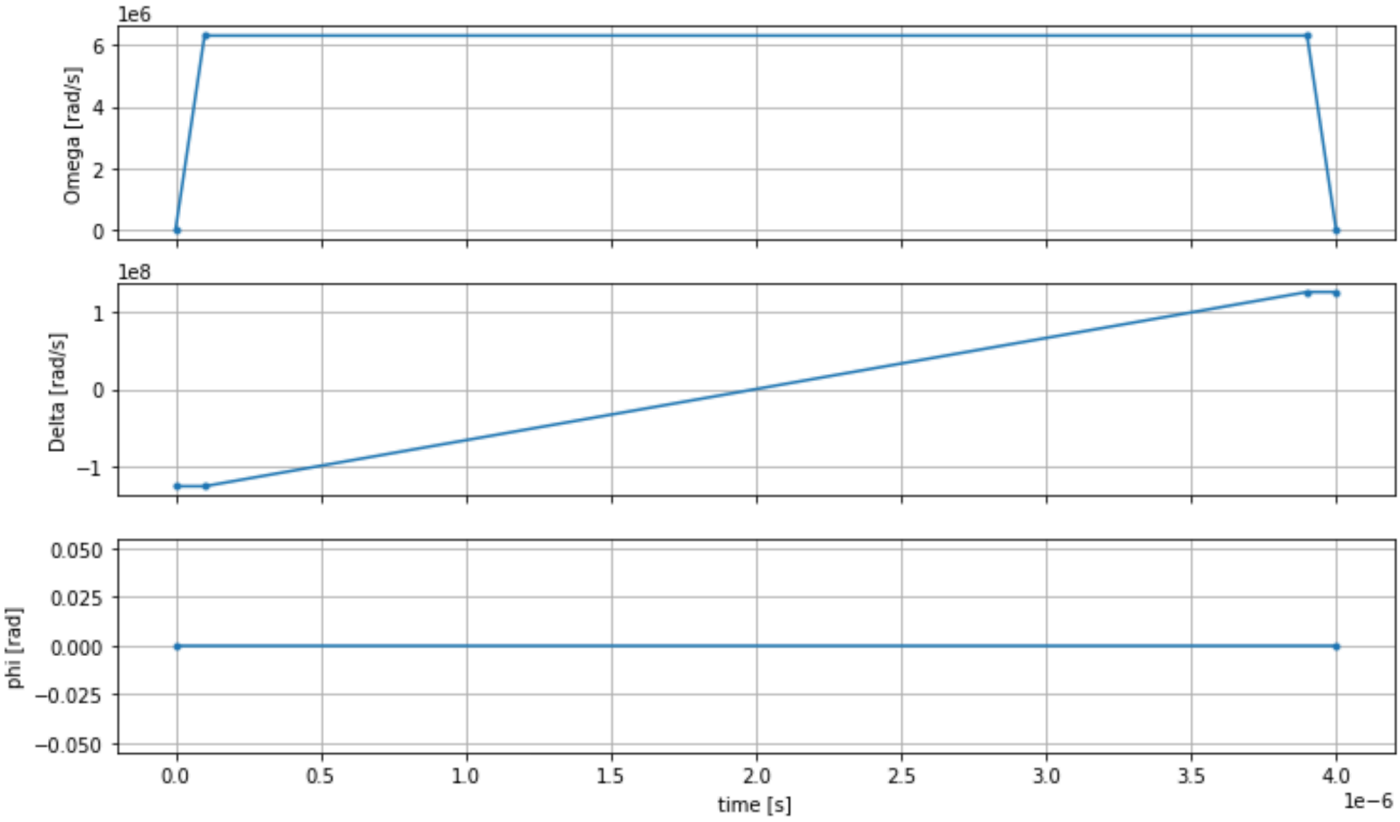
```
from braket.timings.time_series import TimeSeries
from braket.ahs.driving_field import DrivingField

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

omega = TimeSeries()
```

```
omega.put(0.0, 0.0)
omega.put(time_ramp, omega_max)
omega.put(time_max - time_ramp, omega_max)
omega.put(time_max, 0.0)

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

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

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

다음 스크립트를 사용하여 구동장의 시계열을 시각화할 수 있습니다.

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

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

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

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

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



AHS 프로그램

레지스터, 구동장(및 암시적 van der Waals 상호 작용)은 아날로그 해밀토니안 시뮬레이션 프로그램 `ahs_program`을 구성합니다.

```
from braket.ahs.analog_hamiltonian_simulation import AnalogHamiltonianSimulation

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

로컬 시뮬레이터에서 실행

이 예제는 작기 때문에(스핀 수 15회 미만), AHS 호환 QPU에서 실행하기 전에 Braket SDK와 함께 제공되는 로컬 AHS 시뮬레이터에서 실행할 수 있습니다. Braket SDK를 통해 로컬 시뮬레이터를 무료로 사용할 수 있으므로, 이는 코드가 올바르게 실행될 수 있도록 보장하는 모범 사례입니다.

여기서는 로컬 시뮬레이터가 양자 상태의 시간 변화를 추적하고 최종 상태에서 샘플을 추출하므로 샷 수를 높은 값(예: 100만)으로 설정할 수 있습니다. 따라서 샷 수가 늘어나더라도 총 실행 시간은 약간만 증가합니다.

```
from braket.devices import LocalSimulator

device = LocalSimulator("braket_ahs")

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

시뮬레이터 결과 분석

각 스핀의 상태("down"은 "d", "up"은 "u", 빈 위치는 "e"일 수 있음)를 추론하고 각 구성이 샷 전체에서 발생한 횟수를 계산하는 다음 함수를 사용하여 샷 결과를 집계할 수 있습니다.

```
from collections import Counter


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

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

    Args:
      result
 (braket.tasks.analog_hamiltonian_simulation_quantum_task_result.AnalogHamiltonianSimulationQua

    Returns
       dict: number of times each state configuration is measured

    """
    state_counts = Counter()
    states = ['e', 'u', 'd']
    for shot in result.measurements:
        pre = shot.pre_sequence
```

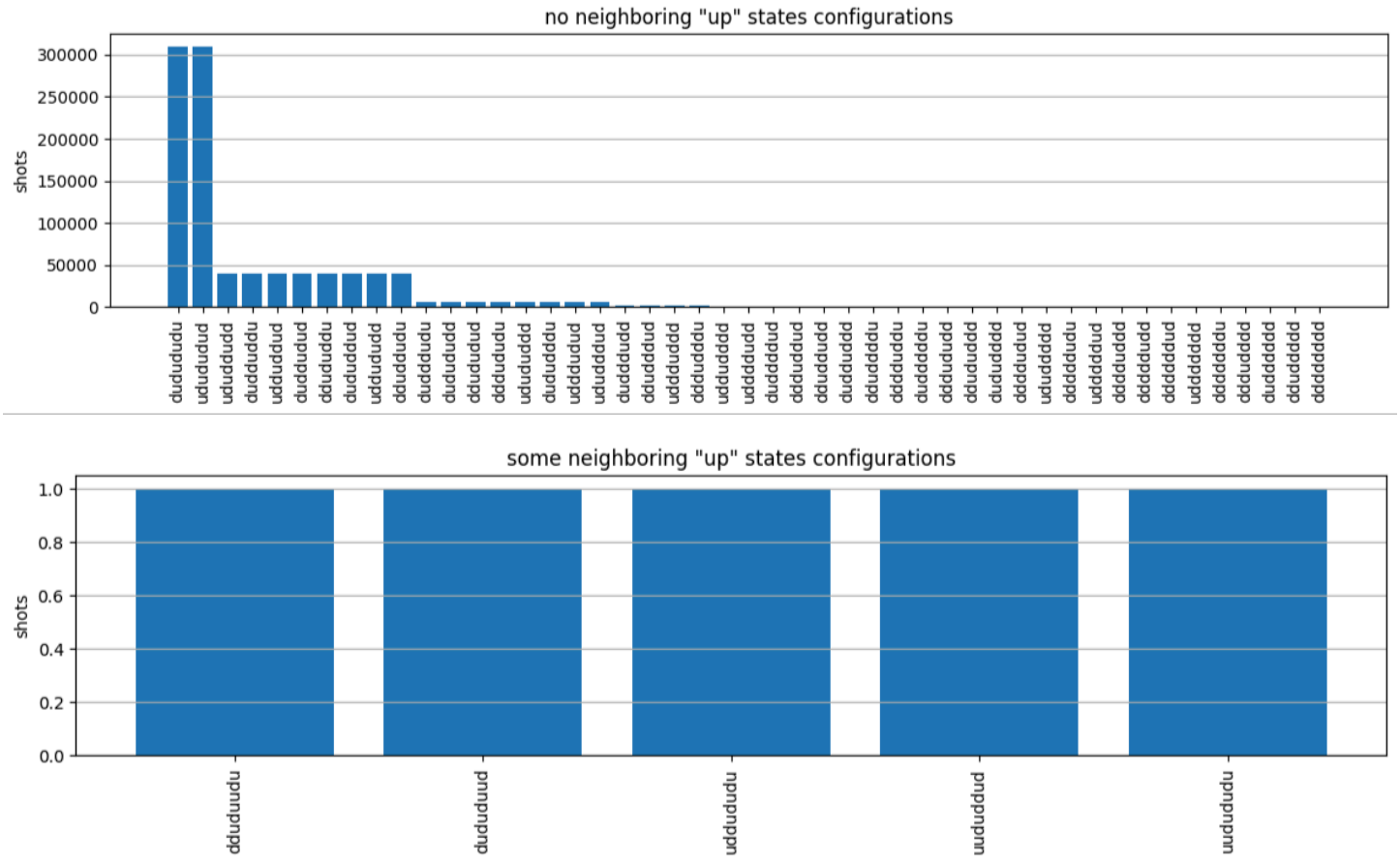
```
        post = shot.post_sequence
        state_idx = np.array(pre) * (1 + np.array(post))
        state = "".join(map(lambda s_idx: states[s_idx], state_idx))
        state_counts.update((state,))
    return dict(state_counts)


counts_simulator = get_counts(result_simulator)  # Takes about 5 seconds
print(counts_simulator)
```

```
*[Output]*
{'dddddddd': 5, 'ddddddddu': 12, 'ddddddud': 15, ...}
```

다음 counts는 샷 전체에서 각 상태 구성이 관찰된 횟수를 계산하는 딕셔너리입니다. 다음 코드로 시각화할 수도 있습니다.

```
from collections import Counter


def has_neighboring_up_states(state):
    if 'uu' in state:
        return True
    if state[0] == 'u' and state[-1] == 'u':
        return True
    return False


def number_of_up_states(state):
    return Counter(state)['u']


def plot_counts(counts):
    non_blockaded = []
    blockaded = []
    for state, count in counts.items():
        if not has_neighboring_up_states(state):
            collection = non_blockaded
        else:
            collection = blockaded
        collection.append((state, count, number_of_up_states(state)))

    blockaded.sort(key=lambda _: _[1], reverse=True)
    non_blockaded.sort(key=lambda _: _[1], reverse=True)
```

```
    for configurations, name in zip((non_blockaded,
                                     blockaded),
                                    ('no neighboring "up" states',
                                     'some neighboring "up" states')):
        plt.figure(figsize=(14, 3))
        plt.bar(range(len(configurations)), [item[1] for item in configurations])
        plt.xticks(range(len(configurations)))
        plt.gca().set_xticklabels([item[0] for item in configurations], rotation=90)
        plt.ylabel('shots')
        plt.grid(axis='y')
        plt.title(f'{name} configurations')
        plt.show()


plot_counts(counts_simulator)
```



플롯에서 다음과 같은 관찰 가능 항목을 읽고 반강자성 위상을 성공적으로 준비했는지 확인할 수 있습니다.

1. 일반적으로, 비차단 상태(두 개의 이웃하는 스핀이 “up” 상태에 있지 않음)는 하나 이상의 이웃하는 스핀 쌍이 모두 “up” 상태에 있는 상태보다 더 일반적입니다.
2. 일반적으로 구성이 차단되지 않는 한 "up" 여기가 더 많은 상태가 선호됩니다.
3. 가장 일반적인 상태는 실제로 완벽한 반강자성 상태 "dudududu" 및 "udududud"입니다.
4. 두 번째로 일반적인 상태는 연속 분리가 1, 2, 2인 3개의 “up” 여기만 존재하는 상태입니다. 이는 van der Waals 상호 작용이 다음으로 가까운 이웃에도 (비록 훨씬 작지만) 영향을 미친다는 것을 보여줍니다.

QuEra의 Aquila QPU에서 실행

사전 조건: Braket SDK를 pip 설치하는 것 외에도, Amazon Braket을 처음 사용하는 경우 필요한 시작하기 단계를 완료했는지 확인합니다.

> **Note**
>
> Braket 호스팅 노트북 인스턴스를 사용하는 경우 Braket SDK는 인스턴스와 함께 사전 설치되어 제공됩니다.

모든 종속성이 설치된 상태에서 Aquila QPU에 연결할 수 있습니다.

```
from braket.aws import AwsDevice

aquila_qpu = AwsDevice("arn:aws:braket:us-east-1::device/qpu/quera/Aquila")
```

AHS 프로그램을 QuEra 시스템에 적합하게 만들려면 Aquila QPU에서 허용하는 정밀도 수준에 부합하도록 모든 값을 반올림해야 합니다. (이러한 요구 사항은 이름에 “분해능”이 있는 디바이스 파라미터에 의해 관리됩니다. 노트북에서 `aquila_qpu.properties.dict()`를 실행하면 확인할 수 있습니다. Aquila의 기능 및 요구 사항에 대한 자세한 내용은 Aquila 소개 노트북을 참조하세요.) `discretize` 메서드를 직접적으로 호출하여 이 작업을 수행할 수 있습니다.

```
discretized_ahs_program = ahs_program.discretize(aquila_qpu)
```

이제 Aquila QPU에서 프로그램(현재는 100회 샷만 실행)을 실행할 수 있습니다.

> **Note**
>
> Aquila 프로세서에서 이 프로그램을 실행하면 비용이 발생합니다. Amazon Braket SDK에는 고객이 비용 한도를 설정하고 거의 실시간으로 비용을 추적할 수 있는 Cost Tracker가 포함되어 있습니다.

```
task = aquila_qpu.run(discretized_ahs_program, shots=100)

metadata = task.metadata()
task_arn = metadata['quantumTaskArn']
task_status = metadata['status']

print(f"ARN: {task_arn}")
print(f"status: {task_status}")
```

```
*[Output]*
ARN: arn:aws:braket:us-east-1:123456789012:quantum-task/12345678-90ab-
cdef-1234-567890abcdef
status: CREATED
```

양자 작업 실행 시간은 가용성 창과 QPU 사용률에 따라 크게 달라질 수 있으므로, 양자 작업 ARN을 기록해 두는 것이 좋습니다. 그러면 나중에 다음 코드 조각을 사용하여 작업 상태를 확인할 수 있습니다.

```
# Optionally, in a new python session
from braket.aws import AwsQuantumTask

SAVED_TASK_ARN = "arn:aws:braket:us-east-1:123456789012:quantum-task/12345678-90ab-
cdef-1234-567890abcdef"

task = AwsQuantumTask(arn=SAVED_TASK_ARN)
metadata = task.metadata()
task_arn = metadata['quantumTaskArn']
task_status = metadata['status']

print(f"ARN: {task_arn}")
print(f"status: {task_status}")
```

```
*[Output]*
```

```
ARN: arn:aws:braket:us-east-1:123456789012:quantum-task/12345678-90ab-
cdef-1234-567890abcdef
status: COMPLETED
```

상태가 COMPLETED(Amazon Braket 콘솔의 양자 작업 페이지에서도 확인할 수 있음)가 되면 다음을 사용하여 결과를 쿼리할 수 있습니다.

```
result_aquila = task.result()
```

QPU 결과 분석

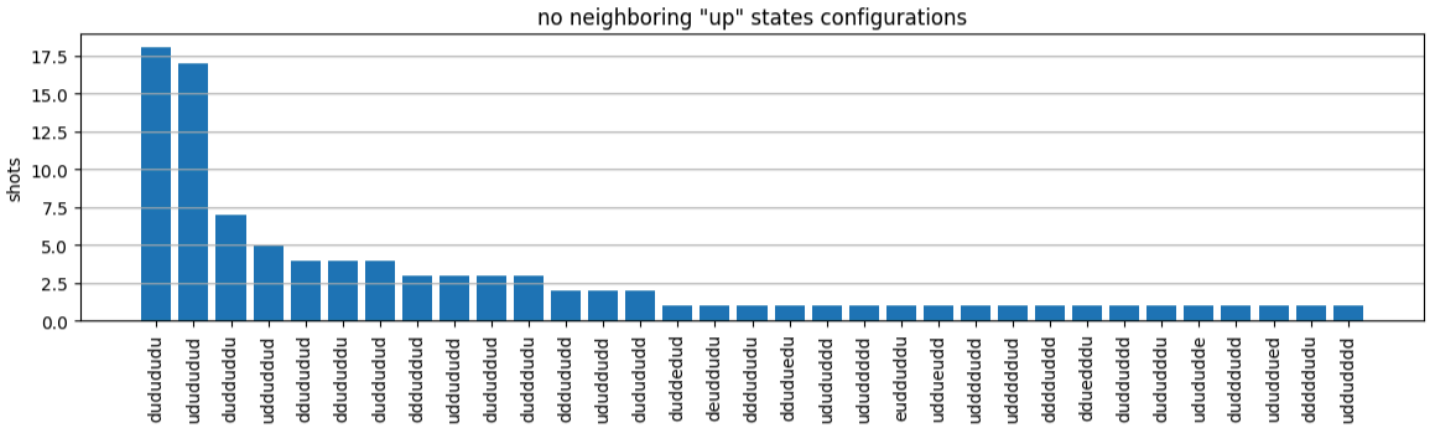
이전과 동일한 `get_counts` 함수를 사용하여 개수를 계산할 수 있습니다.

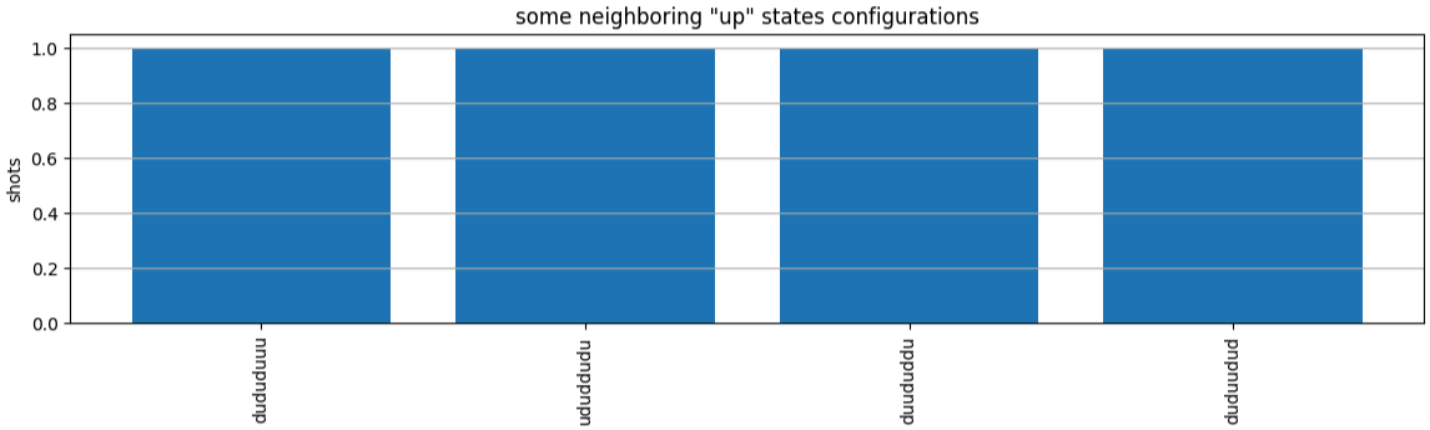
```
counts_aquila = get_counts(result_aquila)
   print(counts_aquila)
```

```
*[Output]*
{'ddddududd': 2, 'dudududu': 18, 'ddududud': 4, ...}
```

그리고 `plot_counts`로 플롯합니다.

```
plot_counts(counts_aquila)
```





일부 샷에는 빈 위치("e"로 표시)가 있습니다. 이는 Aquila QPU의 원자당 1~2%에 달하는 준비 결함에 의한 것입니다. 이 외에도, 결과는 적은 샷 수로 인해 예상되는 통계적 변동 내에서 시뮬레이션과 일치합니다.

다음 단계

축하합니다, 이제 로컬 AHS 시뮬레이터와 Aquila QPU를 사용하여 Amazon Braket에서 첫 번째 AHS 워크로드를 실행하셨습니다.

Rydberg 물리학, 아날로그 해밀토니안 시뮬레이션 및 Aquila 디바이스에 대해 자세히 알아보려면 예제 노트북을 참조하세요.

QuEra Aquila를 사용하여 아날로그 프로그램 제출

이 페이지에서는 QuEra의 Aquila 시스템 기능에 대한 포괄적인 설명서를 제공합니다. 여기서 다루는 세부 정보는 다음과 같습니다.

1. Aquila가 시뮬레이션한 매개변수화된 해밀토니안
2. AHS 프로그램 파라미터
3. AHS 결과 내용
4. Aquila 기능 파라미터

이 섹션의 내용:

- 해밀토니안
- Braket AHS 프로그램 스키마
- Braket AHS 작업 결과 스키마

- QuEra 디바이스 속성 스키마

해밀토니안

QuEra의 Aquila 시스템은 기본적으로 다음 (시간 종속) 해밀토니안을 시뮬레이션합니다.

$$H(t) = \sum_{k=1}^{N} H_{\text{drive},k}(t) + \sum_{k=1}^{N} H_{\text{local detuning},k}(t) + \sum_{k=1}^{N-1} \sum_{l=k+1}^{N} V_{\text{vdw},k,l}$$

> **Note**
>
> 로컬 디튜닝에 대한 액세스는 실험 기능이며 Braket Direct를 통해 요청하여 사용할 수 있습니다.

where

- $H_{drive,k}(t)=({}^{1}/_{2}\ \Omega(t)e^{i\phi(t)}S_{-,k} + {}^{1}/_{2}\ \Omega(t)e^{-i\phi(t)}\ S_{+,k}) + (-\Delta_{global}(t)n_k)$,
  - Ω(t)는 시간 종속 전역 구동 진폭(Rabi 주파수라고도 함)으로, 단위는 (rad/s)입니다.
  - ϕ(t)는 라디안 단위로 측정되는 시간 종속 전역 위상입니다.
  - $S_{-,k}$와 $S_{+,k}$는 원자 k의 스핀 강하 및 상승 연산자입니다(기저 상태에서 |↓#=|g#, |↑#=|r#, 이는 S_-=|g##r|, $S_+=(S_-)^{\dagger}$=|r##g|).
  - $\Delta_{global}(t)$는 시간 종속 전역 디튜닝입니다.
  - n_k는 원자 k의 Rydberg 상태에 대한 사영 연산자입니다(즉, n=|r##r|).
- $H_{local\ detuning,k}(t)=-\Delta_{local}(t)h_k n_k$
  - $\Delta_{local}(t)$는 (rad/s) 단위로 나타낸 로컬 주파수 이동의 시간 종속 인자입니다.
  - h_k는 위치 종속 인자로, 0.0에서 1.0 사이의 무차원수입니다.
- $V_{vdw,k,l}=C_6/(d_{k,l})^6 n_k n_l$,
  - C_6은van der Waals 계수로, 단위는 (rad / s) * (m)^6입니다.
  - $d_{k,l}$은 원자 k와 l 사이의 유클리드 거리로, 미터 단위로 측정됩니다.

사용자는 Braket AHS 프로그램 스키마를 통해 다음 파라미터를 제어할 수 있습니다.

- 2차원 원자 배열(각 원자 k의 x_k 및 y_k 좌표, um 단위)로, k,l=1,2,…N으로 쌍별 원자 거리 $d_{k,l}$을 제어합니다.

- $\Omega(t)$는 시간 종속 전역 구동 주파수로, 단위는 (rad/s)입니다.
- $\phi(t)$는 시간 종속 전역 위상으로, 단위는 (rad)입니다.
- $\Delta_{global}(t)$는 시간 종속 전역 디튜닝으로, 단위는 (rad/s)입니다.
- $\Delta_{local}(t)$는 로컬 디튜닝 크기의 시간 종속 (전역) 인자로, 단위는 (rad/s)입니다.
- h_k는 로컬 디튜닝 크기의 (정적) 위치 종속 인자로, 0.0에서 1.0 사이의 무차원수입니다.

> **Note**
>
> 사용자는 관련된 수준(즉, S_-,S_+, n 연산자가 고정됨)이나 Rydberg-Rydberg 상호 작용 계수(C_6)의 강도를 제어할 수 없습니다.

Braket AHS 프로그램 스키마

braket.ir.ahs.program_v1.Program 객체(예제)

> **Note**
>
> 계정에 대해 로컬 디튜닝 기능이 활성화되지 않은 경우 다음 예제에서 `localDetuning=[]`을 사용합니다.

```
Program(
    braketSchemaHeader=BraketSchemaHeader(
        name='braket.ir.ahs.program',
        version='1'
    ),
    setup=Setup(
        ahs_register=AtomArrangement(
            sites=[
                [Decimal('0'), Decimal('0')],
                [Decimal('0'), Decimal('4e-6')],
                [Decimal('4e-6'), Decimal('0')]
            ],
            filling=[1, 1, 1]
        )
    ),
    hamiltonian=Hamiltonian(
        drivingFields=[
```

```
            DrivingField(
                amplitude=PhysicalField(
                    time_series=TimeSeries(
                        values=[Decimal('0'), Decimal('15700000.0'),
 Decimal('15700000.0'), Decimal('0')],
                        times=[Decimal('0'), Decimal('0.000001'), Decimal('0.000002'),
 Decimal('0.000003')]
                    ),
                    pattern='uniform'
                ),
                phase=PhysicalField(
                    time_series=TimeSeries(
                        values=[Decimal('0'), Decimal('0')],
                        times=[Decimal('0'), Decimal('0.000003')]
                    ),
                    pattern='uniform'
                ),
                detuning=PhysicalField(
                    time_series=TimeSeries(
                        values=[Decimal('-54000000.0'), Decimal('54000000.0')],
                        times=[Decimal('0'), Decimal('0.000003')]
                    ),
                    pattern='uniform'
                )
            )
        ],
        localDetuning=[
            LocalDetuning(
                magnitude=PhysicalField(
                    times_series=TimeSeries(
                        values=[Decimal('0'), Decimal('25000000.0'),
 Decimal('25000000.0'), Decimal('0')],
                        times=[Decimal('0'), Decimal('0.000001'), Decimal('0.000002'),
 Decimal('0.000003')]
                    ),
                    pattern=Pattern([Decimal('0.8'), Decimal('1.0'), Decimal('0.9')])
                )
            )
        ]
    )
)
```

JSON(예제)

> **Note**
>
> 계정에 대해 로컬 디튜닝 기능이 활성화되지 않은 경우 다음 예제에서 `"localDetuning": []`을 사용합니다.

```
{
    "braketSchemaHeader": {
        "name": "braket.ir.ahs.program",
        "version": "1"
    },
    "setup": {
        "ahs_register": {
            "sites": [
                [0E-7, 0E-7],
                [0E-7, 4E-6],
                [4E-6, 0E-7]
            ],
            "filling": [1, 1, 1]
        }
    },
    "hamiltonian": {
        "drivingFields": [
            {
                "amplitude": {
                    "time_series": {
                        "values": [0.0, 15700000.0, 15700000.0, 0.0],
                        "times": [0E-9, 0.000001000, 0.000002000, 0.000003000]
                    },
                    "pattern": "uniform"
                },
                "phase": {
                    "time_series": {
                        "values": [0E-7, 0E-7],
                        "times": [0E-9, 0.000003000]
                    },
                    "pattern": "uniform"
                },
                "detuning": {
                    "time_series": {
                        "values": [-54000000.0, 54000000.0],
                        "times": [0E-9, 0.000003000]
```

```
                    },
                    "pattern": "uniform"
                }
            }
        ],
        "localDetuning": [
            {
                "magnitude": {
                    "time_series": {
                        "values": [0.0, 25000000.0, 25000000.0, 0.0],
                        "times": [0E-9, 0.000001000, 0.000002000, 0.000003000]
                    },
                    "pattern": [0.8, 1.0, 0.9]
                }
            }
        ]
    }
}
```

기본 필드

프로그램 필드	type	description
setup.ahs_register.sites	List[List[Decimal]]	핀셋이 원자를 포획하는 2차원 좌표 목록
setup.ahs_register.filling	List[int]	트랩 위치를 차지하는 원자는 1로, 빈 위치는 0으로 표시
hamiltonian.drivingFields[].amplitude.time_series.times	List[Decimal]	구동 진폭의 시점, Omega(t)
hamiltonian.drivingFields[].amplitude.time_series.values	List[Decimal]	구동 진폭의 값, Omega(t)
hamiltonian.drivingFields[].amplitude.pattern	str	구동 진폭의 공간 패턴, Omega(t). 'uniform'이어야 함

프로그램 필드	type	description
hamiltonian.drivingFields[].phase.time_series.times	List[Decimal]	구동 위상의 시점, phi(t)
hamiltonian.drivingFields[].phase.time_series.values	List[Decimal]	구동 위상의 값, phi(t)
hamiltonian.drivingFields[].phase.pattern	str	구동 위상의 공간 패턴, phi(t). 'uniform'이어야 함
hamiltonian.drivingFields[].detuning.time_series.times	List[Decimal]	구동 디튜닝의 시점, Delta_global(t)
hamiltonian.drivingFields[].detuning.time_series.values	List[Decimal]	구동 디튜닝의 값, Delta_global(t)
hamiltonian.drivingFields[].detuning.pattern	str	구동 디튜닝의 공간 패턴, Delta_global(t). 'uniform'이어야 함
hamiltonian.localDetuning[].magnitude.time_series.times	List[Decimal]	로컬 디튜닝 크기의 시간 종속 인자의 시점, Delta_local(t)
hamiltonian.localDetuning[].magnitude.time_series.values	List[Decimal]	로컬 디튜닝 크기의 시간 종속 인자의 값, Delta_local(t)
hamiltonian.localDetuning[].magnitude.pattern	List[Decimal]	로컬 디튜닝 크기의 위치 종속 인자, h_k(값은 setup.ahs_register.sites의 위치에 해당)

메타데이터 필드

프로그램 필드	type	description
braketSchemaHeader.name	str	스키마의 이름. 'braket.ir.ahs.program'이어야 함
braketSchemaHeader.version	str	스키마 버전

Braket AHS 작업 결과 스키마

braket.tasks.analog_hamiltonian_simulation_quantum_task_result.AnalogHamiltonianSimulationQuantumTas
제)

```
AnalogHamiltonianSimulationQuantumTaskResult(
    task_metadata=TaskMetadata(
        braketSchemaHeader=BraketSchemaHeader(
            name='braket.task_result.task_metadata',
            version='1'
        ),
        id='arn:aws:braket:us-east-1:123456789012:quantum-task/12345678-90ab-
cdef-1234-567890abcdef',
        shots=2,
        deviceId='arn:aws:braket:us-east-1::device/qpu/quera/Aquila',
        deviceParameters=None,
        createdAt='2022-10-25T20:59:10.788Z',
        endedAt='2022-10-25T21:00:58.218Z',
        status='COMPLETED',
        failureReason=None
    ),
    measurements=[
        ShotResult(
            status=<AnalogHamiltonianSimulationShotStatus.SUCCESS: 'Success'>,

            pre_sequence=array([1, 1, 1, 1]),
            post_sequence=array([0, 1, 1, 1])
        ),

        ShotResult(
            status=<AnalogHamiltonianSimulationShotStatus.SUCCESS: 'Success'>,

            pre_sequence=array([1, 1, 0, 1]),
```

```
            post_sequence=array([1, 0, 0, 0])
        )
    ]
)
```

JSON(예제)

```
{
    "braketSchemaHeader": {
        "name": "braket.task_result.analog_hamiltonian_simulation_task_result",
        "version": "1"
    },
    "taskMetadata": {
        "braketSchemaHeader": {
            "name": "braket.task_result.task_metadata",
            "version": "1"
        },
        "id": "arn:aws:braket:us-east-1:123456789012:quantum-task/12345678-90ab-
cdef-1234-567890abcdef",
        "shots": 2,
        "deviceId": "arn:aws:braket:us-east-1::device/qpu/quera/Aquila",

        "createdAt": "2022-10-25T20:59:10.788Z",
        "endedAt": "2022-10-25T21:00:58.218Z",
        "status": "COMPLETED"

    },
    "measurements": [
        {
            "shotMetadata": {"shotStatus": "Success"},
            "shotResult": {
                "preSequence": [1, 1, 1, 1],
                "postSequence": [0, 1, 1, 1]
            }
        },
        {
            "shotMetadata": {"shotStatus": "Success"},
            "shotResult": {
                "preSequence": [1, 1, 0, 1],
                "postSequence": [1, 0, 0, 0]
            }
        }
    ],
```

```
    "additionalMetadata": {
        "action": {...}
        "queraMetadata": {
            "braketSchemaHeader": {
                "name": "braket.task_result.quera_metadata",
                "version": "1"
            },
            "numSuccessfulShots": 100
        }
    }
}
```

기본 필드

작업 결과 필드	type	description
measurements[].shotResult.preSequence	List[int]	각 샷에 대한 사전 시퀀스 측정 비트(각 원자성 위치마다 하나씩): 위치가 비어 있는 경우 0, 위치가 채워진 경우 1, 양자 진화를 실행하는 펄스 시퀀스 전에 측정됨
measurements[].shotResult.postSequence	List[int]	각 샷에 대한 사후 시퀀스 측정 비트: 원자가 Rydberg 상태에 있거나 위치가 비어 있는 경우 0, 원자가 바닥 상태에 있는 경우 1, 양자 진화를 실행하는 펄스 시퀀스 종료 시 측정됨

메타데이터 필드

작업 결과 필드	type	description
braketSchemaHeader.name	str	스키마의 이름. 'braket.task_result.analog_hamiltonian_simula

작업 결과 필드	type	description
		tion_task _result'여야 함
braketSchemaHeader.version	str	스키마 버전
taskMetadata.braketSchemaHeader.name	str	스키마의 이름, ‘braket.task_result.task_metadata'여야 함
taskMetadata.braketSchemaHeader.version	str	스키마 버전
taskMetadata.id	str	양자 작업의 ID입니다. AWS 양자 작업의 경우 양자 작업 ARN입니다.
taskMetadata.shots	int	양자 작업에 대한 샷 수입니다.
taskMetadata.shots.deviceId	str	양자 작업이 실행된 디바이스의 ID입니다. AWS 디바이스의 경우 디바이스 ARN입니다.

작업 결과 필드	type	description
taskMetadata.shots.createdAt	str	생성 타임스탬프입니다. 형식은 ISO-8601/RFC3339 문자열 형식 YYYY-MM-DDTHH:mm:ss.sssZ여야 합니다. 기본값은 '없음'입니다.
taskMetadata.shots.endedAt	str	양자 작업이 종료된 시점의 타임스탬프입니다. 형식은 ISO-8601/RFC3339 문자열 형식 YYYY-MM-DDTHH:mm:ss.sssZ여야 합니다. 기본값은 '없음'입니다.
taskMetadata.shots.status	str	양자 작업의 상태(CREATED, QUEUED, RUNNING, COMPLETED, FAILED)입니다. 기본값은 '없음'입니다.

작업 결과 필드	type	description
taskMetadata.shots.failureReason	str	양자 작업의 실패 이유입니다. 기본값은 '없음'입니다.
additionalMetadata.action	braket.ir.ahs.program_v1.Program	(Braket AHS program schema 섹션 참조)
additionalMetadata.action.braketSchemaHeader.queraMetadata.name	str	스키마의 이름. 'braket.task_result.quera_metadata'여야 함
additionalMetadata.action.braketSchemaHeader.queraMetadata.version	str	스키마 버전
additionalMetadata.action.numSuccessfulShots	int	완전히 성공한 샷 수. 요청된 샷 수와 같아야 함
measurements[].shotMetadata.shotStatus	int	샷 상태(성공, 부분 성공, 실패). "성공"이어야 함

QuEra 디바이스 속성 스키마

braket.device_schema.quera.quera_device_capabilities_v1.QueraDeviceCapabilities(예제)

```
QueraDeviceCapabilities(
    service=DeviceServiceProperties(
        braketSchemaHeader=BraketSchemaHeader(
```

```
            name='braket.device_schema.device_service_properties',
            version='1'
            ),
            executionWindows=[
                DeviceExecutionWindow(
                    executionDay=<ExecutionDay.MONDAY: 'Monday'>,
                    windowStartHour=datetime.time(1, 0),
                    windowEndHour=datetime.time(23, 59, 59)
                ),
                DeviceExecutionWindow(
                    executionDay=<ExecutionDay.TUESDAY: 'Tuesday'>,
                    windowStartHour=datetime.time(0, 0),
                    windowEndHour=datetime.time(12, 0)
                ),
                DeviceExecutionWindow(
                    executionDay=<ExecutionDay.WEDNESDAY: 'Wednesday'>,
                    windowStartHour=datetime.time(0, 0),
                    windowEndHour=datetime.time(12, 0)
                ),
                DeviceExecutionWindow(
                    executionDay=<ExecutionDay.FRIDAY: 'Friday'>,
                    windowStartHour=datetime.time(0, 0),
                    windowEndHour=datetime.time(23, 59, 59)
                ),
                DeviceExecutionWindow(
                    executionDay=<ExecutionDay.SATURDAY: 'Saturday'>,
                    windowStartHour=datetime.time(0, 0),
                    windowEndHour=datetime.time(23, 59, 59)
                ),
                DeviceExecutionWindow(
                    executionDay=<ExecutionDay.SUNDAY: 'Sunday'>,
                    windowStartHour=datetime.time(0, 0),
                    windowEndHour=datetime.time(12, 0)
                )
            ],
            shotsRange=(1, 1000),
            deviceCost=DeviceCost(
                price=0.01,
                unit='shot'
            ),
            deviceDocumentation=
                DeviceDocumentation(
```

```
                    imageUrl='https://
a.b.cdn.console.awsstatic.com/59534b58c709fc239521ef866db9ea3f1aba73ad3ebcf60c23914ad8c5c5c878/
a6cfc6fca26cf1c2e1c6.png',
                    summary='Analog quantum processor based on neutral atom arrays',
                    externalDocumentationUrl='https://www.quera.com/aquila'
                ),
                deviceLocation='Boston, USA',
                updatedAt=datetime.datetime(2024, 1, 22, 12, 0,
 tzinfo=datetime.timezone.utc),
                getTaskPollIntervalMillis=None
    ),
    action={
        <DeviceActionType.AHS: 'braket.ir.ahs.program'>: DeviceActionProperties(
                version=['1'],
                actionType=<DeviceActionType.AHS: 'braket.ir.ahs.program'>
            )
    },
    deviceParameters={},
    braketSchemaHeader=BraketSchemaHeader(
        name='braket.device_schema.quera.quera_device_capabilities',
        version='1'
    ),
    paradigm=QueraAhsParadigmProperties(
        ...
        # See https://github.com/amazon-braket/amazon-braket-schemas-python/blob/main/
src/braket/device_schema/quera/quera_ahs_paradigm_properties_v1.py
        ...
    )
)
```

JSON(예제)

```
{
    "service": {
        "braketSchemaHeader": {
            "name": "braket.device_schema.device_service_properties",
            "version": "1"
        },
        "executionWindows": [
            {
                "executionDay": "Monday",
                "windowStartHour": "01:00:00",
                "windowEndHour": "23:59:59"
```

```
            },
            {
                "executionDay": "Tuesday",
                "windowStartHour": "00:00:00",
                "windowEndHour": "12:00:00"
            },
            {
                "executionDay": "Wednesday",
                "windowStartHour": "00:00:00",
                "windowEndHour": "12:00:00"
            },
            {
                "executionDay": "Friday",
                "windowStartHour": "00:00:00",
                "windowEndHour": "23:59:59"
            },
            {
                "executionDay": "Saturday",
                "windowStartHour": "00:00:00",
                "windowEndHour": "23:59:59"
            },
            {
                "executionDay": "Sunday",
                "windowStartHour": "00:00:00",
                "windowEndHour": "12:00:00"
            }
        ],
        "shotsRange": [
            1,
            1000
        ],
        "deviceCost": {
            "price": 0.01,
            "unit": "shot"
        },
        "deviceDocumentation": {
            "imageUrl": "https://
a.b.cdn.console.awsstatic.com/59534b58c709fc239521ef866db9ea3f1aba73ad3ebcf60c23914ad8c5c5c878/
a6cfc6fca26cf1c2e1c6.png",
            "summary": "Analog quantum processor based on neutral atom arrays",
            "externalDocumentationUrl": "https://www.quera.com/aquila"
        },
        "deviceLocation": "Boston, USA",
        "updatedAt": "2024-01-22T12:00:00+00:00"
```

```
    },
    "action": {
        "braket.ir.ahs.program": {
            "version": [
                "1"
            ],
            "actionType": "braket.ir.ahs.program"
        }
    },
    "deviceParameters": {},
    "braketSchemaHeader": {
        "name": "braket.device_schema.quera.quera_device_capabilities",
        "version": "1"
    },
    "paradigm": {
        ...
        # See Aquila device page > "Calibration" tab > "JSON" page
        ...
    }
}
```

서비스 속성 필드

서비스 속성 필드	type	description
service.executionWindows[].executionDay	ExecutionDay	실행 기간의 일 수. '매일', '평일', '주말', '월요일', '화요일', '수요일', 목요일', '금요일', '토요일' 또는 '일요일'이어야 함
service.executionWindows[].windowStartHour	datetime.time	실행 기간이 시작되는 시간의 UTC 24시간 형식
service.executionWindows[].windowEndHour	datetime.time	실행 기간이 종료되는 시간의 UTC 24시간 형식
service.qpu_capabilities.service.shotsRange	Tuple[int, int]	디바이스의 최소 및 최대 샷 수

서비스 속성 필드	type	description
service.qpu_capabilities.service.deviceCost.price	실수	미국 달러 기준 디바이스 요금
service.qpu_capabilities.service.deviceCost.unit	str	요금 청구 단위, 예: '분', '시간', '샷', '작업'

메타데이터 필드

메타데이터 필드	type	description
action[].version	str	AHS 프로그램 스키마 버전
action[].actionType	ActionType	AHS 프로그램 스키마 이름. 'braket.ir.ahs.program'이어야 함
service.braketSchemaHeader.name	str	스키마의 이름. 'braket.device_schema.device_service_properties'여야 함
service.braketSchemaHeader.version	str	스키마 버전
service.deviceDocumentation.imageUrl	str	디바이스 이미지의 URL
service.deviceDocumentation.summary	str	디바이스에 대한 간략한 설명
service.deviceDocumentation.externalDocumentationUrl	str	외부 설명서 URL
service.deviceLocation	str	디바이스의 지리적 위치

메타데이터 필드	type	description
service.updatedAt	datetime	디바이스 속성이 마지막으로 업데이트된 시간

AWS Boto3 작업

Boto3는 Python용 AWS SDK입니다. Python 개발자는 Boto3를 사용하여 Amazon Braket과 같은를 생성 AWS 서비스, 구성 및 관리할 수 있습니다. Boto3은 객체 지향 API와 Amazon Braket에 대한 저수준 액세스를 제공합니다.

Boto3 Quickstart 가이드의 지침에 따라 Boto3을 설치하고 구성하는 방법을 알아봅니다.

Boto3은 Amazon Braket Python SDK와 함께 작동하면서 양자 작업을 구성하고 실행하는 데 도움이 되는 핵심 기능을 제공합니다. Python 고객은 항상 Boto3을 설치해야 합니다. 이는 핵심 구현이기 때문입니다. 추가 헬퍼 메서드를 사용하려면 Amazon Braket SDK도 설치해야 합니다.

예를 들어 `CreateQuantumTask`를 직접적으로 호출하면 Amazon Braket SDK가 Boto3에 요청을 제출한 다음 AWS API를 직접적으로 호출합니다.

이 섹션의 내용:

- Amazon Braket Boto3 클라이언트 켜기
- Boto3 및 Braket SDK에 대한 AWS CLI 프로파일 구성

Amazon Braket Boto3 클라이언트 켜기

Amazon Braket에서 Boto3을 사용하려면 Boto3을 가져온 다음 Amazon Braket API에 연결하는 데 사용하는 클라이언트를 정의해야 합니다. 다음 예제에서 Boto3 클라이언트의 이름은 `braket`입니다.

```
import boto3
import botocore

braket = boto3.client("braket")
```

> **Note**
>
> Braket은 IPv6를 지원합니다. IPv6 전용 네트워크를 사용하거나 워크로드가 IPv6 트래픽을 사용하도록 하려면 듀얼 스택 및 FIPS 엔드포인트 가이드에 설명된 대로 듀얼 스택 엔드포인트를 사용합니다.

이제 `braket` 클라이언트가 설정되었으므로 Amazon Braket 서비스에서 요청을 하고 응답을 처리할 수 있습니다. API 참조에서 요청 및 응답 데이터에 대한 자세한 내용을 확인할 수 있습니다.

다음 예제에서는 디바이스 및 양자 작업을 사용하는 방법을 보여줍니다.

- 디바이스 검색
- 디바이스 검색
- 양자 작업 생성
- 양자 작업 검색
- 양자 작업 검색
- 양자 작업 취소

디바이스 검색

- `search_devices(**kwargs)`

지정된 필터를 사용하여 디바이스를 검색합니다.

```
# 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'])
```

디바이스 검색

- `get_device(deviceArn)`

Amazon Braket에서 사용 가능한 디바이스를 검색합니다.

```
# Pass the device ARN when sending the request and capture the repsonse
response = braket.get_device(deviceArn='arn:aws:braket:::device/quantum-simulator/
amazon/sv1')

print(f"Device {response['deviceName']} is {response['deviceStatus']}")
```

양자 작업 생성

- `create_quantum_task(**kwargs)`

양자 작업을 생성합니다.

```
# 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")
```

양자 작업 검색

- `get_quantum_task(quantumTaskArn)`

지정된 양자 작업을 검색합니다.

```
# 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'])
```

양자 작업 검색

- `search_quantum_tasks(**kwargs)`

지정된 필터 값과 일치하는 양자 작업을 검색합니다.

```
# 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']}")
```

양자 작업 취소

- `cancel_quantum_task(quantumTaskArn)`

지정된 양자 작업을 취소합니다.

```
# 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']}")
```

Boto3 및 Braket SDK에 대한 AWS CLI 프로파일 구성

달리 명시적으로 지정하지 않는 한 Amazon Braket SDK는 기본 AWS CLI 자격 증명을 사용합니다. 노트북 인스턴스를 시작할 권한이 있는 IAM 역할을 제공해야 하므로 관리형 Amazon Braket 노트북에서 실행할 때는 기본값을 유지하는 것이 좋습니다.

선택적으로 코드를 로컬(예: Amazon EC2 인스턴스에서)로 실행하는 경우 명명된 AWS CLI 프로파일을 설정할 수 있습니다. 기본 프로파일을 정기적으로 덮어쓰지 않고 각 프로파일에 서로 다른 권한 세트를 부여할 수 있습니다.

이 섹션에서는 이러한 CLI `profile`을 구성하는 방법과 해당 프로파일의 권한을 사용하여 API 직접 호출이 이루어지도록 해당 프로파일을 Amazon Braket에 통합하는 방법을 간략하게 설명합니다.

이 섹션의 내용:

- 1단계: 로컬 AWS CLI 구성 profile
- 2단계: Boto3 세션 객체 설정
- 3단계: Boto3 세션을 Braket AwsSession에 통합

1단계: 로컬 AWS CLI 구성 **`profile`**

사용자를 생성하는 방법과 비기본 프로파일을 구성하는 방법에 대한 설명은 이 문서의 범위를 벗어납니다. 이 주제에 대한 자세한 내용은 다음을 참조하세요.

- 시작하기
- AWS CLI 를 사용하도록 구성 AWS IAM Identity Center

Amazon Braket을 사용하려면 이 사용자와 연결된 CLI `profile`에 필요한 Braket 권한을 제공해야 합니다. 예를 들어 AmazonBraketFullAccess 정책을 연결할 수 있습니다.

2단계: Boto3 세션 객체 설정

Boto3 세션 객체를 설정하려면 다음 코드 예제를 활용하세요.

```
from boto3 import Session

# Insert CLI profile name here
boto_sess = Session(profile_name=`profile`)
```

> **Note**
>
> 예상 API 직접 호출에 `profile` 기본 리전과 일치하지 않는 리전 기반 제한이 있는 경우 다음 예제와 같이 Boto3 세션에 대한 리전을 지정할 수 있습니다.

```
# Insert CLI profile name _and_ region
boto_sess = Session(profile_name=`profile`, region_name=`region`)
```

로 지정된 인수의 경우 `region`, 등과 같이 Amazon Braket AWS 리전 을 사용할 수 있는 중 하나에 해당하는 값을 대체`us-east-1``us-west-1`합니다.

3단계: Boto3 세션을 Braket AwsSession에 통합

다음 예제에서는 Boto3 Braket 세션을 초기화하고 해당 세션에서 디바이스를 인스턴스화하는 방법을 보여줍니다.

```
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)
```

이 설정이 완료되면 인스턴스화된 `AwsDevice` 객체에 양자 작업을 제출할 수 있습니다(예: `device.run(…)` 명령을 직접적으로 호출). 해당 디바이스에서 수행한 모든 API 직접 호출은 이전에 `profile`로 지정한 CLI 프로필과 연결된 IAM 자격 증명을 사용할 수 있습니다.

Amazon Braket을 사용한 양자 작업 테스트

Amazon Braket은 실제 양자 하드웨어에서 실행하기 전에 양자 알고리즘을 테스트하고 검증하는 데 도움이 되는 다양한 고성능 양자 회로 시뮬레이터를 제공합니다. 이러한 시뮬레이터는 복잡한 기본 소프트웨어와 인프라, 그리고 Amazon Elastic Compute Cloud(Amazon EC2) 클러스터를 처리하여 고전적인 고성능 컴퓨팅(HPC) 인프라에서 양자 회로를 시뮬레이션하는 부담을 덜어줍니다. 이러한 리소스를 사용하면 양자 애플리케이션을 개발하고 최적화하는 데 집중할 수 있습니다.

Braket의 시뮬레이터를 사용하면 물리적 양자 디바이스의 제약 조건 및 제한 없이 양자 회로와 알고리즘을 철저하게 테스트할 수 있습니다. 이를 통해 기본 양자 게이트 및 회로부터 고급 양자 알고리즘 및 오류 완화 기법에 이르기까지 다양한 양자 컴퓨팅 개념을 탐구할 수 있습니다.

Braket SDK는 시뮬레이터에 양자 작업을 제출하는 과정을 간소화하여 샷 수 및 노이즈 모델과 같은 시뮬레이션 파라미터를 제어할 수 있게 함으로써 양자 알고리즘의 동작을 더욱 잘 이해할 수 있게 합니다. 또한 Amazon Braket Hybrid Job 기능을 사용해 고전 컴퓨팅 요소와 양자 컴퓨팅 요소를 결합하여 테스트 및 검증 범위를 더욱 확장할 수 있습니다.

Braket의 시뮬레이터에서 양자 작업을 철저히 테스트하면 실제 양자 하드웨어에 배포하기 전에 귀중한 인사이트를 얻고 알고리즘을 개선하며 정합성을 보장할 수 있습니다. 이를 통해 개발 시간을 단축하고 오류를 최소화하며 궁극적으로 양자 컴퓨팅 분야에서의 발전을 가속화할 수 있습니다.

이 섹션의 내용:

시뮬레이터에 양자 작업 제출

Amazon Braket은 양자 작업을 테스트할 수 있는 여러 시뮬레이터에 대한 액세스를 제공합니다. 양자 작업을 개별적으로 제출하거나 여러 프로그램을 실행할 수 있습니다.

시뮬레이터

- 밀도 행렬 시뮬레이터, DM1 : `arn:aws:braket:::device/quantum-simulator/amazon/dm1`
- 상태 벡터 시뮬레이터, SV1 : `arn:aws:braket:::device/quantum-simulator/amazon/sv1`

- 텐서 네트워크 시뮬레이터, TN1 : `arn:aws:braket:::device/quantum-simulator/amazon/tn1`
- 로컬 시뮬레이터 : `LocalSimulator()`

> **Note**
>
> QPU 및 온디맨드 시뮬레이터의 경우 CREATED 상태의 양자 작업을 취소할 수 있습니다. 온디맨드 시뮬레이터 및 QPU의 경우 QUEUED 상태의 양자 작업을 최선의 방식으로 취소할 수 있습니다. 참고로 QPU QUEUED 양자 작업은 QPU 가용성 기간 동안 성공적으로 취소될 가능성이 낮습니다.

이 섹션의 내용:

로컬 상태 벡터 시뮬레이터(**braket_sv**)

로컬 상태 벡터 시뮬레이터(`braket_sv`)는 해당 환경에서 로컬로 실행되는 Amazon Braket SDK의 일부입니다. Braket 노트북 인스턴스 또는 로컬 환경의 하드웨어 사양에 따라 소형 회로(최대 25qubits)에서의 신속한 프로토타이핑에 적합합니다.

로컬 시뮬레이터는 Amazon Braket SDK의 모든 게이트를 지원하지만, QPU 디바이스는 더 작은 부분집합을 지원합니다. 디바이스 속성에서 디바이스의 지원되는 게이트를 확인할 수 있습니다.

> **Note**
>
> 로컬 시뮬레이터는 QPU 디바이스 또는 기타 시뮬레이터에서 지원되지 않을 수 있는 고급 OpenQASM 기능을 지원합니다. 지원되는 기능에 대한 자세한 내용은 OpenQASM 로컬 시뮬레이터 노트북에 제공된 예제를 참조하세요.

시뮬레이터 사용 방법에 대한 자세한 내용은 Amazon Braket 예제를 참조하세요.

로컬 밀도 행렬 시뮬레이터(`braket_dm`)

로컬 밀도 행렬 시뮬레이터(`braket_dm`)는 해당 환경에서 로컬로 실행되는 Amazon Braket SDK의 일부입니다. Braket 노트북 인스턴스 또는 로컬 환경의 하드웨어 사양에 따라 노이즈가 있는 소형 회로(최대 12qubits)에서의 신속한 프로토타이핑에 적합합니다.

비트플립 및 탈분극 오류와 같은 게이트 노이즈 연산을 사용하여 일반적인 노이즈 회로를 처음부터 빌드할 수 있습니다. 노이즈가 있거나 없는 상태에서 모두 실행되도록 설계된 기존 회로의 특정 qubits 및 게이트에 노이즈 연산을 적용할 수도 있습니다.

`braket_dm` 로컬 시뮬레이터는 지정된 shots 수에 따라 다음과 같은 결과를 제공할 수 있습니다.

- 축약 밀도 행렬: Shots = 0

> **Note**
>
> 로컬 시뮬레이터는 QPU 디바이스 또는 기타 시뮬레이터에서 지원되지 않을 수 있는 고급 OpenQASM 기능을 지원합니다. 지원되는 기능에 대한 자세한 내용은 OpenQASM 로컬 시뮬레이터 노트북에 제공된 예제를 참조하세요.

로컬 밀도 행렬 시뮬레이터에 대한 자세한 내용은 Braket 입문용 노이즈 시뮬레이터 예제를 참조하세요.

로컬 AHS 시뮬레이터(`braket_ahs`)

로컬 AHS(Analog Hamiltonian Simulation) 시뮬레이터(`braket_ahs`)는 해당 환경에서 로컬로 실행되는 Amazon Braket SDK의 일부입니다. AHS 프로그램의 결과를 시뮬레이션하는 데 사용할 수 있습니다. Braket 노트북 인스턴스 또는 로컬 환경의 하드웨어 사양에 따라 소형 레지스터(최대 10~12개 원자)에서의 프로토타이핑에 적합합니다.

로컬 시뮬레이터는 균일한 구동장 1개, (비균일) 이동장 1개, 임의의 원자 배열을 갖는 AHS 프로그램을 지원합니다. 자세한 내용은 Braket AHS 클래스 및 Braket AHS 프로그램 스키마를 참조하세요.

로컬 AHS 시뮬레이터에 대한 자세한 내용은 Hello AHS: 첫 번째 아날로그 해밀토니안 시뮬레이션 실행 페이지 및 아날로그 해밀토니안 시뮬레이션 예제 노트북을 참조하세요.

상태 벡터 시뮬레이터(SV1)

SV1은 온디맨드 고성능 범용 상태 벡터 시뮬레이터입니다. 최대 34qubits의 회로를 시뮬레이션할 수 있습니다. 사용된 게이트 유형 및 기타 인자에 따라 34-qubit의 고밀도 정방 회로(회로 깊이 = 34)를 완료하는 데 약 1~2시간이 걸릴 것으로 예상할 수 있습니다. all-to-all 게이트가 있는 회로는 SV1에 적합합니다. 전체 상태 벡터 또는 진폭 배열과 같은 형태로 결과를 반환합니다.

SV1의 최대 런타임은 6시간입니다. 기본적으로 35개의 동시 양자 작업이 처리되며 최대 동시 양자 작업 수는 100개(us-west-1 및 eu-west-2에서 50개)입니다.

SV1 결과

SV1은 지정된 shots 수에 따라 다음과 같은 결과를 제공할 수 있습니다.

- 샘플: Shots > 0
- 기대값: Shots >= 0
- 분산: Shots >= 0
- 확률: Shots > 0
- 진폭: Shots = 0
- 수반 기울기: Shots = 0

결과에 대한 자세한 내용은 결과 유형을 참조하세요.

SV1은 항상 사용할 수 있으며 온디맨드로 회로를 실행하고 여러 회로를 병렬로 실행할 수 있습니다. 런타임은 연산 횟수에 따라 선형적으로 증가하고 qubits 수에 따라 기하급수적으로 증가합니다. shots 수는 런타임에 약간의 영향을 미칩니다. 자세한 내용은 시뮬레이터 비교를 참조하세요.

시뮬레이터는 Braket SDK의 모든 게이트를 지원하지만, QPU 디바이스는 더 작은 부분 집합을 지원합니다. 디바이스 속성에서 디바이스의 지원되는 게이트를 확인할 수 있습니다.

밀도 행렬 시뮬레이터(DM1)

DM1은 온디맨드 고성능 밀도 행렬 시뮬레이터입니다. 최대 17qubits의 회로를 시뮬레이션할 수 있습니다.

DM1의 최대 런타임은 6시간이고, 기본적으로 35개의 동시 양자 작업이 처리되며 최대 동시 양자 작업 수는 50개입니다.

DM1 결과

DM1은 지정된 shots 수에 따라 다음과 같은 결과를 제공할 수 있습니다.

- 샘플: Shots > 0
- 기대값: Shots >= 0
- 분산: Shots >= 0
- 확률: Shots > 0
- 축약 밀도 행렬: Shots = 0, 최대 8qubits까지

결과에 대한 자세한 내용은 결과 유형을 참조하세요.

DM1은 항상 사용할 수 있으며 온디맨드로 회로를 실행하고 여러 회로를 병렬로 실행할 수 있습니다. 런타임은 연산 횟수에 따라 선형적으로 증가하고 qubits 수에 따라 기하급수적으로 증가합니다. shots 수는 런타임에 약간의 영향을 미칩니다. 자세한 내용은 시뮬레이터 비교를 참조하세요.

노이즈 게이트 및 제한 사항

```
AmplitudeDamping
    Probability has to be within [0,1]
BitFlip
    Probability has to be within [0,0.5]
Depolarizing
    Probability has to be within [0,0.75]
GeneralizedAmplitudeDamping
    Probability has to be within [0,1]
PauliChannel
    The sum of the probabilities has to be within [0,1]
Kraus
    At most 2 qubits
    At most 4 (16) Kraus matrices for 1 (2) qubit
PhaseDamping
    Probability has to be within [0,1]
```

```
PhaseFlip
    Probability has to be within [0,0.5]
TwoQubitDephasing
    Probability has to be within [0,0.75]
TwoQubitDepolarizing
    Probability has to be within [0,0.9375]
```

텐서 네트워크 시뮬레이터(TN1)

TN1은 온디맨드 고성능 텐서 네트워크 시뮬레이터입니다. TN1은 최대 50qubits 및 1,000 이하의 회로 깊이를 갖는 특정 회로 유형을 시뮬레이션할 수 있습니다. TN1는 희소 회로, 로컬 게이트가 있는 회로, 양자 푸리에 변환(QFT) 회로와 같은 특수 구조의 기타 회로에 특히 강력합니다. TN1은 두 단계로 작동합니다. 먼저 리허설 단계에서는 회로의 효율적인 계산 경로를 식별하려 시도하므로, TN1은 수축 단계라고 하는 다음 단계의 런타임을 추정할 수 있습니다. 추정 수축 시간이 TN1 시뮬레이션 런타임 제한을 초과하는 경우 TN1은 수축을 시도하지 않습니다.

TN1의 런타임 제한은 6시간입니다. 최대 동시 양자 작업 수는 10개(eu-west-2에서 5개)입니다.

TN1 결과

수축 단계는 일련의 행렬 곱셈으로 구성됩니다. 결과에 도달할 때까지 또는 결과에 도달할 수 없는 것으로 확인될 때까지 일련의 곱셈이 계속됩니다.

참고: Shots은 > 0이어야 합니다.

결과 유형은 다음과 같습니다.

- 샘플
- 기대값
- 분산

결과에 대한 자세한 내용은 결과 유형을 참조하세요.

TN1은 항상 사용할 수 있으며 온디맨드로 회로를 실행하고 여러 회로를 병렬로 실행할 수 있습니다. 자세한 내용은 시뮬레이터 비교를 참조하세요.

시뮬레이터는 Braket SDK의 모든 게이트를 지원하지만, QPU 디바이스는 더 작은 부분 집합을 지원합니다. 디바이스 속성에서 디바이스의 지원되는 게이트를 확인할 수 있습니다.

TN1을 시작하는 데 도움이 되는 TN1 예제 노트북을 보려면 Amazon Braket GitHub 리포지토리를 방문하세요.

TN1 작업 모범 사례

- all-to-all 회로를 피합니다.
- 새로운 회로나 shots 수가 적은 회로 클래스를 테스트하여 TN1에 대한 회로의 "난이도"를 알아봅니다.
- 대규모 shot 시뮬레이션을 여러 양자 작업으로 분할합니다.

임베디드 시뮬레이터 정보

임베디드 시뮬레이터는 알고리즘 코드 내에 직접 시뮬레이션을 임베디드하여 작동합니다. 또한 동일한 컨테이너 내에 포함되어 하이브리드 작업 인스턴스에서 직접 시뮬레이션을 실행합니다. 이 접근 방식은 일반적으로 시뮬레이션과 원격 디바이스 간의 통신과 관련된 병목 현상을 제거하는 데 유용합니다. 임베디드 시뮬레이터는 모든 계산을 단일하고 일관된 환경에서 수행함으로써 메모리 요구 사항을 크게 줄이고 목표 결과를 달성하는 데 필요한 회로 실행 횟수를 감소시킬 수 있습니다. 이는 원격 시뮬레이션에 의존하는 기존 설정에 비해 종종 10배 이상의 상당한 성능 개선으로 이어질 수 있습니다. 임베디드 시뮬레이터가 어떻게 성능을 향상하고 간소화된 하이브리드 작업을 가능하게 하는지에 대한 자세한 내용은 Amazon Braket Hybrid Jobs로 하이브리드 작업 실행 설명서 페이지를 참조하세요.

PennyLane의 라이트닝 시뮬레이터

Braket에서 PennyLane의 라이트닝 시뮬레이터를 임베디드 시뮬레이터로 사용할 수 있습니다. PennyLane의 라이트닝 시뮬레이터를 사용하면 수반 미분과 같은 고급 기울기 계산 방법을 사용하여 기울기를 더 빠르게 평가할 수 있습니다. lightning.qubit 시뮬레이터는 Braket NBI를 통해 디바이스로 제공되며 임베디드 시뮬레이터로도 사용할 수 있는 반면, lightning.gpu 시뮬레이터는 GPU 인스턴스가 있는 임베디드 시뮬레이터로 실행되어야 합니다. lightning.gpu 사용 예제는 Braket Hybrid Jobs의 임베디드 시뮬레이터 노트북을 참조하세요.

Amazon Braket 시뮬레이터 비교

이 섹션에서는 몇 가지 개념, 제한 사항 및 사용 사례를 설명하므로 양자 작업에 가장 적합한 Amazon Braket 시뮬레이터를 선택하는 데 도움이 됩니다.

로컬 시뮬레이터와 온디맨드 시뮬레이터 중에서 선택(SV1, TN1, DM1)

로컬 시뮬레이터의 성능은 시뮬레이터를 실행하는 데 사용되는 Braket 노트북 인스턴스와 같이 로컬 환경을 호스팅하는 하드웨어에 따라 달라집니다. 온디맨드 시뮬레이터는 AWS 클라우드에서 실행되며 일반적인 로컬 환경 이상으로 확장되도록 설계되었습니다. 온디맨드 시뮬레이터는 대규모 회로에 최적화되어 있지만, 양자 작업 또는 양자 작업 배치당 약간의 지연 오버헤드가 추가됩니다. 이는 여러 양자 작업이 관련되는 경우 상충 관계가 있음을 의미할 수 있습니다. 이러한 일반적인 성능 특성을 감

안할 때, 다음 지침은 노이즈가 있는 시뮬레이션을 포함하여 시뮬레이션 실행 방법을 선택하는 데 도움이 될 수 있습니다.

시뮬레이션의 경우:

- 18qubits 미만을 사용하는 경우 로컬 시뮬레이터를 사용합니다.
- 18~24qubits를 사용하는 경우 워크로드에 따라 시뮬레이터를 선택합니다.
- 24qubits 이상을 사용하는 경우 온디맨드 시뮬레이터를 사용합니다.

노이즈 시뮬레이션의 경우:

- 9qubits 미만을 사용하는 경우 로컬 시뮬레이터를 사용합니다.
- 9~12qubits를 사용하는 경우 워크로드에 따라 시뮬레이터를 선택합니다.
- 12qubits 이상을 사용하는 경우 DM1을 사용합니다.

상태 벡터 시뮬레이터란 무엇입니까?

SV1은 범용 상태 벡터 시뮬레이터입니다. 양자 상태의 전체 파동 함수를 저장하고 상태에 순차적으로 게이트 연산을 적용합니다. 가능성이 극히 희박한 것까지 포함하여 모든 가능성을 저장합니다. 양자 작업에 대한 SV1 시뮬레이터의 실행 시간은 회로의 게이트 수에 따라 선형적으로 증가합니다.

밀도 행렬 시뮬레이터란 무엇입니까?

DM1은 노이즈가 있는 양자 회로를 시뮬레이션합니다. 시스템의 전체 밀도 행렬을 저장하고 회로의 게이트 및 노이즈 연산을 순차적으로 적용합니다. 최종 밀도 행렬에는 회로 실행 후 양자 상태에 대한 전체 정보가 포함됩니다. 런타임은 일반적으로 연산 횟수에 따라 선형적으로 증가하고 qubits 수에 따라 기하급수적으로 증가합니다.

텐서 네트워크 시뮬레이터란 무엇입니까?

TN1은 양자 회로를 구조화된 그래프로 인코딩합니다.

- 그래프의 노드는 양자 게이트 또는 qubits로 구성됩니다.
- 그래프의 변은 게이트 간의 연결을 나타냅니다.

이러한 구조 덕분에, TN1은 상대적으로 크고 복잡한 양자 회로에 대해 시뮬레이션된 해를 찾을 수 있습니다.

TN1에는 두 단계가 필요합니다.

일반적으로는 TN1은 양자 계산을 시뮬레이션하는 2단계 접근 방식으로 작동합니다.

- 리허설 단계: 이 단계에서 TN1은 그래프를 효율적으로 탐색할 수 있는 방법을 고안합니다. 여기에는 원하는 측정값을 얻을 수 있도록 모든 노드를 방문하는 것이 포함됩니다. TN1은 두 단계를 함께 수행하므로 고객은 이 단계를 볼 수 없습니다. 첫 번째 단계를 완료하고 실제 제약 조건에 따라 두 번째 단계를 자체적으로 수행할지 여부를 결정합니다. 시뮬레이션이 시작된 후에는 해당 결정에 대해 입력을 제공할 수 없습니다.
- 수축 단계:이 단계는 고전 컴퓨터에서의 계산 실행 단계와 유사합니다. 단계는 일련의 행렬 곱셈으로 구성됩니다. 이러한 곱셈의 순서는 계산의 난이도에 큰 영향을 미칩니다. 따라서 그래프 전체에서 가장 효과적인 계산 경로를 찾기 위해 리허설 단계가 먼저 수행됩니다. 리허설 단계에서 수축 경로를 찾은 후 TN1은 회로의 게이트를 함께 수축시켜 시뮬레이션 결과를 생성합니다.

TN1 그래프는 지도와 유사합니다.

비유하자면, 기본 TN1 그래프를 도시의 거리에 비교할 수 있습니다. 격자형으로 계획된 도시에서는 지도를 사용하여 목적지까지의 경로를 쉽게 찾을 수 있습니다. 계획되지 않은 거리, 중복된 도로명 등이 있는 도시에서는 지도만으로 목적지까지 가는 경로를 찾기가 어려울 수 있습니다.

TN1이 리허설 단계를 수행하지 않았다면, 지도를 먼저 보지 않고 목적지를 찾으려 도시의 거리를 헤매는 것과 같을 것입니다. 지도를 보는 데 더 많은 시간을 보내는 것이 실제로 걸어 다니는 시간 측면에서 큰 도움이 될 수 있습니다. 마찬가지로 리허설 단계는 중요한 정보를 제공합니다.

TN1은 통과하는 기본 회로의 구조에 대해 특정한 “인식”을 가지고 있다고 말할 수 있습니다. 이러한 인식은 리허설 단계에서 획득됩니다.

이러한 각 유형의 시뮬레이터에 가장 적합한 문제 유형

SV1은 주로 특정 수의 qubits와 게이트를 갖는 것에 의존하는 모든 종류의 문제에 적합합니다. 일반적으로, 소요 시간은 게이트 수에 따라 선형적으로 증가하지만, shots의 수에는 의존하지 않습니다. SV1은 일반적으로 28qubits 미만의 회로에서 TN1보다 더 빠릅니다.

SV1은 qubit 수가 많아질수록 더 느려질 수 있는데, 이는 실제로 극히 희박한 것까지 포함하여 모든 가능성을 시뮬레이션하기 때문입니다. 어떤 결과가 나올 가능성이 높은지 판단할 수 있는 방법은 없습니다 따라서 30-qubit 평가를 위해 SV1은 2^30 구성을 계산해야 합니다. Amazon Braket SV1 시뮬레이터를 34qubits로 제한하는 것은 메모리 및 스토리지 제한으로 인해 실질적인 제약이 됩니다. 이렇게 생각할 수 있습니다. qubit에 SV1을 더할 때마다 문제가 두 배로 어려워집니다.

많은 종류의 문제에서 TN1은 SV1보다 훨씬 더 큰 회로를 현실적인 시간 안에 평가할 수 있습니다. 왜냐하면 TN1이 그래프의 구조를 활용하기 때문입니다. 시작 지점에서부터 해의 진화를 추적하며 효율

적인 탐색에 기여하는 구성만 유지합니다. 다시 말해, 행렬 곱셈의 순서를 생성하기 위해 구성을 저장하여 평가 과정을 간소화합니다.

TN1의 경우 qubits 및 게이트 수가 중요하지만, 그래프의 구조는 훨씬 더 중요합니다. 예를 들어, TN1은 게이트가 단거리(즉, 각 qubit가 게이트를 통해 가장 가까이 이웃하는 qubits에만 연결됨)인 회로(그래프)와 연결(또는 게이트)의 범위가 유사한 회로(그래프)를 평가하는 데 매우 적합합니다. TN1의 일반적인 범위는 각 qubit가 5qubits 떨어진 다른 qubits와만 통신하도록 하는 것입니다. 대부분의 구조가 이와 같이 더 단순한 관계로 분해될 수 있고, 이를 더 많고, 더 작거나, 더 균일한 행렬로 표현할 수 있다면 TN1은 효율적으로 평가를 수행합니다.

TN1의 제한 사항

TN1은 그래프의 구조적 복잡성에 따라 SV1보다 느릴 수 있습니다. 특정 그래프의 경우 TN1는 리허설 단계 후에 시뮬레이션을 종료하고 다음 두 가지 이유 중 하나로 `FAILED`의 상태를 표시합니다.

- 경로를 찾을 수 없음 — 그래프가 너무 복잡하면 적절한 탐색 경로를 찾기가 너무 어려워 시뮬레이터가 계산을 포기합니다. TN1은 수축을 수행할 수 없습니다. 다음과 유사한 오류 메시지가 표시될 수 있습니다. `No viable contraction path found.`
- 수축 단계가 너무 어려움 — 일부 그래프에서는 TN1이 탐색 경로를 찾을 수 있지만, 경로가 매우 길어 평가하는 데 극도로 시간이 많이 소요됩니다. 이 경우 수축 비용이 너무 높아 비용 부담이 커지게 되며, 대신 TN1은 리허설 단계 이후에 종료됩니다. 다음과 유사한 오류 메시지가 표시될 수 있습니다. `Predicted runtime based on best contraction path found exceeds TN1 limit.`

> **Note**
>
> 수축이 수행되지 않고 `FAILED` 상태가 표시되는 경우에도 TN1의 리허설 단계에 대한 요금이 청구됩니다.

예측 런타임은 또한 shot 수에 따라 달라집니다. 최악의 경우 TN1 수축 시간은 shot 수에 따라 선형적으로 달라집니다. 회로는 더 적은 shots으로 수축될 수 있습니다. 예를 들어, shots이 100개인 양자 작업을 제출하면 TN1이 수축 불가능으로 판단할 수 있지만, 10개만으로 다시 제출하면 수축이 진행됩니다. 이런 상황에서 100개의 샘플을 얻으려면 동일한 회로에 대해 shots이 10개인 양자 작업 10개를 제출하고 최종적으로 결과를 조합하면 됩니다.

가장 좋은 방법은 더 많은 shots 수로 진행하기 전에 항상 적은 shots 수(예: 10개)로 회로나 회로 클래스를 테스트하여 TN1에 대한 회로의 난이도를 파악하는 것입니다.

> **Note**
>
> 수축 단계를 형성하는 일련의 곱셈은 작은 NxN 행렬로 시작됩니다. 예를 들어 2-qubit 게이트에는 4x4 행렬이 필요합니다. 너무 어렵다고 판단되는 수축 과정에서 필요한 중간 행렬은 매우 큽니다. 이러한 계산을 완료하는 데는 며칠이 걸릴 수 있습니다. 따라서 Amazon Braket은 극도로 복잡한 수축은 시도하지 않습니다.

동시성

모든 Braket 시뮬레이터를 사용하면 여러 회로를 동시에 실행할 수 있습니다. 동시성 제한은 시뮬레이터 및 리전에 따라 다릅니다. 동시성 제한에 대한 자세한 내용은 할당량 페이지를 참조하세요.

Amazon Braket의 양자 작업 예제

이 섹션에서는 디바이스 선택부터 결과 보기에 이르기까지 예제 양자 작업을 실행하는 단계를 안내합니다. Amazon Braket의 모범 사례로, SV1과 같은 시뮬레이터에서 회로를 실행하여 시작하는 것이 좋습니다.

이 섹션의 내용:

- 디바이스 지정
- 예제 양자 작업 제출
- 파라미터화된 작업 제출
- shots을 지정합니다.
- 결과에 대한 폴링
- 예제 결과 확인

디바이스 지정

먼저, 양자 작업에 사용할 디바이스를 선택하고 지정합니다. 이 예제에서는 시뮬레이터 SV1을 선택하는 방법을 보여줍니다.

```
from braket.aws import AwsDevice

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

이 디바이스의 몇 가지 속성은 다음과 같이 확인할 수 있습니다.

```
print(device.name)
for iter in device.properties.action['braket.ir.jaqcd.program']:
    print(iter)
```

```
SV1
('version', ['1.0', '1.1'])
('actionType', 'braket.ir.jaqcd.program')
('supportedOperations', ['ccnot', 'cnot', 'cphaseshift', 'cphaseshift00',
 'cphaseshift01', 'cphaseshift10', 'cswap', 'cy', 'cz', 'ecr', 'h', 'i', 'iswap',
 'pswap', 'phaseshift', 'rx', 'ry', 'rz', 's', 'si', 'swap', 't', 'ti', 'unitary', 'v',
 'vi', 'x', 'xx', 'xy', 'y', 'yy', 'z', 'zz'])
('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)])
('disabledQubitRewiringSupported', None)
```

예제 양자 작업 제출

온디맨드 시뮬레이터에서 실행할 예제 양자 작업을 제출합니다.

```
from braket.circuits import Circuit, Observable

# 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-s3-demo-bucket"  # The name of the bucket
my_prefix = "your-folder-name"  # The name of the folder in the bucket
s3_location = (my_bucket, my_prefix)

# Submit the quantum task to run
my_task = device.run(circ, s3_location, shots=1000, poll_timeout_seconds=100,
 poll_interval_seconds=10)
# The positional argument for the S3 bucket is optional if you want to specify a bucket
 other than the default
```

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

`device.run()` 명령은 `CreateQuantumTask` API를 통해 양자 작업을 생성합니다. 짧은 초기화 시간 후, 양자 작업은 디바이스에서 양자 작업을 실행할 수 있는 용량이 생길 때까지 대기열에 추가됩니다. 이 경우 디바이스는 SV1입니다. 디바이스가 계산을 완료한 후 Amazon Braket은 직접 호출 시 지정된 Amazon S3 위치에 결과를 기록합니다. 위치 인수 `s3_location`은 로컬 시뮬레이터를 제외한 모든 디바이스에 필요합니다.

> **Note**
>
> Braket 양자 작업 관련 작업은 크기가 3MB로 제한됩니다.

파라미터화된 작업 제출

Amazon Braket 온디맨드 및 로컬 시뮬레이터와 QPU는 작업 제출 시 자유 파라미터 값을 지정하는 것도 지원합니다. 다음 예제와 같이 `device.run()`에 `inputs` 인수를 사용하여 이 작업을 수행할 수 있습니다. `inputs`는 키가 파라미터 이름인 문자열-부동 소수점 쌍의 딕셔너리여야 합니다.

파라메트릭 컴파일은 특정 QPU에서 파라메트릭 회로 실행 성능을 향상할 수 있습니다. 파라메트릭 회로를 지원되는 QPU에 양자 작업으로 제출할 때 Braket은 회로를 한 번 컴파일하고 결과를 캐싱합니다. 동일한 회로에 대한 후속 파라미터 업데이트는 재컴파일되지 않으므로 동일한 회로를 사용하는 작업의 런타임이 단축됩니다. Braket은 회로를 컴파일할 때 하드웨어 공급자의 업데이트된 보정 데이터를 자동으로 사용하여 최고 품질의 결과를 보장합니다.

> **Note**
>
> 파라메트릭 컴파일은 펄스 레벨 프로그램을 제외한 Rigetti Computing의 모든 초전도 게이트 기반 QPU에서 지원됩니다.

```
from braket.circuits import Circuit, FreeParameter, Observable

# Create the free parameters
alpha = FreeParameter('alpha')
beta = FreeParameter('beta')
```

```
# Create a circuit with a result type
circ = Circuit().rx(0, alpha).ry(1, alpha).cnot(0, 2).xx(0, 2, beta)
circ.variance(observable=Observable.Z(), target=0)

# Add another result type
circ.probability(target=[0, 2])

# Submit the quantum task to run
my_task = device.run(circ, inputs={'alpha': 0.1, 'beta': 0.2}, shots=100)
```

shots을 지정합니다.

shots 인수는 원하는 측정 shots 수를 나타냅니다. SV1과 같은 시뮬레이터는 두 가지 시뮬레이션 모드를 지원합니다.

- shots = 0의 경우 시뮬레이터는 정확한 시뮬레이션을 수행하여 모든 결과 유형에 대한 참 값을 반환합니다. (TN1에서는 아직 사용할 수 없음)
- shots 값이 0이 아닌 경우, 시뮬레이터는 출력 분포에서 샘플링하여 실제 QPU의 shot 노이즈를 에뮬레이션합니다. QPU 디바이스는 shots > 0만 허용합니다.

양자 작업당 최대 샷 수에 대한 자세한 내용은 Braket 할당량을 참조하세요.

결과에 대한 폴링

`my_task.result()`를 실행하면 SDK는 양자 작업 생성 시 정의한 파라미터로 결과에 대한 폴링을 시작합니다.

- `poll_timeout_seconds`는 온디맨드 시뮬레이터 및/또는 QPU 디바이스에서 양자 작업을 실행할 때 시간이 초과되기 전에 양자 작업을 폴링하는 초의 수입니다. 기본값은 432,000초(5일)입니다.
- 참고: Rigetti 및 IonQ와 같은 QPU 디바이스의 경우 며칠 정도 허용하는 것이 좋습니다. 폴링 시간 초과가 너무 짧으면 폴링 시간 내에 결과가 반환되지 않을 수 있습니다. 예를 들어 QPU를 사용할 수 없는 경우 로컬 시간 초과 오류가 반환됩니다.
- `poll_interval_seconds`는 양자 작업이 폴링되는 빈도입니다. 온디맨드 시뮬레이터 및 QPU 디바이스에서 양자 작업이 실행될 때 상태를 가져오기 위해 Braket API를 직접적으로 호출하는 빈도를 지정합니다. 기본값은 1초입니다.

이 비동기 실행은 항상 사용할 수 있는 것은 아닌 QPU 디바이스와의 상호 작용을 용이하게 합니다. 예를 들어 정기 유지 관리 기간 동안에는 디바이스를 사용하지 못할 수 있습니다.

반환된 결과에는 양자 작업과 관련된 다양한 메타데이터가 포함됩니다. 다음 명령을 사용하여 측정 결과를 확인할 수 있습니다.

```
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]
 [0 0 0]
 ...
 [0 0 0]
 [0 0 0]
 [1 0 1]]
Counts for collapsed states:
 Counter({'000': 766, '101': 220, '010': 11, '111': 3})
Probabilities for collapsed states:
 {'101': 0.22, '000': 0.766, '010': 0.011, '111': 0.003}
```

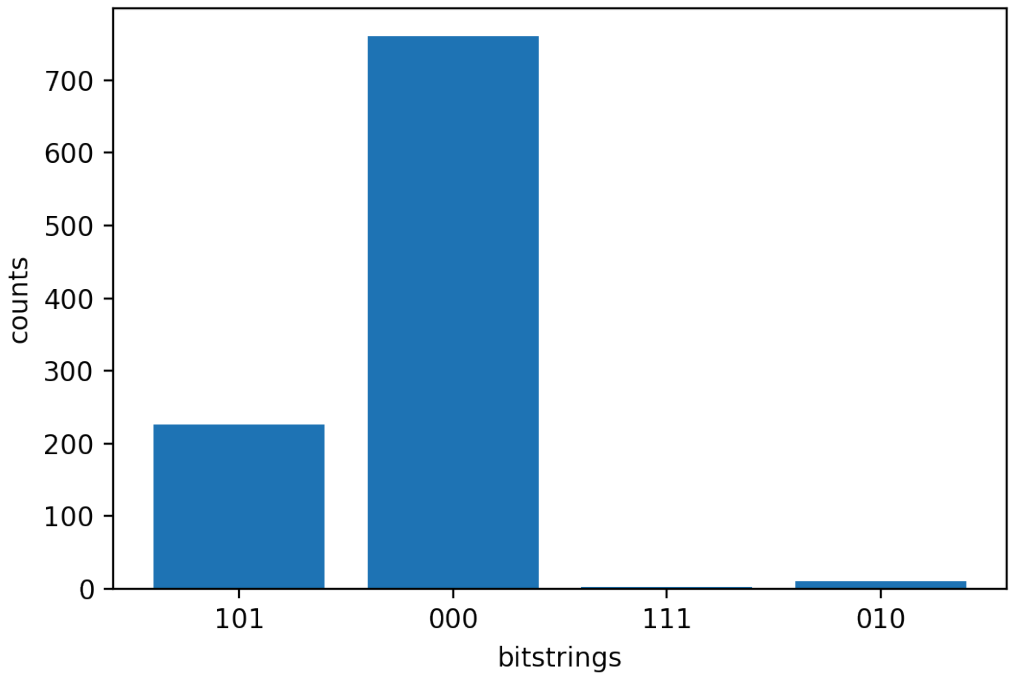
예제 결과 확인

ResultType도 지정했으므로 반환된 결과를 볼 수 있습니다. 결과 유형은 회로에 추가된 순서대로 표시됩니다.

```
print('Result types include:\n', result.result_types)
print('Variance=', result.values[0])
print('Probability=', result.values[1])

# 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=Variance(observable=['z'], targets=[0], type=<Type.variance:
 'variance'>), value=0.693084), ResultTypeValue(type=Probability(targets=[0, 2],
 type=<Type.probability: 'probability'>), value=array([0.777, 0.   , 0.   , 0.223]))]
Variance= 0.693084
Probability= [0.777 0.    0.    0.223]
```

```
Text(0, 0.5, 'counts')
```



로컬 시뮬레이터를 사용한 양자 작업 테스트

양자 작업을 로컬 시뮬레이터에 직접 전송하여 신속한 프로토타이핑 및 테스트를 수행할 수 있습니다. 이 시뮬레이터는 로컬 환경에서 실행되므로 Amazon S3 위치를 지정할 필요가 없습니다. 결과는 세션에서 직접 계산됩니다. 로컬 시뮬레이터에서 양자 작업을 실행하려면 shots 파라미터만 지정해야 합니다.

> **Note**
>
> 로컬 시뮬레이터가 처리할 수 있는 실행 속도와 최대 qubits 수는 Amazon Braket 노트북 인스턴스 유형 또는 로컬 하드웨어 사양에 따라 다릅니다.

다음 명령은 모두 동일하며 상태 벡터(노이즈 없음) 로컬 시뮬레이터를 인스턴스화합니다.

```
# Import the LocalSimulator module
from braket.devices import LocalSimulator

# The following are identical commands
```

```
device = LocalSimulator()
device = LocalSimulator("default")
device = LocalSimulator(backend="default")
device = LocalSimulator(backend="braket_sv")
```

그런 다음 다음을 사용하여 양자 작업을 실행합니다.

```
my_task = device.run(circ, shots=1000)
```

로컬 밀도 행렬(노이즈) 시뮬레이터를 인스턴스화하려면 고객은 다음과 같이 백엔드를 변경합니다.

```
# Import the LocalSimulator module
from braket.devices import LocalSimulator

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

로컬 시뮬레이터에서 특정 큐비트 측정

로컬 상태 벡터 시뮬레이터 및 로컬 밀도 행렬 시뮬레이터는 회로의 큐비트 부분 집합을 측정할 수 있는 회로 실행을 지원하는데, 이를 부분 측정이라고도 합니다.

예를 들어 다음 코드에서는 2큐비트 회로를 생성하고 회로 끝에 대상 큐비트를 지정하는 `measure` 명령을 추가하여 첫 번째 큐비트만 측정할 수 있습니다.

```
# Import the LocalSimulator module
from braket.devices import LocalSimulator

# Use the local simulator device
device = LocalSimulator()

# Define a bell circuit and only measure
circuit = Circuit().h(0).cnot(0, 1).measure(0)

# Run the circuit
task = device.run(circuit, shots=10)

# Get the results
result = task.result()

# Print the measurement counts for qubit 0
print(result.measurement_counts)
```

로컬 양자 디바이스 에뮬레이터

Amazon Braket의 로컬 에뮬레이터 도구를 사용하면 실제 양자 하드웨어에서 실행하기 전에 축어적 양자 프로그램을 로컬에서 에뮬레이션할 수 있습니다. 에뮬레이터는 디바이스 보정 데이터를 사용하여 축어적 회로를 검증하므로 호환성 문제를 조기에 파악할 수 있습니다.

또한 로컬 에뮬레이터는 다음 프로세스를 통해 양자 하드웨어 노이즈를 시뮬레이션합니다.

- 디바이스 보정 데이터를 사용하여 노이즈 모델 구성
- 회로의 각 게이트에 탈분극 노이즈 적용
- 회로 끝에 판독 오류 적용
- 로컬 밀도 행렬 시뮬레이터를 사용하여 노이즈가 있는 회로 시뮬레이션

로컬 에뮬레이터 사용에 대한 자세한 내용은 amazon-braket-examples GitHub 리포지토리의 Amazon Braket에서 축어적 회로의 로컬 에뮬레이션을 참조하세요.

이 섹션의 내용:

로컬 에뮬레이션의 이점

- 실시간 또는 과거 보정 데이터를 사용하여 디바이스 제약 조건에 대해 축어적 회로를 검증합니다.
- 양자 하드웨어에 작업을 제출하기 전에 문제를 디버깅합니다.
- 노이즈 없는 에뮬레이션과 노이즈 있는 에뮬레이션을 하드웨어 결과와 비교하여 노이즈 효과를 이해합니다.
- 노이즈 인식 양자 알고리즘을 개발하는 워크플로를 간소화합니다.

로컬 에뮬레이터 생성

로컬 양자 디바이스 에뮬레이터는 양자 디바이스 또는 디바이스 속성 집합에서 직접 생성할 수 있습니다. 디바이스를 직접 에뮬레이션할 때 에뮬레이터는 인스턴스화된 디바이스의 최신 보정 데이터를 사용합니다. 다음 코드 예제에서는 Rigetti's Ankaa-3 디바이스를 직접 에뮬레이션하는 방법을 보여줍니다.

```
from braket.aws.aws_device import AwsDevice

ankaa3 = AwsDevice("arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3")
ankaa3_emulator = ankaa3.emulator()
```

다음 예제에서는 JSON 형식의 Ankaa-3 디바이스 속성 집합에서 로컬 디바이스 에뮬레이터를 생성하는 방법을 보여줍니다.

```
from braket.aws import AwsDevice
from braket.emulation.local_emulator import LocalEmulator
import json

# Instantiate the device
ankaa3 = AwsDevice("arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3")
ankaa3_properties = ankaa3.properties

# Put the Ankaa-3 properties in a file named ankaa3_device_properties.json
with open("ankaa3_device_properties.json", "w") as f:
    json.dump(ankaa3_properties.json(), f)

# Load the json into the ankaa3_data_json variable
with open("ankaa3_device_properties.json", "r") as json_file:
    ankaa3_data_json = json.load(json_file)

# Create the Ankaa-3 local emulator from the json file you created
ankaa3_emulator = LocalEmulator.from_json(ankaa3_data_json)
```

다음을 통해 예제를 사용자 지정할 수 있습니다.

- 다른 QPU 디바이스의 속성 사용
- 에뮬레이터에 다른 JSON 파일 지정
- 에뮬레이터를 인스턴스화하기 전에 디바이스 속성 값 변경

Amazon Braket으로 양자 작업 실행

Braket은 다양한 유형의 양자 컴퓨터에 대한 안전한 온디맨드 액세스를 제공합니다. AQT, IonQIQMRigetti, 및의 게이트 기반 양자 컴퓨터와 QuEra의 아날로그 해밀턴 시뮬레이터에 액세스할 수 있습니다. 또한 선결제 약정이 없으며 개별 공급자를 통해 액세스 권한을 확보할 필요가 없습니다.

- Amazon Braket 콘솔은 리소스 및 양자 작업을 생성, 관리 및 모니터링하는 데 도움이 되는 디바이스 정보와 상태를 제공합니다.
- Amazon Braket Python SDK와 콘솔을 통해 양자 작업을 제출하고 실행합니다. SDK는 사전 구성된 Amazon Braket 노트북을 통해 액세스할 수 있습니다.
- Amazon Braket API는 Amazon Braket Python SDK 및 노트북을 통해 액세스할 수 있습니다. 양자 컴퓨팅을 프로그래밍 방식으로 활용하는 애플리케이션을 빌드하는 경우 API를 직접적으로 호출할 수 있습니다.

이 섹션의 예제에서는 AWS Python SDK for Braket(Boto3)과 함께 Amazon Braket Python SDK를 사용하여 Amazon Braket API를 직접 활용하는 방법을 보여줍니다.

Amazon Braket Python SDK에 대해 자세히 알아보기

Amazon Braket Python SDK를 사용하려면 먼저 AWS 와 통신할 수 있도록 Python SDK for Braket(Boto3)을 설치합니다 AWS API. Amazon Braket Python SDK는 양자 고객을 위한 Boto3 주변의 편리한 래퍼라고 생각할 수 있습니다.

- Boto3에는를 활용해야 하는 인터페이스가 포함되어 있습니다 AWS API. (Boto3는와 통신하는 대형 Python SDK입니다 AWS API. 대부분은 Boto3 인터페이스를 AWS 서비스 지원합니다.)
- Amazon Braket Python SDK에는 회로, 게이트, 디바이스, 결과 유형 및 양자 작업의 기타 부분을 위한 소프트웨어 모듈이 포함되어 있습니다. 프로그램을 생성할 때마다 해당 양자 작업에 필요한 모듈을 가져옵니다.
- Amazon Braket Python SDK는 양자 작업을 실행하는 데 필요한 모든 모듈 및 종속성이 사전 로드된 노트북을 통해 액세스할 수 있습니다.
- 노트북을 활용하지 않으려면 Amazon Braket Python SDK에서 모든 Python 스크립트로 모듈을 가져올 수 있습니다.

Boto3을 설치한 후 Amazon Braket Python SDK를 통해 양자 작업을 생성하는 단계의 개요는 다음과 유사합니다.

1. (선택 사항) 노트북을 엽니다.
2. 회로에 필요한 SDK 모듈을 가져옵니다.
3. QPU 또는 시뮬레이터를 지정합니다.
4. 회로를 인스턴스화합니다.
5. 회로를 실행합니다.
6. 결과를 수집합니다.

이 섹션의 예제는 각 단계의 세부 정보를 보여줍니다.

더 많은 예제는 GitHub에서 Amazon Braket 예제 리포지토리를 참조하세요.

이 섹션의 내용:

QPU에 양자 작업 제출

Amazon Braket은 양자 작업을 실행할 수 있는 여러 디바이스에 대한 액세스를 제공합니다. 양자 작업을 개별적으로 제출하거나 양자 작업 일괄 처리를 설정할 수 있습니다.

양자 처리 장치(Quantum Processing Unit, QPU)

언제든지 양자 작업을 QPU에 제출할 수 있지만, 해당 작업은 Amazon Braket 콘솔의 디바이스 페이지에 표시되는 특정 가용성 기간 내에 실행됩니다. 다음 섹션에 소개된 양자 작업 ID를 사용하여 양자 작업의 결과를 검색할 수 있습니다.

- AQT IBEX-Q1 : `arn:aws:braket:eu-north-1::device/qpu/aqt/Ibex-Q1`
- IonQ Aria-1 : `arn:aws:braket:us-east-1::device/qpu/ionq/Aria-1`
- IonQ Forte-1 : `arn:aws:braket:us-east-1::device/qpu/ionq/Forte-1`
- IonQ Forte-Enterprise-1 : `arn:aws:braket:us-east-1::device/qpu/ionq/Forte-Enterprise-1`
- IQM Garnet : `arn:aws:braket:eu-north-1::device/qpu/iqm/Garnet`

- IQM Emerald : `arn:aws:braket:eu-north-1::device/qpu/iqm/Emerald`
- QuEra Aquila : `arn:aws:braket:us-east-1::device/qpu/quera/Aquila`
- Rigetti Ankaa-3 : `arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3`

> **Note**
>
> QPU 및 온디맨드 시뮬레이터의 경우 CREATED 상태의 양자 작업을 취소할 수 있습니다. 온디맨드 시뮬레이터 및 QPU의 경우 QUEUED 상태의 양자 작업을 최선의 방식으로 취소할 수 있습니다. 참고로 QPU QUEUED 양자 작업은 QPU 가용성 기간 동안 성공적으로 취소될 가능성이 낮습니다.

이 섹션의 내용:

AQT

AQT의 IBEX-Q1 QPU는 초고진공실에 배치된 거시적 무선 주파수 트랩에서 $^{40}Ca^{+}$의 수정을 기반으로 합니다. 디바이스는 실내에서 실행되며 2개의 19인치 데이터 센터 호환 랙에 맞습니다.

높은 충실도 게이트는 트랩의 낮은 난방 속도와 쿼비트 회전을 위한 직접 광학 전환을 사용하여 활성화됩니다. 쿼비트 전환은 상대 주파수 안정성이 매우 높은 좁은 선폭 레이저에 의해 구동됩니다. 또한 쿼비트는 효율적인 상태 준비 및 광학 선반을 통한 읽기 기능을 제공합니다. All-to-all 연결은 이온 수정의 장거리 Coulomb 상호 작용을 통해 달성됩니다. 단이온 주소 지정 및 판독은 높은 수치의 렌즈 렌즈를 사용하여 이루어집니다.

AQT 디바이스는 다음 양자 게이트를 지원합니다.

```
'ccnot', 'cnot', 'cphaseshift', 'cphaseshift00', 'cphaseshift01', 'cphaseshift10',
 'cswap', 'swap', 'iswap', 'pswap', 'ecr', 'cy', 'cz', 'xy', 'xx', 'yy', 'zz', 'h',
```

```
 'i', 'phaseshift', 'rx', 'ry', 'rz', 's', 'si', 't', 'ti', 'v', 'vi', 'x', 'y', 'z',
 'prx'
```

축어적 컴파일을 사용하면 AQT 디바이스는 다음과 같은 네이티브 게이트를 지원합니다.

```
'prx', 'xx', 'rz'
```

> **Note**
>
> 다음은 AQT네이티브 게이트와 Amazon Braket 간의 동일한 게이트를 설명합니다.
>
> - AQT M™lmer-S™rensen(MS 또는 RXX) 게이트는 Braket의 `'xx'` 게이트에 해당합니다.
> - AQT R 게이트는 Braket의 `'prx'` 게이트에 해당합니다.
> - `'rz'` 게이트 이름 지정은 동일합니다.

IonQ

IonQ는 이온 트랩 기술을 기반으로 게이트 기반 QPU를 제공합니다. IonQ's의 트랩 이온 QPU는 진공 챔버 내 미세 가공된 표면 전극 트랩에 의해 공간적으로 제한되는 트랩된 171Yb+ 이온 체인을 기반으로 빌드됩니다.

IonQ 디바이스는 다음 양자 게이트를 지원합니다.

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

축어적 컴파일을 통해 IonQ QPU는 다음과 같은 네이티브 게이트를 지원합니다.

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

네이티브 MS 게이트를 사용할 때 두 개의 위상 파라미터만 지정하면 완전 얽힘을 생성하는 MS 게이트가 실행됩니다. 완전 얽힘을 생성하는 MS 게이트는 항상 π/2 회전을 수행합니다. 다른 각도를 지정하고 부분 얽힘을 생성하는 MS 게이트를 실행하려면, 세 번째 파라미터를 추가하여 원하는 각도를 지정합니다. 자세한 내용은 braket.circuits.gate 모듈을 참조하세요.

이러한 네이티브 게이트는 축어적 컴파일에만 사용할 수 있습니다. 축어적 컴파일에 대한 자세한 내용은 축어적 컴파일을 참조하세요.

IQM

IQM 양자 프로세서는 초전도 트랜스몬 큐비트를 기반으로 하는 범용 게이트 모델 디바이스입니다. IQM Garnet은 20큐비트 디바이스이며 IQM Emerald는 54큐비트 디바이스입니다. 이 두 디바이스 모두 결정 격자 토폴로지라고도 하는 정사각형 격자 토폴로지를 사용합니다.

IQM 디바이스는 다음 양자 게이트를 지원합니다.

```
"ccnot", "cnot", "cphaseshift", "cphaseshift00", "cphaseshift01", "cphaseshift10",
 "cswap", "swap", "iswap", "pswap", "ecr", "cy", "cz", "xy", "xx", "yy", "zz", "h",
 "i", "phaseshift", "rx", "ry", "rz", "s", "si", "t", "ti", "v", "vi", "x", "y", "z"
```

축어적 컴파일을 통해 IQM 디바이스는 다음과 같은 네이티브 게이트를 지원합니다.

```
'cz', 'prx'
```

Rigetti

Rigetti 양자 프로세서는 완전히 조정 가능한 초전도 qubits를 기반으로 하는 범용 게이트 모델 시스템입니다.

- Ankaa-3 시스템은 확장 가능한 멀티칩 기술을 활용하는 84큐비트 디바이스입니다.

Rigetti 디바이스는 다음 양자 게이트를 지원합니다.

```
'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'
```

축어적 컴파일을 통해 Ankaa-3은 다음과 같은 네이티브 게이트를 지원합니다.

```
'rx', 'rz', 'iswap'
```

Rigetti 초전도 양자 프로세서는 각도가 ±π/2 또는 ±π인 'rx' 게이트를 실행할 수 있습니다.

펄스 수준 제어는 Ankaa-3 시스템에 대해 다음 유형의 사전 정의된 프레임 세트를 지원하는 Rigetti 디바이스에서 사용할 수 있습니다.

```
`flux_tx`, `charge_tx`, `readout_rx`, `readout_tx`
```

QuEra

QuEra는 아날로그 해밀토니안 시뮬레이션(Analog Hamiltonian Simulation, AHS) 양자 작업을 실행할 수 있는 중성 원자 기반 디바이스를 제공합니다. 이러한 특수 용도 디바이스는 동시에 상호 작용하는 수백여 큐비트의 시간 종속 양자 동역학을 충실하게 재현합니다.

큐비트 레지스터의 레이아웃과 조작장의 시간 및 공간 종속성을 고려하여 AHS의 패러다임으로 이러한 디바이스를 프로그래밍할 수 있습니다. Amazon Braket은 python SDK, `braket.ahs`의 AHS 모듈을 통해 이러한 프로그램을 구성하는 유틸리티를 제공합니다.

자세한 내용은 아날로그 해밀토니안 시뮬레이션 예제 노트북 또는 QuEra Aquila를 사용하여 아날로그 프로그램 제출 페이지를 참조하세요.

예: QPU에 양자 작업 제출

Amazon Braket을 사용하면 QPU 디바이스에서 양자 회로를 실행할 수 있습니다. 다음 예제에서는 Rigetti 또는 IonQ 디바이스에 양자 작업을 제출하는 방법을 보여줍니다.

Rigetti Ankaa-3 디바이스를 선택한 다음 연결된 연결 그래프를 확인합니다.

```
# import the QPU module
from braket.aws import AwsDevice
# choose the Rigetti device
device = AwsDevice("arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3")

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

```
{'fullyConnected': False,
 'connectivityGraph': {'0': ['1', '7'],
  '1': ['0', '2', '8'],
  '2': ['1', '3', '9'],
  '3': ['2', '4', '10'],
  '4': ['3', '5', '11'],
  '5': ['4', '6', '12'],
  '6': ['5', '13'],
  '7': ['0', '8', '14'],
  '8': ['1', '7', '9', '15'],
  '9': ['2', '8', '10', '16'],
  '10': ['3', '9', '11', '17'],
  '11': ['4', '10', '12', '18'],
  '12': ['5', '11', '13', '19'],
```

```
    '13': ['6', '12', '20'],
    '14': ['7', '15', '21'],
    '15': ['8', '14', '22'],
    '16': ['9', '17', '23'],
    '17': ['10', '16', '18', '24'],
    '18': ['11', '17', '19', '25'],
    '19': ['12', '18', '20', '26'],
    '20': ['13', '19', '27'],
    '21': ['14', '22', '28'],
    '22': ['15', '21', '23', '29'],
    '23': ['16', '22', '24', '30'],
    '24': ['17', '23', '25', '31'],
    '25': ['18', '24', '26', '32'],
    '26': ['19', '25', '33'],
    '27': ['20', '34'],
    '28': ['21', '29', '35'],
    '29': ['22', '28', '30', '36'],
    '30': ['23', '29', '31', '37'],
    '31': ['24', '30', '32', '38'],
    '32': ['25', '31', '33', '39'],
    '33': ['26', '32', '34', '40'],
    '34': ['27', '33', '41'],
    '35': ['28', '36', '42'],
    '36': ['29', '35', '37', '43'],
    '37': ['30', '36', '38', '44'],
    '38': ['31', '37', '39', '45'],
    '39': ['32', '38', '40', '46'],
    '40': ['33', '39', '41', '47'],
    '41': ['34', '40', '48'],
    '42': ['35', '43', '49'],
    '43': ['36', '42', '44', '50'],
    '44': ['37', '43', '45', '51'],
    '45': ['38', '44', '46', '52'],
    '46': ['39', '45', '47', '53'],
    '47': ['40', '46', '48', '54'],
    '48': ['41', '47', '55'],
    '49': ['42', '56'],
    '50': ['43', '51', '57'],
    '51': ['44', '50', '52', '58'],
    '52': ['45', '51', '53', '59'],
    '53': ['46', '52', '54'],
    '54': ['47', '53', '55', '61'],
    '55': ['48', '54', '62'],
    '56': ['49', '57', '63'],
```

```
  '57': ['50', '56', '58', '64'],
  '58': ['51', '57', '59', '65'],
  '59': ['52', '58', '60', '66'],
  '60': ['59'],
  '61': ['54', '62', '68'],
  '62': ['55', '61', '69'],
  '63': ['56', '64', '70'],
  '64': ['57', '63', '65', '71'],
  '65': ['58', '64', '66', '72'],
  '66': ['59', '65', '67'],
  '67': ['66', '68'],
  '68': ['61', '67', '69', '75'],
  '69': ['62', '68', '76'],
  '70': ['63', '71', '77'],
  '71': ['64', '70', '72', '78'],
  '72': ['65', '71', '73', '79'],
  '73': ['72', '80'],
  '75': ['68', '76', '82'],
  '76': ['69', '75', '83'],
  '77': ['70', '78'],
  '78': ['71', '77', '79'],
  '79': ['72', '78', '80'],
  '80': ['73', '79', '81'],
  '81': ['80', '82'],
  '82': ['75', '81', '83'],
  '83': ['76', '82']}}
```

위의 딕셔너리 `connectivityGraph`은 Rigetti 디바이스의 각 큐비트에 대한 이웃 큐비트를 나열합니다.

IonQ Aria-1 디바이스를 선택합니다

IonQ Aria-1 디바이스의 경우 디바이스가 all-to-all 연결을 제공하므로 다음 예제와 같이 `connectivityGraph`가 비어 있습니다. 따라서 상세한 `connectivityGraph`는 필요하지 않습니다.

```
# or choose the IonQ Aria-1 device
device = AwsDevice("arn:aws:braket:us-east-1::device/qpu/ionq/Aria-1")

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

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

다음 예제와 같이 기본 버킷 이외의 위치를 지정하도록 선택하면 shots(기본값 = 1,000), `poll_timeout_seconds`(기본값 = 432,000 = 5일), `poll_interval_seconds`(기본값 = 1) 및 결과가 저장될 S3 버킷(`s3_location`)의 위치를 조정할 수 있습니다.

```
my_task = device.run(circ, s3_location = 'amazon-braket-my-folder', shots=100,
 poll_timeout_seconds = 100, poll_interval_seconds = 10)
```

IonQ 및 Rigetti 디바이스는 제공된 회로를 해당 네이티브 게이트 세트로 자동으로 컴파일하고 추상 qubit 인덱스를 해당 QPU의 물리적 qubits에 매핑합니다.

> **Note**
>
> QPU 디바이스는 용량이 제한되어 있습니다. 용량에 도달하면 대기 시간이 길어질 수 있습니다.

Amazon Braket은 특정 가용성 기간 내에 QPU 양자 작업을 실행할 수 있지만, 모든 해당 데이터와 메타데이터가 적절한 S3 버킷에 안정적으로 저장되므로 언제든지(24/7) 양자 작업을 제출할 수 있습니다. 다음 섹션에서 나와 있듯이, `AwsQuantumTask` 및 고유한 양자 작업 ID를 사용하여 양자 작업을 복구할 수 있습니다.

컴파일된 회로 검사

양자 처리 장치(QPU)와 같은 하드웨어 디바이스에서 양자 회로를 실행해야 하는 경우 먼저 디바이스가 이해하고 처리할 수 있는 허용 가능한 형식으로 회로를 컴파일해야 합니다. 예를 들어, 상위 수준 양자 회로를 대상 QPU 하드웨어에서 지원하는 특정 네이티브 게이트로 변환합니다. 양자 회로의 실제 컴파일된 출력을 검사하는 것은 디버깅 및 최적화 목적에 매우 유용할 수 있습니다. 이러한 지식은 양자 애플리케이션의 성능과 효율성을 개선하기 위한 잠재적인 문제, 병목 현상 또는 기회를 식별하는 데 도움이 될 수 있습니다. 아래 제공된 코드를 사용하여 Rigetti 및 IQM 양자 컴퓨팅 디바이스 모두에 대한 양자 회로의 컴파일된 출력을 보고 분석할 수 있습니다.

```
task = AwsQuantumTask(arn=task_id, aws_session=session)
# After the task has finished running
task_result = task.result()
compiled_circuit = task_result.get_compiled_circuit()
```

> **Note**
>
> 현재 IonQ 디바이스에 대한 컴파일된 회로 출력 보기는 지원되지 않습니다.

여러 프로그램 실행

Amazon Braket은 여러 양자 프로그램을 효율적으로 실행하기 위한 두 가지 접근 방식, 즉 프로그램 세트와 양자 작업 일괄 처리를 제공합니다.

프로그램 세트는 여러 프로그램으로 워크로드를 실행하는 데 선호되는 방법입니다. 이를 통해 여러 프로그램을 단일 Amazon Braket 양자 작업으로 패키징할 수 있습니다. 프로그램 세트는 특히 프로그램 실행 수가 100개에 가까울 때 프로그램을 개별적으로 제출하는 것에 비해 성능을 개선하고 비용을 절감합니다.

현재 IQM 및 Rigetti 디바이스가 프로그램 세트를 지원합니다. 프로그램 세트를 QPU에 제출하기 전에 먼저 Amazon Braket Local Simulator에서 테스트하는 것이 좋습니다. 디바이스가 프로그램 세트를 지원하는지 확인하려면 Amazon Braket SDK를 사용하여 디바이스의 속성을 보거나 Amazon Braket 콘솔에서 디바이스 페이지를 볼 수 있습니다.

다음 예제에서는 프로그램 세트를 실행하는 방법을 보여줍니다.

```
from math import pi
from braket.devices import LocalSimulator
from braket.program_sets import ProgramSet
from braket.circuits import Circuit

program_set = ProgramSet([
    Circuit().h(0).cnot(0,1),
    Circuit().rx(0, pi/4).ry(1, pi/8).cnot(1,0),
    Circuit().t(0).t(1).cz(0,1).s(0).cz(1,2).s(1).s(2),
])

device = LocalSimulator()
result = device.run(program_set, shots=300).result()
print(result[0][0].counts)  # The result of the first program in the program set
```

프로그램 세트를 구성하는 다양한 방법(예: 단일 프로그램으로 여러 관찰 가능 항목 또는 파라미터에서 프로그램 세트를 구성) 및 프로그램 세트 결과를 검색하는 방법에 대해 자세히 알아보려면 Amazon

Braket 개발자 안내서의 프로그램 세트 섹션과 Braket 예제 Github 리포지토리의 프로그램 세트 폴더를 참조하세요.

양자 작업 일괄 처리는 모든 Amazon Braket 디바이스에서 사용할 수 있습니다. 일괄 처리는 여러 양자 작업을 병렬로 처리할 수 있으므로 온디맨드 시뮬레이터(SV1, DM1 또는 TN1)에서 실행하는 양자 작업에 특히 유용합니다. 일괄 처리를 사용하면 양자 작업을 병렬로 시작할 수 있습니다. 예를 들어 10개의 양자 작업이 필요하고 해당 양자 작업의 프로그램이 서로 독립적이어야 하는 계산을 수행하려는 경우 작업 일괄 처리를 사용하는 것이 좋습니다. 프로그램 세트를 지원하지 않는 디바이스에서 여러 프로그램으로 워크로드를 실행할 때는 양자 작업 일괄 처리를 사용하십시오.

다음 예제에서는 양자 작업 배체를 실행하는 방법을 보여줍니다.

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

bell = Circuit().h(0).cnot(0, 1)
circuits = [bell for _ in range(5)]

device = LocalSimulator()
batch = device.run_batch(circuits, shots=100)
print(batch.results()[0].measurement_counts)  # The result of the first quantum task in
 the batch
```

일괄 처리에 대한 자세한 내용은 GitHub의 Amazon Braket 예제를 참조하세요.

이 섹션의 내용:

- 프로그램 세트 및 비용 정보
- 양자 작업 일괄 처리 및 비용 정보
- 양자 작업 일괄 처리 및 PennyLane
- 작업 일괄 처리 및 파라미터화된 회로

프로그램 세트 및 비용 정보

프로그램 세트는 최대 100개의 프로그램 또는 파라미터 세트를 단일 양자 작업으로 패키징하여 여러 양자 프로그램을 효율적으로 실행합니다. 프로그램 세트를 사용하면 모든 프로그램의 총 샷을 기준으로 작업당 요금과 샷당 요금만 지불하면 되므로 프로그램을 개별적으로 제출하는 것보다 비용이 크게 절감됩니다. 이 접근 방식은 프로그램 수가 많고 프로그램당 샷 수가 적은 워크로드에 특히 유용합니다. 프로그램 세트는 현재 IQM 및 Rigetti 디바이스와 Amazon Braket Local Simulator에서 지원됩니다.

자세한 내용은 프로그램 세트 섹션에서 상세한 구현 단계, 모범 사례 및 코드 예제를 참조하세요.

양자 작업 일괄 처리 및 비용 정보

양자 작업 일괄 처리 및 결제 비용과 관련하여 유의해야 할 몇 가지 주의 사항입니다.

- 기본적으로, 양자 작업 일괄 처리는 모든 시간 초과 또는 실패 양자 작업을 3회 재시도합니다.
- SV1의 34qubits와 같이 장기 실행 양자 작업 배치에는 많은 비용이 발생할 수 있습니다. 양자 작업 배치를 시작하기 전에 `run_batch` 할당 값을 주의 깊게 다시 확인해야 합니다. TN1을 `run_batch`와 함께 사용하지 않는 것이 좋습니다.
- TN1은 실패한 리허설 단계 작업에 대한 비용을 발생시킬 수 있습니다(자세한 내용은 TN1 description 참조). 자동 재시도는 비용을 증가시킬 수 있으므로 TN1을 사용할 때는 일괄 처리 시 'max_retries' 수를 0으로 설정하는 것이 좋습니다(Quantum Task Batching, Line 186 참조).

양자 작업 일괄 처리 및 PennyLane

다음 예제와 같이, Amazon Braket에서 PennyLane 사용 시 Amazon Braket 디바이스를 인스턴스화할 때 `parallel = True`을 설정하여 일괄 처리를 활용합니다.

```
import pennylane as qml

# Define the number of wires (qubits) you want to use
wires = 2  # For example, using 2 qubits

# Define your S3 bucket
my_bucket = "amazon-braket-s3-demo-bucket"
my_prefix = "pennylane-batch-output"
s3_folder = (my_bucket, my_prefix)

device = qml.device("braket.aws.qubit",
                    device_arn="arn:aws:braket:::device/quantum-simulator/amazon/sv1",
                    wires=wires,
                    s3_destination_folder=s3_folder,
                    parallel=True)
```

PennyLane을 사용한 일괄 처리에 대한 자세한 내용은 양자 회로의 병렬화된 최적화를 참조하세요.

작업 일괄 처리 및 파라미터화된 회로

파라미터화된 회로가 포함된 양자 작업 배치를 제출할 때 배치의 모든 양자 작업에 사용되는 `inputs` 딕셔너리 또는 입력 딕셔너리의 `list`를 제공할 수 있습니다. 이 경우 다음 예제와 같이 `i`번째 딕셔너리가 `i`번째 작업과 쌍을 이룹니다.

```
from braket.circuits import Circuit, FreeParameter, Observable
from braket.aws import AwsQuantumTaskBatch, AwsDevice

# Define your quantum device
device = AwsDevice("arn:aws:braket:::device/quantum-simulator/amazon/sv1")

# Create the free parameters
alpha = FreeParameter('alpha')
beta = FreeParameter('beta')

# Create two circuits
circ_a = Circuit().rx(0, alpha).ry(1, alpha).cnot(0, 2).xx(0, 2, beta)
circ_a.variance(observable=Observable.Z(), target=0)

circ_b = Circuit().rx(0, alpha).rz(1, alpha).cnot(0, 2).zz(0, 2, beta)
circ_b.expectation(observable=Observable.Z(), target=2)

# Use the same inputs for both circuits in one batch
tasks = device.run_batch([circ_a, circ_b], inputs={'alpha': 0.1, 'beta': 0.2})

# Or provide each task its own set of inputs
inputs_list = [{'alpha': 0.3, 'beta': 0.1}, {'alpha': 0.1, 'beta': 0.4}]

tasks = device.run_batch([circ_a, circ_b], inputs=inputs_list)
```

단일 파라메트릭 회로에 대한 입력 딕셔너리 목록을 준비하여 양자 작업 배치로 제출할 수도 있습니다. 목록에 N개의 입력 딕셔너리가 있는 경우 배치에는 N개의 양자 작업이 포함됩니다. `i`번째 양자 작업은 `i`번째 입력 딕셔너리로 실행되는 회로에 해당합니다.

```
from braket.circuits import Circuit, FreeParameter

# Create a parametric circuit
circ = Circuit().rx(0, FreeParameter('alpha'))

# Provide a list of inputs to execute with the circuit
inputs_list = [{'alpha': 0.1}, {'alpha': 0.2}, {'alpha': 0.3}]
```

```
tasks = device.run_batch(circ, inputs=inputs_list, shots=100)
```

양자 작업은 언제 실행되나요?

회로를 제출하면 Amazon Braket이 지정한 디바이스로 회로를 전송합니다. QPU 및 온디맨드 시뮬레이터 양자 작업은 수신된 순서대로 대기열에 추가되고 처리됩니다. 양자 작업을 제출한 후 처리하는 데 필요한 시간은 다른 Amazon Braket 고객이 제출한 작업의 수와 복잡성 및 선택한 QPU의 가용성에 따라 달라집니다.

이 섹션의 내용:

- QPU 가용성 기간 및 상태
- 대기열 가시성
- 이메일 또는 SMS 알림 설정

QPU 가용성 기간 및 상태

QPU 가용성은 디바이스마다 다릅니다.

Amazon Braket 콘솔의 디바이스 페이지에서 현재 및 예정된 가용성 기간과 디바이스 상태를 볼 수 있습니다. 또한 각 디바이스 페이지에는 양자 작업 및 하이브리드 작업에 대한 개별 대기열 깊이가 표시됩니다.

가용성 기간에 관계없이 고객이 사용할 수 없는 경우 디바이스는 오프라인으로 간주됩니다. 예를 들어 예약된 유지 관리, 업그레이드 또는 운영 문제로 인해 오프라인 상태일 수 있습니다.

대기열 가시성

양자 작업 또는 하이브리드 작업을 제출하기 전에 디바이스 대기열 깊이를 확인하여 앞에 있는 양자 작업 또는 하이브리드 작업 수를 확인할 수 있습니다.

대기열 깊이

Queue depth는 특정 디바이스에 대해 대기열에 있는 양자 작업 및 하이브리드 작업 수를 나타냅니다. 디바이스의 양자 작업 및 하이브리드 작업 대기열 수는 Amazon Braket Management Console 또는 Braket Software Development Kit (SDK)를 통해 액세스할 수 있습니다.

1. 작업 대기열 깊이는 현재 일반 우선순위에서 실행되기를 기다리는 양자 작업의 총 수를 나타냅니다.

2. 우선순위 작업 대기열 깊이는 Amazon Braket Hybrid Jobs을 통해 실행되기를 기다리는 제출된 양자 작업의 총 수를 나타냅니다. 이러한 작업은 독립 실행형 작업보다 먼저 실행됩니다.
3. 하이브리드 작업 대기열 깊이는 현재 디바이스에 대기 중인 하이브리드 작업의 총 수를 나타냅니다. 하이브리드 작업의 일부로 제출된 Quantum tasks에 우선순위가 있으며 Priority Task Queue에서 집계됩니다.

Braket SDK를 통해 대기열 깊이를 보고자 하는 고객은 다음 코드 조각을 수정하여 양자 작업 또는 하이브리드 작업의 대기열 위치를 가져올 수 있습니다.

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

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


# returns the number of hybrid jobs queued on the device
print(device.queue_depth().jobs)
'3'
```

양자 작업 또는 하이브리드 작업을 QPU에 제출하면 워크로드가 QUEUED 상태가 될 수 있습니다. Amazon Braket은 고객에게 양자 작업 및 하이브리드 작업 대기열 위치에 대한 가시성을 제공합니다.

대기열 위치

Queue position는 각 디바이스 대기열 내에서 양자 작업 또는 하이브리드 작업의 현재 위치를 나타냅니다. 이는 Braket Software Development Kit (SDK) 또는 Amazon Braket Management Console을 통해 양자 작업 또는 하이브리드 작업에 대해 얻을 수 있습니다.

Braket SDK를 통해 대기열 위치를 보고자 하는 고객은 다음 코드 조각을 수정하여 양자 작업 또는 하이브리드 작업의 대기열 위치를 가져올 수 있습니다.

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

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

# retrieve the queue position information
```

```
print(task.queue_position().queue_position)

# Returns the number of Quantum Tasks queued ahead of you
'2'


from braket.aws import AwsQuantumJob

job = AwsQuantumJob.create(
    "arn:aws:braket:eu-north-1::device/qpu/iqm/Garnet",
    source_module="algorithm_script.py",
    entry_point="algorithm_script:start_here",
    wait_until_complete=False
)

# retrieve the queue position information
print(job.queue_position().queue_position)
'3' # returns the number of hybrid jobs queued ahead of you
```

이메일 또는 SMS 알림 설정

Amazon Braket은 QPU의 가용성이 변경되거나 양자 작업의 상태가 변경될 때 Amazon EventBridge로 이벤트를 전송합니다. 디바이스 및 양자 작업 상태 변경 알림을 이메일 또는 SMS 메시지로 수신하려면 다음 단계를 따릅니다.

1. Amazon SNS 주제와 이메일 또는 SMS 구독을 생성합니다. 이메일 또는 SMS의 가용성은 리전에 따라 다릅니다. 자세한 내용은 Amazon SNS 시작하기 및 SMS 메시지 전송을 참조하세요.
2. EventBridge에서 SNS 주제에 대한 알림을 트리거하는 규칙을 생성합니다. 자세한 내용은 Amazon EventBridge를 사용한 Amazon Braket 모니터링을 참조하세요.

(선택 사항) SNS 알림 설정

Amazon Braket 양자 작업이 완료되면 알림을 받도록 Amazon Simple Notification Service(SNS)를 통해 알림을 설정할 수 있습니다. 활성 알림은 대기 시간이 길어질 것으로 예상되는 경우에 유용합니다. 예를 들어 대규모 양자 작업을 제출하거나 디바이스의 가용성 기간 외에 양자 작업을 제출할 때 유용합니다. 양자 작업이 완료될 때까지 기다리는 것을 원치 않는다면 SNS 알림을 설정할 수 있습니다.

Amazon Braket 노트북의 설정 단계 안내를 따릅니다. 자세한 내용은 GitHub의 Amazon Braket 예제 및 특히 알림 설정을 위한 예제 노트북을 참조하세요.

예약 작업

예약을 통해 선택한 양자 디바이스에 독점적으로 액세스할 수 있습니다. 편리한 시간에 예약을 할 수 있으므로 워크로드 실행이 시작되고 종료되는 시기를 정확히 알 수 있습니다. 예약은 모든 Braket 디바이스에 대해 1시간 단위로 제공되며 추가 비용 없이 최대 48시간 전에 취소할 수 있습니다. 예정된 예약의 양자 작업 및 하이브리드 작업을 미리 대기열에 추가하거나, Braket Direct Reservation ARN을 사용하거나, 예약 중에 워크로드를 제출하는 것이 좋습니다.

전용 디바이스 액세스 비용은 양자 처리 장치(QPU)에서 실행하는 양자 작업 및 하이브리드 작업 수에 관계없이 예약 기간을 기준으로 합니다. 예약에 사용할 수 있는 업데이트된 양자 컴퓨터 목록은 요금 페이지 또는 Amazon Braket 관리 콘솔을 통해 확인할 수 있습니다.

> **Note**
>
> 예약에는 게이트샷 제한이 없습니다. 또한 IonQ 디바이스의 경우 오류 완화 작업의 최소 샷 수가 500개(온디맨드의 경우 2500개)로 줄어듭니다.

예약 사용 시기

예약 액세스를 활용하면 양자 워크로드의 실행 시작 및 종료 시기를 정확하게 알 수 있는 편리하고 예측 가능한 기능을 제공합니다. 온디맨드 방식으로 작업 및 하이브리드 작업을 제출하는 것에 비해 다른 고객 작업이 있는 대기열에서 대기할 필요가 없습니다. 예약 기간 동안 디바이스에 독점적으로 액세스할 수 있으므로, 전체 예약 기간에 걸쳐 디바이스에서는 해당 사용자의 워크로드만 실행됩니다.

연구의 설계 및 프로토타입 작성 단계에 온디맨드 액세스를 사용하여 알고리즘을 빠르고 비용 효율적으로 반복할 수 있도록 하는 것이 좋습니다. 최종 실험 결과를 생성할 준비가 되면 프로젝트 또는 게시 기한을 충족할 수 있도록 편리한 시간에 디바이스 예약을 고려하세요. 또한 양자 컴퓨터에서 라이브 데모 또는 워크숍을 실행하는 경우와 같이 특정 시간 동안 작업 실행을 원하는 경우 예약을 사용하는 것이 좋습니다.

이 섹션의 내용:

- 예약을 생성하는 방법
- 예약 기간 동안 양자 작업 실행
- 예약 기간 동안 하이브리드 작업 실행
- 예약 종료 시 발생하는 일
- 기존 예약 취소 또는 일정 조정

예약을 생성하는 방법

예약을 생성하려면 다음 단계에 따라 Braket 팀에 문의하세요.

1. Amazon Braket 콘솔을 엽니다.
2. 왼쪽 창에서 Braket Direct를 선택한 다음 예약 섹션에서 디바이스 예약을 선택합니다.
3. 예약하려는 디바이스를 선택합니다.
4. 이름 및 이메일을 포함한 연락처 정보를 제공합니다. 정기적으로 확인하는 유효한 이메일 주소를 제공해야 합니다.
5. 워크로드에 대한 정보 제공에서 예약을 사용하여 실행할 워크로드에 대한 세부 정보를 제공합니다. 예를 들어, 원하는 예약 기간, 관련 제약 조건 또는 원하는 일정 등이 있습니다.

양식을 제출하면 Braket 팀으로부터 다음 단계가 포함된 이메일을 받게 됩니다. 예약이 확인되면 이메일을 통해 예약 ARN을 받게 됩니다. 예약 ARN을 사용하여 예약 작업을 생성하려면이 필요합니다. 예약 ARN 없이 생성된 작업은 일반 온디맨드 대기열에 제출되며 예약 중에 실행되지 않습니다.

> **Note**
>
> 예약 ARN을 받은 후에만 예약이 확정됩니다.

예약은 최소 1시간 단위로 가능하며 특정 디바이스에는 추가 예약 기간 제약 조건(최소 및 최대 예약 기간 포함)이 적용될 수 있습니다. Braket 팀은 예약을 확정하기 전에 관련 정보를 공유합니다.

Braket 팀이 이메일을 통해 연락하여 Braket 전문가와 30분 세션을 예약합니다.

예약 기간 동안 양자 작업 실행

예약 생성에서 유효한 예약 ARN을 얻은 후 예약 기간 동안 실행할 양자 작업을 생성할 수 있습니다. 예약 ARN과 함께 제출된 양자 작업 및 하이브리드 작업은 디바이스 대기열에 표시되지 않습니다. 예약 시작 시간 이전에 제출된 작업은 예약이 시작될 때까지 QUEUED 상태로 유지됩니다.

> **Note**
>
> 예약은 AWS 계정 및 디바이스별로 다릅니다. 예약을 생성한 AWS 계정만 예약 ARN을 사용할 수 있습니다.
>
> 예약 중에 예약 작업과 일반 작업을 모두 생성할 수 있습니다. 생성된 Braket 양자 작업이 예약과 연결되어 있는지 확인하려면 Braket 콘솔의 양자 작업 페이지에서 “예약 ARN” 필드를 확인

하거나 SDK를 사용하여 작업 메타데이터에서 동일한 필드를 쿼리합니다. 이 페이지의 나머지 부분에서는 재개와 연결된 작업을 지정하는 방법을 설명합니다.

Python SDKs Braket, , CUDA-Q, PennyLane Qiskit또는 boto3(Boto3 작업)과 직접 같은를 사용하여 양자 작업을 생성할 수 있습니다. 예약을 사용하려면 버전 v1.79.0 이상의 Amazon Braket Python SDK가 있어야 합니다. 다음 코드를 사용하여 최신 Braket SDK, Qiskit 공급자 및 PennyLane 플러그인으로 업데이트할 수 있습니다.

```
pip install --upgrade amazon-braket-sdk amazon-braket-pennylane-plugin qiskit-braket-
provider
```

`DirectReservation` 컨텍스트 관리자를 사용하여 작업 실행

예정된 예약 내에서 작업을 실행하는 데 권장되는 방법은 `DirectReservation` 컨텍스트 관리자를 사용하는 것입니다. 컨텍스트 관리자는 대상 디바이스와 예약 ARN을 지정하여 Python `with` 문 내에서 생성된 모든 작업이 디바이스에 대한 독점적 액세스 권한으로 실행되도록 합니다.

먼저 양자 회로와 디바이스를 정의합니다. 그런 다음 예약 컨텍스트를 사용하고 작업을 실행합니다. 전체 워크로드가 **`with`** 블록 내에서 실행되어야 합니다. **`with`** 블록 범위 외부에서 실행되는 모든 것은 예약과 연결되지 않습니다.

```
from braket.aws import AwsDevice, DirectReservation
from braket.circuits import Circuit
from braket.devices import Devices

bell = Circuit().h(0).cnot(0, 1)
device = AwsDevice(Devices.IonQ.Aria1)

# run the circuit in a reservation
with DirectReservation(device, reservation_arn="<my_reservation_arn>"):
    task = device.run(bell, shots=100)
```

양자 태스크를 생성하는 동안 `DirectReservation` 컨텍스트가 활성 상태인 한 , CUDA-Q PennyLane및 Qiskit 플러그인을 사용하여 예약에서 양자 태스크를 생성할 수 있습니다. 예를 들어 Qiskit-Braket 공급자를 사용하면 다음과 같이 작업을 실행할 수 있습니다.

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

```
from qiskit import QuantumCircuit
from qiskit_braket_provider import BraketProvider


qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0, 1)

aria = BraketProvider().get_backend("Aria 1")

# run the circuit in a reservation
with DirectReservation(Devices.IonQ.Aria1, reservation_arn="<my_reservation_arn>"):
    aria_task = aria.run(qc, shots=10)
```

마찬가지로, 다음 코드는 예약 기간 동안 Braket-PennyLane 플러그인을 사용하여 회로를 실행합니다.

```
from braket.devices import Devices
from braket.aws import DirectReservation
import pennylane as qml


dev = qml.device("braket.aws.qubit", device_arn=Devices.IonQ.Aria1.value, wires=2,
 shots=10)

@qml.qnode(dev)
def bell_state():
    qml.Hadamard(wires=0)
    qml.CNOT(wires=[0, 1])
    return qml.probs(wires=[0, 1])

# run the circuit in a reservation
with DirectReservation(Devices.IonQ.Aria1, reservation_arn="<my_reservation_arn>"):
    probs = bell_state()
```

예약 컨텍스트 수동 설정

또는 다음 코드를 사용하여 예약 컨텍스트를 수동으로 설정할 수 있습니다.

```
# set reservation context
reservation_context = DirectReservation(device,
 reservation_arn="<my_reservation_arn>").start()

# run circuit during reservation
```

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

이는 컨텍스트를 첫 번째 셀에서 실행할 수 있고 모든 후속 작업이 예약에서 실행되는 Jupyter Notebook에 적합합니다.

> **Note**
>
> `.start()` 직접 호출이 포함된 셀은 한 번만 실행해야 합니다.

온디맨드 모드로 다시 전환하는 방법: Jupyter Notebook을 다시 시작하거나 다음을 직접적으로 호출하여 컨텍스트를 온디맨드 모드로 다시 변경합니다.

```
reservation_context.stop()  # unset reservation context
```

> **Note**
>
> 예약에는 미리 결정된 시작 및 종료 시간이 있습니다(예약 생성 참조). `reservation_context.start()` 및 `reservation_context.stop()` 메서드는 예약을 시작하거나 종료하지 않습니다. 대신 컨텍스트가 활성화되어 있는 동안 생성한 양자 작업은 예약과 연결되며 예약된 예약 중에만 실행됩니다. 예약 컨텍스트는 예약된 예약 시간에 영향을 미치지 않습니다.

작업 생성 시 예약 ARN 명시적으로 전달

예약 기간 동안 작업을 생성하는 또 다른 방법은 `device.run()`를 직접적으로 호출할 때 예약 ARN을 명시적으로 전달하는 것입니다.

```
task = device.run(bell, shots=100, reservation_arn="<my_reservation_arn>")
```

이 메서드는 양자 작업을 예약 ARN과 직접 연결하여 예약 기간 동안 실행되도록 합니다. 이 옵션의 경우 예약 기간 동안 실행하려는 각 작업에 예약 ARN을 추가합니다. 그러나 Qiskit 또는와 같은 타사 라이브러리를 사용하는 경우 제출된 작업이 올바른 예약 ARN을 사용하고 있는지 확인하기 어려울 PennyLane수 있습니다. 따라서 DirectReservation 컨텍스트 관리자를 사용하는 것이 좋습니다.

boto3을 직접 사용하는 경우 작업을 생성할 때 예약 ARN을 연결로 전달합니다.

```
import boto3

braket_client = boto3.client("braket")


kwargs["associations"] = [
    {
        "arn": "<my_reservation_arn>",
        "type": "RESERVATION_TIME_WINDOW_ARN",
    }
]

response = braket_client.create_quantum_task(**kwargs)
```

예약 기간 동안 하이브리드 작업 실행

하이브리드 작업으로 실행할 Python 함수가 있으면 `reservation_arn` 키워드 인수를 전달하여 예약에서 하이브리드 작업을 실행할 수 있습니다. 하이브리드 작업 내의 모든 작업은 예약 ARN을 사용합니다. 중요한 점은, `reservation_arn`이 있는 하이브리드 작업은 예약이 시작된 후에만 고전적 컴퓨팅을 가동한다는 것입니다.

> **Note**
>
> 예약 기간 동안 실행되는 하이브리드 작업은 예약된 디바이스에서만 양자 작업을 성공적으로 실행합니다. 다른 온디맨드 Braket 디바이스를 사용하려고 하면 오류가 발생합니다. 동일한 하이브리드 작업 내에서 온디맨드 시뮬레이터와 예약된 디바이스 모두에서 작업을 실행해야 하는 경우 대신 `DirectReservation`을 사용합니다.

다음 코드는 예약 기간 동안 하이브리드 작업을 실행하는 방법을 보여줍니다.

```
from braket.aws import AwsDevice
from braket.devices import Devices
from braket.jobs import get_job_device_arn, hybrid_job

@hybrid_job(device=Devices.IonQ.Aria1, reservation_arn="<my_reservation_arn>")
def example_hybrid_job():
    # declare AwsDevice within the hybrid job
    device = AwsDevice(get_job_device_arn())
    bell = Circuit().h(0).cnot(0, 1)
```

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

Python 스크립트를 사용하는 하이브리드 작업의 경우(개발자 안내서의 Creating your first Hybrid Job 섹션 참조) 작업을 생성할 때 `reservation_arn` 키워드 인수를 전달하여 예약 내에서 실행할 수 있습니다.

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

job = AwsQuantumJob.create(
    Devices.IonQ.Aria1,
    source_module="algorithm_script.py",
    entry_point="algorithm_script:start_here",
    reservation_arn="<my_reservation_arn>"
)
```

예약 종료 시 발생하는 일

예약이 종료된 후에는 더 이상 디바이스에 대한 전용 액세스 권한이 부여되지 않습니다. 이 예약과 함께 대기 중인 나머지 워크로드는 자동으로 취소됩니다.

> **Note**
>
> 예약이 종료되면 `RUNNING` 상태였던 모든 작업이 취소됩니다. 편리한 시간에 체크포인트를 사용하여 작업을 저장하고 다시 시작하는 것이 좋습니다.

각 예약은 독립 실행형 전용 디바이스 액세스를 의미하므로, 예약 시작 후 및 예약 종료 전과 같은 진행 중인 예약은 연장할 수 없습니다. 예를 들어 두 개의 back-to-back 예약은 별도의 예약으로 간주되며 첫 번째 예약에서 보류 중인 모든 작업은 자동으로 취소됩니다. 해당 작업은 두 번째 예약에서 재개되지 않습니다.

> **Note**
>
> 예약은 AWS 계정의 전용 디바이스 액세스를 나타냅니다. 디바이스가 유휴 상태로 유지되더라도 다른 고객은 디바이스를 사용할 수 없습니다. 따라서 실제 사용 시간에 관계없이 예약 시간 전체에 대해 요금이 부과됩니다.

기존 예약 취소 또는 일정 조정

예정된 예약 시작 시간 최소 48시간 전에 예약을 취소할 수 있습니다. 취소하려면 취소 요청과 함께 받은 예약 확인 이메일에 회신합니다.

일정을 조정하려면 기존 예약을 취소한 다음 새 예약을 생성해야 합니다.

오류 완화 기법

양자 오류 완화는 양자 컴퓨터에서 오류의 영향을 줄이기 위한 일련의 기법입니다.

양자 디바이스는 수행되는 계산의 품질을 저하시키는 환경 노이즈의 영향을 받습니다. 내결함성 양자 컴퓨팅이 이 문제에 대한 솔루션을 제시할 것으로 기대되지만, 현재 양자 디바이스는 큐비트 수와 상대적으로 높은 오류율에 의해 제한을 받고 있습니다. 단기적으로 이를 해결하기 위해 연구원들은 노이즈가 있는 양자 계산의 정확도를 개선하기 위한 방법을 조사하고 있습니다. 양자 오류 완화라고 하는 이 접근 방식에는 다양한 기법을 사용하여 노이즈가 있는 측정 데이터에서 최상의 신호를 추출하는 것이 포함됩니다.

이 섹션의 내용:

- IonQ 디바이스에서의 오류 완화 기법

IonQ 디바이스에서의 오류 완화 기법

오류 완화에는 여러 물리적 회로를 실행하고 측정값을 결합하여 결과를 개선하는 작업이 포함됩니다.

> **Note**
>
> 모든 IonQ 디바이스: 온디맨드 모델을 사용하는 경우 게이트샷 제한은 1백만 개이고 오류 완화 작업을 위한 최소 샷 수는 2,500개입니다. 직접 예약의 경우 게이트샷 제한이 없으며 오류 완화 작업을 위한 최소 샷 수는 500개입니다.

편향 제거

IonQ 디바이스에는 편향 제거라고 하는 오류 완화 방법이 있습니다.

편향 제거는 다양한 큐비트 순열 또는 다양한 게이트 분해에서 작동하는 여러 변형으로 회로를 매핑합니다. 이를 통해 측정 결과에 편향을 일으킬 수 있는 회로의 다양한 구현을 사용하여 게이트 과도 회전

또는 단일 결함 큐비트와 같은 체계적인 오류의 영향을 줄일 수 있습니다. 이는 여러 큐비트와 게이트를 보정하는 데 추가적인 오버헤드가 발생한다는 단점이 있습니다.

편향 제거에 대한 자세한 내용은 대칭화를 통한 양자 컴퓨터 성능 향상을 참조하세요.

> **Note**
>
> 편향 제거를 사용하려면 최소 2,500개의 샷이 필요합니다.

다음 코드를 사용하여 IonQ 디바이스에서 편향 제거를 사용하여 양자 작업을 실행할 수 있습니다.

```
from braket.aws import AwsDevice
from braket.circuits import Circuit
from braket.error_mitigation import Debias

# choose an IonQ device
device = AwsDevice("arn:aws:braket:us-east-1::device/qpu/ionq/Forte-Enterprise-1")
circuit = Circuit().h(0).cnot(0, 1)

task = device.run(circuit, shots=2500, device_parameters={"errorMitigation": Debias()})

result = task.result()
print(result.measurement_counts)
>>> {"00": 1245, "01": 5, "10": 10 "11": 1240} # result from debiasing
```

양자 작업이 완료되면 양자 작업의 측정 확률과 결과 유형을 볼 수 있습니다. 모든 변형의 측정 확률과 수는 단일 분포로 집계됩니다. 예상 값과 같이 회로에 지정된 모든 결과 유형은 집계 측정 수를 사용하여 계산됩니다.

선명화

또한 선명화라고 하는 다른 사후 처리 전략으로 계산된 측정 확률에 액세스할 수 있습니다. 선명화는 각 변형의 결과를 비교하고 일관되지 않은 샷을 폐기하며, 변형 전반에서 가장 가능성이 높은 측정 결과를 우선시합니다. 자세한 내용은 대칭화를 통한 양자 컴퓨터 성능 향상을 참조하세요.

중요한 점은, 선명화는 출력 분포의 형태를 희소하게 가정한다는 것입니다. 즉, 고확률 상태가 거의 없으며 제로 확률 상태가 많습니다. 이 가정이 유효하지 않으면 확률 분포가 왜곡될 수 있습니다.

Braket Python SDK의 `GateModelTaskResult`에 있는 `additional_metadata` 필드의 선명화된 분포에서 확률에 액세스할 수 있습니다. 단, 선명화는 측정 수를 반환하지 않고 대신 다시 정규화된 확률 분포를 반환합니다. 다음 코드 조각은 선명화 후 분포에 액세스하는 방법을 보여줍니다.

```
print(result.additional_metadata.ionqMetadata.sharpenedProbabilities)
>>> {"00": 0.51, "11": 0.549} # sharpened probabilities
```

Amazon Braket Hybrid Jobs 작업

Amazon Braket Hybrid Jobs는 클래식 AWS 리소스와 양자 처리 장치(QPUs)가 모두 필요한 하이브리드 양자 클래식 알고리즘을 실행할 수 있는 방법을 제공합니다. 하이브리드 작업은 요청된 고전적 리소스를 구동하고 알고리즘을 실행하며 완료 후 인스턴스를 해제하도록 설계되었으므로 사용한 만큼만 비용을 지불하면 됩니다.

하이브리드 작업은 고전 컴퓨팅 리소스와 양자 컴퓨팅 리소스를 모두 사용하는 장기 실행 반복 알고리즘에 적합합니다. 하이브리드 작업에서는 실행할 알고리즘을 제출하고 나면, Braket이 확장 가능하고 컨테이너화된 환경에서 해당 알고리즘을 실행합니다. 알고리즘이 완료되면 결과를 검색할 수 있습니다.

또한 하이브리드 작업에서 생성된 양자 작업은 대상 QPU 디바이스에 대한 대기열에서 더 높은 우선순위로 지정되는 이점을 누릴 수 있습니다. 이러한 우선순위 지정 덕분에 대기열에 있는 다른 작업들보다 양자 계산이 먼저 처리되고 실행됩니다. 이는 하나의 양자 작업의 결과가 이전 양자 작업의 결과에 따라 달라지는 반복 하이브리드 알고리즘에 특히 유용합니다. 이러한 알고리즘의 예로는 양자 근사 최적화 알고리즘(Quantum Approximate Optimization Algorithm, QAOA), 변분 양자 고유해석기 또는 양자 기계 학습이 있습니다. 또한 알고리즘 진행 상황을 거의 실시간으로 모니터링하여 비용, 예산 또는 훈련 손실이나 기대값과 같은 사용자 지정 지표를 추적할 수 있습니다.

다음을 사용하여 Braket에서 하이브리드 작업에 액세스할 수 있습니다.

- Amazon Braket Python SDK
- Amazon Braket 콘솔
- Amazon Braket API

이 섹션의 내용:

- Amazon Braket에서 CUDA-Q 사용

Amazon Braket Hybrid Jobs를 사용해야 하는 시점

Amazon Braket Hybrid Jobs를 사용하면 변분 양자 고유해석기(Variational Quantum Eigensolver, VQE) 및 양자 근사 최적화 알고리즘(Quantum Approximate Optimization Algorithm, QAOA)과 같은 하이브리드 양자-고전 알고리즘을 실행하여 고전 컴퓨팅 리소스를 양자 컴퓨팅 디바이스와 결합해 오늘날 양자 시스템의 성능을 최적화할 수 있습니다. Amazon Braket Hybrid Jobs는 다음과 같은 세 가지 주요 이점을 제공합니다.

1. 성능: Amazon Braket Hybrid Jobs는 자체 환경에서 하이브리드 알고리즘을 실행하는 것보다 더 나은 성능을 제공합니다. 작업이 실행되는 동안 선택한 대상 QPU에 우선적으로 액세스할 수 있습니다. 따라서 해당 양자 작업의 작업이 디바이스에 대기 중인 다른 작업보다 먼저 실행됩니다. 따라서 하이브리드 알고리즘의 런타임이 단축되고 예측 가능성은 더 높아집니다. Amazon Braket Hybrid Jobs는 파라메트릭 컴파일도 지원합니다. 자유 파라미터를 사용하여 회로를 제출할 수 있으며 Braket은 동일한 회로에 대한 후속 파라미터 업데이트를 위해 다시 컴파일할 필요 없이 회로를 한 번 컴파일하므로 런타임이 훨씬 빨라집니다.
2. 편의성: Amazon Braket Hybrid Jobs는 컴퓨팅 환경 설정 및 관리를 간소화하고 하이브리드 알고리즘이 실행되는 동안 환경을 계속 가동 상태로 유지합니다. 알고리즘 스크립트를 제공하고 실행할 양자 디바이스(양자 처리 장치 또는 시뮬레이터)를 선택하면 됩니다. Amazon Braket은 대상 디바이스가 사용 가능 상태가 될 때까지 대기하고 고전 리소스를 구동하며 사전 빌드된 컨테이너 환경에서 워크로드를 실행하고 결과를 Amazon Simple Storage Service(Amazon S3)에 반환한 다음 컴퓨팅 리소스를 해제합니다.
3. 지표: Amazon Braket Hybrid Jobs는 실행 중인 알고리즘에 대한 즉각적인 인사이트를 제공하고 사용자 지정 가능한 알고리즘 지표를 거의 실시간으로 Amazon CloudWatch 및 Amazon Braket 콘솔에 전달하여 알고리즘의 진행 상황을 추적할 수 있게 합니다.

Amazon Braket Hybrid Jobs를 사용한 하이브리드 작업 실행

Amazon Braket Hybrid Jobs로 하이브리드 작업을 실행하려면 먼저 알고리즘을 정의해야 합니다. Amazon Braket Python SDK 또는 PennyLane을 사용하여 알고리즘 스크립트와 기타 종속성 파일(선택 사항)을 작성하여 정의할 수 있습니다. 다른 (오픈 소스 또는 독점) 라이브러리를 사용하려면 이러한 라이브러리가 포함된 Docker를 사용하여 자체 사용자 지정 컨테이너 이미지를 정의할 수 있습니다. 자세한 내용은 자체 컨테이너 사용(BYOC)을 참조하세요.

어느 경우든, 다음으로 Amazon Braket API를 사용하여 하이브리드 작업을 생성합니다. 여기서 알고리즘 스크립트 또는 컨테이너를 제공하고 하이브리드 작업이 사용할 대상 양자 디바이스를 선택한 다음 다양한 옵션 설정 중에서 선택합니다. 이러한 옵션 설정에 제공된 기본값은 대부분의 사용 사례에 적합합니다. 하이브리드 작업을 실행할 대상 디바이스로 QPU, 온디맨드 시뮬레이터(예: SV1, DM1 또는 TN1) 또는 고전적인 하이브리드 작업 인스턴스 자체 중에서 선택할 수 있습니다. 온디맨드 시뮬레이터 또는 QPU를 사용하면 하이브리드 작업 컨테이너가 원격 디바이스에 대한 API 직접 호출을 수행합니다. 임베디드 시뮬레이터를 사용하면, 시뮬레이터가 알고리즘 스크립트와 동일한 컨테이너에 임베디드됩니다. PennyLane의 라이트닝 시뮬레이터에는 사용자를 위해 사전 빌드된 기본 하이브리드 작업 컨테이너가 임베디드되어 있습니다. 임베디드 PennyLane 시뮬레이터 또는 사용자 지정 시뮬레이터를 사용하여 코드를 실행하는 경우 인스턴스 유형과 사용하려는 인스턴스 수를 지정할 수 있습니다. 각 선택과 관련된 비용은 Amazon Braket 요금 페이지를 참조하세요.

대상 디바이스가 온디맨드 또는 임베디드 시뮬레이터인 경우 Amazon Braket은 하이브리드 작업을 즉시 실행하기 시작합니다. 하이브리드 작업 인스턴스를 구동하고(API 직접 호출 시 인스턴스 유형을 사용자 지정할 수 있음) 알고리즘을 실행하며 Amazon S3에 결과를 쓰고 리소스를 해제합니다. 이러한 리소스 해제 방식 덕분에 사용한 만큼만 비용을 지불하면 됩니다.

QPU당 동시 하이브리드 작업의 총 수는 제한됩니다. 현재 QPU에서는 한 번에 하나의 하이브리드 작업만 실행할 수 있습니다. 대기열은 허용된 제한을 초과하지 않도록 실행이 허용되는 하이브리드 작업의 수를 제어하는 데 사용됩니다. 대상 디바이스가 QPU인 경우 하이브리드 작업은 먼저 선택한 QPU의 작업 대기열에 들어갑니다. Amazon Braket은 필요한 하이브리드 작업 인스턴스를 구동하고 디바이스에서 하이브리드 작업을 실행합니다. 알고리즘 실행 기간 동안에는 하이브리드 작업이 우선적으로 처리됩니다. 즉, 하이브리드 작업의 양자 작업이 디바이스에 대기 중인 다른 Braket 양자 작업보다 먼

저 실행됩니다. 단, 하이브리드 작업의 양자 작업은 몇 분 간격으로 QPU에 제출되어야 합니다. 하이브리드 작업이 완료되면 리소스가 해제되므로, 사용한 만큼만 비용을 지불하면 됩니다.

> **Note**
>
> 디바이스는 리전별이며 하이브리드 작업은 기본 디바이스 AWS 리전 와 동일한에서 실행됩니다.

시뮬레이터 및 QPU 대상 시나리오 모두에서 알고리즘의 일부로 해밀토니안의 에너지와 같은 사용자 지정 알고리즘 지표를 정의할 수 있는 옵션이 있습니다. 이러한 지표는 Amazon CloudWatch에 자동으로 보고되며 거기서 Amazon Braket 콘솔에 거의 실시간으로 표시됩니다.

> **Note**
>
> GPU 기반 인스턴스를 사용하려면 Braket의 임베디드 시뮬레이터와 함께 사용할 수 있는 GPU 기반 시뮬레이터 중 하나를 사용해야 합니다(예: `lightning.gpu`). CPU 기반 임베디드 시뮬레이터(예: `lightning.qubit`또는 `braket:default-simulator`) 중 하나를 선택하면 GPU가 사용되지 않으므로 불필요한 비용이 발생할 수 있습니다.

하이브리드 작업의 주요 개념

이 섹션에서는 Amazon Braket Python SDK에서 제공하는 `AwsQuantumJob.create` 함수의 주요 개념과 컨테이너 파일 구조에 대한 매핑을 설명합니다.

전체 알고리즘 스크립트를 구성하는 파일 외에도, 하이브리드 작업에는 추가 입력과 출력이 있을 수 있습니다. 하이브리드 작업이 시작되면 Amazon Braket은 하이브리드 작업 생성의 일부로 제공된 입력을 알고리즘 스크립트를 실행하는 컨테이너에 복사합니다. 하이브리드 작업이 완료되면 알고리즘 중에 정의된 모든 출력이 지정된 Amazon S3 위치로 복사됩니다.

> **Note**
>
> 알고리즘 지표는 실시간으로 보고되며 이 출력 절차를 따르지 않습니다.

또한 Amazon Braket은 컨테이너 입력 및 출력과의 상호 작용을 간소화하는 여러 환경 변수 및 헬퍼 함수를 제공합니다. 자세한 내용은 Amazon Braket SDK의 braket.jobs 패키지를 참조하세요.

이 섹션의 내용:

입력

입력 데이터: 입력 데이터는 딕셔너리로 설정된 입력 데이터 파일을 `input_data` 인수와 함께 지정하여 하이브리드 알고리즘에 제공할 수 있습니다. 사용자는 SDK의 `AwsQuantumJob.create` 함수 내에서 `input_data` 인수를 정의합니다. 그러면 환경 변수 `"AMZN_BRAKET_INPUT_DIR"`에서 지정한 위치의 컨테이너 파일 시스템에 입력 데이터가 복사됩니다. 하이브리드 알고리즘에서 입력 데이터가 사용되는 방법에 대한 몇 가지 예제는 Amazon Braket Hybrid Jobs와 PennyLane을 활용한 QAOA 및 Amazon Braket Hybrid Jobs에서의 양자 기계 학습 Jupyter notebooks를 참조하세요.

> **Note**
>
> 입력 데이터가 크면(>1GB) 하이브리드 작업이 제출되기까지 대기 시간이 길어집니다. 이는 로컬 입력 데이터가 먼저 S3 버킷에 업로드된 다음 S3 경로가 하이브리드 작업 요청에 추가되고, 마지막으로 하이브리드 작업 요청이 Braket 서비스에 제출되기 때문입니다.

하이퍼파라미터: `hyperparameters`를 전달하면 환경 변수 `"AMZN_BRAKET_HP_FILE"`에서 사용할 수 있습니다.

> **Note**
>
> 하이퍼파라미터 및 입력 데이터를 생성한 다음 이 정보를 하이브리드 작업 스크립트에 전달하는 방법에 대한 자세한 내용은 하이퍼파라미터 사용 섹션 및 이 github 페이지를 참조하세요.

체크포인트: 새 하이브리드 작업에서 사용할 체크포인트를 갖는 `job-arn`을 지정하려면 `copy_checkpoints_from_job` 명령을 사용합니다. 이 명령은 체크포인트 데이터를 새 하이브리드 작업의 `checkpoint_configs3Uri`으로 복사하여 작업이 실행되는 동안 환경 변수 `AMZN_BRAKET_CHECKPOINT_DIR`이 지정한 경로에서 사용할 수 있게 합니다. 기본값은 `None`입니다. 즉, 다른 하이브리드 작업의 체크포인트 데이터는 새 하이브리드 작업에 사용되지 않습니다.

출력

양자 작업: 양자 작업 결과는 S3 위치 `s3://amazon-braket-<region>-<accountID>/jobs/<job-name>/tasks`에 저장됩니다.

작업 결과: 알고리즘 스크립트가 환경 변수 "AMZN_BRAKET_JOB_RESULTS_DIR"에 의해 지정된 디렉터리에 저장하는 모든 내용은 `output_data_config`에 지정된 S3 위치로 복사됩니다. 값이 지정되지 않은 경우 기본값은 `s3://amazon-braket-<region>-<accountID>/jobs/<job-name>/<timestamp>/data`입니다. 알고리즘 스크립트에서 직접적으로 호출할 때 결과를 딕셔너리 형태로 편리하게 저장하는 데 사용할 수 있는 SDK 헬퍼 함수 **`save_job_result`**를 제공합니다.

체크포인트: 체크포인트를 사용하려면 환경 변수 "AMZN_BRAKET_CHECKPOINT_DIR"에 의해 지정된 디렉터리에 체해당 크포인트를 저장할 수 있습니다. SDK 헬퍼 함수 `save_job_checkpoint`를 대신 사용할 수도 있습니다.

알고리즘 지표: 하이브리드 작업이 실행되는 동안 Amazon CloudWatch로 내보내지고 Amazon Braket 콘솔에 실시간으로 표시되는 알고리즘 스크립트의 일부로 알고리즘 지표를 정의할 수 있습니다. 알고리즘 지표를 사용하는 방법의 예제는 Amazon Braket Hybrid Jobs를 사용하여 QAOA 알고리즘 실행을 참조하세요.

작업 출력 저장에 대한 자세한 내용은 하이브리드 작업 설명서의 결과 저장을 참조하세요.

환경 변수

Amazon Braket은 컨테이너 입력 및 출력과의 상호 작용을 간소화하는 여러 환경 변수를 제공합니다. 아래 코드는 Braket이 사용하는 환경 변수를 나열합니다.

- `AMZN_BRAKET_INPUT_DIR` - 입력 데이터 디렉터리 opt/braket/input/data
- `AMZN_BRAKET_JOB_RESULTS_DIR` - 작업 결과를 쓸 출력 디렉터리 opt/braket/model
- `AMZN_BRAKET_JOB_NAME` - 작업의 이름
- `AMZN_BRAKET_CHECKPOINT_DIR` - 체크포인트 디렉터리
- `AMZN_BRAKET_HP_FILE` - 하이퍼파라미터가 포함된 파일
- `AMZN_BRAKET_DEVICE_ARN` - 디바이스 ARN(AWS 리소스 이름).
- `AMZN_BRAKET_OUT_S3_BUCKET` - `CreateJob` 요청의 `OutputDataConfig`에 지정된 출력 Amazon S3 버킷
- `AMZN_BRAKET_SCRIPT_ENTRY_POINT` - `CreateJob` 요청의 `ScriptModeConfig`에 지정된 진입점

- AMZN_BRAKET_SCRIPT_COMPRESSION_TYPE - CreateJob 요청의 ScriptModeConfig에 지정된 압축 유형
- AMZN_BRAKET_SCRIPT_S3_URI - CreateJob 요청의 ScriptModeConfig에 지정된 사용자 스크립트의 Amazon S3 위치
- AMZN_BRAKET_TASK_RESULTS_S3_URI - SDK가 기본적으로 작업에 대한 양자 작업 결과를 저장하는 Amazon S3 위치
- AMZN_BRAKET_JOB_RESULTS_S3_PATH - CreateJob 요청의 OutputDataConfig에 지정된 대로 작업 결과가 저장될 Amazon S3 위치
- AMZN_BRAKET_JOB_TOKEN - 작업 컨테이너에 생성된 양자 작업에 대해 CreateQuantumTask의 jobToken 파라미터에 전달되어야 하는 문자열

헬퍼 함수

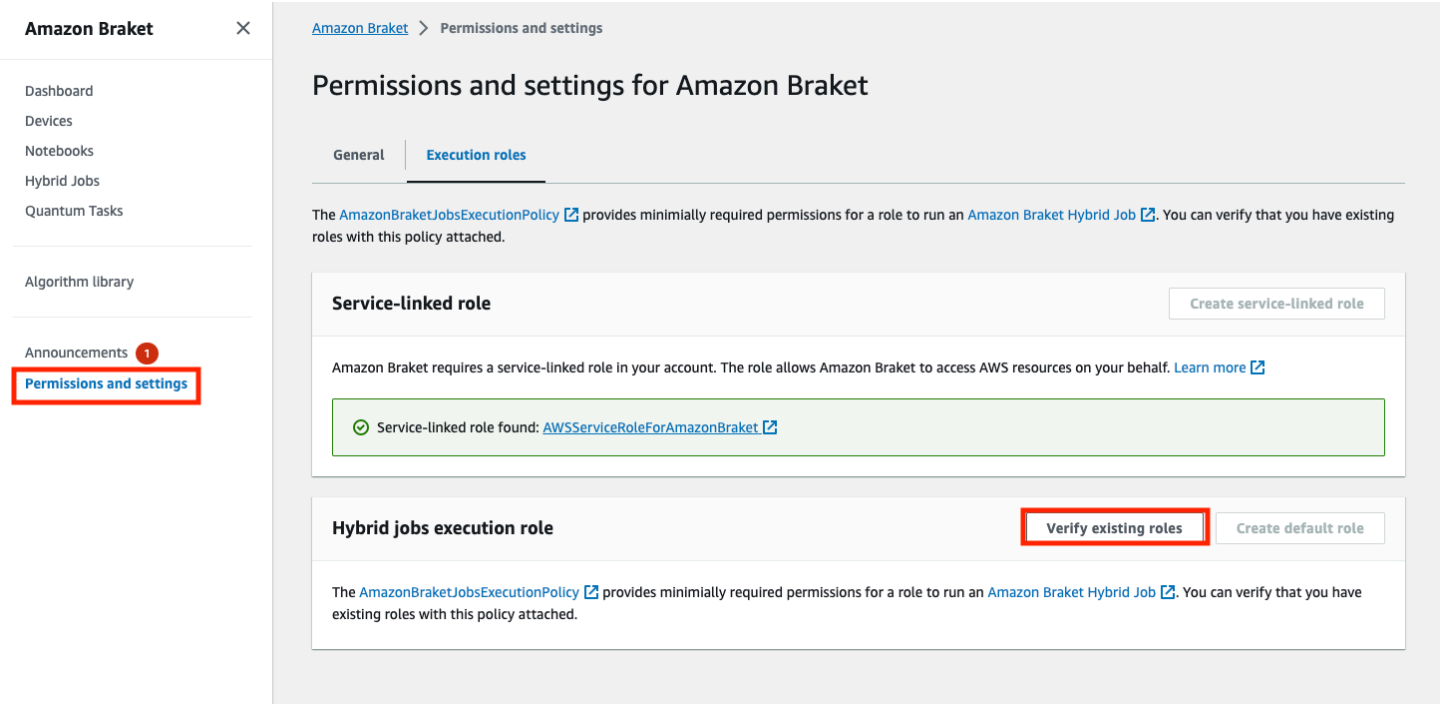
Amazon Braket은 컨테이너 입력 및 출력과의 상호 작용을 간소화하는 여러 헬퍼 함수를 제공합니다. 이러한 헬퍼 함수는 하이브리드 작업을 실행하는 데 사용되는 알고리즘 스크립트 내에서 직접적으로 호출됩니다. 다음 예제에서는 이러한 함수를 사용하는 방법을 보여줍니다.

```
from braket.jobs import get_checkpoint_dir, get_hyperparameters, get_input_data_dir,
 get_job_device_arn, get_job_name, get_results_dir, save_job_result,
 save_job_checkpoint, load_job_checkpoint

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

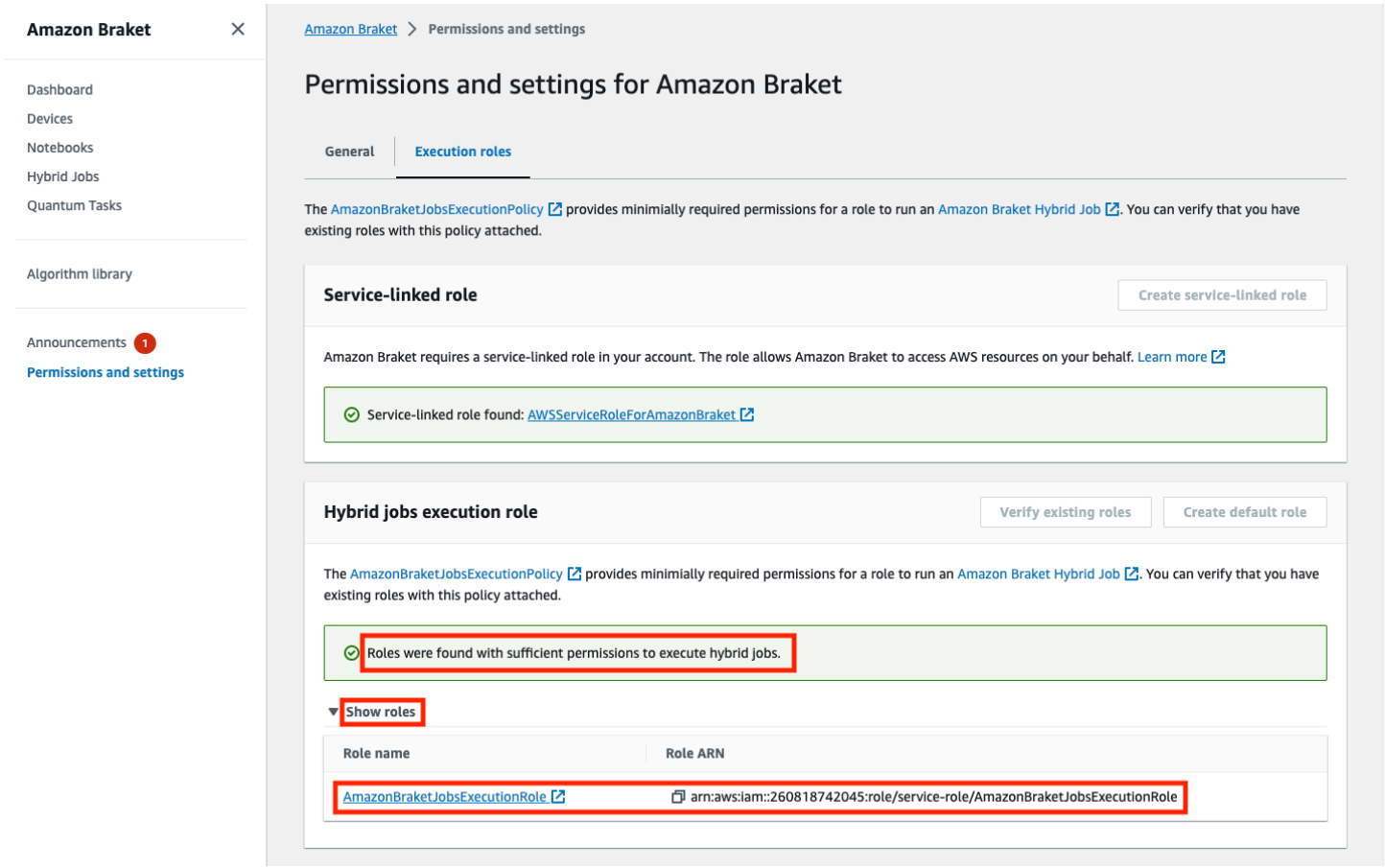
사전 조건

첫 번째 하이브리드 작업을 실행하기 전에, 이 작업을 진행할 수 있는 충분한 권한이 있는지 확인해야 합니다. 올바른 권한이 있는지 확인하려면 Braket 콘솔 왼쪽의 메뉴에서 권한을 선택합니다. Amazon Braket에 대한 권한 관리 페이지는 기존 역할 중 하나에 하이브리드 작업을 실행하기 충분한 권한이

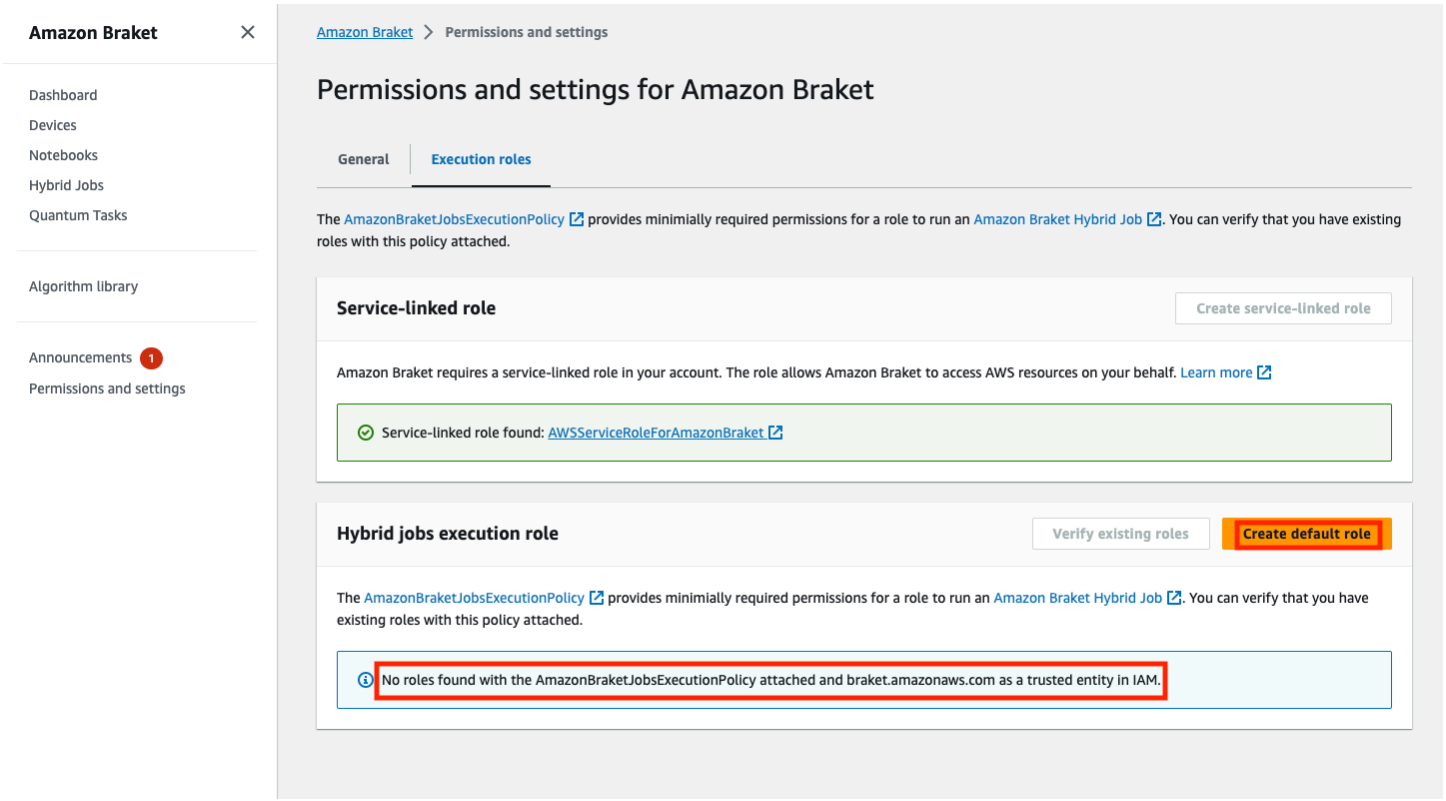
있는지 확인하는 데 도움을 주거나, 해당 역할이 아직 없는 경우 하이브리드 작업을 실행하는 데 사용할 수 있는 기본 역할 생성 과정을 안내합니다.



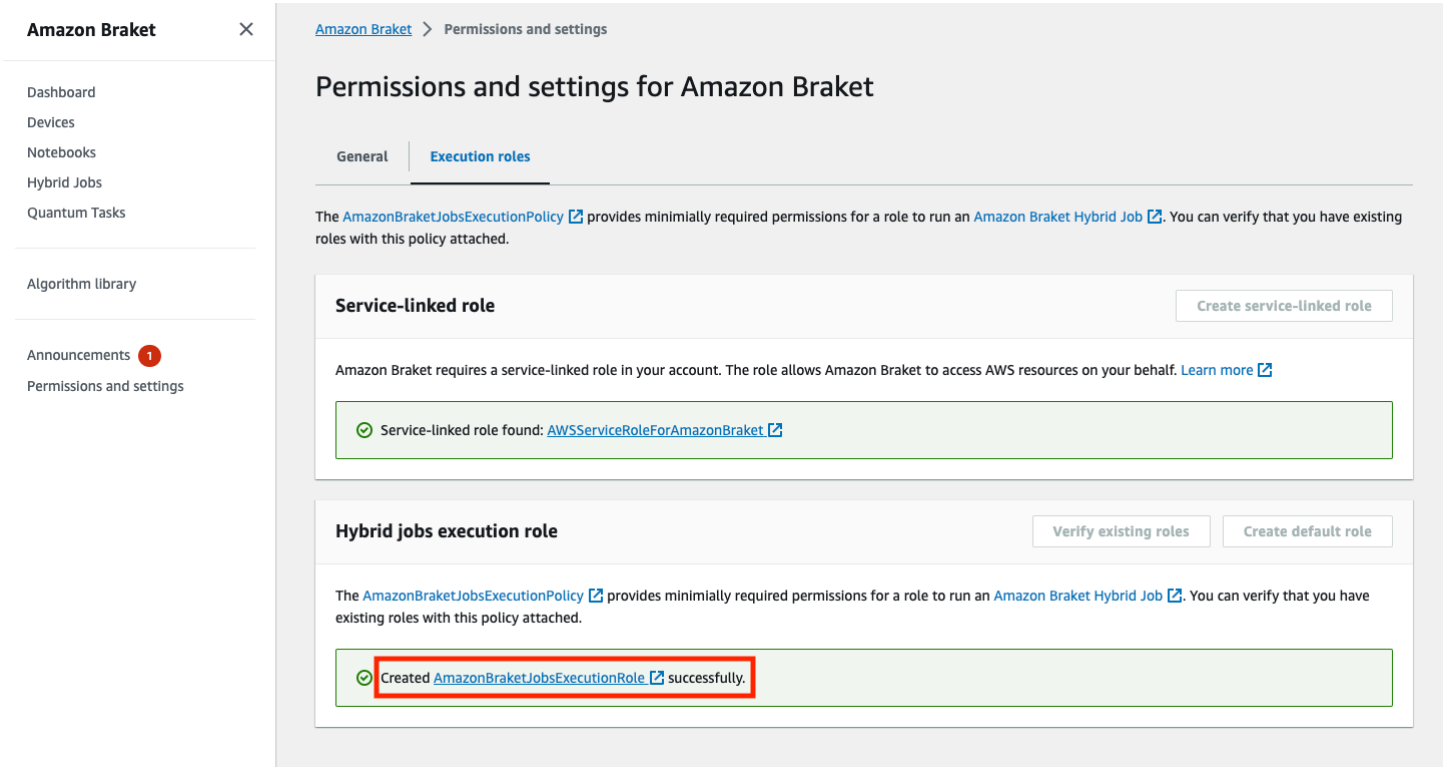
하이브리드 작업을 실행할 수 있는 충분한 권한을 가진 역할이 있는지 확인하려면 기존 역할 확인 버튼을 선택합니다. 이렇게 하면 역할을 찾았다는 메시지가 표시됩니다. 역할의 이름과 역할 ARN을 보려면 역할 표시 버튼을 선택합니다.



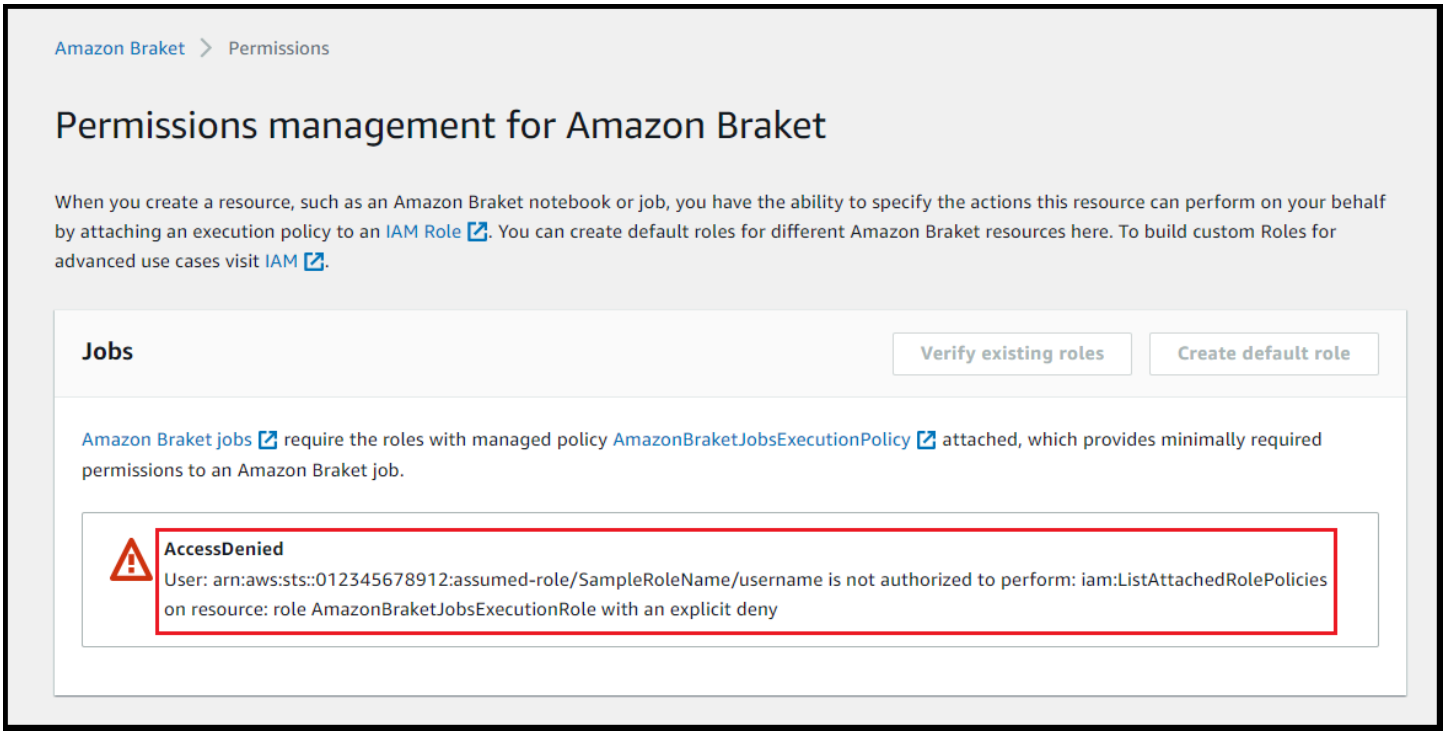
하이브리드 작업을 실행할 수 있는 충분한 권한이 있는 역할이 없는 경우 해당 역할을 찾을 수 없다는 메시지가 표시됩니다. 충분한 권한이 있는 역할을 얻으려면 기본 역할 생성 버튼을 선택합니다.



역할이 성공적으로 생성되면 이를 확인하는 메시지가 표시됩니다.



이 문의를 수행할 권한이 없는 경우 액세스가 거부됩니다. 이 경우 내부 AWS 관리자에게 문의하세요.



하이브리드 작업 생성

이 섹션에서는 Python 스크립트를 사용하여 하이브리드 작업을 생성하는 방법을 보여줍니다. 또는 선호하는 통합 개발 환경(IDE)이나 Braket 노트북 같은 로컬 Python 코드에서 하이브리드 작업을 생성하려면 로컬 코드를 하이브리드 작업으로 실행을 참조하세요.

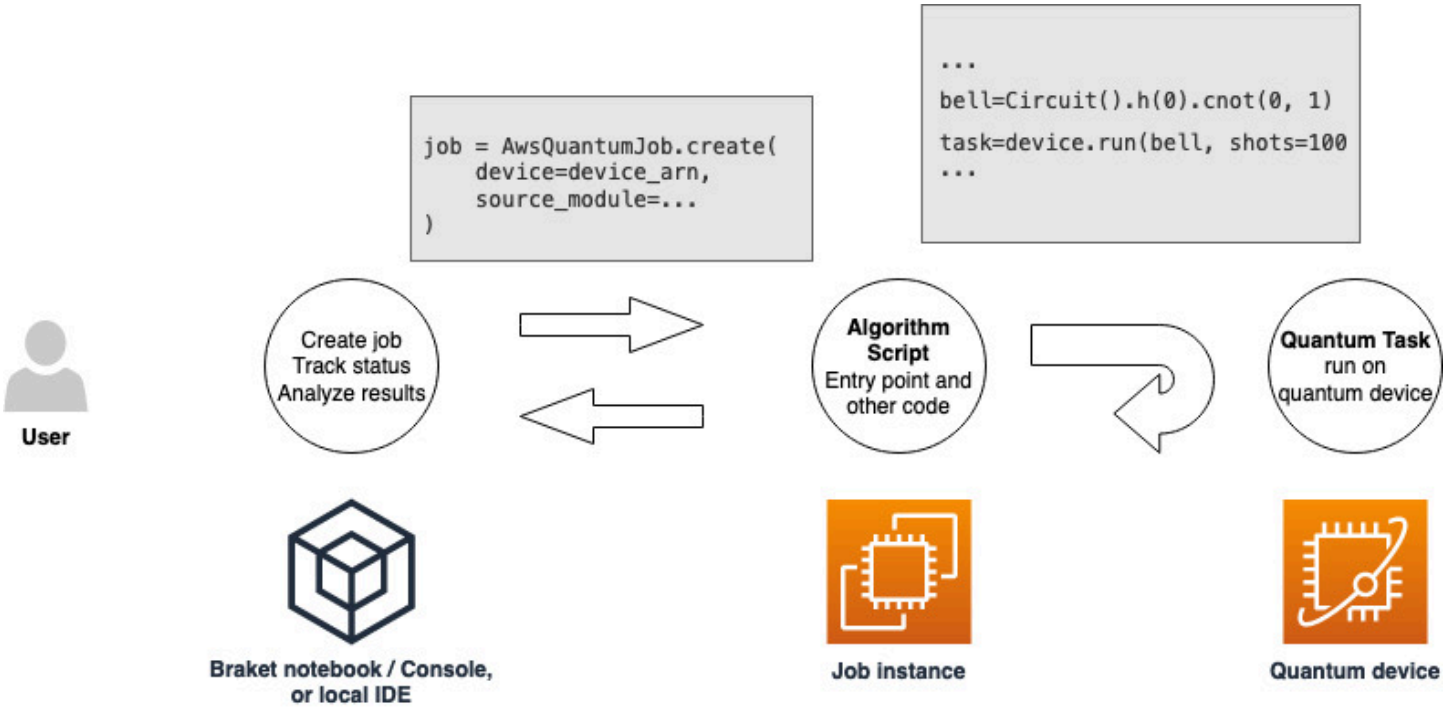
이 섹션의 내용:

- 생성 및 실행
- 결과 모니터링
- 결과 저장
- 체크포인트 사용
- 로컬 코드를 하이브리드 작업으로 실행
- 하이브리드 작업에 API 사용
- 로컬 모드로 하이브리드 작업 생성 및 디버깅

생성 및 실행

하이브리드 작업을 실행할 권한이 있는 역할이 있으면 바로 진행할 수 있습니다. 첫 번째 Braket 하이브리드 작업의 핵심은 알고리즘 스크립트입니다. 실행하려는 알고리즘을 정의하고 알고리즘의 일부인

고전 논리 및 양자 작업을 포함합니다. 알고리즘 스크립트 외에도 다른 종속성 파일을 제공할 수 있습니다. 알고리즘 스크립트와 해당 종속성을 합쳐서 소스 모듈이라고 합니다. 진입점은 하이브리드 작업이 시작될 때 소스 모듈에서 실행할 첫 번째 파일 또는 함수를 정의합니다.



먼저 5개의 벨 상태를 생성하고 해당 측정 결과를 출력하는 알고리즘 스크립트의 다음 기본 예제를 살펴보세요.

```
import os

from braket.aws import AwsDevice
from braket.circuits import Circuit


def start_here():

    print("Test job started!")

    # Use the device declared in the job script
    device = AwsDevice(os.environ["AMZN_BRAKET_DEVICE_ARN"])

    bell = Circuit().h(0).cnot(0, 1)
    for count in range(5):
        task = device.run(bell, shots=100)
        print(task.result().measurement_counts)
```

```
    print("Test job completed!")
```

이 파일을 Braket 노트북 또는 로컬 환경의 현재 작업 디렉터리에 algorithm_script.py라는 이름으로 저장합니다. algorithm_script.py 파일은 `start_here()`를 계획된 진입점으로 사용합니다.

그런 다음 algorithm_script.py 파일과 동일한 디렉터리에 Python 파일 또는 Python 노트북을 생성합니다. 이 스크립트는 하이브리드 작업을 시작하고 상태나 관심 대상 주요 결과 출력과 같은 모든 비동기 처리를 해결합니다. 최소한 이 스크립트는 하이브리드 작업 스크립트와 기본 디바이스를 지정해야 합니다.

> **Note**
>
> 노트북과 동일한 디렉터리에 Braket 노트북을 생성하거나 algorithm_script.py 파일과 같은 파일을 업로드하는 방법에 대한 자세한 내용은 Amazon Braket Python SDK를 사용하여 첫 번째 회로 실행을 참조하세요.

이 기본적인 첫 번째 사례의 경우 시뮬레이터를 대상으로 합니다. 대상 양자 디바이스 유형, 시뮬레이터 또는 실제 QPU에 관계없이 다음 스크립트에서 `device`로 지정하는 디바이스는 하이브리드 작업을 예약하는 데 사용되며 알고리즘 스크립트에서 환경 변수 `AMZN_BRAKET_DEVICE_ARN`으로 사용할 수 있습니다.

> **Note**
>
> 하이브리드 작업 AWS 리전 의에서 사용할 수 있는 디바이스만 사용할 수 있습니다. Amazon Braket SDK는 이 AWS 리전을 자동으로 선택합니다. 예를 들어 us-east-1의 하이브리드 작업은 IonQ, SV1, DM1 및 TN1 디바이스를 사용할 수 있지만 Rigetti 디바이스는 사용할 수 없습니다.

시뮬레이터 대신 양자 컴퓨터를 선택하면, Braket은 하이브리드 작업을 예약하여 모든 양자 작업을 우선적으로 실행하도록 합니다.

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

job = AwsQuantumJob.create(
    Devices.Amazon.SV1,
    source_module="algorithm_script.py",
    entry_point="algorithm_script:start_here",
```

```
    wait_until_complete=True
)
```

파라미터 `wait_until_complete=True`는 작업이 실행 중인 실제 작업의 출력을 인쇄하도록 자세한 정보 표시 모드를 설정합니다. 다음 예제와 유사한 출력이 표시됩니다.

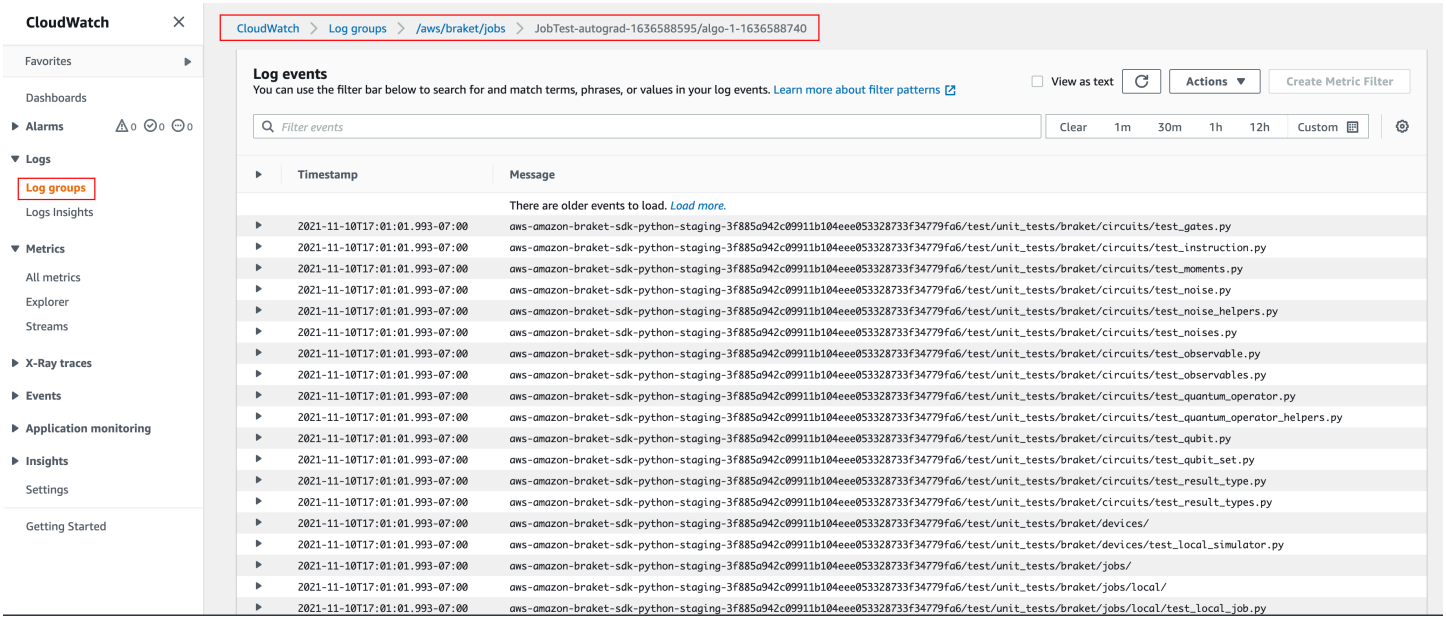
```
Initializing Braket Job: arn:aws:braket:us-west-2:111122223333:job/braket-job-
default-123456789012
Job queue position: 1
Job queue position: 1
Job queue position: 1
..............
.
.
.
Beginning Setup
Checking for Additional Requirements
Additional Requirements Check Finished
Running Code As Process
Test job started!
Counter({'00': 58, '11': 42})
Counter({'00': 55, '11': 45})
Counter({'11': 51, '00': 49})
Counter({'00': 56, '11': 44})
Counter({'11': 56, '00': 44})
Test job completed!
Code Run Finished
2025-09-24 23:13:40,962 sagemaker-training-toolkit INFO     Reporting training SUCCESS
```

> **Note**
>
> 위치(로컬 디렉터리 또는 파일의 경로, 또는 tar.gz 파일의 S3 URI)를 전달하여 AwsQuantumJob.create 메서드와 함께 사용자 지정 모듈을 사용할 수도 있습니다. 작업 예제는 Amazon Braket 예제 Github 리포지토리의 하이브리드 작업 폴더에서 Parallelize_training_for_QML.ipynb 파일을 참조하세요.

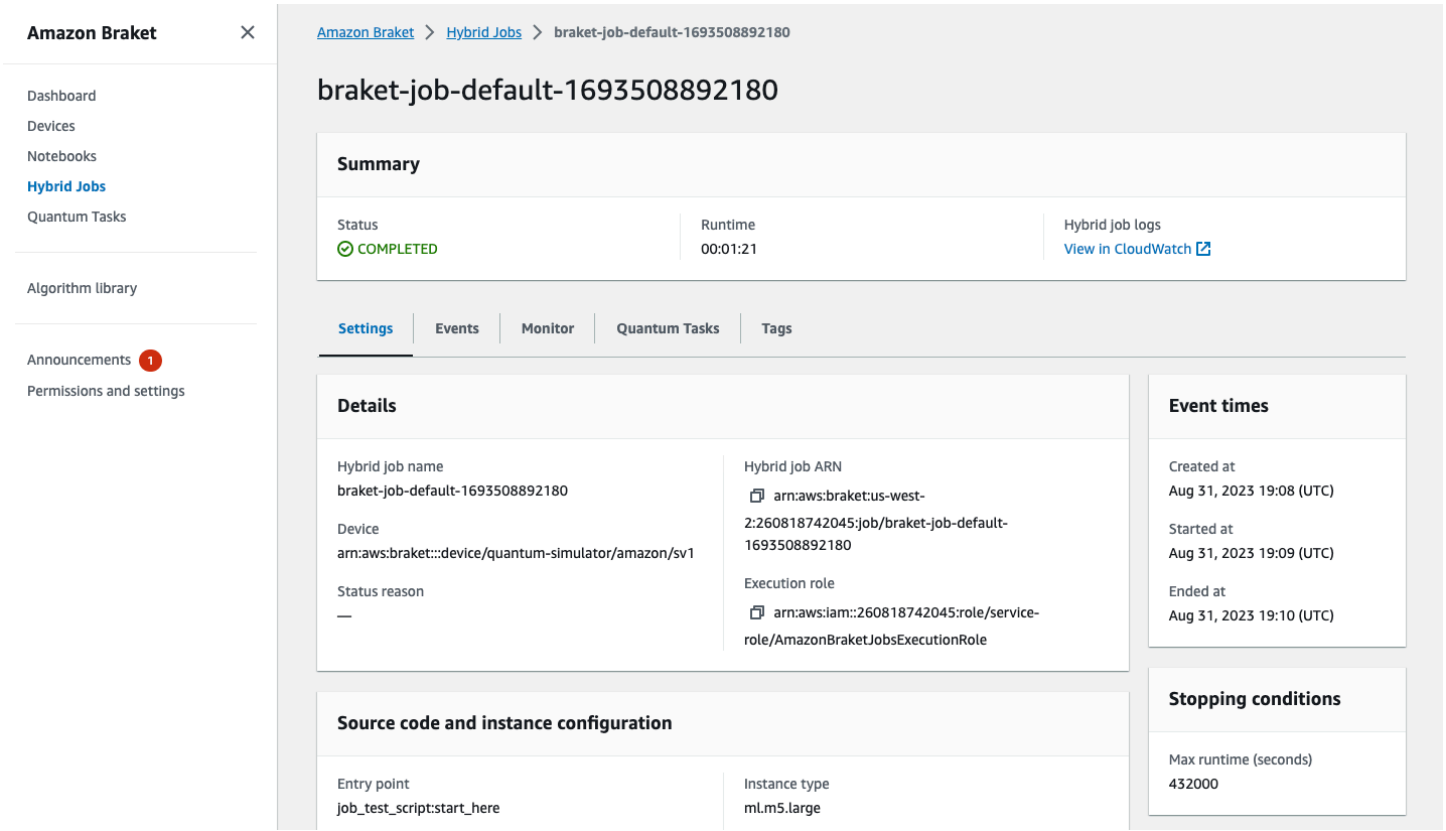
결과 모니터링

또는, Amazon CloudWatch의 로그 출력에 액세스할 수 있습니다. 이렇게 하려면 작업 세부 정보 페이지의 왼쪽 메뉴에 있는 로그 그룹 탭으로 이동하여 로그 그룹 `aws/braket/jobs`를

선택한 다음, 작업 이름이 포함된 로그 스트림을 선택합니다. 위 예제에서는 `braket-job-default-1631915042705/algo-1-1631915190`입니다.



하이브리드 작업 페이지를 선택한 다음 설정을 선택하여 콘솔에서 하이브리드 작업의 상태를 볼 수도 있습니다.



하이브리드 작업은 실행되는 동안 Amazon S3에서 일부 아티팩트를 생성합니다. 기본 S3 버킷 이름은 amazon-braket-<region>-<accountid>이고 콘텐츠는 jobs/<jobname>/<timestamp> 디렉터리에 있습니다. Braket Python SDK를 사용하여 하이브리드 작업을 생성할 때 다른 code_location을 지정하여 이러한 아티팩트가 저장되는 S3 위치를 구성할 수 있습니다.

> **Note**
>
> 이 S3 버킷은 작업 스크립트 AWS 리전 와 동일한에 있어야 합니다.

jobs/<jobname>/<timestamp> 디렉터리에는 model.tar.gz 파일에 담긴 진입점 스크립트의 출력이 있는 하위 폴더가 포함되어 있습니다. 또한 source.tar.gz 파일에 알고리즘 스크립트 아티팩트가 포함된 script라고 하는 디렉터리도 있습니다. 실제 양자 작업의 결과는 jobs/<jobname>/tasks라는 디렉터리에 있습니다.

결과 저장

하이브리드 작업 스크립트의 하이브리드 작업 객체와 Amazon S3의 출력 폴더(model.tar.gz라는 이름의 tar-zipped 파일)에서 사용할 수 있도록 알고리즘 스크립트에서 생성된 결과를 저장할 수 있습니다.

출력은 JavaScript Object Notation(JSON) 형식을 사용하여 파일에 저장해야 합니다. 넘파이 배열의 경우처럼 데이터가 텍스트로 쉽게 직렬화될 수 없는 경우, 피클링된 데이터 형식을 사용하여 직렬화하는 옵션을 전달할 수 있습니다. 자세한 내용은 braket.jobs.data_persistence 모듈을 참조하세요.

하이브리드 작업의 결과를 저장하려면 #ADD로 주석이 달린 다음 줄을 algorithm_script.py 파일에 추가합니다.

```
import os

from braket.aws import AwsDevice
from braket.circuits import Circuit
from braket.jobs import save_job_result  # ADD


def start_here():

    print("Test job started!")

    device = AwsDevice(os.environ['AMZN_BRAKET_DEVICE_ARN'])
```

```
    results = []  # ADD

    bell = Circuit().h(0).cnot(0, 1)
    for count in range(5):
        task = device.run(bell, shots=100)
        print(task.result().measurement_counts)
        results.append(task.result().measurement_counts)  # ADD

        save_job_result({"measurement_counts": results})  # ADD

    print("Test job completed!")
```

그런 다음 #ADD로 주석이 달린 줄 **print(job.result())**를 추가하여 작업 스크립트의 작업 결과를 표시할 수 있습니다.

```
import time
from braket.aws import AwsQuantumJob

job = AwsQuantumJob.create(
    source_module="algorithm_script.py",
    entry_point="algorithm_script:start_here",
    device="arn:aws:braket:::device/quantum-simulator/amazon/sv1",
)

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

print(job.state())
print(job.result())   # ADD
```

이 예제에서는 자세한 정보 출력을 억제하기 위해 wait_until_complete=True를 제거했습니다. 디버깅을 위해 다시 추가할 수 있습니다. 이 하이브리드 작업을 실행하면 식별자와 job-arn을 출력한 다음 하이브리드 작업이 COMPLETED 상태가 될 때까지 10초마다 하이브리드 작업의 상태를 출력한 후에 벨 회로의 결과가 보여줍니다. 다음 예제를 참조하세요.

```
arn:aws:braket:us-west-2:111122223333:job/braket-job-default-123456789012
INITIALIZED
RUNNING
RUNNING
RUNNING
```

```
RUNNING
RUNNING
RUNNING
RUNNING
RUNNING
RUNNING
RUNNING
...
RUNNING
RUNNING
COMPLETED
{'measurement_counts': [{'11': 53, '00': 47},..., {'00': 51, '11': 49}]}
```

체크포인트 사용

체크포인트를 사용하여 하이브리드 작업의 중간 반복을 저장할 수 있습니다. 이전 섹션의 알고리즘 스크립트 예제에서 체크포인트 파일을 생성하려면 #ADD로 주석이 달린 다음 줄을 추가하면 됩니다.

```
from braket.aws import AwsDevice
from braket.circuits import Circuit
from braket.jobs import save_job_checkpoint  # ADD
import os


def start_here():

    print("Test job starts!")

    device = AwsDevice(os.environ["AMZN_BRAKET_DEVICE_ARN"])

    # ADD the following code
    job_name = os.environ["AMZN_BRAKET_JOB_NAME"]
    save_job_checkpoint(checkpoint_data={"data": f"data for checkpoint from
 {job_name}"}, checkpoint_file_suffix="checkpoint-1")  # End of ADD

    bell = Circuit().h(0).cnot(0, 1)
    for count in range(5):
        task = device.run(bell, shots=100)
        print(task.result().measurement_counts)

    print("Test hybrid job completed!")
```

하이브리드 작업을 실행하면 기본 /opt/jobs/checkpoints 경로의 체크포인트 디렉터리 내 하이브리드 작업 아티팩트에 <jobname>-checkpoint-1.json 파일이 생성됩니다. 이 기본 경로를 변경하지 않는 한 하이브리드 작업 스크립트는 변경되지 않습니다.

이전 하이브리드 작업에서 생성된 체크포인트에서 하이브리드 작업을 로드하려는 경우 알고리즘 스크립트는 `from braket.jobs import load_job_checkpoint`를 사용합니다. 알고리즘 스크립트에 로드할 논리는 다음과 같습니다.

```
from braket.jobs import load_job_checkpoint

checkpoint_1 = load_job_checkpoint(
    "previous_job_name",
    checkpoint_file_suffix="checkpoint-1",
)
```

이 체크포인트를 로드한 후 checkpoint-1에 로드된 콘텐츠를 기반으로 논리를 계속 진행할 수 있습니다.

> **Note**
>
> checkpoint_file_suffix는 체크포인트 생성 시 이전에 지정한 접미사와 일치해야 합니다.

오케스트레이션 스크립트는 #ADD로 주석이 달린 줄로 이전 하이브리드 작업의 job-arn을 지정해야 합니다.

```
from braket.aws import AwsQuantumJob

job = AwsQuantumJob.create(
    source_module="source_dir",
    entry_point="source_dir.algorithm_script:start_here",
    device="arn:aws:braket:::device/quantum-simulator/amazon/sv1",
    copy_checkpoints_from_job="<previous-job-ARN>", #ADD
    )
```

로컬 코드를 하이브리드 작업으로 실행

Amazon Braket Hybrid Jobs는 Amazon EC2 컴퓨팅 리소스와 Amazon Braket Quantum Processing Unit(QPU) 액세스를 결합하여 하이브리드 양자-고전 알고리즘의 완전관리형 오케스트레이션을 제공합니다. 하이브리드 작업에서 생성된 양자 작업은 개별 양자 작업보다 대기열에서 우선순위를 가지므

로, 양자 작업 대기열의 변동으로 인해 알고리즘이 중단되지 않습니다. 각 QPU는 별도의 하이브리드 작업 대기열을 유지하여 한 번에 하나의 하이브리드 작업만 실행되도록 보장합니다.

이 섹션의 내용:

- 로컬 Python 코드에서 하이브리드 작업 생성
- 추가 Python 패키지 및 소스 코드 설치
- 하이브리드 작업 인스턴스에 데이터 저장 및 로드
- 하이브리드 작업 데코레이터 모범 사례

로컬 Python 코드에서 하이브리드 작업 생성

로컬 Python 코드를 Amazon Braket Hybrid Jobs로 실행할 수 있습니다. 다음 코드 예제와 같이 `@hybrid_job` 데코레이터로 코드에 주석을 달아 이 작업을 수행할 수 있습니다. 사용자 지정 환경의 경우 Amazon Elastic Container Registry(ECR)의 사용자 지정 컨테이너를 사용하도록 선택할 수 있습니다.

> **Note**
>
> Python 3.12만 기본적으로 지원됩니다.

`@hybrid_job` 데코레이터를 사용하여 함수에 주석을 달 수 있습니다. Braket은 데코레이터 내부의 코드를 Braket 하이브리드 작업 알고리즘 스크립트로 변환합니다. 그런 다음 하이브리드 작업은 Amazon EC2 인스턴스의 데코레이터 내에서 함수를 간적접으로 호출합니다. `job.state()` 또는 Braket 콘솔을 통해 작업 진행 상황을 모니터링할 수 있습니다. 다음 코드 예제는 State Vector Simulator (SV1) device에서 다섯 개의 상태 시퀀스를 실행하는 방법을 보여줍니다.

```
from braket.aws import AwsDevice
from braket.circuits import Circuit, FreeParameter, Observable
from braket.devices import Devices
from braket.jobs.hybrid_job import hybrid_job
from braket.jobs.metrics import log_metric

device_arn = Devices.Amazon.SV1


@hybrid_job(device=device_arn)  # Choose priority device
def run_hybrid_job(num_tasks=1):
    device = AwsDevice(device_arn)  # Declare AwsDevice within the hybrid job
```

```
    # Create a parametric circuit
    circ = Circuit()
    circ.rx(0, FreeParameter("theta"))
    circ.cnot(0, 1)
    circ.expectation(observable=Observable.X(), target=0)

    theta = 0.0  # Initial parameter

    for i in range(num_tasks):
        task = device.run(circ, shots=100, inputs={"theta": theta})  # Input parameters
        exp_val = task.result().values[0]

        theta += exp_val  # Modify the parameter (possibly gradient descent)

        log_metric(metric_name="exp_val", value=exp_val, iteration_number=i)

    return {"final_theta": theta, "final_exp_val": exp_val}
```

일반적인 Python 함수와 마찬가지로 함수를 간접적으로 호출하여 하이브리드 작업을 생성합니다. 그러나 데코레이터 함수는 함수의 결과가 아닌 하이브리드 작업 핸들을 반환합니다. 완료 후 결과를 검색하려면 `job.result()`를 사용합니다.

```
job = run_hybrid_job(num_tasks=1)
result = job.result()
```

`@hybrid_job` 데코레이터의 디바이스 인수는 하이브리드 작업이 우선적으로 액세스할 수 있는 디바이스를 지정합니다. 이 경우 SV1 시뮬레이터입니다. QPU 우선순위를 얻으려면 함수 내에서 사용되는 디바이스 ARN이 데코레이터에 지정된 것과 일치하는지 확인해야 합니다. 편의를 위해 헬퍼 함수 `get_job_device_arn()`을 사용하여 `@hybrid_job`에 선언된 디바이스 ARN을 캡처할 수 있습니다.

> **Note**
>
> 각 하이브리드 작업은 Amazon EC2에서 컨테이너화된 환경을 생성하기 때문에 시작 시간이 최소 1분입니다. 따라서 단일 회로 또는 여러 회로의 배치와 같이 매우 짧은 워크로드의 경우 양자 작업을 사용하기에 충분할 수 있습니다.

하이퍼파라미터

run_hybrid_job() 함수는 인수 num_tasks를 사용하여 생성된 양자 작업 수를 제어합니다. 하이브리드 작업은 이를 하이퍼파라미터로 자동으로 캡처합니다.

> **Note**
>
> 하이퍼파라미터는 Braket 콘솔에 문자열로 표시되며, 문자열은 2,500자로 제한됩니다.

지표 및 로깅

run_hybrid_job() 함수 내에서 반복 알고리즘의 지표는 log_metrics를 사용하여 기록됩니다. 지표는 하이브리드 작업 탭 아래의 Braket 콘솔 페이지에 자동으로 플롯됩니다. Braket 비용 추적기를 사용하면 하이브리드 작업 실행 중에 지표를 사용하여 거의 실시간으로 양자 작업 비용을 추적할 수 있습니다. 위 예제에서는 결과 유형의 첫 번째 확률을 기록하는 지표 이름 “probability”를 사용합니다.

결과 검색

하이브리드 작업이 완료되면 job.result()를 사용하여 하이브리드 작업 결과를 검색합니다. 반환 문의 모든 객체는 Braket에 의해 자동으로 캡처됩니다. 함수에서 반환하는 객체는 각 요소가 직렬화 가능한 튜플이어야 합니다. 예를 들어, 다음 코드는 작동하는 예제와 실패하는 예제를 보여줍니다.

```
import numpy as np


# Working example
@hybrid_job(device=Devices.Amazon.SV1)
def passing():
    np_array = np.random.rand(5)
    return np_array  # Serializable

# # Failing example
# @hybrid_job(device=Devices.Amazon.SV1)
# def failing():
#     return MyObject() # Not serializable
```

작업 이름

기본적으로 이 하이브리드 작업의 이름은 함수 이름에서 추론됩니다. 최대 50자까지 사용자 지정 이름을 지정할 수도 있습니다. 예를 들어, 다음 코드에서 작업 이름은 "my-job-name"입니다.

```
@hybrid_job(device=Devices.Amazon.SV1, job_name="my-job-name")
def function():
```

```
    pass
```

로컬 모드

로컬 작업은 데코레이터에 인수 `local=True`를 추가하여 생성됩니다. 그러면 랩톱과 같은 로컬 컴퓨팅 환경의 컨테이너화된 환경에서 하이브리드 작업이 실행됩니다. 로컬 작업에는 양자 작업에 대한 우선순위 대기열이 없습니다. 다중 노드 또는 MPI와 같은 고급 사례의 경우 로컬 작업이 필요한 Braket 환경 변수에 액세스할 수 있습니다. 다음 코드는 디바이스를 SV1 시뮬레이터로 사용하여 로컬 하이브리드 작업을 생성합니다.

```
@hybrid_job(device=Devices.Amazon.SV1, local=True)
def run_hybrid_job(num_tasks=1):
    return ...
```

다른 모든 하이브리드 작업 옵션도 지원됩니다. 옵션 목록은 braket.jobs.quantum_job_creation 모듈을 참조하세요.

추가 Python 패키지 및 소스 코드 설치

원하는 Python 패키지를 사용하도록 런타임 환경을 사용자 지정할 수 있습니다. `requirements.txt` 파일, 패키지 이름 목록을 사용하거나 자체 컨테이너를 사용할 수 있습니다(BYOC). 예를 들어 `requirements.txt` 파일에는 설치할 다른 패키지가 포함될 수 있습니다.

```
qiskit
pennylane >= 0.31
mitiq == 0.29
```

`requirements.txt` 파일을 사용하여 런타임 환경을 사용자 지정하려면 다음 코드 예제를 참조하세요.

```
@hybrid_job(device=Devices.Amazon.SV1, dependencies="requirements.txt")
def run_hybrid_job(num_tasks=1):
    return ...
```

또는 다음과 같이 패키지 이름을 Python 목록으로 제공할 수 있습니다.

```
@hybrid_job(device=Devices.Amazon.SV1, dependencies=["qiskit", "pennylane>=0.31",
 "mitiq==0.29"])
def run_hybrid_job(num_tasks=1):
    return ...
```

추가 소스 코드는 다음 코드 예제와 같이 모듈 목록 또는 단일 모듈로 지정할 수 있습니다.

```
@hybrid_job(device=Devices.Amazon.SV1, include_modules=["my_module1", "my_module2"])
def run_hybrid_job(num_tasks=1):
    return ...
```

하이브리드 작업 인스턴스에 데이터 저장 및 로드

입력 훈련 데이터 지정

하이브리드 작업을 생성할 때 Amazon Simple Storage Service(Amazon S3) 버킷을 지정하여 입력 훈련 데이터세트를 제공할 수 있습니다. 로컬 경로를 지정할 수도 있으며, 그러면 Braket이 자동으로 데이터를 `s3://<default_bucket_name>/jobs/<job_name>/<timestamp>/data/<channel_name>`의 Amazon S3에 업로드합니다. 로컬 경로를 지정하는 경우 채널 이름은 기본적으로 "input"으로 설정됩니다. 다음 코드는 로컬 경로 `data/file.npy`의 넘파이 파일을 보여줍니다.

```
import numpy as np


@hybrid_job(device=Devices.Amazon.SV1, input_data="data/file.npy")
def run_hybrid_job(num_tasks=1):
    data = np.load("data/file.npy")
    return ...
```

S3의 경우 `get_input_data_dir()` 헬퍼 함수를 사용해야 합니다.

```
import numpy as np
from braket.jobs import get_input_data_dir

s3_path = "s3://amazon-braket-us-east-1-123456789012/job-data/file.npy"


@hybrid_job(device=None, input_data=s3_path)
def job_s3_input():
    np.load(get_input_data_dir() + "/file.npy")


@hybrid_job(device=None, input_data={"channel": s3_path})
def job_s3_input_channel():
    np.load(get_input_data_dir("channel") + "/file.npy")
```

채널 값 및 S3 URI 또는 로컬 경로의 딕셔너리를 제공하여 여러 입력 데이터 소스를 지정할 수 있습니다.

```
import numpy as np
from braket.jobs import get_input_data_dir

input_data = {
    "input": "data/file.npy",
    "input_2": "s3://amzn-s3-demo-bucket/data.json"
}


@hybrid_job(device=None, input_data=input_data)
def multiple_input_job():
    np.load(get_input_data_dir("input") + "/file.npy")
    np.load(get_input_data_dir("input_2") + "/data.json")
```

> **Note**
>
> 입력 데이터가 크면(>1GB) 작업이 생성되기까지 대기 시간이 길어집니다. 이는 로컬 입력 데이터가 S3 버킷에 처음 업로드된 후 S3 경로가 작업 요청에 추가되기 때문입니다. 마지막으로, 작업 요청이 Braket 서비스에 제출됩니다.

S3에 결과 저장

데코레이션된 함수의 반환 문에 포함되지 않은 결과를 저장하려면 모든 파일 쓰기 작업에 올바른 디렉터리를 추가해야 합니다. 다음 예제에서는 넘파이 배열과 matplotlib 그림을 저장하는 방법을 보여줍니다.

```
import matplotlib.pyplot as plt
import numpy as np


@hybrid_job(device=Devices.Amazon.SV1)
def run_hybrid_job(num_tasks=1):
    result = np.random.rand(5)

    # Save a numpy array
    np.save("result.npy", result)
```

```
    # Save a matplotlib figure
    plt.plot(result)
    plt.savefig("fig.png")
    return ...
```

모든 결과는 model.tar.gz라는 파일로 압축됩니다. Python 함수 job.result()를 사용하거나 Braket 관리 콘솔의 하이브리드 작업 페이지에서 결과 폴더로 이동하여 결과를 다운로드할 수 있습니다.

체크포인트에서 저장 및 재개

장기 실행 하이브리드 작업의 경우 알고리즘의 중간 상태를 주기적으로 저장하는 것이 좋습니다. 기본 제공 save_job_checkpoint() 헬퍼 함수를 사용하거나 파일을 AMZN_BRAKET_JOB_RESULTS_DIR 경로에 저장할 수 있습니다. 후자의 경우 헬퍼 함수 get_job_results_dir()과 함께 사용할 수 있습니다.

다음은 하이브리드 작업 데코레이터를 사용하여 체크포인트를 저장하고 로드하는 최소 작업 예제입니다.

```
from braket.jobs import save_job_checkpoint, load_job_checkpoint, hybrid_job


@hybrid_job(device=None, wait_until_complete=True)
def function():
    save_job_checkpoint({"a": 1})


job = function()
job_name = job.name
job_arn = job.arn


@hybrid_job(device=None, wait_until_complete=True, copy_checkpoints_from_job=job_arn)
def continued_function():
    load_job_checkpoint(job_name)


continued_job = continued_function()
```

첫 번째 하이브리드 작업에서는 save_job_checkpoint()가 저장하려는 데이터가 포함된 딕셔너리와 함께 직접적으로 호출됩니다. 기본적으로 모든 값은 텍스트로 직렬화 가능해야 합니다. 넘파이 배열과 같은 더 복잡한 Python 객체를 체크포인트하려면 data_format =

`PersistedJobDataFormat.PICKLED_V4`를 설정할 수 있습니다. 이 코드는 "checkpoints"라는 하위 폴더 아래의 하이브리드 작업 아티팩트에서 기본 이름 `<jobname>.json`으로 체크포인트 파일을 생성하고 덮어씁니다.

체크포인트에서 계속 진행하기 위해 새로운 하이브리드 작업을 생성하려면 `copy_checkpoints_from_job=job_arn`을 전달해야 합니다. 여기서 `job_arn`은 이전 작업의 하이브리드 작업 ARN입니다. 그런 다음 `load_job_checkpoint(job_name)`을 사용하여 체크포인트에서를 로드합니다.

하이브리드 작업 데코레이터 모범 사례

비동기성 수용

데코레이터 주석으로 생성된 하이브리드 작업은 비동기적입니다. 즉, 고전 리소스와 양자 리소스를 사용할 수 있게 되면 실행됩니다. Braket Management Console 또는 Amazon CloudWatch를 사용하여 알고리즘의 진행 상황을 모니터링합니다. 실행할 알고리즘을 제출하면 Braket은 확장 가능한 컨테이너화된 환경에서 해당 알고리즘을 실행하고 알고리즘이 완료되면 결과가 검색됩니다.

반복 변분 알고리즘 실행

하이브리드 작업은 반복적인 양자-고전 알고리즘을 실행할 수 있는 도구를 제공합니다. 순수하게 양자 문제인 경우 양자 작업 또는 양자 작업 배치를 사용합니다. 특정 QPU에 대한 우선 액세스는 중간에 고전적 처리를 수행하면서 QPU에 대한 여러 번의 반복 직접 호출을 필요로 하는 장기 실행 변분 알고리즘에 가장 유용합니다.

로컬 모드를 사용한 디버깅

QPU에서 하이브리드 작업을 실행하기 전에 먼저 시뮬레이터 SV1에서를 실행하여 예상대로 실행되는지 확인하는 것이 좋습니다. 소규모 테스트의 경우 빠른 반복 및 디버깅을 위해 로컬 모드로 실행할 수 있습니다.

BYOC(Bring Your Own Container)를 통한 재현성 향상

컨테이너화된 환경 내에서 소프트웨어와 해당 종속성을 캡슐화하여 재현 가능한 실험을 생성합니다. 컨테이너에 모든 코드, 종속성 및 설정을 패키징하면 잠재적인 충돌 및 버전 관리 문제를 방지할 수 있습니다.

다중 인스턴스 분산 시뮬레이터

많은 수의 회로를 실행하려면 기본 제공 MPI 지원을 사용하여 단일 하이브리드 작업 내의 여러 인스턴스에서 로컬 시뮬레이터를 실행하는 것이 좋습니다. 자세한 내용은 임베디드 시뮬레이터를 참조하세요.

파라메트릭 회로 사용

하이브리드 작업에서 제출하는 파라메트릭 회로는 파라메트릭 컴파일을 사용하여 특정 QPU에서 자동으로 컴파일되어 알고리즘의 런타임을 개선합니다.

주기적인 체크포인트

장기 실행 하이브리드 작업의 경우 알고리즘의 중간 상태를 주기적으로 저장하는 것이 좋습니다.

추가 예제, 사용 사례 및 모범 사례는 Amazon Braket 예제 GitHub를 참조하세요.

하이브리드 작업에 API 사용

API를 사용하여 Amazon Braket Hybrid Jobs에 직접 액세스하고 상호 작용할 수 있습니다. 그러나 API를 직접 사용할 때는 기본값 및 편의 메서드를 사용할 수 없습니다.

> **Note**
>
> Amazon Braket Python SDK를 사용하여 Amazon Braket Hybrid Jobs와 상호 작용하는 것이 좋습니다. 이 SDK는하이브리드 작업을 성공적으로 실행하는 데 도움이 되는 편의 기본값과 보호 기능을 제공합니다.

이 주제에서는 API 사용의 기본 사항을 다룹니다. API를 사용하기로 선택한 경우 이 접근 방식은 더 복잡할 수 있으며 하이브리드 작업이 실행되도록 하기 위해 여러 번의 반복에 대비해야 함을 염두에 두세요.

API를 사용하려면 계정에 `AmazonBraketFullAccess` 관리형 정책이 적용된 역할이 있어야 합니다.

> **Note**
>
> `AmazonBraketFullAccess` 관리형 정책이 적용된 역할을 얻는 방법에 대한 자세한 내용은 Amazon Braket 활성화 페이지를 참조하세요.

또한 실행 역할이 필요합니다. 이 역할은 서비스에 전달됩니다. Amazon Braket 콘솔을 사용하여 역할을 생성할 수 있습니다. 권한 및 설정 페이지의 실행 역할 탭을 사용하여 하이브리드 작업에 대한 기본 역할을 생성합니다.

`CreateJob` API에서는 하이브리드 작업에 필요한 모든 파라미터를 지정해야 합니다. Python을 사용하려면 알고리즘 스크립트 파일을 input.tar.gz 파일과 같은 tar 번들로 압축하고 다음 스크립트를 실행

합니다. 하이브리드 작업이 시작되는 경로, 파일 및 메서드를 지정하는 계정 정보 및 진입점과 일치하도록 꺾쇠 대괄호(<>) 안의 코드 부분을 업데이트합니다.

```
from braket.aws import AwsDevice, AwsSession
import boto3
from datetime import datetime

s3_client = boto3.client("s3")
client = boto3.client("braket")

project_name = "job-test"
job_name = project_name + "-" + datetime.strftime(datetime.now(), "%Y%m%d%H%M%S")
bucket = "amazon-braket-<your_bucket>"
s3_prefix = job_name

job_script = "input.tar.gz"
job_object = f"{s3_prefix}/script/{job_script}"
s3_client.upload_file(job_script, bucket, job_object)

input_data = "inputdata.csv"
input_object = f"{s3_prefix}/input/{input_data}"
s3_client.upload_file(input_data, bucket, input_object)

job = client.create_job(
    jobName=job_name,
    roleArn="arn:aws:iam::<your_account>:role/service-role/
AmazonBraketJobsExecutionRole",  # https://docs.aws.amazon.com/braket/latest/
developerguide/braket-manage-access.html#about-amazonbraketjobsexecution
    algorithmSpecification={
        "scriptModeConfig": {
            "entryPoint": "<your_execution_module>:<your_execution_method>",
            "containerImage": {"uri": "292282985366.dkr.ecr.us-west-1.amazonaws.com/
amazon-braket-base-jobs:1.0-cpu-py37-ubuntu18.04"},   # Change to the specific region
 you are using
            "s3Uri": f"s3://{bucket}/{job_object}",
            "compressionType": "GZIP"
        }
    },
    inputDataConfig=[
        {
            "channelName": "hellothere",
            "compressionType": "NONE",
            "dataSource": {
```

```
                "s3DataSource": {
                    "s3Uri": f"s3://{bucket}/{s3_prefix}/input",
                    "s3DataType": "S3_PREFIX"
                }
            }
        }
    ],
    outputDataConfig={
        "s3Path": f"s3://{bucket}/{s3_prefix}/output"
    },
    instanceConfig={
        "instanceType": "ml.m5.large",
        "instanceCount": 1,
        "volumeSizeInGb": 1
    },
    checkpointConfig={
        "s3Uri":  f"s3://{bucket}/{s3_prefix}/checkpoints",
        "localPath": "/opt/omega/checkpoints"
    },
    deviceConfig={
        "priorityAccess": {
            "devices": [
                "arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3"
            ]
        }
    },
    hyperParameters={
        "hyperparameter key you wish to pass": "<hyperparameter value you wish to
 pass>",
    },
    stoppingCondition={
        "maxRuntimeInSeconds": 1200,
        "maximumTaskLimit": 10
    },
)
```

하이브리드 작업을 생성한 후에는 GetJob API 또는 콘솔을 통해 하이브리드 작업 세부 정보에 액세스할 수 있습니다. 이전 예제와 같이 createJob 코드를 실행한 Python 세션에서 하이브리드 작업 세부 정보를 가져오려면 다음 Python 명령을 사용합니다.

```
getJob = client.get_job(jobArn=job["jobArn"])
```

하이브리드 작업을 취소하려면 작업('JobArn')의 Amazon Resource Name을 사용하여 `CancelJob` API를 직접적으로 호출합니다.

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

`checkpointConfig` 파라미터를 사용하여 체크포인트를 `createJob` API의 일부로 지정할 수 있습니다.

```
    checkpointConfig = {
        "localPath" : "/opt/omega/checkpoints",
        "s3Uri": f"s3://{bucket}/{s3_prefix}/checkpoints"
    },
```

> **Note**
>
> `checkpointConfig`의 localPath는 다음의 예약된 경로 `/opt/ml`, `/opt/braket`, `/tmp` 또는 `/usr/local/nvidia` 중 어느 것으로도 시작할 수 없습니다.

로컬 모드로 하이브리드 작업 생성 및 디버깅

새 하이브리드 알고리즘을 빌드할 때 로컬 모드를 사용하면 알고리즘 스크립트를 디버깅하고 테스트할 수 있습니다. 로컬 모드는 Amazon Braket Hybrid Jobs에서 사용하려는 코드를 실행할 수 있게 해주지만, 하이브리드 작업 실행을 위한 인프라 관리를 Braket에 의존하지 않아도 되는 기능입니다. 대신, Amazon Braket Notebook 인스턴스 또는 랩톱이나 데스크톱 컴퓨터와 같은 기본 클라이언트에서 로컬로 하이브리드 작업을 실행합니다.

로컬 모드에서는 여전히 양자 작업을 실제 디바이스로 전송할 수 있지만, 로컬 모드에서 실제 QPU를 대상으로 실행할 때 성능상의 이점을 얻을 수 없습니다.

로컬 모드를 사용하려면 프로그램 내 어디에서 발생하든, `LocalQuantumJob`을 `AwsQuantumJob`으로 수정합니다. 예를 들어 첫 번째 하이브리드 작업 생성의 예제를 실행하려면 다음과 같이 코드에서 하이브리드 작업 스크립트를 편집합니다.

```
from braket.jobs.local import LocalQuantumJob

job = LocalQuantumJob.create(
    device="arn:aws:braket:::device/quantum-simulator/amazon/sv1",
```

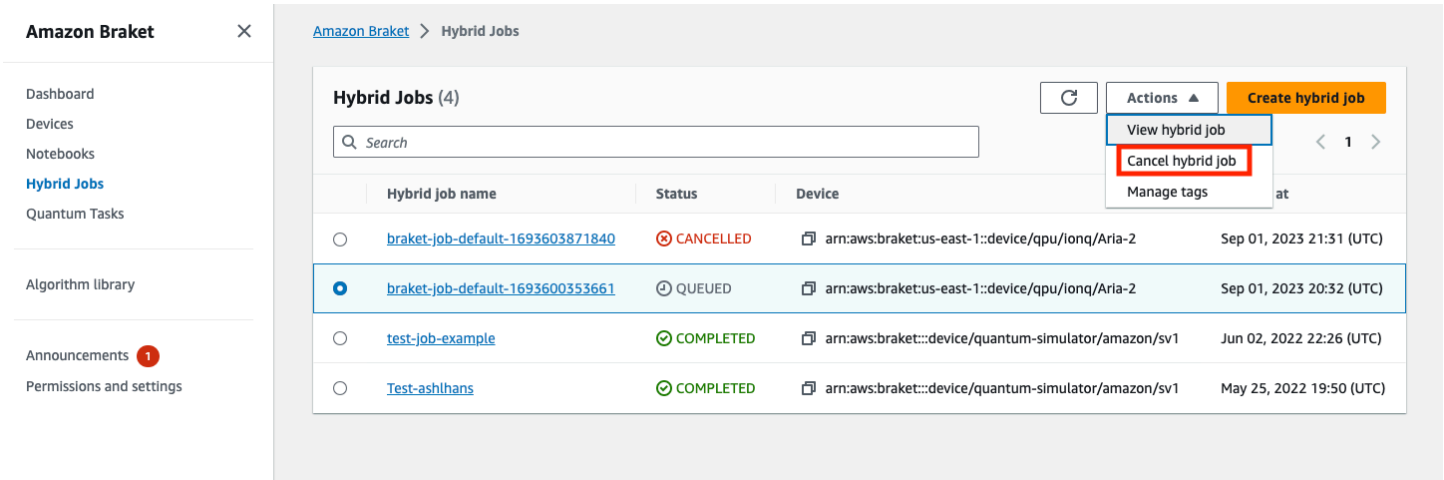
```
    source_module="algorithm_script.py",
    entry_point="algorithm_script:start_here",
)
```

> **Note**
>
> 이 기능을 사용하려면 Amazon Braket 노트북에 이미 사전 설치되어 있는 Docker를 로컬 환경에 설치해야 합니다. Docker 설치 지침은 Get Docker 페이지에서 확인할 수 있습니다. 또한 모든 파라미터가 로컬 모드에서 지원되는 것은 아닙니다.

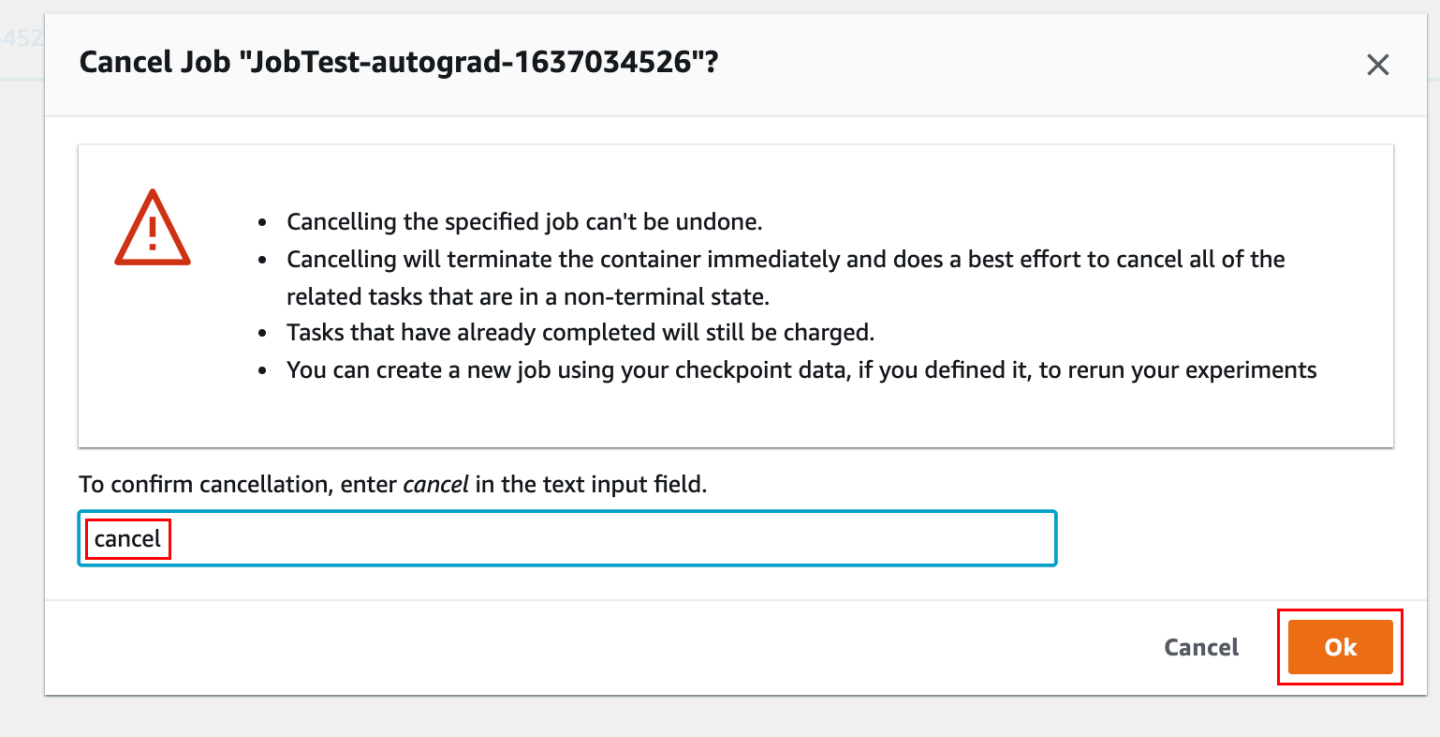
하이브리드 작업 취소

비종료 상태에서 하이브리드 작업을 취소해야 할 수 있습니다. 콘솔에서 또는 코드를 사용하여 이 작업을 수행할 수 있습니다.

콘솔에서 하이브리드 작업을 취소하려면 하이브리드 작업 페이지에서 취소할 하이브리드 작업을 선택한 다음 작업 드롭다운 메뉴에서 하이브리드 작업 취소를 선택합니다.



취소를 확인하려면 메시지가 표시될 때 입력 필드에 cancel를 입력한 다음 확인을 선택합니다.



Braket Python SDK의 코드를 사용하여 하이브리드 작업을 취소하려면 `job_arn`을 사용하여 하이브리드 작업을 식별한 다음, 다음 코드에서와 같이 `cancel` 명령을 직접적으로 호출합니다.

```
job = AwsQuantumJob(arn=job_arn)
job.cancel()
```

이 `cancel` 명령은 고전 하이브리드 작업 컨테이너를 즉시 종료하며 아직 비종료 상태인 모든 관련 양자 작업을 취소하기 위한 작업을 실행합니다.

하이브리드 작업 사용자 지정

Amazon Braket은 하이브리드 작업 실행 방식을 사용자 지정하여 특정 요구 사항에 맞게 환경을 조정할 수 있는 여러 가지 방법을 제공합니다. 이 섹션에서는 알고리즘 스크립트 환경 정의부터 자체 컨테이너 사용(BYOC)에 이르기까지 하이브리드 작업을 사용자 지정하는 옵션을 살펴봅니다. 하이퍼파라미터를 사용하여 워크플로를 최적화하고 작업 인스턴스를 구성하며 파라메트릭 컴파일을 활용하여 성능을 개선하는 방법을 알아봅니다. 이러한 사용자 지정 기법은 Amazon Braket에서 하이브리드 양자 계산의 잠재력을 극대화하는 데 도움이 됩니다.

이 섹션의 내용:

- 알고리즘 스크립트의 환경 정의
- 하이퍼파라미터 사용

- 하이브리드 작업 인스턴스 구성
- 파라메트릭 컴파일을 사용하여 하이브리드 작업 가속화

알고리즘 스크립트의 환경 정의

Amazon Braket은 알고리즘 스크립트의 컨테이너에서 정의한 환경을 지원합니다.

- 기본 컨테이너(`image_uri`가 지정되지 않은 경우의 기본값)
- CUDA-Q가 포함된 컨테이너
- Tensorflow 및 PennyLane이 포함된 컨테이너
- PyTorch, PennyLane 및 CUDA-Q가 포함된 컨테이너

다음 표에는 컨테이너와 컨테이너에 포함된 라이브러리에 대한 세부 정보가 나와 있습니다.

Amazon Braket 컨테이너

Type	기본	CUDA-Q	TensorFlow	PyTorch
이미지 URI	292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-base-jobs:latest	292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-cudaq-jobs:latest	292282985366.dkr.ecr.us-east-1.amazonaws.com/amazon-braket-tensorflow-jobs:latest	292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-pytorch-jobs:latest
상속된 라이브러리		• amazon-braket-default-simulator • amazon-braket-pennylane-plugin • amazon-braket-schemas • amazon-braket-sdk	• awscli • numpy • pandas • scipy	• awscli • numpy • pandas • scipy

Type	기본	CUDA-Q	TensorFlow	PyTorch
		• awscli • botocore • boto3 • dask • matplotlib • numpy • pandas • PennyLane • PennyLane-Lightning • qiskit-braket-provider • requests • sagemaker-training • scikit-learn • scipy		

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

aws/amazon-braket-containers에서 오픈 소스 컨테이너 정의를 보고 액세스할 수 있습니다. 사용 사례에 가장 적합한 컨테이너를 선택합니다. Braket에서 사용 가능한 AWS 리전(us-east-1, us-west-1, us-west-2, eu-north-1, eu-west-2)을 사용할 수 있지만 컨테이너 리전은 하이브리드 작업의 리전과 일치해야 합니다. 하이브리드 작업을 생성할 때 하이브리드 작업 스크립트의 `create(…)` 직접 호출에 다음

세 인수 중 하나를 추가하여 컨테이너 이미지를 지정합니다. Amazon Braket 컨테이너는 인터넷 연결이 가능하므로 런타임(시작 또는 런타임 시 비용 발생) 시 선택한 컨테이너에 추가 종속성을 설치할 수 있습니다. 다음 예제는 us-west-2 리전에 대한 것입니다.

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

Amazon Braket SDK의 `retrieve_image()` 함수를 사용하여 `image-uris`를 검색할 수도 있습니다. 다음 예제는 us-west-2에서 검색하는 방법을 보여줍니다 AWS 리전.

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

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

자체 컨테이너 사용(Bring Your Own Container, BYOC)

Amazon Braket Hybrid Jobs는 다양한 환경에서 코드를 실행하기 위해 사전 빌드된 세 개의 컨테이너를 제공합니다. 이러한 컨테이너 중 하나가 사용 사례를 지원하는 경우 하이브리드 작업을 생성할 때 알고리즘 스크립트만 제공하면 됩니다. 사소한 누락 종속성은 `pip`을 사용하여 알고리즘 스크립트 또는 `requirements.txt` 파일에서 추가할 수 있습니다.

이러한 컨테이너 중 사용 사례를 지원하는 컨테이너가 없거나 컨테이너를 확장하려는 경우 Braket Hybrid Jobs는 자체 사용자 지정 Docker 컨테이너 이미지로 하이브리드 작업을 실행하거나 자체 컨테이너를 사용(BYOC)할 수 있도록 지원합니다. 사용 사례에 적합한 기능인지 확인하세요.

이 섹션의 내용:

- 자체 컨테이너 사용이 올바른 결정인 시기는 언제인가요?
- 자체 컨테이너 사용 방법

- 자체 컨테이너에서 Braket 하이브리드 작업 실행

자체 컨테이너 사용이 올바른 결정인 시기는 언제인가요?

Braket Hybrid Jobs에서 자체 컨테이너를 사용(BYOC)하면 자체 소프트웨어를 패키지 환경에 설치하여 유연하게 사용할 수 있습니다. 특정 요구 사항에 따라, 전체 BYOC Docker 빌드 - Amazon ECR 업로드 - 사용자 지정 이미지 URI 주기를 거치지 않고도 동일한 유연성을 달성할 수 있는 방법이 있을 수 있습니다.

> **Note**
>
> 공개적으로 사용할 수 있는 소수의 추가 Python 패키지(일반적으로 10개 미만)를 추가하려는 경우 BYOC가 적합하지 않을 수 있습니다. 예를 들어 PyPi를 사용하는 경우입니다.

이 경우 사전 빌드된 Braket 이미지 중 하나를 사용한 다음, 작업 제출 시 소스 디렉터리에 `requirements.txt` 파일을 포함할 수 있습니다. 파일은 자동으로 읽히며 `pip`는 지정된 버전이 포함된 패키지를 정상적으로 설치합니다. 많은 수의 패키지를 설치하는 경우 해당 작업의 런타임이 크게 증가할 수 있습니다. 사용하려는 사전 빌드된 컨테이너의 Python 버전과 CUDA 버전(해당하는 경우)을 확인하여 소프트웨어가 작동하는지 테스트합니다.

작업 스크립트에 비Python 언어(예: C++ 또는 Rust)를 사용하거나 Braket 사전 빌드 컨테이너를 통해 제공되지 않는 Python 버전을 사용하려는 경우 BYOC가 필요합니다. 이는 다음과 같은 경우에도 좋은 선택입니다.

- 라이선스 키와 함께 소프트웨어를 사용 중이며 소프트웨어를 실행하려면 라이선스 서버에 대해 해당 키를 인증해야 합니다. BYOC를 사용하면 Docker 이미지에 라이선스 키를 임베드하고 이를 인증하는 코드를 포함할 수 있습니다.
- 공개적으로 사용할 수 없는 소프트웨어를 사용하고 있습니다. 예를 들어, 소프트웨어가 특정 SSH 키가 있어야 액세스할 수 있는 비공개 GitLab 또는 GitHub 리포지토리에 호스팅되는 경우입니다.
- Braket 제공 컨테이너에 패키징되지 않은 대규모 소프트웨어 제품군을 설치해야 합니다. BYOC를 사용하면 소프트웨어 설치로 인해 하이브리드 작업 컨테이너의 시작 시간이 길어지는 문제를 해결할 수 있습니다.

또한 BYOC를 사용하면 소프트웨어로 Docker 컨테이너를 빌드하여 사용자에게 제공함으로써 고객이 사용자 지정 SDK 또는 알고리즘을 사용하도록 할 수 있습니다. Amazon ECR에서 적절한 권한을 설정하여 이 작업을 수행할 수 있습니다.

> **Note**
>
> 해당하는 모든 소프트웨어 라이선스를 준수해야 합니다.

자체 컨테이너 사용 방법

이 섹션에서는 Braket Hybrid Jobs에 bring your own container (BYOC)를 적용하는 데 필요한 사항, 즉 스크립트, 파일 및 이를 결합하여 사용자 지정 Docker 이미지를 시작하고 실행하는 단계에 대한 단계별 가이드를 제공합니다. 두 가지 일반적인 사례에 대한 방법:

1. Docker 이미지에 추가 소프트웨어를 설치하고 해당 작업에 Python 알고리즘 스크립트만 사용합니다.
2. 하이브리드 작업 또는 x86 이외의 CPU 아키텍처와 함께 비Python 언어로 작성된 알고리즘 스크립트를 사용합니다.

컨테이너 항목 스크립트 정의는 두 번째 사례에서 더 복잡합니다.

Braket은 하이브리드 작업을 실행할 때 요청된 수와 유형의 Amazon EC2 인스턴스를 시작한 다음, 이미지 URI 입력으로 지정된 Docker 이미지를 실행하여 해당 인스턴스에서 작업을 생성합니다. BYOC 기능을 사용할 때는 읽기 액세스 권한이 있는 비공개 Amazon ECR 리포지토리에 호스팅된 이미지 URI를 지정합니다. Braket Hybrid Jobs는 해당 사용자 지정 이미지를 사용하여 작업을 실행합니다.

하이브리드 작업과 함께 사용할 수 있는 Docker 이미지를 빌드하는 데 필요한 특정 구성 요소입니다. `Dockerfiles` 작성 및 빌드에 익숙하지 않은 경우 Dockerfile 설명서 및 Amazon ECR CLI 설명서를 참조하세요.

요구 사항:

- Dockerfile의 기본 이미지
- (선택 사항) 수정된 컨테이너 진입점 스크립트
- Dockerfile을 사용하여 필요한 소프트웨어 및 컨테이너 스크립트 설치

Dockerfile의 기본 이미지

Python을 사용 중이고 Braket 제공 컨테이너에 제공된 것 외에도 추가로 소프트웨어를 설치하려는 경우, 기본 이미지에 대한 옵션은 GitHub 리포지토리와 Amazon ECR에 호스팅된 Braket 컨테이너 이미지 중 하나입니다. 이미지를 가져와 이를 기반으로 빌드하려면 Amazon ECR에 인증해야 합니다. 예를 들어 BYOC Docker 파일의 첫 번째 줄은 다음과 같을 수 있습니다. `FROM [IMAGE_URI_HERE]`

다음으로, Dockerfile의 나머지 부분을 채워 컨테이너에 추가하려는 소프트웨어를 설치하고 설정합니다. 사전 빌드된 Braket 이미지에는 이미 적절한 컨테이너 진입점 스크립트가 포함되어 있으므로 이를 포함하는 것에 대해 걱정할 필요가 없습니다.

C++, Rust 또는 Julia와 같은 비Python 언어를 사용하거나 ARM과 같은 비x86 CPU 아키텍처용 이미지를 빌드하려는 경우 기본 공개 이미지를 기반으로 빌드해야 할 수 있습니다. 이러한 이미지는 Amazon Elastic Container Registry Public Gallery에서 찾을 수 있습니다. CPU 아키텍처에 적합한 아키텍처와 필요시 사용하려는 GPU를 선택해야 합니다.

(선택 사항) 수정된 컨테이너 진입점 스크립트

> **Note**
>
> 사전 빌드된 Braket 이미지에 추가 소프트웨어만 추가하는 경우 이 섹션을 건너뛸 수 있습니다.

하이브리드 작업의 일부로 비Python 코드를 실행하려면 컨테이너 진입점을 정의하는 Python 스크립트를 수정합니다. 예를 들면, `braket_container.py` Amazon Braket Github의 python 스크립트입니다. 이는 Braket이 사전 빌드한 이미지가 알고리즘 스크립트를 실행하고 적절한 환경 변수를 설정하는 데 사용하는 스크립트입니다. 컨테이너 진입점 스크립트 자체는 Python으로 작성되어야 하지만, 비Python 스크립트를 실행할 수 있습니다. 사전 빌드된 예제에서 볼 수 있듯이, Python 알고리즘 스크립트는 Python 하위 프로세스 또는 완전히 새로운 프로세스로 실행됩니다. 이 논리를 수정하면 진입점 스크립트가 비Python 알고리즘 스크립트를 실행하도록 할 수 있습니다. 예를 들어, 파일 확장자 끝에 따라 Rust 프로세스를 실행하도록 `thekick_off_customer_script()` 함수를 수정할 수 있습니다.

완전히 새로운 `braket_container.py`를 작성하도록 선택할 수도 있습니다. 입력 데이터, 소스 아카이브 및 기타 필요한 파일을 Amazon S3에서 컨테이너로 복사하고 적절한 환경 변수를 정의해야 합니다.

`Dockerfile`을 사용하여 필요한 소프트웨어 및 컨테이너 스크립트 설치

> **Note**
>
> 사전 빌드된 Braket 이미지를 Docker 기본 이미지로 사용하는 경우 컨테이너 스크립트가 이미 있습니다.

이전 단계에서 수정된 컨테이너 스크립트를 생성한 경우 컨테이너에 이를 복사하고 환경 변수 SAGEMAKER_PROGRAM을 braket_container.py로 정의하거나 새 컨테이너 진입점 스크립트에 지정한 이름으로 정의해야 합니다.

다음은 GPU 가속 작업 인스턴스에서 Julia를 사용할 수 있게 하는 Dockerfile의 예제입니다.

```
FROM nvidia/cuda:12.2.0-devel-ubuntu22.04


 ARG DEBIAN_FRONTEND=noninteractive
 ARG JULIA_RELEASE=1.8
 ARG JULIA_VERSION=1.8.3


 ARG PYTHON=python3.11
 ARG PYTHON_PIP=python3-pip
 ARG PIP=pip


 ARG JULIA_URL = https://julialang-s3.julialang.org/bin/linux/x64/${JULIA_RELEASE}/
 ARG TAR_NAME = julia-${JULIA_VERSION}-linux-x86_64.tar.gz


 ARG PYTHON_PKGS = # list your Python packages and versions here


 RUN curl -s -L ${JULIA_URL}/${TAR_NAME} | tar -C /usr/local -x -z --strip-components=1
 -f -


 RUN apt-get update \

    && apt-get install -y --no-install-recommends \

    build-essential \

    tzdata \

    openssh-client \

    openssh-server \

    ca-certificates \
```

```
    curl \
    git \
    libtemplate-perl \
    libssl1.1 \
    openssl \
    unzip \
    wget \
    zlib1g-dev \
    ${PYTHON_PIP} \
    ${PYTHON}-dev \



 RUN ${PIP} install --no-cache --upgrade ${PYTHON_PKGS}


 RUN ${PIP} install --no-cache --upgrade sagemaker-training==4.1.3


 # Add EFA and SMDDP to LD library path
 ENV LD_LIBRARY_PATH="/opt/conda/lib/python${PYTHON_SHORT_VERSION}/site-packages/
smdistributed/dataparallel/lib:$LD_LIBRARY_PATH"
 ENV LD_LIBRARY_PATH=/opt/amazon/efa/lib/:$LD_LIBRARY_PATH


 # Julia specific installation instructions
 COPY Project.toml /usr/local/share/julia/environments/v${JULIA_RELEASE}/
 RUN JULIA_DEPOT_PATH=/usr/local/share/julia \

    julia -e 'using Pkg; Pkg.instantiate(); Pkg.API.precompile()'
 # generate the device runtime library for all known and supported devices
 RUN JULIA_DEPOT_PATH=/usr/local/share/julia \
```

```
    julia -e 'using CUDA; CUDA.precompile_runtime()'


 # Open source compliance scripts
 RUN HOME_DIR=/root \

 && curl -o ${HOME_DIR}/oss_compliance.zip https://aws-dlinfra-
utilities.s3.amazonaws.com/oss_compliance.zip \

 && unzip ${HOME_DIR}/oss_compliance.zip -d ${HOME_DIR}/ \

 && cp ${HOME_DIR}/oss_compliance/test/testOSSCompliance /usr/local/bin/
testOSSCompliance \

 && chmod +x /usr/local/bin/testOSSCompliance \

 && chmod +x ${HOME_DIR}/oss_compliance/generate_oss_compliance.sh \

 && ${HOME_DIR}/oss_compliance/generate_oss_compliance.sh ${HOME_DIR} ${PYTHON} \

 && rm -rf ${HOME_DIR}/oss_compliance*


 # Copying the container entry point script
 COPY braket_container.py /opt/ml/code/braket_container.py
 ENV SAGEMAKER_PROGRAM braket_container.py
```

이 예제에서는에서 제공하는 스크립트를 다운로드하고 실행 AWS 하여 모든 관련 오픈 소스 라이선스를 준수하는지 확인합니다. 예를 들어, MIT license에 의해 관리되는 모든 설치된 코드를 적절하게 속성화합니다.

비공개 코드(예: 비공개 GitHub 또는 GitLab 리포지토리에 호스팅된 코드)를 포함해야 하는 경우, 해당 코드에 액세스하기 위해 SSH 키를 Docker 이미지에 임베드하지 마세요. 대신, 빌드할 때 Docker Compose를 사용하여 Docker가 빌드된 호스트 시스템에서 SSH에 액세스할 수 있도록 허용합니다. 자세한 내용은 비공개 GitHub 리포지토리에 액세스하기 위해 Docker에서 SSH 키를 안전하게 사용하는 방법 가이드를 참조하세요.

Docker 이미지 빌드 및 업로드

`Dockerfile`이 올바르게 정의되었으므로, 이제 다음 단계에 따라 비공개 Amazon ECR 리포지토리를 생성할 준비가 되었습니다(아직 없는 경우). 컨테이너 이미지를 빌드하고 태그를 지정하여 리포지토리에 업로드할 수도 있습니다.

이미지를 빌드, 태그 지정 및 푸시할 준비가 되었습니다. `docker build`에 대한 옵션의 자세한 설명과 몇 가지 예제는 Docker 빌드 설명서를 참조하세요.

위에 정의된 샘플 파일의 경우 다음을 실행할 수 있습니다.

```
aws ecr get-login-password --region ${your_region} | docker login --username AWS --
password-stdin ${aws_account_id}.dkr.ecr.${your_region}.amazonaws.com
 docker build -t braket-julia .
 docker tag braket-julia:latest ${aws_account_id}.dkr.ecr.${your_region}.amazonaws.com/
braket-julia:latest
 docker push ${aws_account_id}.dkr.ecr.${your_region}.amazonaws.com/braket-julia:latest
```

적절한 Amazon ECR 권한 할당

Braket Hybrid Jobs Docker 이미지는 비공개 Amazon ECR 리포지토리에 호스팅되어야 합니다. 기본적으로 비공개 Amazon ECR 리포지토리는 Braket Hybrid Jobs IAM role 또는 공동 작업자나 학생 등 해당 이미지를 사용하려는 다른 사용자에게 읽기 액세스 권한을 제공하지 않습니다. 적절한 권한을 부여하려면 리포지토리 정책을 설정해야 합니다. 일반적으로, 이미지에 액세스하려는 특정 사용자와 IAM 역할에만 권한을 부여하고, image URI를 가진 사람 누구나 이미지를 가져오는 것을 허용하지 마세요.

자체 컨테이너에서 Braket 하이브리드 작업 실행

자체 컨테이너로 하이브리드 작업을 생성하려면 지정된 인수 `image_uri`로 `AwsQuantumJob.create()`를 직접적으로 호출합니다. QPU, 온디맨드 시뮬레이터를 사용하거나 Braket Hybrid Jobs에서 사용할 수 있는 고전 프로세서에서 로컬로 코드를 실행할 수 있습니다. 실제 QPU에서 실행하기 전에 SV1, DM1 또는 TN1과 같은 시뮬레이터에서 코드를 테스트하는 것이 좋습니다.

고전 프로세서에서 코드를 실행하려면 `InstanceConfig`를 업데이트하여 사용하는 `instanceType` 및 `instanceCount`를 지정합니다. `instance_count` > 1을 지정하는 경우 코드가 여러 호스트에서 실행될 수 있는지 확인해야 합니다. 선택할 수 있는 인스턴스 수의 상한은 5입니다. 예제:

```
job = AwsQuantumJob.create(
    source_module="source_dir",
    entry_point="source_dir.algorithm_script:start_here",
    image_uri="111122223333.dkr.ecr.us-west-2.amazonaws.com/my-byoc-container:latest",
    instance_config=InstanceConfig(instanceType="ml.g4dn.xlarge", instanceCount=3),
    device="local:braket/braket.local.qubit",
    # ...)
```

> **Note**
>
> 디바이스 ARN을 사용하여 하이브리드 작업 메타데이터로 사용한 시뮬레이터를 추적합니다. 허용되는 값은 형식 `device = "local:<provider>/<simulator_name>"`을 따라야 합니다. `<provider>` 및 `<simulator_name>`은 문자, 숫자, `_`, `-`, `.`으로만 구성되어야 합니다. 문자열은 256자로 제한됩니다.
> BYOC를 사용할 계획이고 Braket SDK를 사용하여 양자 작업을 생성하지 않는 경우 환경 변수 `AMZN_BRAKET_JOB_TOKEN`의 값을 `CreateQuantumTask` 요청의 `jobToken` 파라미터에 전달해야 합니다. 그렇지 않으면 양자 작업에 우선순위가 부여되지 않으며 일반 독립 실행형 양자 작업으로 청구됩니다.

하이퍼파라미터 사용

하이브리드 작업을 생성할 때 학습률 또는 단계 크기와 같이 알고리즘에 필요한 하이퍼파라미터를 정의할 수 있습니다. 하이퍼파라미터 값은 일반적으로 알고리즘의 다양한 측면을 제어하는 데 사용되며 알고리즘의 성능을 최적화하기 위해 조정되는 경우가 많습니다. Braket 하이브리드 작업에서 하이퍼파라미터를 사용하려면 이름과 값을 딕셔너리로 명시적으로 지정해야 합니다. 최적의 값 집합을 검색할 때 테스트할 하이퍼파라미터 값을 지정합니다. 하이퍼파라미터를 사용하기 위한 첫 번째 단계는 하이퍼파라미터를 딕셔너리로 설정하고 정의하는 것이며, 이는 다음 코드에서 확인할 수 있습니다.

```
from braket.devices import Devices

device_arn = Devices.Amazon.SV1

hyperparameters = {"shots": 1_000}
```

그런 다음 위에 제공된 코드 조각에 정의된 하이퍼파라미터를 전달하여 원하는 알고리즘에 사용합니다. 다음 코드 예제를 실행하려면 하이퍼파라미터 파일과 동일한 경로에 “src”라는 디렉터리를 생성합니다. "src" 디렉터리 내에 0_Getting_started_papermill.ipynb, notebook_runner.py 및 requirements.txt 코드 파일을 추가합니다.

```
import time
from braket.aws import AwsQuantumJob

job = AwsQuantumJob.create(
    device=device_arn,
    source_module="src",
    entry_point="src.notebook_runner:run_notebook",
```

```
    input_data="src/0_Getting_started_papermill.ipynb",
    hyperparameters=hyperparameters,
    job_name=f"papermill-job-demo-{int(time.time())}",
)

# Print job to record the ARN
print(job)
```

하이브리드 작업 스크립트 내에서 하이퍼파라미터에 액세스하려면 notebook_runner.py python 파일의 `load_jobs_hyperparams()` 함수를 참조하세요. 하이브리드 작업 스크립트 외부에서 하이퍼파라미터에 액세스하려면 다음 코드를 실행합니다.

```
from braket.aws import AwsQuantumJob

# Get the job using the ARN
job_arn = "arn:aws:braket:us-east-1:111122223333:job/5eabb790-d3ff-47cc-98ed-
b4025e9e296f"  # Replace with your job ARN
job = AwsQuantumJob(arn=job_arn)

# Access the hyperparameters
job_metadata = job.metadata()
hyperparameters = job_metadata.get("hyperParameters", {})
print(hyperparameters)
```

하이퍼파라미터 사용 방법에 대한 자세한 내용은 Amazon Braket Hybrid Jobs와 PennyLane을 활용한 QAOA 및 Amazon Braket Hybrid Jobs에서의 양자 기계 학습 자습서를 참조하세요.

하이브리드 작업 인스턴스 구성

알고리즘에 따라 요구 사항이 다를 수 있습니다. 기본적으로 Amazon Braket은 `ml.m5.large` 인스턴스에서 알고리즘 스크립트를 실행합니다. 그러나 다음 가져오기 및 구성 인수를 사용하여 하이브리드 작업을 생성할 때 이 인스턴스 유형을 사용자 지정할 수 있습니다.

```
from braket.jobs.config import InstanceConfig

job = AwsQuantumJob.create(
    ...
    instance_config=InstanceConfig(instanceType="ml.g4dn.xlarge"), # Use NVIDIA T4
 instance with 4 GPUs.
    ...
    ),
```

임베디드 시뮬레이션을 실행 중이고 디바이스 구성에서 로컬 디바이스를 지정한 경우, `instanceCount`를 지정하고 둘 이상으로 설정하여 `InstanceConfig`에서 둘 이상의 인스턴스를 추가로 요청할 수 있습니다. 상한은 5입니다. 예를 들어 다음과 같이 인스턴스 3개를 선택할 수 있습니다.

```
from braket.jobs.config import InstanceConfig
job = AwsQuantumJob.create(
    ...
    instance_config=InstanceConfig(instanceType="ml.g4dn.xlarge", instanceCount=3), #
 Use 3 NVIDIA T4 instances
    ...
    ),
```

여러 인스턴스를 사용하는 경우 데이터 병렬 기능을 사용하여 하이브리드 작업을 배포하는 것이 좋습니다. 이 QML을 위한 병렬화 훈련 예제를 확인하는 방법에 대한 자세한 내용은 다음 예제 노트북을 참조하세요.

다음 세 개의 표에는 표준, 고성능 및 GPU 가속 인스턴스에 사용할 수 있는 인스턴스 유형과 사양이 나와 있습니다.

> **Note**
>
> 하이브리드 작업에 대한 기본 고전 컴퓨팅 인스턴스 할당량을 보려면 Amazon Braket 할당량 페이지를 참조하세요.

표준 인스턴스	vCPU	메모리(GiB)
ml.m5.large(기본값)	4	16
ml.m5.xlarge	4	16
ml.m5.2xlarge	8	32
ml.m5.4xlarge	16	64
ml.m5.12xlarge	48	192
ml.m5.24xlarge	96	384

고성능 인스턴스	vCPU	메모리(GiB)
ml.c5.xlarge	4	8
ml.c5.2xlarge	8	16
ml.c5.4xlarge	16	32
ml.c5.9xlarge	36	72
ml.c5.18xlarge	72	144
ml.c5n.xlarge	4	10.5
ml.c5n.2xlarge	8	21
ml.c5n.4xlarge	16	32
ml.c5n.9xlarge	36	72
ml.c5n.18xlarge	72	192

GPU 가속 인스턴스	GPU	vCPU	메모리(GiB)	GPU 메모리(GiB)
ml.p4d.24xlarge	8	96	1152	320
ml.g4dn.xlarge	1	4	16	16
ml.g4dn.2xlarge	1	8	32	16
ml.g4dn.4xlarge	1	16	64	16
ml.g4dn.8xlarge	1	32	128	16
ml.g4dn.12xlarge	4	48	192	64
ml.g4dn.16xlarge	1	64	256	16

각 인스턴스는 30GB의 기본 데이터 스토리지(SSD) 구성을 사용합니다. 그러나 instanceType을 구성하는 것과 동일한 방식으로 스토리지를 조정할 수 있습니다. 다음 예제에서는 총 스토리지를 50GB로 늘리는 방법을 보여줍니다.

```
from braket.jobs.config import InstanceConfig

job = AwsQuantumJob.create(
    ...
    instance_config=InstanceConfig(
        instanceType="ml.g4dn.xlarge",
        volumeSizeInGb=50,
    ),
    ...
    ),
```

AwsSession에서 기본 버킷 구성

자체 AwsSession 인스턴스를 활용하면 기본 Amazon S3 버킷의 사용자 지정 위치를 지정하는 등 향상된 유연성을 누릴 수 있습니다. 기본적으로 AwsSession에는 "amazon-braket-{id}-{region}"의 사전 구성된 Amazon S3 버킷 위치가 있습니다. 그러나 AwsSession을 생성할 때 기본 Amazon S3 버킷 위치를 재정의할 수 있는 옵션이 있습니다. 다음 코드 예제에서 볼 수 있듯이, 사용자는 aws_session 파라미터를 제공하여 선택적으로 AwsSession 객체를 AwsQuantumJob.create() 메서드로 전달할 수 있습니다.

```
aws_session = AwsSession(default_bucket="amazon-braket-s3-demo-bucket")

# Then you can use that AwsSession when creating a hybrid job
job = AwsQuantumJob.create(
    ...
    aws_session=aws_session
)
```

파라메트릭 컴파일을 사용하여 하이브리드 작업 가속화

Amazon Braket은 특정 QPU에서 파라메트릭 컴파일을 지원합니다. 따라서 하이브리드 알고리즘의 모든 반복에 대해 회로를 한 번만 컴파일하여 계산 비용이 많이 드는 컴파일 단계와 관련된 오버헤드를 줄일 수 있습니다. 각 단계에서 회로를 다시 컴파일할 필요가 없으므로 하이브리드 작업의 런타임이 크게 향상됩니다. 파라미터화된 회로를 Braket Hybrid Job으로 지원되는 QPU 중 하나에 제출하기만 하면 됩니다. 장기 실행 하이브리드 작업의 경우, Braket은 회로를 컴파일할 때 하드웨어 공급자의 업데이트된 보정 데이터를 자동으로 사용하여 최고 품질의 결과를 보장합니다.

파라메트릭 회로를 생성하려면 먼저 알고리즘 스크립트에 파라미터를 입력으로 제공해야 합니다. 이 예제에서는 작은 파라미터 회로를 사용하고 각 반복 사이의 모든 고전적 처리를 무시합니다. 일반적인 워크로드의 경우 여러 회로를 일괄적으로 제출하고 각 반복에서 파라미터를 업데이트하는 등의 고전적 처리를 수행합니다.

```
import os

from braket.aws import AwsDevice
from braket.circuits import Circuit, FreeParameter

def start_here():

    print("Test job started.")

    # Use the device declared in the job script
    device = AwsDevice(os.environ["AMZN_BRAKET_DEVICE_ARN"])

    circuit = Circuit().rx(0, FreeParameter("theta"))
    parameter_list = [0.1, 0.2, 0.3]

    for parameter in parameter_list:
        result = device.run(circuit, shots=1000, inputs={"theta": parameter})

    print("Test job completed.")
```

다음 작업 스크립트를 사용하여 하이브리드 작업으로 실행할 알고리즘 스크립트를 제출할 수 있습니다. 파라메트릭 컴파일을 지원하는 QPU에서 하이브리드 작업을 실행할 때 회로는 첫 번째 실행 시에만 컴파일됩니다. 다음 실행에서는 컴파일된 회로가 재사용되므로 추가 코드 줄 없이 하이브리드 작업의 런타임 성능이 향상됩니다.

```
from braket.aws import AwsQuantumJob

job = AwsQuantumJob.create(
    device=device_arn,
    source_module="algorithm_script.py",
)
```

> **Note**
>
> 파라메트릭 컴파일은 펄스 레벨 프로그램을 제외한 Rigetti Computing의 모든 초전도 게이트 기반 QPU에서 지원됩니다.

Amazon Braket에서 PennyLane 사용

하이브리드 알고리즘은 고전 명령과 양자 명령을 모두 포함하는 알고리즘입니다. 고전 명령은 고전 하드웨어(EC2 인스턴스 또는 랩톱)에서 실행되고 양자 명령은 시뮬레이터 또는 양자 컴퓨터에서 실행됩니다. 하이브리드 작업 기능을 사용하여 하이브리드 알고리즘을 실행하는 것이 좋습니다. 자세한 내용은 Amazon Braket Jobs를 사용해야 하는 시점을 참조하세요.

Amazon Braket을 사용하면 Amazon Braket PennyLane 플러그인의 지원 또는 Amazon Braket Python SDK 및 예제 노트북 리포지토리를 통해 하이브리드 양자 알고리즘을 설정하고 실행할 수 있습니다. SDK를 기반으로 하는 Amazon Braket 예제 노트북을 사용하면 PennyLane 플러그인 없이 특정 하이브리드 알고리즘을 설정하고 실행할 수 있습니다. 그러나 PennyLane은 더 풍부한 경험을 제공하므로 PennyLane을 사용하는 것이 좋습니다.

하이브리드 양자 알고리즘 정보

하이브리드 양자 알고리즘은 오늘날 산업계에 중요합니다. 현대 양자 컴퓨팅 디바이스는 일반적으로 노이즈를 생성하고 이로 인해 오류가 발생하기 때문입니다. 계산에 양자 게이트를 추가할 때마다 노이즈가 추가될 가능성이 커지며, 따라서 장기 실행 알고리즘은 노이즈에 압도되어 잘못된 계산을 초래할 수 있습니다.

Shor's(양자 위상 추정 예제) 또는 Grover's(Grover 예제)와 같은 순수 양자 알고리즘에는 수천, 수백만 회의 연산이 필요합니다. 이러한 이유로, 일반적으로 노이즈가 많은 중간 규모 양자 (NISQ) 디바이스라고 하는 기존 양자 디바이스에는 실용적이지 않을 수 있습니다.

하이브리드 양자 알고리즘에서 QPU는 고전 CPU의 공동 프로세서로 작동하여, 특히 고전적 알고리즘에서 특정 계산 속도를 높입니다. 회로 실행 시간이 현저히 단축되어 오늘날 디바이스의 성능에 도달할 수 있게 되었습니다.

이 섹션의 내용:

- PennyLane을 지원하는 Amazon Braket
- Amazon Braket 예제 노트북의 하이브리드 알고리즘
- PennyLane 시뮬레이터가 임베디드된 하이브리드 알고리즘

- Amazon Braket 시뮬레이터를 사용한 PennyLane의 수반 기울기
- 하이브리드 작업 및 PennyLane을 사용하여 QAOA 알고리즘 실행
- PennyLane 임베디드 시뮬레이터를 사용한 하이브리드 워크로드 실행

PennyLane을 지원하는 Amazon Braket

Amazon Braket은 양자 미분 프로그래밍 개념을 기반으로 빌드된 오픈 소스 소프트웨어 프레임워크인 PennyLane에 대한 지원을 제공합니다. 이 프레임워크를 사용하면 양자 화학, 양자 기계 학습, 최적화 분야의 계산 문제에 대한 해법을 찾기 위해 신경망을 훈련시키는 것과 동일한 방식으로 양자 회로를 훈련시킬 수 있습니다.

PennyLane 라이브러리는 PyTorch 및 TensorFlow를 비롯한 친숙한 기계 학습 도구에 대한 인터페이스를 제공하여 양자 회로 훈련을 빠르고 직관적으로 수행할 수 있게 합니다.

- PennyLane 라이브러리 -- PennyLane은 Amazon Braket 노트북에 사전 설치되어 있습니다. PennyLane에서 Amazon Braket 디바이스에 액세스하려면 노트북을 열고 다음 명령을 사용하여 PennyLane 라이브러리를 가져옵니다.

```
import pennylane as qml
```

자습서 노트북을 사용하면 빠르게 시작할 수 있습니다. 또는 원하는 IDE에서 Amazon Braket의 PennyLane을 사용할 수 있습니다.

- Amazon Braket PennyLane 플러그인 — 자체 IDE를 사용하려면 Amazon Braket PennyLane 플러그인을 수동으로 설치할 수 있습니다. 플러그인은 PennyLane을 Amazon Braket Python SDK와 연결하므로 Amazon Braket 디바이스의 PennyLane에서 회로를 실행할 수 있습니다. PennyLane 플러그인을 설치하려면 다음 명령을 사용합니다.

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

다음 예제에서는 PennyLane에서 Amazon Braket 디바이스에 대한 액세스를 설정하는 방법을 보여줍니다.

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

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

result = circuit(0.543)


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

PennyLane에 대한 자습서 예제와 자세한 내용은 Amazon Braket 예제 리포지토리를 참조하세요.

Amazon Braket PennyLane 플러그인을 사용하면 단 한 줄의 코드로 PennyLane에서 Amazon Braket QPU와 임베디드 시뮬레이터 디바이스 간에 전환할 수 있습니다. PennyLane과 함께 사용할 수 있는 2개의 Amazon Braket 양자 디바이스를 제공합니다.

- QPU 및 시뮬레이터를 포함한 Amazon Braket 서비스의 양자 디바이스로 실행하기 위한 `braket.aws.qubit`
- Amazon Braket SDK의 로컬 시뮬레이터로 실행하기 위한 `braket.local.qubit`

Amazon Braket PennyLane 플러그인은 오픈 소스입니다. PennyLane Plugin GitHub 리포지토리에서 설치할 수 있습니다.

PennyLane에 대한 자세한 내용은 PennyLane 웹 사이트의 설명서를 참조하세요.

Amazon Braket 예제 노트북의 하이브리드 알고리즘

Amazon Braket은 하이브리드 알고리즘 실행을 위해 PennyLane 플러그인에 의존하지 않는 다양한 예제 노트북을 제공합니다. 양자 근사 최적화 알고리즘(Quantum Approximate Optimization Algorithm, QAOA) 또는 변분 양자 고유해석기(Variational Quantum Eigensolver, VQE)와 같은 변분법을 설명하는 Amazon Braket 하이브리드 예제 노트북 중 하나를 사용하여 시작할 수 있습니다.

Amazon Braket 예제 노트북은 Amazon Braket Python SDK를 사용합니다. SDK는 Amazon Braket을 통해 양자 컴퓨팅 하드웨어 디바이스와 상호 작용하는 프레임워크를 제공합니다. 하이브리드 워크플로의 양자 부분을 지원하도록 설계된 오픈 소스 라이브러리입니다.

예제 노트북을 통해 Amazon Braket을 더 자세히 살펴볼 수 있습니다.

PennyLane 시뮬레이터가 임베디드된 하이브리드 알고리즘

Amazon Braket Hybrid Jobs는 이제 PennyLane의 고성능 CPU 및 GPU 기반 임베디드 시뮬레이터와 함께 제공됩니다. 이 임베디드 시뮬레이터 제품군은 하이브리드 작업 컨테이너 내에 직접 임베디드할 수 있으며 빠른 상태 벡터 `lightning.qubit` 시뮬레이터, NVIDIA의 cuQuantum 라이브러리를 사용하여 가속화된 `lightning.gpu` 시뮬레이터 등을 포함합니다. 이러한 임베디드 시뮬레이터는 수반 미분법과 같은 고급 방법을 활용할 수 있는 양자 기계 학습과 같은 변분 알고리즘에 적합합니다. 하나 이상의 CPU 또는 GPU 인스턴스에서 이러한 임베디드 시뮬레이터를 실행할 수 있습니다.

하이브리드 작업을 통해 이제 고전 공동 프로세서와 QPU를 조합하거나 SV1과 같은 Amazon Braket 온디맨드 시뮬레이터를 사용하거나 PennyLane의 임베디드 시뮬레이터를 직접 사용하여 변분 알고리즘 코드를 실행할 수 있습니다.

임베디드 시뮬레이터는 하이브리드 작업 컨테이너에 이미 제공되어 있으므로 `@hybrid_job` 데코레이터로 기본 Python 함수를 데코레이션해야 합니다. PennyLane `lightning.gpu` 시뮬레이터를 사용하려면 다음 코드 조각과 같이 `InstanceConfig`에서 GPU 인스턴스도 지정해야 합니다.

```
import pennylane as qml
from braket.jobs import hybrid_job
from braket.jobs.config import InstanceConfig


@hybrid_job(device="local:pennylane/lightning.gpu",
 instance_config=InstanceConfig(instanceType="ml.g4dn.xlarge"))
def function(wires):
    dev = qml.device("lightning.gpu", wires=wires)
    ...
```

하이브리드 작업에서 PennyLane 임베디드 시뮬레이터 사용을 시작하려면 예제 노트북을 참조하세요.

Amazon Braket 시뮬레이터를 사용한 PennyLane의 수반 기울기

Amazon Braket용 PennyLane 플러그인을 사용하면 로컬 상태 벡터 시뮬레이터 또는 SV1에서 실행할 때 수반 미분법을 사용하여 기울기를 계산할 수 있습니다.

참고: 수반 미분법을 사용하려면 `diff_method='adjoint'`가 아닌 qnode에 `diff_method='device'`를 지정해야 합니다. 다음 예제를 참조하세요.

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

```
dev = qml.device("braket.aws.qubit", wires=wires, shots=0, device_arn=device_arn)

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

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

> **Note**
>
> 현재 PennyLane은 QAOA 해밀토니안에 대한 그룹화 인덱스를 계산하고 이를 사용하여 해밀토니안을 여러 기대값으로 분할합니다. PennyLane에서 QAOA를 실행할 때 SV1의 수반 미분 기능을 사용하려면 다음과 같이 그룹화 인덱스를 제거하여 비용 해밀토니안을 재구성해야 합니다. `cost_h, mixer_h = qml.qaoa.max_clique(g, constrained=False)`
> `cost_h = qml.Hamiltonian(cost_h.coeffs, cost_h.ops)`

하이브리드 작업 및 PennyLane을 사용하여 QAOA 알고리즘 실행

이 섹션에서는 학습한 내용을 바탕으로 파라메트릭 컴파일과 PennyLane을 함께 사용하여 실제 하이브리드 프로그램을 작성하게 됩니다. 알고리즘 스크립트를 사용하여 QAOA 문제를 해결합니다. 이 프로그램은 고전적 Max Cut 최적화 문제에 해당하는 비용 함수를 생성하고 파라미터화된 양자 회로를 지정하며 경사 하강법을 사용해 파라미터를 최적화하여 비용 함수가 최소화되도록 합니다. 이 예제에서는 단순화를 위해 알고리즘 스크립트에 문제 그래프를 생성하지만, 보다 일반적인 사용 사례에서는 입력 데이터 구성의 전용 채널을 통해 문제 사양을 제공하는 것이 가장 좋습니다. 플래그 `parametrize_differentiable`은 기본적으로 `True`로 설정되므로 지원되는 QPU에서 파라메트릭 컴파일을 통해 향상된 런타임 성능의 이점을 자동으로 얻을 수 있습니다.

```
import os
import json
import time

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

import networkx as nx
import pennylane as qml
from pennylane import numpy as np
```

```
from matplotlib import pyplot as plt

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

def start_here():
    input_dir = os.environ["AMZN_BRAKET_INPUT_DIR"]
    output_dir = os.environ["AMZN_BRAKET_JOB_RESULTS_DIR"]
    job_name = os.environ["AMZN_BRAKET_JOB_NAME"]
    checkpoint_dir = os.environ["AMZN_BRAKET_CHECKPOINT_DIR"]
    hp_file = os.environ["AMZN_BRAKET_HP_FILE"]
    device_arn = os.environ["AMZN_BRAKET_DEVICE_ARN"]

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

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

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

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

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

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

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

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

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

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

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

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

    t1 = time.time()

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

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

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

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

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

> **Note**
>
> 파라메트릭 컴파일은 펄스 레벨 프로그램을 제외한 Rigetti Computing의 모든 초전도 게이트 기반 QPU에서 지원됩니다.

PennyLane 임베디드 시뮬레이터를 사용한 하이브리드 워크로드 실행

Amazon Braket Hybrid Jobs에서 PennyLane의 임베디드 시뮬레이터를 사용하여 하이브리드 워크로드를 실행하는 방법을 살펴보겠습니다. Pennylane의 GPU 기반 임베디드 시뮬레이터인 `lightning.gpu`는 Nvidia cuQuantum 라이브러리를 사용하여 회로 시뮬레이션을 가속화합니다. 임베디드 GPU 시뮬레이터는 사용자가 즉시 사용할 수 있는 모든 Braket 작업 컨테이너에 미리 구성되어 있습니다. 이 페이지에서는 `lightning.gpu`를 사용하여 하이브리드 워크로드의 속도를 높이는 방법을 보여줍니다.

QAOA 워크로드 `lightning.gpu`에 사용

이 노트북의 양자 근사 최적화 알고리즘(Quantum Approximate Optimization Algorithm, QAOA) 예제를 살펴보세요. 임베디드 시뮬레이터를 선택하려면 `device` 인수를 다음과 같은 형식의 문자열로 지정합니다. `"local:<provider>/<simulator_name>"` 예를 들어 `lightning.gpu`에 대해 `"local:pennylane/lightning.gpu"`를 설정합니다. 시작 시 하이브리드 작업에 제공하는 디바이스 문자열은 환경 변수 `"AMZN_BRAKET_DEVICE_ARN"`으로 해당 작업에 전달됩니다.

```
device_string = os.environ["AMZN_BRAKET_DEVICE_ARN"]
prefix, device_name = device_string.split("/")
```

```
device = qml.device(simulator_name, wires=n_wires)
```

이 페이지에서 2개의 임베디드 PennyLane 상태 벡터 시뮬레이터 `lightning.qubit`(CPU 기반)와 `lightning.gpu`(GPU 기반)를 비교합니다. 시뮬레이터에 사용자 지정 게이트 분해를 제공하여 다양한 기울기를 계산합니다.

이제 하이브리드 작업 시작 스크립트를 준비할 수 있습니다. `ml.m5.2xlarge` 및 `ml.g4dn.xlarge`의 두 가지 인스턴스 유형을 사용하여 QAOA 알고리즘을 실행합니다. `ml.m5.2xlarge` 인스턴스 유형은 표준 개발자 랩톱과 비슷합니다. 는 메모리가 16GB인 단일 NVIDIA T4 GPU가 있는 가속 컴퓨팅 인스턴스`ml.g4dn.xlarge`입니다.

GPU를 실행하려면 먼저 호환되는 이미지와 올바른 인스턴스(기본값은 `ml.m5.2xlarge` 인스턴스)를 지정해야 합니다.

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

image_uri = retrieve_image(Framework.PL_PYTORCH, AwsSession().region)
instance_config = InstanceConfig(instanceType="ml.g4dn.xlarge")
```

그런 다음 시스템 및 하이브리드 작업 인수 모두에서 업데이트된 디바이스 파라미터와 함께 하이브리드 작업 데코레이터에 이를 입력해야 합니다.

```
@hybrid_job(
        device="local:pennylane/lightning.gpu",
        input_data=input_file_path,
        image_uri=image_uri,
        instance_config=instance_config)
def run_qaoa_hybrid_job_gpu(p=1, steps=10):
    params = np.random.rand(2, p)

    braket_task_tracker = Tracker()

    graph = nx.read_adjlist(input_file_path, nodetype=int)
    wires = list(graph.nodes)
    cost_h, _mixer_h = qaoa.maxcut(graph)

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

> **Note**
>
> 를 GPU 기반 인스턴스를 사용하는 `instance_config` 것으로 지정하지만를 임베디드 CPU 기반 시뮬레이터(`lightning.qubit`)device로 선택하면 GPU가 사용되지 않습니다. GPU를 대상으로 지정하려면 임베디드 GPU 기반 시뮬레이터를 사용해야 합니다!

`m5.2xlarge` 인스턴스의 평균 반복 시간은 약 73초이고 `ml.g4dn.xlarge` 인스턴스의 경우 약 0.6초입니다. 이 21비트 워크플로의 경우 GPU 인스턴스는 100배의 속도 향상을 제공합니다. Amazon Braket Hybrid Jobs 요금 페이지를 보면 `m5.2xlarge` 인스턴스의 분당 비용이 0.00768 USD인 반면 `ml.g4dn.xlarge` 인스턴스의 경우 0.01227 USD임을 알 수 있습니다. 이 인스턴스에서는 GPU 인스턴스에서 실행하는 것이 더 빠르고 저렴합니다.

양자 기계 학습 및 데이터 병렬 처리

워크로드 유형이 데이터세트를 기반으로 훈련하는 양자 기계 학습(QML)인 경우 데이터 병렬 처리를 사용하여 워크로드를 더욱 가속화할 수 있습니다. QML에서 모델은 하나 이상의 양자 회로를 포함합니다. 모델은 고전 신경망을 포함할 수도 있고 포함하지 않을 수도 있습니다. 데이터세트로 모델을 훈련할 때 모델의 파라미터는 손실 함수를 최소화하도록 업데이트됩니다. 손실 함수는 일반적으로 단일 데이터 포인트에 대해 정의되며 총 손실은 전체 데이터세트에 대한 평균 손실에 대해 정의됩니다. QML에서 손실은 일반적으로 기울기 계산을 위한 총 손실로 평균화하기 전에 직렬로 계산됩니다. 이 절차는 특히 수백 개의 데이터 포인트가 있는 경우 시간이 많이 걸립니다.

한 데이터 포인트의 손실은 다른 데이터 포인트에 의존하지 않으므로 손실을 병렬로 평가할 수 있습니다. 서로 다른 데이터 포인트와 관련된 손실 및 기울기를 동시에 평가할 수 있습니다. 이를 데이터 병렬 처리라고 합니다. SageMaker의 분산 데이터 병렬 라이브러리를 통해, Amazon Braket Hybrid Jobs는 데이터 병렬 처리를 사용하여 훈련을 더욱 쉽게 가속화할 수 있습니다.

바이너리 분류의 예로 잘 알려진 UCI 리포지토리의 Sonar 데이터세트 데이터세트를 사용하는 데이터 병렬 처리를 위한 다음 QML 워크로드를 고려해 보세요. Sonar 데이터세트에는 각각 60개의 특징을 가진 208개의 데이터 포인트가 있으며, 이는 물질에서 반사되는 Sonar 신호로부터 수집됩니다. 각 데이터 포인트는 mines의 "M", rocks의 "R"로 표시됩니다. QML 모델은 입력 계층, 은닉 계층으로서 양자 회로 및 출력 계층으로 구성됩니다. 입력 및 출력 계층은 PyTorch에서 구현된 고전 신경망입니다. 양자 회로는 PennyLane의 qml.qnn 모듈을 사용하여 PyTorch 신경망과 통합됩니다. 워크로드에 대한 자세한 내용은 예제 노트북을 참조하세요. 위의 QAOA 예제와 마찬가지로, PennyLane의 `lightning.gpu`와 같은 임베디드 GPU 기반 시뮬레이터를 사용하면 GPU의 성능을 활용하여 임베디드 CPU 기반 시뮬레이터보다 성능을 개선할 수 있습니다.

하이브리드 작업을 생성하려면 AwsQuantumJob.create를 직접적으로 호출하고 키워드 인수를 통해 알고리즘 스크립트, 디바이스 및 기타 구성을 지정할 수 있습니다.

```
instance_config = InstanceConfig(instanceType='ml.g4dn.xlarge')

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

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

데이터 병렬 처리를 사용하려면, SageMaker 분산 라이브러리의 알고리즘 스크립트에서 몇 줄의 코드를 수정하여 훈련을 올바르게 병렬 처리해야 합니다. 먼저, 여러 GPU와 여러 인스턴스에 걸쳐 워크로드를 분산하는 데 필요한 대부분의 작업을 수행하는 smdistributed 패키지를 가져옵니다. 이 패키지는 Braket PyTorch 및 TensorFlow 컨테이너에 미리 구성되어 있습니다. dist 모듈은 알고리즘 스크립트에 훈련(world_size)을 위한 GPU의 총 수와 GPU 코어의 rank 및 local_rank를 알려줍니다. rank는 모든 인스턴스에서 GPU의 절대 인덱스이며, local_rank는 인스턴스 내 GPU의 인덱스입니다. 예를 들어 훈련에 각각 8개의 GPU가 할당된 인스턴스가 4개 있는 경우 rank 범위는 0~31이고 local_rank 범위는 0~7입니다.

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

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

다음으로, world_size와 rank에 따라 DistributedSampler를 정의한 후 데이터 로더에 전달합니다. 이 샘플러는 GPU가 데이터세트의 동일한 조각에 액세스하는 것을 방지합니다.

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

다음으로, `DistributedDataParallel` 클래스를 사용하여 데이터 병렬 처리를 활성화합니다.

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

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

위의 내용은 데이터 병렬 처리를 사용하는 데 필요한 변경 사항입니다. QML에서는 결과를 저장하고 훈련 진행 상황을 출력하고 싶은 경우가 종종 있습니다. 각 GPU가 저장 및 인쇄 명령을 실행하면 로그에 반복되는 정보가 넘쳐나고 결과가 서로 덮어쓰기됩니다. 이를 방지하기 위해 `rank 0`이 있는 GPU에서만 저장하고 출력할 수 있습니다.

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

Amazon Braket Hybrid Jobs는 SageMaker 분산 데이터 병렬 라이브러리를 위한 `ml.g4dn.12xlarge` 인스턴스 유형을 지원합니다. 하이브리드 작업의 `InstanceConfig` 인수를 통해 인스턴스 유형을 구성합니다. SageMaker 분산 데이터 병렬 라이브러리가 데이터 병렬 처리가 활성화되었음을 인식할 수 있게 하려면, 2개의 추가 하이퍼파라미터를 추가해야 합니다. `"sagemaker_distributed_dataparallel_enabled"`를 `"true"`로 설정하고 `"sagemaker_instance_type"`을 사용 중인 인스턴스 유형으로 설정합니다. 이 두 하이퍼파라미터

는 smdistributed 패키지에서 사용됩니다. 알고리즘 스크립트에서 이를 명시적으로 사용할 필요는 없습니다. Amazon Braket SDK에서는 편리한 키워드 인수 distribution를 제공합니다. 하이브리드 작업 생성에서 distribution="data_parallel"을 사용하면 Amazon Braket SDK가 자동으로 2개의 하이퍼파라미터를 삽입합니다. Amazon Braket API를 사용하는 경우 이 두 하이퍼파라미터를 포함해야 합니다.

인스턴스 및 데이터 병렬 처리가 구성되었으므로 이제 하이브리드 작업을 제출할 수 있습니다. ml.g4dn.12xlarge 인스턴스에는 4개의 GPUs. instanceCount=1을 설정하면 워크로드가 인스턴스의 8개 GPU에 분산됩니다. instanceCount을 1보다 크게 설정하면 워크로드가 모든 인스턴스에서 사용 가능한 GPU에 분산됩니다. 여러 인스턴스를 사용하는 경우 사용 시간에 따라 각 인스턴스에 요금이 부과됩니다. 예를 들어, 4개의 인스턴스를 사용하는 경우 워크로드를 동시에 실행하는 인스턴스가 4개이므로 청구 가능한 시간은 인스턴스당 실행 시간의 4배입니다.

```
instance_config = InstanceConfig(instanceType='ml.g4dn.12xlarge',
                                 instanceCount=1,
)

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

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

> **Note**
>
> 위의 하이브리드 작업 생성에서 train_dp.py는 데이터 병렬 처리를 사용하기 위한 수정된 알고리즘 스크립트입니다. 데이터 병렬 처리는 위 섹션에 따라 알고리즘 스크립트를 수정할 때만 올바르게 작동합니다. 데이터 병렬 처리 옵션이 올바르게 수정된 알고리즘 스크립트 없이 활성화된 경우, 하이브리드 작업에서 오류가 발생하거나 각 GPU가 동일한 데이터 조각을 반복적으로 처리하게 되어 비효율적일 수 있습니다.

올바르게 사용하면 여러 인스턴스를 사용하면 시간과 비용이 크게 감소할 수 있습니다. 자세한 내용은 예제 노트북을 참조하세요.

Amazon Braket에서 CUDA-Q 사용

NVIDIA's CUDA-Q는 CPU, GPU 및 QPU를 결합한 하이브리드 양자 알고리즘을 프로그래밍하기 위해 설계된 소프트웨어 라이브러리입니다. 이 라이브러리는 통합 프로그래밍 모델을 제공하여 개발자가 단일 프로그램 내에서 고전 명령과 양자 명령을 모두 표현할 수 있게 함으로써 워크플로를 간소화합니다. CUDA-Q는 기본 제공 CPU 및 GPU 시뮬레이터를 통해 양자 프로그램 시뮬레이션과 런타임을 가속화합니다. CUDA-Q는 네이티브 Braket 노트북 인스턴스(NBI)와 Amazon Braket Hybrid Jobs에서 사용할 수 있습니다.

이 섹션의 내용:

- NBI에서의 CUDA-Q
- 하이브리드 작업에서의 CUDA-Q

NBI에서의 CUDA-Q

CUDA-Q는 기본적으로 Braket NBI 환경에 설치됩니다. Jupyter 시작 관리자 페이지로 이동하고 CUDA-Q 및 Braket 타일을 선택하여 CUDA-Q 예제 노트북을 열 수 있습니다. 그러면 기본 창에서 예제 노트북 `0_hello_cudaq_jobs.ipynb`이 열립니다. 자세한 CUDA-Q 예제는 `nvidia_cuda_q/` 디렉터리의 왼쪽 패널을 참조하세요.

NBI에 설치된 CUDA-Q 또는 다른 타사 패키지의 버전을 확인할 수도 있습니다. 예를 들어 노트북 코드 셀에서 다음 명령을 실행하여 환경에 설치된 CUDA-Q, Qiskit, PennyLane 및 Braket 패키지의 버전을 확인할 수 있습니다.

```
%pip freeze | grep -i -e cudaq -e qiskit -e pennylane -e braket
```

하이브리드 작업에서의 CUDA-Q

Amazon Braket Hybrid Jobs에서 CUDA-Q를 사용하면 유연한 온디맨드 컴퓨팅 환경을 제공할 수 있습니다. 계산 인스턴스는 워크로드 기간 동안에만 실행되므로 사용한 만큼만 비용을 지불하면 됩니다. Amazon Braket Hybrid Jobs는 또한 확장 가능한 환경을 제공합니다. 사용자는 프로토타이핑 및 테스트를 위해 작은 인스턴스로 시작한 다음, 전체 실험을 위해 더 큰 워크로드를 처리할 수 있는 더 큰 인스턴스로 확장할 수 있습니다.

Amazon Braket Hybrid Jobs는 CUDA-Q의 잠재력을 극대화하는 데 필수적인 GPU를 지원합니다. GPU는 CPU 기반 시뮬레이터에 비해 양자 프로그램 시뮬레이션의 속도를 크게 높입니다. 특히 큐비트 수가 많은 회로 작업 시 더욱 그렇습니다. Amazon Braket Hybrid Jobs에서 CUDA-Q를 사용하면 병렬화가 간단해집니다. 하이브리드 작업은 여러 계산 노드에 걸쳐 회로 샘플링 및 관찰 가능 항목 평가의 분산을 단순화합니다. 이렇게 원활한 CUDA-Q 워크로드의 병렬화를 통해 사용자는 대규모 실험을 위한 인프라를 설정하는 대신 워크로드 개발에 더 집중할 수 있습니다.

시작하려면 Amazon Braket 예제 GitHub에서 CUDA-Q 스타터 예제를 참조하여 Braket에서 제공하는 CUDA-Q 하이브리드 작업 컨테이너를 사용하세요.

다음 코드 조각은 Amazon Braket Hybrid Jobs를 사용하여 CUDA-Q 프로그램을 실행하는 `hello-world` 예제입니다.

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

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

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

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

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

    return measurement_probabilities
```

위 예제에서는 CPU 시뮬레이터에서 벨 회로를 시뮬레이션합니다. 이 예제는 렙톱 또는 Braket Jupyter Notebook에서 로컬로 실행됩니다. `local=True` 설정으로 인해 이 스크립트를 실행하면 로컬 환경에서 컨테이너가 시작되어 테스트 및 디버깅을 위한 CUDA-Q 프로그램을 실행합니다. 테스트를 완료한 후 `local=True` 플래그를 제거하고 AWS에서 작업을 실행할 수 있습니다. 자세한 내용은 Amazon Braket Hybrid Jobs 작업을 참조하세요.

워크로드의 큐비트 수가 많거나 회로 수가 많거나 반복 횟수가 많은 경우, instance_config 설정을 지정하여 더 강력한 CPU 컴퓨팅 리소스를 사용할 수 있습니다. 다음 코드 조각은 hybrid_job 데코레이터에서 instance_config 설정을 구성하는 방법을 보여줍니다. 지원되는 인스턴스 유형에 대한 자세한 내용은 하이브리드 작업 인스턴스 구성을 참조하세요. 인스턴스 유형 목록은 Amazon EC2 인스턴스 유형을 참조하세요.

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

더 까다로운 워크로드의 경우 CUDA-Q GPU 시뮬레이터에서 워크로드를 실행할 수 있습니다. GPU 시뮬레이터를 활성화하려면 백엔드 이름 nvidia를 사용합니다. nvidia 백엔드는 CUDA-Q GPU 시뮬레이터로 작동합니다. 다음으로, NVIDIA GPU를 지원하는 Amazon EC2 인스턴스 유형을 선택합니다. 다음 코드 조각은 GPU로 구성된 hybrid_job 데코레이터를 보여줍니다.

```
@hybrid_job(
    device="local:nvidia/nvidia",
    image_uri=image_uri,
    instance_config=InstanceConfig(instanceType="ml.g4dn.xlarge"),
)
def my_job_script():
    ...
```

Amazon Braket Hybrid Jobs 및 NBI는 CUDA-Q를 사용한 병렬 GPU 시뮬레이션을 지원합니다. 여러 관찰 가능 항목 또는 여러 회로의 평가를 병렬화하여 워크로드의 성능을 높일 수 있습니다. 여러 관찰 가능 항목을 병렬화하려면 알고리즘 스크립트를 다음과 같이 변경합니다.

nvidia 백엔드의 mgpu 옵션을 설정합니다. 이는 관찰 가능 항목을 병렬화하는 데 필요합니다. 병렬화는 GPU 간 통신에 MPI를 사용하므로, MPI는 실행 전에 초기화되고 실행 후에 종료되어야 합니다.

다음으로, execution=cudaq.parallel.mpi를 설정하여 실행 모드를 지정합니다. 다음 코드 조각은 이러한 변경 사항을 보여줍니다.

```
cudaq.set_target("nvidia", option="mqpu")
cudaq.mpi.initialize()
result = cudaq.observe(
```

```
    kernel, hamiltonian, shots_count=n_shots, execution=cudaq.parallel.mpi
)
cudaq.mpi.finalize()
```

다음 코드 조각에서와 같이 hybrid_job 데코레이터에서 여러 GPU를 호스팅하는 인스턴스 유형을 지정합니다.

```
@hybrid_job(
    device="local:nvidia/nvidia-mqpu",
    instance_config=InstanceConfig(instanceType="ml.g4dn.12xlarge", instanceCount=1),
    image_uri=image_uri,
)
def parallel_observables_gpu_job(sagemaker_mpi_enabled=True):
    ...
```

Amazon Braket 예제 GitHub의 병렬 시뮬레이션 노트북은 GPU 백엔드에서 양자 프로그램 시뮬레이션을 실행하고 관찰 가능 항목 및 회로 배치에 대한 병렬 시뮬레이션을 수행하는 방법을 보여주는 종단 간 예제를 제공합니다.

양자 컴퓨터에서 워크로드 실행

시뮬레이터 테스트를 완료한 후 QPU에서 실행 중인 실험으로 전환할 수 있습니다. 대상을 IQM, IonQ 또는 Rigetti 디바이스와 같은 Amazon Braket QPU로 바꾸기만 하면 됩니다. 다음 코드 조각은 대상을 IQM Garnet 디바이스로 설정하는 방법을 보여줍니다. 사용 가능한 QPU 목록은 Amazon Braket 콘솔을 참조하세요.

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

하이브리드 작업에 대한 자세한 내용은 개발자 안내서의 Amazon Braket Hybrid Jobs 작업을 참조하세요. CUDA-Q에 대한 자세한 내용은 NVIDIA CUDA-Q 설명서를 참조하세요.

Amazon Braket 문제 해결

이 섹션의 문제 해결 정보와 솔루션을 사용하여 Amazon Braket 관련 문제를 해결합니다.

이 섹션의 내용:

AccessDeniedException

Braket을 활성화하거나 사용할 때 AccessDeniedException을 수신하는 경우 제한된 역할에 액세스 권한이 없는 리전에서 Braket을 활성화하거나 사용하려 시도하고 있을 가능성이 높습니다.

이러한 경우 내부 AWS 관리자에게 문의하여 다음 중 어떤 조건이 적용되는지 알아보십시오.

- 리전에 대한 액세스를 금지하는 역할 제한이 있는 경우
- 사용하려는 역할에 Braket 사용이 허용된 경우

Braket을 사용할 때 역할이 지정된 리전에 액세스할 수 없는 경우 해당 리전의 디바이스를 사용할 수 없습니다.

CreateQuantumTask 연산을 직접적으로 호출할 때 오류(ValidationException)가 발생함

`An error occurred (ValidationException) when calling the CreateQuantumTask operation: Caller doesn't have access to amazon-braket-…` 과 유사한 오류가 발생하는 경우 기존 s3_folder를 참조하고 있는지 확인합니다. Braket은 새 Amazon S3 버킷과 접두사를 자동으로 생성하지 않습니다.

API에 직접 액세스하고 `Failed to create quantum task: Caller doesn't have access to s3://MY_BUCKET`과 유사한 오류가 수신되는 경우: Amazon S3 버킷 경로에 `s3://`이 포함되어 있지 않은지 확인합니다.

SDK 기능이 작동하지 않음

Python 버전은 3.10 이상이어야 합니다. Amazon Braket Hybrid Jobs의 경우 Python 3.12를 사용하는 것이 좋습니다.

SDK 및 스키마가 최신 상태인지 확인합니다. 노트북 또는 Python 편집기에서 SDK를 업데이트하려면 다음 명령을 실행합니다.

```
pip install amazon-braket-sdk --upgrade --upgrade-strategy eager
```

스키마를 업데이트하려면 다음 명령을 실행합니다.

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

자체 클라이언트에서 Amazon Braket에 액세스하는 경우 AWS 리전이 Amazon Braket에서 지원하는 리전으로 설정되어 있는지 확인합니다.

ServiceQuotaExceededException으로 인해 하이브리드 작업이 실패함

대상 시뮬레이터 디바이스의 동시 양자 작업 제한을 초과하면 Amazon Braket 시뮬레이터에 대해 양자 작업을 실행하는 하이브리드 작업이 생성되지 않을 수 있습니다. 서비스 제한에 대한 자세한 내용은 할당량 주제를 참조하세요.

계정의 여러 하이브리드 작업에서 시뮬레이터 디바이스에 대해 동시 작업을 실행하는 경우 이 오류가 발생할 수 있습니다.

특정 시뮬레이터 디바이스에 대한 동시 양자 작업 수를 보려면 다음 코드 예제와 같이 `search-quantum-tasks` API를 사용합니다.

```
DEVICE_ARN=arn:aws:braket:::device/quantum-simulator/amazon/sv1
task_list=""
for status_value in "CREATED" "QUEUED" "RUNNING" "CANCELLING"; do
```

```
    tasks=$(aws braket search-quantum-tasks --filters
 name=status,operator=EQUAL,values=${status_value}
 name=deviceArn,operator=EQUAL,values=$DEVICE_ARN --max-results 100 --query
 'quantumTasks[*].quantumTaskArn' --output text)
    task_list="$task_list $tasks"
done;
echo "$task_list" | tr -s ' \t' '[\n*]' | sort | uniq
```

Amazon CloudWatch 지표: Braket > 내 디바이스를 사용하여 디바이스에 대해 생성된 양자 작업을 볼 수도 있습니다.

이러한 오류가 발생하지 않도록 하려면 다음과 같이 하세요.

1. 시뮬레이터 디바이스의 동시 양자 작업 수에 대한 서비스 할당량 증가를 요청합니다. 이는 SV1 디바이스에만 적용됩니다.
2. 코드의 `ServiceQuotaExceeded` 예외를 처리하고 다시 시도합니다.

노트북 인스턴스에서 구성 요소가 작동을 멈춤

노트북의 일부 구성 요소가 작동을 멈추면 다음을 시도하세요.

1. 생성하거나 수정한 노트북을 로컬 드라이브에 다운로드합니다.
2. 노트북 인스턴스를 중지합니다.
3. 노트북 인스턴스를 삭제합니다.
4. 다른 이름으로 새 노트북 인스턴스를 생성합니다.
5. 노트북을 새 인스턴스에 업로드합니다.

Python 3.12 업그레이드 문제 해결

발효일: 2026년 1월 21일

개요

2026년 1월 21일부터 Amazon Braket은 모든 노트북 인스턴스 및 관리형 컨테이너 이미지(기본, CUDA-Q, TensorFlow 및 PyTorch)에 대해 Python 런타임을 3.10에서 3.12로 업그레이드합니다. 이 가이드에서는 일반적인 호환성 문제에 대한 솔루션을 제공합니다.

이 섹션의 내용:

일반 오류 메시지

SDK Python 버전 불일치 오류

오류:

```
RuntimeError: Python version must match between local environment and container. Client
 is running Python 3.10 locally, but container uses Python 3.12.
```

원인: Braket SDK가 노트북이 Python 3.10을 실행하고 있지만 하이브리드 작업 컨테이너가 Python 3.12를 실행하고 있음을 감지했습니다.

해결 방법: 노트북을 Python 3.12로 업그레이드하거나 Python 3.10 컨테이너에 고정합니다.

Cloudpickle 직렬화 오류

오류:

```
TypeError: code() argument 13 must be str, not int
```

원인: SDK 검증을 우회하면 Python 3.12의 CodeType 생성자 변경으로 인해 cloudpickle이 Python 3.10과 3.12 사이의 코드를 직렬화하지 못합니다.

해결 방법: 노트북과 컨테이너가 동일한 Python 버전을 사용하는지 확인합니다.

Braket 관리형 노트북

Python 3.10에서 Braket 노트북 인스턴스를 실행하고 하이브리드 작업을 제출하는 경우 작업 컨테이너가 이제 기본적으로 Python 3.12를 사용하기 때문에 버전 불일치 오류가 발생합니다.

여기에는 두 가지 옵션이 있습니다.

1. [권장] Python 3.12를 사용하여 새 노트북 인스턴스 생성 - Braket 노트북 인스턴스 업그레이드를 참조하세요.
2. Python 3.10 컨테이너에 고정 - 하이브리드 작업 데코레이터 참조

하이브리드 작업 데코레이터

`@hybrid_job` 데코레이터를 사용하려면 환경의 Python 버전이 컨테이너의 Python 버전과 일치해야 합니다.

옵션 1: Python 3.12 컨테이너 사용(권장)

환경을 Python 3.12로 업그레이드한 경우 최신 태그(기본 동작)를 사용합니다.

옵션 2: Python 3.10 컨테이너 사용

Python 3.10을 유지해야 하는 경우 `@hybrid_job` 데코레이터에서 `image_uri` 파라미터를 명시적으로 지정합니다.

Python 3.10 컨테이너 태그:

이미지 이름	태그
기본	1.0-cpu-py310-ubuntu22.04
CUDA-Q	0.12.0-cpu-py310-0.12.0
PyTorch	2.2.0-gpu-py310-cu121-ubuntu20.04
TensorFlow	2.14.1-gpu-py310-cu118-ubuntu20.04

다음 예제는 us-west-2 리전에 대한 것입니다.

전체 이미지 URIs:

```
Base:       292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-base-jobs:1.0-cpu-py310-ubuntu22.04
CUDA-Q:     292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-cudaq-jobs:0.12.0-cpu-py310-0.12.0
```

```
PyTorch:    292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-pytorch-
jobs:2.2.0-gpu-py310-cu121-ubuntu20.04
TensorFlow: 292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-tensorflow-
jobs:2.14.1-gpu-py310-cu118-ubuntu20.04
```

예:

```
from braket.jobs.hybrid_job import hybrid_job
from braket.devices import Devices

device_arn = Devices.Amazon.SV1

@hybrid_job(
    device=device_arn,
    image_uri="292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-base-
jobs:1.0-cpu-py310-ubuntu22.04"
)
def my_job():
    pass
```

Note

- Python 3.10 컨테이너는 계속 사용할 수 있지만 업데이트는 받지 않습니다.
- 알고리즘 스크립트의 환경 정의를 참조하세요.

기존 Bring-Your-Own-Container(BYOC)

Dockerfile이 최신 태그와 함께 Braket 관리형 이미지를 사용하는 경우 2026년 1월 21일 이후에 재구축하면 Python 3.12 지원 이미지가 풀링됩니다.

Python 3.10 지원 Braket 관리형 이미지를 유지하려면 Dockerfile을 업데이트합니다.

이전(업그레이드 후 Python 3.12 획득):

```
FROM 292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-base-jobs:latest
FROM 292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-cudaq-jobs:latest
FROM 292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-tensorflow-jobs:latest
FROM 292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-pytorch-jobs:latest
```

이후(Python 3.10 유지):

```
FROM 292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-base-jobs:1.0-cpu-
py310-ubuntu22.04
FROM 292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-cudaq-jobs:0.12.0-cpu-
py310-0.12.0
FROM 292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-pytorch-jobs:2.2.0-gpu-
py310-cu121-ubuntu20.04
FROM 292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-tensorflow-jobs:2.14.1-
gpu-py310-cu118-ubuntu20.04
```

Braket 노트북 인스턴스 업그레이드

다음 단계에 따라 Python 3.12로 업그레이드합니다.

> **Important**
>
> 노트북 인스턴스를 삭제하기 전에 보관하려는 모든 노트북과 파일을 다운로드했는지 확인합니다. 삭제 후에는이 데이터를 복구할 수 없습니다.

1. 생성하거나 수정한 노트북을 로컬 드라이브에 다운로드합니다.
2. 노트북 인스턴스를 중지합니다.
3. 노트북 인스턴스를 삭제합니다.
4. 다른 이름으로 새 노트북 인스턴스를 생성합니다.
5. 노트북을 새 인스턴스에 업로드합니다.

OpenQASM 문제 해결

이 섹션에서는 OpenQASM 3.0 사용 중 오류가 발생할 때 유용할 수 있는 문제 해결 지침을 제공합니다.

이 섹션의 내용:

- 문 포함 오류
- 비연속 qubits 오류
- 물리적 qubits를 가상 qubits와 혼합 오류

- 결과 유형 요청 및 동일한 프로그램에서 qubits 측정 오류
- 고전적 및 qubit 레지스터 제한 초과 오류
- 상자 앞에 축어적 pragma 없음 오류
- 축어적 상자에 네이티브 게이트 누락 오류
- 축어적 상자에 물리적 qubits 누락 오류
- 축어적 pragma에 “브래킷” 누락 오류
- 단일 qubits는 인덱싱할 수 없음 오류
- 두 qubit 게이트의 물리적 qubits가 연결되지 않음 오류
- 로컬 시뮬레이터 지원 경고

문 포함 오류

현재 Braket에 OpenQASM 프로그램에 포함할 표준 게이트 라이브러리 파일이 없습니다. 예를 들어 다음 예제에서는 구문 분석기 오류가 발생합니다.

```
OPENQASM 3;
include "standardlib.inc";
```

이 코드는 다음 오류 메시지를 생성합니다. `No terminal matches '"' in the current parser context, at line 2 col 17.`

비연속 qubits 오류

디바이스 기능에서 `requiresContiguousQubitIndices`가 `true`로 설정된 디바이스에서 비연속 qubits를 사용하면 오류가 발생합니다.

시뮬레이터 및 IonQ에서 양자 작업을 실행할 때 다음 프로그램이 오류를 트리거합니다.

```
OPENQASM 3;

qubit[4] q;

h q[0];
cnot q[0], q[2];
cnot q[0], q[3];
```

이 코드는 다음 오류 메시지를 생성합니다. `Device requires contiguous qubits. Qubit register q has unused qubits q[1], q[4].`

물리적 qubits를 가상 qubits와 혼합 오류

동일한 프로그램에서 물리적 qubits를 가상 qubits와 혼합하는 것은 허용되지 않으며 오류가 발생합니다. 다음 코드는 오류를 생성합니다.

```
OPENQASM 3;

qubit[2] q;
cnot q[0], $1;
```

이 코드는 다음 오류 메시지를 생성합니다. `[line 4] mixes physical qubits and qubits registers.`

결과 유형 요청 및 동일한 프로그램에서 qubits 측정 오류

결과 유형을 요청하고 동일한 프로그램에서 qubits가 명시적으로 측정되면 오류가 발생합니다. 다음 코드는 오류를 생성합니다.

```
OPENQASM 3;

qubit[2] q;

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

#pragma braket result expectation x(q[0]) @ z(q[1])
```

이 코드는 다음 오류 메시지를 생성합니다. `Qubits should not be explicitly measured when result types are requested.`

고전적 및 qubit 레지스터 제한 초과 오류

고전적 레지스터 하나와 qubit 레지스터 하나만 허용됩니다. 다음 코드는 오류를 생성합니다.

```
OPENQASM 3;
```

```
qubit[2] q0;
qubit[2] q1;
```

이 코드는 다음 오류 메시지를 생성합니다. `[line 4] cannot declare a qubit register. Only 1 qubit register is supported.`

상자 앞에 축어적 pragma 없음 오류

모든 상자 앞에 축어적 pragma를 붙여야 합니다. 다음 코드는 오류를 생성합니다.

```
box{
rx(0.5) $0;
}
```

이 코드는 다음 오류 메시지를 생성합니다. `In verbatim boxes, native gates are required. x is not a device native gate.`

축어적 상자에 네이티브 게이트 누락 오류

축어적 상자에는 네이티브 게이트와 물리적 qubits가 있어야 합니다. 다음 코드는 네이티브 게이트 오류를 생성합니다.

```
#pragma braket verbatim
box{
x $0;
}
```

이 코드는 다음 오류 메시지를 생성합니다. `In verbatim boxes, native gates are required. x is not a device native gate.`

축어적 상자에 물리적 qubits 누락 오류

축어적 상자에는 물리적 qubits가 있어야 합니다. 다음 코드는 누락된 물리적 qubits 오류를 생성합니다.

```
qubit[2] q;

#pragma braket verbatim
box{
rx(0.1) q[0];
```

```
}
```

이 코드는 다음 오류 메시지를 생성합니다. `Physical qubits are required in verbatim box.`

축어적 pragma에 “브래킷” 누락 오류

축어적 pragma에 “브래킷”을 포함해야 합니다. 다음 코드는 오류를 생성합니다.

```
#pragma braket verbatim          // Correct
#pragma verbatim                 // wrong
```

이 코드는 다음 오류 메시지를 생성합니다. `You must include “braket” in the verbatim pragma`

단일 qubits는 인덱싱할 수 없음 오류

단일 qubits는 인덱싱할 수 없습니다. 다음 코드는 오류를 생성합니다.

```
OPENQASM 3;

qubit q;
h q[0];
```

이 코드는 오류를 생성합니다. `[line 4] single qubit cannot be indexed.`

그러나 단일 qubit 배열은 다음과 같이 인덱싱할 수 있습니다.

```
OPENQASM 3;

qubit[1] q;
h q[0];   // This is valid
```

두 qubit 게이트의 물리적 qubits가 연결되지 않음 오류

물리적 qubits를 사용하려면, 먼저 `device.properties.action[DeviceActionType.OPENQASM].supportPhysicalQubits`를 확인하여 디바이스가 물리적 qubits를 사용하는지 확인한 다음 `device.properties.paradigm.connectivity.connectivityGraph` 또

는device.properties.paradigm.connectivity.fullyConnected를 확인하여 연결 그래프를 확인합니다.

```
OPENQASM 3;

cnot $0, $14;
```

이 코드는 다음 오류 메시지를 생성합니다. `[line 3] has disconnected qubits 0 and 14`

로컬 시뮬레이터 지원 경고

LocalSimulator는 QPU 또는 온디맨드 시뮬레이터에서 사용할 수 없는 OpenQASM의 고급 기능을 지원합니다. 다음 예제와 같이 프로그램에 LocalSimulator에만 해당하는 언어 기능이 포함된 경우 경고가 표시됩니다.

```
qasm_string = """
qubit[2] q;

h q[0];
ctrl @ x q[0], q[1];
"""
qasm_program = Program(source=qasm_string)
```

이 코드는 `이 프로그램은 LocalSimulator에서만 지원되는 OpenQASM 언어 기능을 사용합니다. 이러한 기능 중 일부는 QPU 또는 온디맨드 시뮬레이터에서 지원되지 않을 수 있습니다.

지원되는 OpenQASM 기능에 대한 자세한 내용은 로컬 시뮬레이터에서의 OpenQASM에 대한 고급 기능 지원 페이지를 참조하세요.

Amazon Braket의 보안

의 클라우드 보안 AWS 이 최우선 순위입니다. AWS 고객은 보안에 가장 민감한 조직의 요구 사항을 충족하도록 구축된 데이터 센터 및 네트워크 아키텍처의 이점을 누릴 수 있습니다.

보안은 AWS 와 사용자 간의 공동 책임입니다. 공동 책임 모델은 이 사항을 클라우드의 보안 및 클라우드 내 보안으로 설명합니다.

- 클라우드 보안 - AWS 는에서 AWS 서비스를 실행하는 인프라를 보호할 책임이 있습니다 AWS 클라우드. AWS 또한는 안전하게 사용할 수 있는 서비스를 제공합니다. 타사 감사자는 AWS 규정 준수 프로그램 일환으로 보안의 효과를 정기적으로 테스트하고 확인합니다. Amazon Braket에 적용되는 규정 준수 프로그램에 대한 자세한 내용은 규정 준수 프로그램 AWS 제공 범위 내 서비스규정 준수 프로그램 .
- 클라우드의 보안 - 사용자의 책임은 사용하는 AWS 서비스에 따라 결정됩니다. 또한 귀하는 데이터의 민감도, 회사 요구 사항, 관련 법률 및 규정을 비롯한 기타 요소에 대해서도 책임이 있습니다.

이 설명서는 Braket 사용 시 공동 책임 모델을 적용하는 방법을 이해하는 데 도움이 됩니다. 다음 주제에서는 보안 및 규정 준수 목표를 충족하도록 Braket을 구성하는 방법을 보여줍니다. 또한 Braket 리소스를 모니터링하고 보호하는 데 도움이 되는 다른 AWS 서비스를 사용하는 방법을 알아봅니다.

이 섹션의 내용:

보안에 대한 공동 책임

보안은 AWS 와 사용자 간의 공동 책임입니다. 공동 책임 모델은 이를 클라우드의 보안과 클라우드 내 보안으로 설명합니다.

- 클라우드 보안 - AWS 는 AWS 서비스 에서 실행되는 인프라를 보호할 책임이 있습니다 AWS 클라우드. AWS 또한는 안전하게 사용할 수 있는 서비스를 제공합니다. 서드 파티 감사자는 정기적으로 AWS 규정 준수 프로그램의 일환으로 보안 효과를 테스트하고 검증합니다. Amazon Braket에 적용되는 규정 준수 프로그램에 대한 자세한 내용은 AWS 규정 준수 프로그램별 범위 내 서비스를 참조하세요.
- 클라우드의 보안 -이 AWS 인프라에서 호스팅되는 콘텐츠를 제어할 책임은 사용자에게 있습니다. 이 콘텐츠에는 사용하는에 대한 보안 구성 및 관리 작업이 포함되어 AWS 서비스 있습니다.

데이터 보호

AWS 공동 책임 모델 Amazon Braket의 데이터 보호에 적용됩니다. 이 모델에 설명된 대로 AWS 는 모든를 실행하는 글로벌 인프라를 보호할 책임이 있습니다 AWS 클라우드. 사용자는 이 인프라에 호스팅되는 콘텐츠에 대한 통제 권한을 유지할 책임이 있습니다. 사용하는 AWS 서비스 의 보안 구성과 관리 태스크에 대한 책임도 사용자에게 있습니다. 데이터 프라이버시에 관한 자세한 내용은 데이터 프라이버시 FAQ를 참조하세요. 유럽의 데이터 보호에 대한 자세한 내용은 AWS 보안 블로그의 AWS 공동 책임 모델 및 GDPR 블로그 게시물을 참조하세요.

데이터 보호를 위해 자격 증명을 보호하고 AWS 계정 AWS IAM Identity Center 또는 AWS Identity and Access Management (IAM)를 사용하여 개별 사용자를 설정하는 것이 좋습니다. 이렇게 하면 개별 사용자에게 자신의 직무를 충실히 이행하는 데 필요한 권한만 부여됩니다. 또한 다음과 같은 방법으로 데이터를 보호하는 것이 좋습니다.

- 각 계정에 다중 인증(MFA)을 사용합니다.
- SSL/TLS를 사용하여 AWS 리소스와 통신합니다. TLS 1.2는 필수이며 TLS 1.3을 권장합니다.
- 를 사용하여 API 및 사용자 활동 로깅을 설정합니다 AWS CloudTrail. CloudTrail 추적을 사용하여 AWS 활동을 캡처하는 방법에 대한 자세한 내용은 AWS CloudTrail 사용 설명서의 CloudTrail 추적 작업을 참조하세요.
- 내부의 모든 기본 보안 제어와 함께 AWS 암호화 솔루션을 사용합니다 AWS 서비스.
- Amazon S3에 저장된 민감한 데이터를 검색하고 보호하는 데 도움이 되는 Amazon Macie와 같은 고급 관리형 보안 서비스를 사용합니다.

- 명령줄 인터페이스 또는 API를 AWS 통해에 액세스할 때 FIPS 140-3 검증 암호화 모듈이 필요한 경우 FIPS 엔드포인트를 사용합니다. 사용 가능한 FIPS 엔드포인트에 대한 자세한 내용은 연방 정보 처리 표준(FIPS) 140-3을 참조하세요.

고객의 이메일 주소와 같은 기밀 정보나 중요한 정보는 태그나 이름 필드와 같은 자유 형식 텍스트 필드에 입력하지 않는 것이 좋습니다. 여기에는 Amazon Braket 또는 기타 AWS 서비스 에서 콘솔 AWS CLI, API 또는 AWS SDKs를 사용하여 작업하는 경우가 포함됩니다. 이름에 사용되는 태그 또는 자유 형식 텍스트 필드에 입력하는 모든 데이터는 청구 또는 진단 로그에 사용될 수 있습니다. 외부 서버에 URL을 제공할 때 해당 서버에 대한 요청을 검증하기 위해 자격 증명을 URL에 포함해서는 안 됩니다.

데이터 보존

90일이 지나면 Amazon Braket은 양자 작업과 연결된 모든 양자 작업 ID 및 기타 메타데이터를 자동으로 제거합니다. 이 데이터 보존 정책의 결과로, 해당 작업 및 결과는 Amazon Braket 콘솔에서 검색을 통해 더 이상 검색할 수 없게 되지만, 해당 데이터는 S3 버킷에 계속 저장됩니다.

S3 버킷에 90일 이상 저장된 과거 양자 작업 및 결과에 액세스해야 하는 경우, 해당 데이터와 연결된 작업 ID 및 기타 메타데이터를 별도로 기록해 두어야 합니다. 90일 전에 정보를 저장해야 합니다. 저장된 정보를 사용하여 기록 데이터를 검색할 수 있습니다.

Amazon Braket에 대한 액세스 관리

이 장에서는 Amazon Braket을 실행하거나 특정 사용자 및 역할의 액세스를 제한하는 데 필요한 권한을 설명합니다. 계정의 모든 사용자 또는 역할에 필요한 권한을 부여(또는 거부)할 수 있습니다. 이렇게 하려면 다음 섹션에 설명된 대로 계정의 해당 사용자 또는 역할에 적절한 Amazon Braket 정책을 연결합니다.

사전 조건으로, Amazon Braket을 활성화해야 합니다. Braket을 활성화하려면 (1) 관리자 권한이 있거나 (2) AmazonBraketFullAccess 정책이 할당되고 Amazon Simple Storage Service(Amazon S3) 버킷을 생성할 권한이 있는 사용자 또는 역할로 로그인해야 합니다.

이 섹션의 내용:

- Amazon Braket 리소스
- 노트북 및 역할
- AWS Amazon Braket에 대한 관리형 정책
- 특정 디바이스에 대한 사용자 액세스 제한
- 특정 노트북 인스턴스에 대한 사용자 액세스 제한

- 특정 S3 버킷에 대한 사용자 액세스 제한

Amazon Braket 리소스

Braket은 quantum-task 리소스라는 한 가지 유형의 리소스를 생성합니다. 이 AWS 리소스 유형의 리소스 이름(ARN)은 다음과 같습니다.

- 리소스 이름: AWS::Service::Braket
- ARN 정규식: arn:${Partition}:braket:${Region}:${Account}:quantum-task/${RandomId}

노트북 및 역할

Braket에서 노트북 리소스 유형을 사용할 수 있습니다. 노트북은 Braket이 공유할 수 있는 Amazon SageMaker AI 리소스입니다. Braket에서 노트북을 사용하려면 이름이 `AmazonBraketServiceSageMakerNotebook`으로 시작하는 IAM 역할을 지정해야 합니다.

노트북을 생성하려면 관리자 권한이 있거나 다음과 같은 인라인 정책이 연결된 역할을 사용해야 합니다.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "CreateTheRole",
      "Effect": "Allow",
      "Action": "iam:CreateRole",
      "Resource": "arn:aws:iam::*:role/service-role/
AmazonBraketServiceSageMakerNotebookRole*"
    },
    {
      "Sid": "CreateThePolicy",
      "Effect": "Allow",
      "Action": "iam:CreatePolicy",
      "Resource": [
        "arn:aws:iam::*:policy/service-role/
AmazonBraketServiceSageMakerNotebookAccess*",
```

```
        "arn:aws:iam::*:policy/service-role/
AmazonBraketServiceSageMakerNotebookRole*"
      ]
    },
    {
      "Sid": "AttachTheRolePolicy",
      "Effect": "Allow",
      "Action": "iam:AttachRolePolicy",
      "Resource": "arn:aws:iam::*:role/service-role/
AmazonBraketServiceSageMakerNotebookRole*",
      "Condition": {
        "ArnLike": {
          "iam:PolicyARN": [
            "arn:aws:iam::aws:policy/AmazonBraketFullAccess",
            "arn:aws:iam::*:policy/service-role/
AmazonBraketServiceSageMakerNotebookAccess*",
            "arn:aws:iam::*:policy/service-role/
AmazonBraketServiceSageMakerNotebookRole*"
          ]
        }
      }
    }
  ]
}
```

역할을 생성하려면 노트북 생성 페이지에 제공된 단계를 따르거나 관리자가 자동으로 역할을 생성하도록 하세요. AmazonBraketFullAccess 정책이 역할에 연결되었는지 확인합니다.

역할을 생성한 후 나중에 시작하는 모든 노트북에 해당 역할을 재사용할 수 있습니다.

AWS Amazon Braket에 대한 관리형 정책

AWS 관리형 정책은에서 생성하고 관리하는 독립 실행형 정책입니다 AWS. AWS 관리형 정책은 사용자, 그룹 및 역할에 권한 할당을 시작할 수 있도록 많은 일반적인 사용 사례에 대한 권한을 제공하도록 설계되었습니다.

AWS 관리형 정책은 모든 AWS 고객이 사용할 수 있으므로 특정 사용 사례에 대해 최소 권한 권한을 부여하지 않을 수 있습니다. 사용 사례에 고유한 고객 관리형 정책을 정의하여 권한을 줄이는 것이 좋습니다.

AWS 관리형 정책에 정의된 권한은 변경할 수 없습니다. 가 관리형 정책에 정의된 권한을 AWS 업데이트하는 AWS 경우 업데이트는 정책이 연결된 모든 보안 주체 자격 증명(사용자, 그룹 및 역할)에 영향을 미칩니다. AWS AWS 서비스 는 새가 시작되거나 기존 서비스에 새 API 작업을 사용할 수 있게 될 때 AWS 관리형 정책을 업데이트할 가능성이 높습니다.

자세한 내용은 IAM 사용자 가이드의 AWS 관리형 정책을 참조하세요.

주제

- AWS 관리형 정책: AmazonBraketFullAccess
- AWS 관리형 정책: AmazonBraketJobsExecutionPolicy
- AWS 관리형 정책: AmazonBraketServiceRolePolicy
- AWS 관리형 정책에 대한 Amazon Braket 업데이트

AWS 관리형 정책: AmazonBraketFullAccess

AmazonBraketFullAccess 정책은 다음 작업에 대한 권한을 포함하여 Amazon Braket 작업에 대한 권한을 부여합니다.

- Amazon Elastic Container Registry에서 컨테이너 다운로드 - Amazon Braket Hybrid Jobs 기능에 사용되는 컨테이너 이미지를 읽고 다운로드합니다. 컨테이너는 "arn:aws:ecr:::repository/amazon-braket" 형식을 따라야 합니다.
- AWS CloudTrail 로그 유지 - 쿼리 시작 및 중지, 지표 필터 테스트, 로그 이벤트 필터링 외에도 모든 설명, 가져오기 및 나열 작업에 적용됩니다. AWS CloudTrail 로그 파일에는 계정에서 발생하는 모든 Amazon Braket API 활동에 대한 레코드가 포함됩니다.
- 역할을 활용하여 리소스 제어 - 계정에서 서비스 연결 역할을 생성합니다. 서비스 연결 역할은 사용자를 대신하여 AWS 리소스에 액세스할 수 있습니다. Amazon Braket 서비스에서만 사용할 수 있습니다. 또한 IAM 역할을 Amazon Braket `CreateJob` API에 전달하고 역할을 생성하여 해당 역할에 AmazonBraketFullAccess로 범위가 지정된 정책을 연결합니다.
- 로그 그룹, 로그 이벤트 및 쿼리 로그 그룹을 생성하여 계정의 사용 로그 파일 유지 – 계정에서 Amazon Braket 사용에 대한 로깅 정보를 생성, 저장하고 확인합니다. 하이브리드 작업 로그 그룹에 대한 지표를 쿼리합니다. 적절한 Braket 경로를 포함하고 로그 데이터 입력을 허용합니다. CloudWatch에 지표 데이터를 넣습니다.
- Amazon S3 버킷에 데이터를 생성 및 저장하고 모든 버킷 나열 - S3 버킷을 생성하고 계정의 S3 버킷을 나열하며 이름이 amazon-braket-으로 시작하는 계정의 모든 버킷에 객체를 넣고 가져옵니다. 이러한 권한은 Braket이 처리된 양자 작업의 결과가 포함된 파일을 버킷에 넣고 버킷에서 검색하는 데 필요합니다.

- IAM 역할 전달 - IAM 역할을 `CreateJob` API에 전달합니다.
- Amazon SageMaker AI 노트북 - "arn:aws:sagemaker:::notebook-instance/amazon-braket-"에서 리소스로 범위가 지정된 SageMaker 노트북 인스턴스를 생성하고 관리합니다.
- 서비스 할당량 검증 - SageMaker AI 노트북 및 Amazon Braket Hybrid Jobs를 생성하려면 리소스 수가 계정의 할당량을 초과할 수 없습니다.
- 제품 요금 보기 - 워크로드를 제출하기 전에 양자 하드웨어 비용을 검토하고 계획합니다.

이 정책의 권한을 보려면 AWS 관리형 정책 참조에서 AmazonBraketFullAccess를 참조하세요.

AWS 관리형 정책: AmazonBraketJobsExecutionPolicy

AmazonBraketJobsExecutionPolicy 정책은 다음과 같이 Amazon Braket Hybrid Jobs에서 사용되는 실행 역할에 대한 권한을 부여합니다.

- Amazon Elastic Container Registry에서 컨테이너 다운로드 - Amazon Braket Hybrid Jobs 기능에 사용되는 컨테이너 이미지를 읽고 다운로드할 수 있는 권한입니다. 컨테이너는 "arn:aws:ecr:*:*:repository/amazon-braket*" 형식을 따라야 합니다.
- 로그 그룹, 로그 이벤트 및 쿼리 로그 그룹을 생성하여 계정의 사용 로그 파일 유지 – 계정에서 Amazon Braket 사용에 대한 로깅 정보를 생성, 저장하고 확인합니다. 하이브리드 작업 로그 그룹에 대한 지표를 쿼리합니다. 적절한 Braket 경로를 포함하고 로그 데이터 입력을 허용합니다. CloudWatch에 지표 데이터를 넣습니다.
- Amazon S3 버킷에 데이터 저장 – 계정의 S3 버킷을 나열하고 이름이 amazon-braket-으로 시작하는 계정의 모든 버킷에 객체를 넣고 가져옵니다. 이러한 권한은 Braket이 처리된 양자 작업의 결과가 포함된 파일을 버킷에 넣고 버킷에서 검색하는 데 필요합니다.
- IAM 역할 전달 - IAM 역할을 CreateJob API에 전달합니다. 역할은 arn:aws:iam::*:role/service-role/AmazonBraketJobsExecutionRole* 형식을 따라야 합니다.

이 정책의 권한을 보려면 AWS 관리형 정책 참조의 AmazonBraketJobsExecutionPolicy를 참조하세요.

AWS 관리형 정책: AmazonBraketServiceRolePolicy

AmazonBraketServiceRolePolicy 정책은 다음 작업에 대한 권한을 포함하여 Amazon Braket 작업에 대한 권한을 부여합니다.

- Amazon S3 - 계정의 버킷을 나열하고 이름이 `amazon-braket-`으로 시작하는 계정의 모든 버킷에 객체를 넣고 가져올 수 있는 권한입니다.

- Amazon CloudWatch Logs - 로그 그룹을 나열 및 생성하고 연결된 로그 스트림을 생성하며 Amazon Braket용으로 생성된 로그 그룹에 이벤트를 넣을 수 있는 권한입니다.

서비스 연결 역할에 대한 자세한 내용은 Amazon Braket 서비스 연결 역할을 참조하세요.

이 정책의 권한을 보려면 AWS 관리형 정책 참조의 AmazonBraketServiceRolePolicy를 참조하세요.

AWS 관리형 정책에 대한 Amazon Braket 업데이트

다음 표에는이 서비스가 이러한 변경 사항을 추적하기 시작한 시점부터 Amazon Braket의 AWS 관리형 정책 업데이트에 대한 세부 정보가 나와 있습니다.

변경 사항	설명	날짜
AmazonBraketServiceRolePolicy - 리소스 관리 정책	Amazon S3 및 CloudWatch 로그 작업에 "aws:ResourceAccount" : "${aws:PrincipalAccount}" 조건 범위를 추가했습니다.	2025년 7월 11일
AmazonBraketFullAccess - Braket에 대한 전체 액세스 정책	"pricing:GetProducts" 작업을 추가했습니다.	2025년 4월 14일
AmazonBraketFullAccess - Braket에 대한 전체 액세스 정책	S3 작업에 "aws:ResourceAccount": "${aws:PrincipalAccount}" 조건 범위를 추가했습니다.	2025년 3월 7일
AmazonBraketFullAccess - Braket에 대한 전체 액세스 정책	servicequotas:GetServiceQuota 및 cloudwatch:GetMetricData 작업을 추가했습니다.	2023년 3월 24일
AmazonBraketFullAccess - Braket에 대한 전체 액세스 정책	사용된 Amazon S3 버킷을 보고 검사할 수 있는 s3:ListAllMyBuckets 권한을 추가했습니다.	2022년 3월 31일
AmazonBraketFullAccess - Braket에 대한 전체 액세스 정책	Braket이 `service-role/` 경로를 포함하도록 AmazonBraketFullAccess에 대한 iam:PassRole 권한을 조정했습니다.	2021년 11월 29일

변경 사항	설명	날짜
AmazonBraketJobsExecutionPolicy - Amazon Braket Hybrid Jobs에 대한 하이브리드 작업 실행 정책	Braket이 `service-role/` 경로를 포함하도록 하이브리드 작업 실행 역할 ARN을 업데이트했습니다.	2021년 11월 29일
Braket, 변경 사항 추적 시작	Braket이 AWS 관리형 정책에 대한 변경 사항 추적을 시작했습니다.	2021년 11월 29일

특정 디바이스에 대한 사용자 액세스 제한

특정 Braket 디바이스에 대한 사용자 액세스를 제한하려면 특정 IAM 역할에 권한 거부 정책을 추가할 수 있습니다.

다음 작업은 제한될 수 있습니다.

- `CreateQuantumTask` - 지정된 디바이스에서 양자 작업 생성을 거부합니다.
- `CreateJob` - 지정된 디바이스에서 하이브리드 작업 생성을 거부합니다.
- `GetDevice` - 지정된 디바이스의 세부 정보 가져오기를 거부합니다.

다음 예제에서는 AWS 계정 `123456789012`의 모든 QPU 대한 액세스를 제한합니다.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
    "Effect": "Deny",
    "Action": [
      "braket:CreateQuantumTask",
      "braket:CreateJob",
      "braket:GetDevice"
    ],
    "Resource": [
    "arn:aws:braket:*:*:device/qpu/*"
```

```
    ],
    "Condition": {
            "StringEquals": {
                "aws:PrincipalAccount": "123456789012"
            }
      }
    }
  ]
}
```

> **Note**
>
> Braket 콘솔을 통해 디바이스 가용성, 보정 데이터 및 요금과 같은 디바이스 속성에 대한 사용자의 읽기 액세스를 활성화하려면 정책에서 `braket:GetDevice` 작업을 제외합니다.

이 코드를 조정하려면 제한된 디바이스의 Amazon 리소스 번호(ARN)를 이전 예제에 표시된 문자열로 대체합니다. 이 문자열은 리소스 값을 제공합니다. Braket에서 디바이스는 양자 작업을 실행하기 위해 직접적으로 호출할 수 있는 QPU 또는 시뮬레이터를 나타냅니다. 사용 가능한 디바이스는 디바이스 페이지에 나열되어 있습니다. 이러한 디바이스에 대한 액세스를 지정하는 데 사용되는 두 가지 스키마가 있습니다.

- `arn:aws:braket:<region>:*:device/qpu/<provider>/<device_id>`
- `arn:aws:braket:<region>:*:device/quantum-simulator/<provider>/<device_id>`

다음은 다양한 유형의 디바이스 액세스에 대한 예제입니다.

- 모든 리전에서 모든 QPU를 선택: `arn:aws:braket:*:*:device/qpu/*`
- us-west-2 리전에서만 모든 QPU를 선택: `arn:aws:braket:us-west-2:*:device/qpu/*`
- 마찬가지로, us-west-2 리전에서만 모든 QPU를 선택(디바이스는 고객 리소스가 아닌 서비스 리소스이므로): `arn:aws:braket:us-west-2:*:device/qpu/*`
- 모든 온디맨드 시뮬레이터 디바이스에 대한 액세스를 제한: `arn:aws:braket:*:*:device/quantum-simulator/*`
- 특정 공급자의 디바이스(예: Rigetti QPU 디바이스)에 대한 액세스를 제한: `arn:aws:braket:*:*:device/qpu/rigetti/*`

- TN1 디바이스에 대한 액세스를 제한: `arn:aws:braket:*:*:device/quantum-simulator/amazon/tn1`
- 모든 `Create` 작업에 대한 액세스를 제한: `braket:Create*`

특정 노트북 인스턴스에 대한 사용자 액세스 제한

특정 사용자의 특정 Braket 노트북 인스턴스 액세스를 제한하려면 특정 역할, 사용자 또는 그룹에 권한 거부 정책을 추가할 수 있습니다.

다음 예제에서는 정책 변수를 사용하여 액세스 권한이 필요한 사용자에 따라 AWS 계정 `123456789012`이름이 지정된의 특정 노트북 인스턴스를 시작, 중지 및 액세스할 수 있는 권한을 효율적으로 제한합니다(예: 사용자가 라는 노트북 인스턴스에 액세스할 `Alice` 수 있음`amazon-braket-Alice`).

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "DenyCreateDeleteUpdateNotebookInstances",
      "Effect": "Deny",
      "Action": [
        "sagemaker:CreateNotebookInstance",
        "sagemaker:DeleteNotebookInstance",
        "sagemaker:UpdateNotebookInstance",
        "sagemaker:CreateNotebookInstanceLifecycleConfig",
        "sagemaker:DeleteNotebookInstanceLifecycleConfig",
        "sagemaker:UpdateNotebookInstanceLifecycleConfig"
      ],
      "Resource": "*"
    },
    {
      "Sid": "DenyDescribeStartStopNotebookInstances",
      "Effect": "Deny",
      "Action": [
        "sagemaker:DescribeNotebookInstance",
        "sagemaker:StartNotebookInstance",
        "sagemaker:StopNotebookInstance"
      ],
```

```
      "NotResource": [
        "arn:aws:sagemaker:*:123456789012:notebook-instance/amazon-braket-
${aws:username}"
      ]
    },
    {
      "Sid": "DenyNotebookInstanceUrl",
      "Effect": "Deny",
      "Action": [
        "sagemaker:CreatePresignedNotebookInstanceUrl"
      ],
      "NotResource": [
        "arn:aws:sagemaker:*:123456789012:notebook-instance/amazon-braket-
${aws:username}*"
      ]
    }
  ]
}
```

특정 S3 버킷에 대한 사용자 액세스 제한

특정 사용자의 특정 Amazon S3 버컷 액세스를 제한하려면 특정 역할, 사용자 또는 그룹에 거부 정책을 추가할 수 있습니다.

다음 예제에서는 객체를 검색하고 특정 S3 버킷(`arn:aws:s3:::amazon-braket-us-east-1-123456789012-Alice`)에 배치하는 권한을 제한하며 해당 객체를 나열하는 것도 제한합니다.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Action": [
        "s3:ListBucket"
      ],
      "NotResource": [
        "arn:aws:s3:::amazon-braket-us-east-1-123456789012-Alice"
      ]
```

```
    },
    {
      "Effect": "Deny",
      "Action": [
        "s3:GetObject"
      ],
      "NotResource": [
        "arn:aws:s3:::amazon-braket-us-east-1-123456789012-Alice/*"
      ]
    }
  ]
}
```

특정 노트북 인스턴스의 버킷에 대한 액세스를 제한하려면 이전 정책을 노트북 실행 역할에 추가할 수 있습니다.

Amazon Braket 서비스 연결 역할

Amazon Braket을 활성화하면 계정에 서비스 연결 역할이 생성됩니다.

서비스 연결 역할은 IAM 역할의 고유한 유형으로, 이 경우 Amazon Braket에 직접 연결됩니다. Amazon Braket 서비스 연결 역할은 Braket이 사용자를 대신하여 다른 AWS 서비스를 직접적으로 호출할 때 필요한 모든 권한을 포함하도록 미리 정의되어 있습니다.

서비스 연결 역할을 사용하면 필요한 권한을 수동으로 추가할 필요가 없으므로 Amazon Braket을 더 쉽게 설정할 수 있습니다. Amazon Braket은 서비스 연결 역할의 권한을 정의합니다. 이러한 정의를 변경하지 않는 한, Amazon Braket만 해당 역할을 수임할 수 있습니다. 정의된 권한에는 신뢰 정책과 권한 정책이 포함됩니다. 권한 정책은 다른 어떤 IAM 엔터티에도 연결할 수 없습니다.

Amazon Braket이 설정하는 서비스 연결 역할은 AWS Identity and Access Management(IAM) 서비스 연결 역할 기능의 일부입니다. 서비스 연결 역할을 지원하는 기타 AWS 서비스에 대한 자세한 내용은 IAM으로 작업하는 AWS 서비스를 참조하고 서비스 연결 역할 열에 Yes가 있는 서비스를 찾아보세요. 해당 서비스에 대한 서비스 연결 역할 설명서를 보려면 예 링크를 선택합니다.

서비스 연결 역할의 AWS 관리형 정책에 대한 자세한 내용은 AmazonBraketServiceRolePolicy를 참조하세요.

Amazon Braket의 규정 준수 검증

> **Note**
>
> AWS 규정 준수 보고서는 자체 독립 감사를 받기로 선택할 수 있는 타사 하드웨어 공급자 QPU에는 적용되지 않습니다.

AWS 서비스가 특정 규정 준수 프로그램의 범위에 포함되는지 알아보려면 규정 준수 프로그램 제공 범위 내 AWS 서비스를 참조하고 관심 있는 규정 준수 프로그램을 선택하십시오. 일반적인 정보는 AWS 규정 준수 프로그램을 참조하세요.

AWS Artifact를 사용하여 타사 감사 보고서를 다운로드할 수 있습니다. 자세한 내용은 AWS Artifact에서 보고서 다운로드를 참조하세요.

AWS 서비스 사용 시 규정 준수 책임은 데이터의 민감도, 회사의 규정 준수 목표 및 관련 법률 및 규정에 따라 결정됩니다. AWS 서비스 사용 시 규정 준수 책임에 대한 자세한 내용은 AWS 보안 설명서를 참조하세요.

Amazon Braket의 인프라 보안

관리형 서비스인 Amazon Braket은 AWS 글로벌 네트워크 보안으로 보호됩니다. AWS 보안 서비스와 AWS의 인프라 보호 방법에 대한 자세한 내용은 AWS 클라우드 보안을 참조하세요. 인프라 보안에 대한 모범 사례를 사용하여 AWS 환경을 설계하려면 보안 원칙 AWS Well-Architected 프레임워크의 인프라 보호를 참조하세요.

AWS에서 게시한 API 직접 호출을 사용하여 네트워크를 통해 Amazon Braket에 액세스합니다. 클라이언트는 다음을 지원해야 합니다.

- Transport Layer Security(TLS) TLS 1.2는 필수이며 TLS 1.3을 권장합니다.
- DHE(Ephemeral Diffie-Hellman) 또는 ECDHE(Elliptic Curve Ephemeral Diffie-Hellman)와 같은 완전 전송 보안(PFS)이 포함된 암호 제품군 Java 7 이상의 최신 시스템은 대부분 이러한 모드를 지원합니다.

이러한 API 연산은 어떤 네트워크 위치에서든 직접적으로 호출할 수 있지만, Braket은 소스 IP 주소에 따른 제한 사항을 포함할 수 있는 리소스 기반 액세스 정책을 지원합니다. Braket 정책을 사용하여 특정 Amazon Virtual Private Cloud(Amazon VPC) 엔드포인트 또는 특정 VPC에서 액세스를 제어할 수도

있습니다. 그러면 AWS 네트워크 내에 있는 특정 VPC에서만 특정 Braket 리소스에 대한 네트워크 액세스가 효과적으로 격리됩니다.

Amazon Braket 하드웨어 공급자의 보안

Amazon Braket의 QPU는 타사 하드웨어 공급자가 호스팅합니다. QPU에서 양자 작업을 실행하면, Amazon Braket은 처리를 위해 지정된 QPU로 회로를 전송할 때 DeviceARN을 식별자로 사용합니다.

Amazon Braket을 사용하여 타사 하드웨어 공급자 중 하나가 운영하는 양자 컴퓨팅 하드웨어에 액세스하는 경우 회로 및 연결된 데이터는 AWS에서 운영하는 시설 외부의 하드웨어 공급자가 처리합니다. 각 QPU를 사용할 수 있는 물리적 위치 및 AWS 리전에 대한 정보는 Amazon Braket 콘솔의 디바이스 세부 정보 섹션에서 확인할 수 있습니다.

콘텐츠는 익명화됩니다. 회로를 처리하는 데 필요한 콘텐츠만 타사로 전송됩니다. AWS 계정 정보는 타사로 전송되지 않습니다.

모든 데이터는 전송 및 저장 시 암호화됩니다. 데이터는 처리용으로만 해독됩니다. Amazon Braket 타사 공급자는 회로 처리 이외의 목적으로 콘텐츠를 저장하거나 사용할 수 없습니다. 회로가 완료되면 결과가 Amazon Braket으로 반환되고 S3 버킷에 저장됩니다.

Amazon Braket 타사 양자 하드웨어 공급자의 보안은 네트워크 보안, 액세스 제어, 데이터 보호 및 물리적 보안의 표준을 충족하는지 확인하기 위해 주기적으로 감사됩니다.

Amazon Braket용 Amazon VPC 엔드포인트

인터페이스 VPC 엔드포인트를 만들어 VPC와 Amazon Braket 간에 비공개 연결을 설정할 수 있습니다. 인터페이스 엔드포인트는 인터넷 게이트웨이AWS PrivateLink, NAT 디바이스, VPN 연결 또는 Direct Connect 연결 없이 Braket APIs에 액세스할 수 있는 기술로 구동됩니다. VPC의 인스턴스는 Braket API와 통신하는 데 공개 IP 주소가 필요하지 않습니다.

각 인터페이스 엔드포인트는 서브넷에서 하나 이상의 Elastic Network Interfaces로 표현됩니다.

를 사용하면 VPC와 Braket 간의 AWS PrivateLink트래픽이 Amazon 네트워크를 벗어나지 않으므로 퍼블릭 인터넷에 대한 데이터의 노출이 줄어들기 때문에 클라우드 기반 애플리케이션과 공유하는 데이터의 보안이 향상됩니다. 자세한 내용은 Amazon VPC 사용 설명서의 인터페이스 VPC 엔드포인트를 사용하여 AWS 서비스 액세스를 참조하세요.

이 섹션의 내용:

Amazon Braket VPC 엔드포인트에 대한 고려 사항

인터페이스 VPC 엔드포인트를 설정하기 전에 Amazon VPC 사용자 가이드의 인터페이스 엔드포인트 사전 조건을 검토해야 합니다.

Braket은 VPC에서 모든 API 작업에 대한 직접 호출을 지원합니다.

기본적으로 VPC 엔드포인트를 통해 Braket에 대한 전체 액세스가 허용됩니다. VPC 엔드포인트 정책을 지정하는 경우 액세스를 제어할 수 있습니다. 자세한 내용은 Amazon VPC 사용자 가이드의 엔드포인트 정책을 사용하여 VPC 엔드포인트로 액세스 제어를 참조하세요.

Braket 및 PrivateLink 설정

Amazon Braket AWS PrivateLink 에서를 사용하려면 Amazon Virtual Private Cloud(VPC) 엔드포인트를 인터페이스로 생성한 다음 Amazon Braket API 서비스를 통해 엔드포인트에 연결해야 합니다.

다음은 이 프로세스의 일반적인 단계로, 이후 섹션에서 자세히 설명합니다.

- AWS 리소스를 호스팅할 Amazon VPC를 구성하고 시작합니다. VPC가 이미 있는 경우에는 이 단계를 건너뛸 수 있습니다.
- Braket용 Amazon VPC 엔드포인트 생성
- 엔드포인트를 통해 Braket 양자 작업 연결 및 실행

1단계: 필요한 경우 Amazon VPC 시작

계정에 이미 VPC가 작동 중인 경우 이 단계를 건너뛸 수 있습니다.

VPC는 IP 주소 범위, 서브넷, 라우팅 테이블, 네트워크 게이트웨이와 같은 네트워크 설정을 제어합니다. 기본적으로 사용자 지정 가상 네트워크에서 AWS 리소스를 시작합니다. VPC에 대한 자세한 내용은 Amazon VPC 사용자 가이드를 참조하세요.

Amazon VPC 콘솔을 열고 서브넷, 보안 그룹 및 네트워크 게이트웨이가 포함된 새 VPC를 생성합니다.

2단계: Braket용 인터페이스 VPC 엔드포인트 생성

Amazon VPC 콘솔 또는 AWS Command Line Interface ()를 사용하여 Braket 서비스에 대한 VPC 엔드포인트를 생성할 수 있습니다AWS CLI. 자세한 정보는 Amazon VPC 사용 설명서의 VPC 엔드포인트 생성을 참조하세요.

콘솔에서 VPC 엔드포인트를 생성하려면 Amazon VPC 콘솔을 열고 엔드포인트 페이지를 연 다음, 새 엔드포인트 생성을 진행합니다. 나중에 참조할 수 있도록 엔드포인트 ID를 기록해 둡니다. Braket API에 특정 직접 호출을 수행할 때 –endpoint-url 플래그의 일부로 필요합니다.

다음 서비스 이름을 사용하여 Braket용 VPC 엔드포인트를 생성합니다.

- `com.amazonaws.substitute_your_region.braket`

자세한 내용은 Amazon VPC 사용 설명서의 인터페이스 VPC 엔드포인트를 사용하여 AWS 서비스 액세스를 참조하세요.

3단계: 엔드포인트를 통해 Braket 양자 작업 연결 및 실행

VPC 엔드포인트를 생성한 후에는 다음 예제와 같이 `endpoint-url` 파라미터를 포함하는 CLI 명령을 실행하여 API 또는 런타임에 대한 인터페이스 엔드포인트를 지정할 수 있습니다.

```
aws braket search-quantum-tasks --endpoint-url
 VPC_Endpoint_ID.braket.substituteYourRegionHere.vpce.amazonaws.com
```

VPC 엔드포인트에 대해 비공개 DNS 호스트 이름을 활성화하면 CLI 명령에서 엔드포인트를 URL로 지정할 필요가 없습니다. 대신, CLI 및 Braket SDK가 기본적으로 사용하는 Amazon Braket API DNS 호스트 이름이 VPC 엔드포인트로 확인됩니다. 다음 예제와 같은 형식을 갖습니다.

```
https://braket.substituteYourRegionHere.amazonaws.com
```

AWS PrivateLink 엔드포인트를 사용하여 Amazon VPC에서 Amazon SageMaker AI 노트북에 직접 액세스라는 블로그 게시물은 엔드포인트를 설정하여 SageMaker 노트북에 안전하게 연결하는 방법의 예를 제공하며, 이는 Amazon Braket 노트북과 유사합니다.

블로그 게시물의 단계를 따르는 경우 Amazon SageMaker AI 대신 Amazon Braket이라는 이름을 사용해야 합니다. 리전이 us-east-1이 아닌 경우 서비스 이름에 해당 문자열에 올바른 AWS 리전 이름을 입력`com.amazonaws.us-east-1.braket`하거나 대체합니다.

엔드포인트 생성에 대한 추가 정보

- 비공개 서브넷이 포함된 VPC를 생성하는 방법에 대한 자세한 내용은 프라이빗 서브넷이 포함된 VPC 생성을 참조하세요.
- Amazon VPC 콘솔 또는를 사용하여 엔드포인트를 생성하고 구성하는 방법에 대한 자세한 내용은 Amazon VPC 사용 설명서의 VPC 엔드포인트 생성을 AWS CLI참조하세요.
- 를 사용하여 엔드포인트를 생성하고 구성하는 방법에 대한 자세한 내용은 CloudFormation 사용 설명서의 AWS::EC2::VPCEndpoint 리소스를 CloudFormation참조하세요.

Amazon VPC 엔드포인트 정책으로 액세스 제어

Amazon Braket에 대한 연결 액세스를 제어하기 위해 Amazon VPC 엔드포인트에 AWS Identity and Access Management (IAM) 엔드포인트 정책을 연결할 수 있습니다. 이 정책은 다음 정보를 지정합니다.

- 작업을 수행할 수 있는 위탁자(사용자, 역할)
- 수행할 수 있는 작업
- 작업을 수행할 수 있는 리소스

자세한 내용은 Amazon VPC 사용자 가이드의 엔드포인트 정책을 사용하여 VPC 엔드포인트로 액세스 제어를 참조하세요.

예제: Braket 작업에 대한 VPC 엔드포인트 정책

다음 예제는 Braket에 대한 엔드포인트 정책을 보여줍니다. 이 정책은 엔드포인트에 연결될 경우 모든 리소스의 모든 위탁자에 대해 나열된 Braket 작업에 대한 액세스 권한을 부여합니다.

```
{
 "Statement":[
 {
   "Principal":"*",
   "Effect":"Allow",
   "Action":[
     “braket:action-1",
     “braket:action-2",
     “braket:action-3”
     ],
   "Resource":"*"
```

```
    }
  ]
}
```

엔드포인트 정책 여러 개를 연결하여 복잡한 IAM 규칙을 만들 수 있습니다. 자세한 내용과 예제는 다음을 참조하세요.

- Step Functions에 대한 Amazon Virtual Private Cloud 엔드포인트 정책
- 관리자가 아닌 사용자를 위한 세분화된 IAM 권한 생성
- 엔드포인트 정책을 사용하여 VPC 엔드포인트에 대한 액세스 제어

로깅 및 모니터링

Amazon Braket 서비스를 통해 양자 작업을 제출한 후에는 Amazon Braket SDK 및 콘솔을 통해 해당 작업의 상태와 진행 상황을 면밀히 모니터링할 수 있습니다. 이를 통해 워크로드 구현을 추적하고 잠재적인 병목 현상이나 문제를 식별하며 양자 애플리케이션의 성능과 신뢰성을 최적화하기 위한 적절한 조치를 취할 수 있는 중앙 집중식 인터페이스가 제공됩니다. 양자 작업이 완료되면 Braket은 지정된 Amazon S3 위치에 결과를 저장합니다. 양자 작업의 완료 시간은 다를 수 있으며, 특히 양자 처리 장치(Quantum Processing Unit, QPU) 장치에서 실행되는 작업의 경우 더욱 그렇습니다. 이는 주로 실행 대기열의 길이에 기인하는 것으로, 양자 하드웨어 리소스가 여러 사용자 간에 공유되기 때문입니다.

상태 유형 목록:

- `CREATED` - Amazon Braket에서 양자 작업을 수신했습니다.
- `QUEUED` - Amazon Braket이 양자 작업을 처리했으며 이제 디바이스에서 실행되기를 기다리고 있습니다.
- `RUNNING` - 양자 작업이 QPU 또는 온디맨드 시뮬레이터에서 실행 중입니다.
- `COMPLETED` - 양자 작업이 QPU 또는 온디맨드 시뮬레이터에서 실행을 완료했습니다.
- `FAILED` - 양자 작업이 실행을 시도했으나, 실패했습니다. 양자 작업이 실패한 이유에 따라 양자 작업을 다시 제출해 보세요.
- `CANCELLED` - 양자 작업을 취소했습니다. 양자 작업이 실행되지 않았습니다.

이 섹션의 내용:

Amazon Braket SDK에서 양자 작업 추적

명령 `device.run(…)` 은 고유한 양자 작업 ID로 양자 작업을 정의합니다. 다음 예제와 같이 `task.state()`를 사용하여 상태를 쿼리하고 추적할 수 있습니다.

참고: `task = device.run()`은 비동기 연산이므로, 시스템이 백그라운드에서 양자 작업을 처리하는 동안에도 작업을 계속할 수 있습니다.

결과 검색

`task.result()`를 직접적으로 호출하면 SDK가 Amazon Braket 폴링을 시작하여 양자 작업이 완료되었는지 확인하기 시작합니다. SDK는 `.run()`에서 정의한 폴링 파라미터를 사용합니다. 양자 작업이 완료되면 SDK는 S3 버킷에서 결과를 검색하여 `QuantumTaskResult` 객체로 반환합니다.

```
# create a circuit, specify the device and run the circuit
circ = Circuit().rx(0, 0.15).ry(1, 0.2).cnot(0,2)
device = AwsDevice("arn:aws:braket:::device/quantum-simulator/amazon/sv1")
task = device.run(circ, s3_location, shots=1000)

# get ID and status of submitted task
task_id = task.id
status = task.state()
print('ID of task:', task_id)
print('Status of task:', status)
# wait for job to complete
while status != 'COMPLETED':
    status = task.state()
    print('Status:', status)
```

```
ID of task:
arn:aws:braket:us-west-2:123412341234:quantum-task/b68ae94b-1547-4d1d-aa92-1500b82c300d
Status of task: QUEUED
Status: QUEUED
Status: QUEUED
Status: QUEUED
Status: QUEUED
Status: QUEUED
Status: QUEUED
Status: QUEUED
Status: RUNNING
Status: RUNNING
Status: COMPLETED
```

양자 작업 취소

양자 작업을 취소하려면 다음 예제와 같이 `cancel()` 메서드를 직접적으로 호출합니다.

```
# cancel quantum task
task.cancel()
status = task.state()
print('Status of task:', status)
```

```
Status of task: CANCELLING
```

메타데이터 확인

다음 예제와 같이 완료된 양자 작업의 메타데이터를 확인할 수 있습니다.

```
# get the metadata of the quantum task
metadata = task.metadata()
# example of metadata
shots = metadata['shots']
date = metadata['ResponseMetadata']['HTTPHeaders']['date']
# print example metadata
print("{} shots taken on {}.".format(shots, date))

# print name of the s3 bucket where the result is saved
results_bucket = metadata['outputS3Bucket']
print('Bucket where results are stored:', results_bucket)
# print the s3 object key (folder name)
results_object_key = metadata['outputS3Directory']
print('S3 object key:', results_object_key)

# the entire look-up string of the saved result data
look_up = 's3://'+results_bucket+'/'+results_object_key
print('S3 URI:', look_up)
```

```
1000 shots taken on Wed, 05 Aug 2020 14:44:22 GMT.
Bucket where results are stored: amazon-braket-123412341234
S3 object key: simulation-output/b68ae94b-1547-4d1d-aa92-1500b82c300d
S3 URI: s3://amazon-braket-123412341234/simulation-output/b68ae94b-1547-4d1d-
aa92-1500b82c300d
```

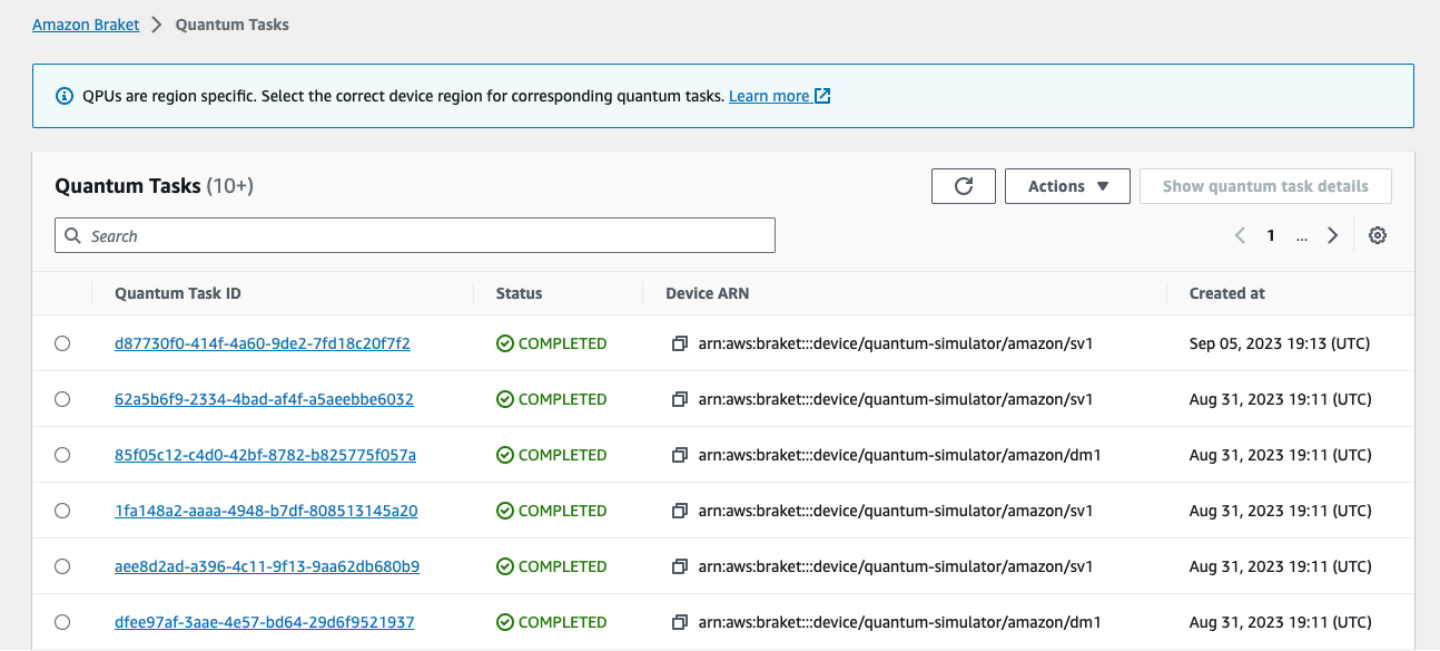
양자 작업 또는 결과 검색

양자 작업을 제출한 후 커널이 종료되거나 노트북 또는 컴퓨터를 닫은 경우 고유한 ARN(양자 작업 ID)으로 `task` 객체를 재구성할 수 있습니다. 그런 다음 `task.result()`를 직접적으로 호출하여 저장된 S3 버킷에서 결과를 가져올 수 있습니다.

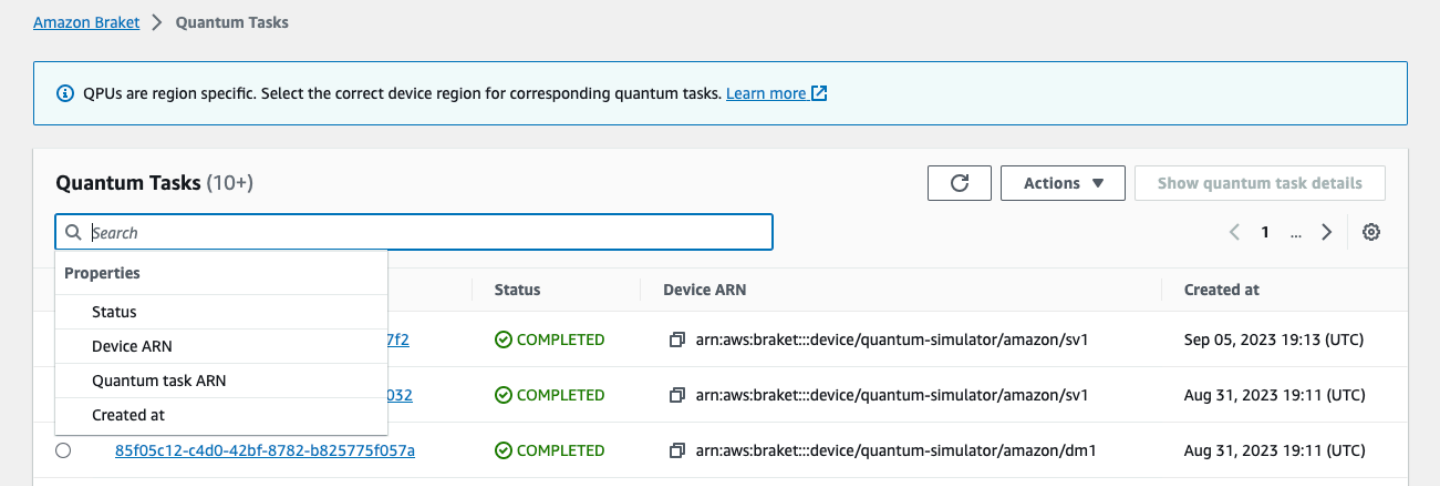
```
from braket.aws import AwsSession, AwsQuantumTask

# restore task with unique arn
task_load = AwsQuantumTask(arn=task_id)
# retrieve the result of the task
result = task_load.result()
```

Amazon Braket 콘솔을 통한 양자 작업 모니터링

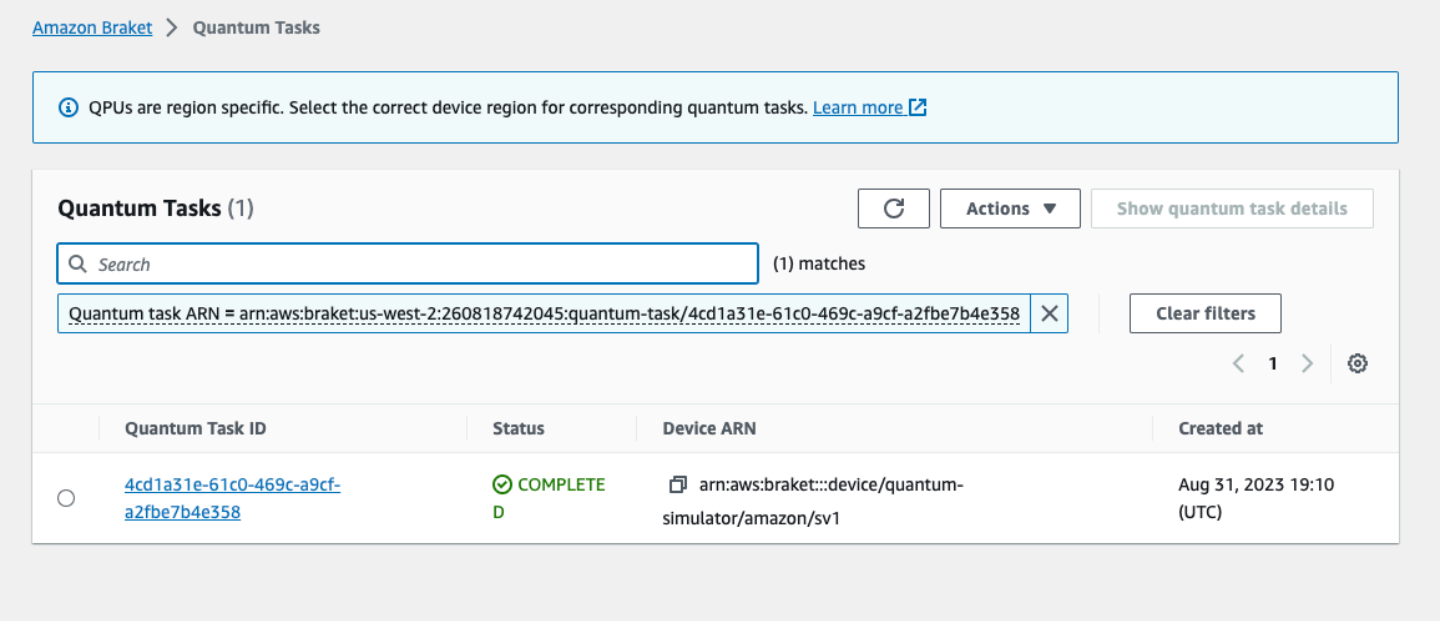
Amazon Braket은 Amazon Braket 콘솔을 통해 양자 작업을 모니터링하는 편리한 방법을 제공합니다. 제출된 모든 양자 작업은 다음 그림과 같이 양자 작업 필드에 나열됩니다. 이 서비스는 리전별로 다르므로 특정에서 생성된 양자 작업만 볼 수 있습니다 AWS 리전.



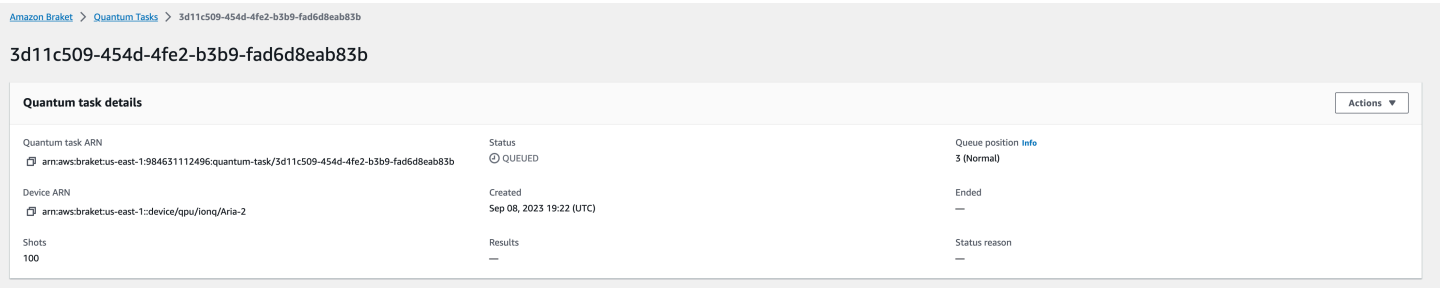
탐색 모음을 통해 특정 양자 작업을 검색할 수 있습니다. 검색은 양자 작업 ARN(ID), 상태, 디바이스 및 생성 시간을 기준으로 수행할 수 있습니다. 다음 예제와 같이 탐색 모음을 선택하면 옵션이 자동으로 나타납니다.



다음 이미지는 `task.id`를 직접적으로 호출하여 얻을 수 있는 고유한 양자 작업 ID를 기준으로 양자 작업을 검색하는 예제를 보여줍니다.



또한 아래 그림에서 볼 수 있듯이 양자 작업의 상태는 QUEUED 상태에 있는 동안 모니터링할 수 있습니다. 양자 작업 ID를 클릭하면 세부 정보 페이지가 표시됩니다. 이 페이지는 처리할 디바이스를 기준으로 양자 작업의 동적 대기열 위치를 표시합니다.



하이브리드 작업의 일부로 제출된 양자 작업은 대기열에 있을 때 우선순위를 갖습니다. 하이브리드 작업 외부에서 제출된 양자 작업은 일반적인 대기열 우선순위를 갖습니다.

Braket SDK를 쿼리하려는 고객은 프로그래밍 방식으로 양자 작업 및 하이브리드 작업 대기열 위치를 얻을 수 있습니다. 자세한 내용은 내 작업은 언제 실행되나요 페이지를 참조하세요.

Amazon Braket 리소스 태그 지정

태그는 사용자가 할당하거나가 AWS 리소스에 AWS 할당하는 사용자 지정 속성 레이블입니다. 태그는 리소스에 대한 자세한 정보를 알려주는 메타데이터입니다. 각 태그는 키와 값으로 구성됩니다. 태그 키와 태그 값을 합해서 키-값 쌍이라고 합니다. 할당하는 태그에 대해 키와 값을 직접 정의합니다.

Amazon Braket 콘솔에서 양자 작업 또는 노트북으로 이동하여 연결된 태그 목록을 볼 수 있습니다. 태그를 추가하거나, 태그를 제거하거나, 태그를 수정할 수 있습니다. 생성 시 양자 작업 또는 노트북에 태그를 지정한 다음 콘솔 AWS CLI, 또는를 통해 연결된 태그를 관리할 수 있습니다API.

AWS 및 태그에 대한 자세한 내용

- 이름 지정 및 사용 규칙을 포함하여 태그 지정에 대한 일반적인 내용은 AWS 리소스 태그 지정 및 태그 편집기 사용 설명서의 태그 편집기란 무엇입니까?를 참조하세요.
- 태그 지정 제한에 대한 자세한 내용은 AWS 리소스 태그 지정 및 태그 편집기 사용 설명서의 태그 이름 지정 제한 및 요구 사항을 참조하세요.
- 모범 사례 및 태그 지정 전략은 AWS 리소스 태그 지정 모범 사례를 참조하세요.
- 태그 사용을 지원하는 서비스 목록은 리소스 그룹 태깅 API 참조를 참조하세요.

다음 섹션에서는 Amazon Braket용 태그에 대한 자세한 정보를 제공합니다.

이 섹션의 내용:

태그 사용

태그를 사용하면 리소스를 유용한 범주로 구성할 수 있습니다. 예를 들어 "Department" 태그를 할당하여 이 리소스를 소유하는 부서를 지정할 수 있습니다.

각 태그는 두 부분으로 구성됩니다.

- 태그 키(예: CostCenter, Environment 또는 Project). 태그 키는 대소문자를 구별합니다.
- 태그 값(예: 111122223333 또는 Production)으로 알려진 선택적 필드 태그 값을 생략하는 것은 빈 문자열을 사용하는 것과 같습니다. 태그 키처럼 태그 값은 대/소문자를 구별합니다.

태그는 다음과 같은 작업을 수행하는 데 도움이 됩니다.

- AWS 리소스를 식별하고 구성합니다. 많은가 태그 지정을 AWS 서비스 지원하므로 서로 다른 서비스의 리소스에 동일한 태그를 할당하여 리소스가 관련이 있음을 나타낼 수 있습니다.
- AWS 비용을 추적합니다. AWS 결제 및 비용 관리 대시보드에서 이러한 태그를 활성화합니다.는 태그를 AWS 사용하여 비용을 분류하고 월별 비용 할당 보고서를 제공합니다. 자세한 내용은 AWS 결제 및 비용 관리 사용자 가이드의 비용 할당 태그 사용을 참조하세요.
- AWS 리소스에 대한 액세스를 제어합니다. 자세한 내용은 태그를 사용한 액세스 제어를 참조하세요.

Amazon Braket에서의 태그 지정을 위해 지원되는 리소스

Amazon Braket의 다음 리소스 유형은 태그 지정을 지원합니다.

- `quantum-task` 리소스
- 리소스 이름: `AWS::Service::Braket`
- ARN 정규식: `arn:${Partition}:braket:${Region}:${Account}:quantum-task/${RandomId}`

참고: 노트북은 실제로 Amazon SageMaker AI 리소스이지만, 콘솔을 사용하여 노트북 리소스로 이동하면 Amazon Braket 콘솔에서 Amazon Braket 노트북에 태그를 적용하고 관리할 수 있습니다. 자세한 내용은 SageMaker 설명서의 노트북 인스턴스 메타데이터를 참조하세요.

Amazon Braket API로 태그 지정

- Amazon Braket API를 사용하여 리소스에 태그를 설정하는 경우 `TagResource`API를 직접적으로 호출합니다.

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

- 리소스에서 태그를 제거하려면 UntagResourceAPI를 직접적으로 호출합니다.

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

- 특정 리소스에 연결된 모든 태그를 나열하려면 ListTagsForResourceAPI를 직접적으로 호출합니다.

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

태그 지정 제한 사항

Amazon Braket 리소스의 태그에는 다음과 같은 기본 제한 사항이 적용됩니다.

- 리소스에 할당할 수 있는 최대 태그 수: 50
- 최대 키 길이: 유니코드 128자
- 최대 값 길이: 유니코드 256자
- 키 및 값에 유효한 문자: `a-z, A-Z, 0-9, space` 및 다음 문자: `_ . : / = + -` 및 `@`
- 키와 값은 대/소문자를 구분합니다.
- 를 키의 접두사`aws`로 사용하지 마세요.이 키는 AWS 사용하기 위해 예약되어 있습니다.

Amazon Braket에서의 태그 관리

태그를 리소스의 속성으로 설정합니다. Amazon Braket 콘솔, Amazon Braket API 또는 AWS CLI를 통해 태그를 보고 추가하고 수정하며 나열하고 삭제할 수 있습니다. 자세한 내용은 Amazon Braket API 참조를 참조하세요.

이 섹션의 내용:

- 태그 추가
- 태그 보기
- 태그 편집
- 태그 제거

태그 추가

태그 지정이 가능한 리소스에 태그를 추가할 수 있는 시점은 다음과 같습니다.

- 리소스 생성 시: 콘솔을 사용하거나 AWS API에서 `Create` 연산에 `Tags` 파라미터를 포함합니다.
- 리소스 생성 후: 콘솔을 사용하여 양자 작업 또는 노트북 리소스로 이동하거나 AWS API에서 `TagResource` 연산을 직접적으로 호출합니다.

리소스를 생성할 때 리소스에 태그를 추가하려면 지정된 유형의 리소스를 생성할 수 있는 권한도 필요합니다.

태그 보기

콘솔을 사용하여 작업 또는 노트북 리소스로 이동하거나 `ListTagsForResource` API 작업을 호출하여 Amazon Braket의 태그 지정 가능한 리소스에서 태그를 볼 수 있습니다 AWS .

다음 AWS API 명령을 사용하여 리소스의 태그를 볼 수 있습니다.

- AWS API: `ListTagsForResource`

태그 편집

콘솔을 사용하여 양자 작업 또는 노트북 리소스로 이동해 태그를 편집하거나 다음 명령을 사용하여 태그 지정이 가능한 리소스에 연결된 태그의 값을 수정할 수 있습니다. 이미 존재하는 태그 키를 지정하면 해당 키의 값이 덮어쓰기됩니다.

- AWS API: `TagResource`

태그 제거

제거할 키를 지정하거나 콘솔을 사용하여 양자 작업 또는 노트북 리소스로 이동하거나 `UntagResource` 연산을 직접적으로 호출하여 리소스에서 태그를 제거할 수 있습니다.

- AWS API: `UntagResource`

Amazon Braket의 AWS CLI 태그 지정 예제

Amazon Braket과 상호 작용하기 위해 AWS Command Line Interface (AWS CLI)로 작업하는 경우 다음 코드는 생성한 양자 작업에 적용되는 태그를 생성하는 방법을 보여주는 예제 명령입니다. 이 예제에서 작업은 Rigetti QPU에 지정된 파라미터 설정을 사용하여 SV1 양자 시뮬레이터에서 실행되고 있습니다. 예제 명령 내에서 태그는 다른 모든 필수 파라미터 뒤의 맨 끝에 지정되어야 합니다. 이 경우 태그는 `state`의 키이고 `Washington`의 값입니다. 이러한 태그를 사용하면 이 특정 양자 작업을 분류하거나 식별하는 데 도움이 될 수 있습니다.

```
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\"}
```

이 예제는 AWS CLI를 통해 양자 작업을 실행할 때 태그를 적용하는 방법을 보여줍니다. 이는 Braket 리소스를 정리하고 추적하는 데 유용합니다.

EventBridge를 사용한 양자 작업 모니터링

Amazon EventBridge는 Amazon Braket 양자 작업의 상태 변경 이벤트를 모니터링합니다. Amazon Braket의 이벤트는 거의 실시간으로 EventBridge에 전달됩니다. 이벤트가 규칙과 일치할 때 수행할 자동화된 작업을 포함하여 관심 있는 이벤트를 나타내는 규칙을 작성할 수 있습니다. 트리거할 수 있는 자동 작업은 다음과 같습니다.

- AWS Lambda 함수 호출

- AWS Step Functions 상태 시스템 활성화
- Amazon SNS 주제 알림

EventBridge는 다음과 같은 Amazon Braket 상태 변경 이벤트를 모니터링합니다.

- 양자 작업의 상태 변경

Amazon Braket은 양자 작업 상태 변경 이벤트의 전달을 보장합니다. 이러한 이벤트는 최소 한 번 이상 전달되지만, 순서가 뒤바뀔 수 있습니다.

자세한 내용은 Amazon EventBridge의 이벤트를 참조하세요.

이 섹션의 내용:

- EventBridge를 사용한 양자 작업 상태 모니터링
- Amazon Braket EventBridge 이벤트 예제

EventBridge를 사용한 양자 작업 상태 모니터링

EventBridge를 사용하면 Amazon Braket이 Braket 양자 작업과 관련된 상태 변경 알림을 보낼 때 수행할 작업을 정의하는 규칙을 생성할 수 있습니다. 예를 들어 양자 작업의 상태가 변경될 때마다 이메일 메시지를 전송하는 규칙을 생성할 수 있습니다.

1. EventBridge 및 Amazon Braket을 사용할 권한이 있는 계정을 AWS 사용하여에 로그인합니다.
2. Amazon EventBridge 콘솔을 엽니다.
3. 다음 값을 사용하여 EventBridge 규칙을 생성합니다.
   - 규칙 유형에서 이벤트 패턴이 있는 규칙을 생성합니다.
   - 이벤트 소스에서 기타를 선택합니다.
   - 이벤트 패턴 섹션에서 사용자 지정 패턴(JSON 편집기)을 선택하고 다음 이벤트 패턴을 텍스트 영역에 붙여 넣습니다.

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

```
}
```

Amazon Braket에서 모든 이벤트를 캡처하려면 다음 코드와 같이 `detail-type` 섹션을 제외합니다.

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

- 대상 유형에서 AWS 서비스를 선택하고 대상 선택에서 Amazon SNS 주제 또는 AWS Lambda 함수와 같은 대상을 선택합니다. Amazon Braket에서 양자 작업 상태 변경 이벤트를 수신하면 대상이 트리거됩니다.

  예를 들어, 이벤트 발생 시 Amazon Simple Notification Service(SNS) 주제를 사용하여 이메일 또는 텍스트 메시지를 보낼 수 있습니다. 이렇게 하려면, 먼저 Amazon SNS 콘솔을 사용하여 Amazon SNS 주제를 생성합니다. 자세한 내용은 사용자 알림에 Amazon SNS 사용을 참조하세요.

규칙 생성에 대한 자세한 내용은 이벤트에 반응하는 Amazon EventBridge 규칙 생성을 참조하세요.

Amazon Braket EventBridge 이벤트 예제

Amazon Braket 양자 작업 상태 변경 이벤트의 필드에 대한 자세한 내용은 Amazon EventBridge의 이벤트를 참조하세요.

JSON "세부 정보" 필드에 다음 속성이 나타납니다.

- **`quantumTaskArn`** (str): 이 이벤트가 생성된 양자 작업
- **`status`** (선택 사항[str]): 양자 작업이 전환된 상태
- **`deviceArn`** (str): 이 양자 작업이 생성되도록 사용자가 지정한 디바이스
- shots (int): 사용자가 요청한 shots의 수
- **`outputS3Bucket`** (str): 사용자가 지정한 출력 버킷
- **`outputS3Directory`** (str): 사용자가 지정한 출력 키 접두사
- **`createdAt`** (str): ISO-8601 문자열로서의 양자 작업 생성 시간
- **`endedAt`** (선택 사항[str]): 양자 작업이 터미널 상태에 도달한 시간 이 필드는 양자 작업이 터미널 상태로 전환된 경우에만 표시됩니다.

다음 JSON 코드는 Amazon Braket 양자 작업 상태 변경 이벤트의 예제를 보여줍니다.

```
{
    "version":"0",
    "id":"6101452d-8caf-062b-6dbc-ceb5421334c5",
    "detail-type":"Braket Task State Change",
    "source":"aws.braket",
    "account":"012345678901",
    "time":"2021-10-28T01:17:45Z",
    "region":"us-east-1",
    "resources":[
        "arn:aws:braket:us-east-1:012345678901:quantum-task/834b21ed-77a7-4b36-a90c-
c776afc9a71e"
    ],
    "detail":{
        "quantumTaskArn":"arn:aws:braket:us-east-1:012345678901:quantum-
task/834b21ed-77a7-4b36-a90c-c776afc9a71e",
        "status":"COMPLETED",
        "deviceArn":"arn:aws:braket:::device/quantum-simulator/amazon/sv1",
        "shots":"100",
        "outputS3Bucket":"amazon-braket-0260a8bc871e",
        "outputS3Directory":"sns-testing/834b21ed-77a7-4b36-a90c-c776afc9a71e",
        "createdAt":"2021-10-28T01:17:42.898Z",
        "eventName":"MODIFY",
        "endedAt":"2021-10-28T01:17:44.735Z"
      }
}
```

CloudWatch를 사용한 지표 모니터링

원시 데이터를 수집하여 읽을 수 있는 근실시간 지표로 처리하는 Amazon CloudWatch를 사용하여 Amazon Braket을 모니터링할 수 있습니다. Amazon CloudWatch 콘솔에서 최대 15개월 전에 생성된 과거 정보를 보거나 지난 2주 동안 업데이트된 지표를 검색하여 Amazon Braket의 성능에 대한 더 나은 관점을 확보할 수 있습니다. 자세한 내용은 CloudWatch 지표 사용을 참조하세요.

> **Note**
>
> Amazon SageMaker AI 콘솔의 노트북 세부 정보 페이지로 이동하여 Amazon Braket 노트북의 CloudWatch 로그 스트림을 볼 수 있습니다. SageMaker 콘솔을 통해 추가 Amazon Braket 노트북 설정을 사용할 수 있습니다.

이 섹션의 내용:

Amazon Braket 지표 및 차원

지표는 CloudWatch의 기본 개념입니다. 지표는 CloudWatch에 게시된 시간 순서별 데이터 포인트 집합을 나타냅니다. 모든 지표는 일련의 차원으로 특징지어집니다. CloudWatch의 지표 차원에 대한 자세한 내용은 CloudWatch 차원을 참조하세요.

Amazon Braket은 Amazon Braket에만 해당하는 다음과 같은 지표 데이터를 Amazon CloudWatch 지표로 전송합니다.

양자 작업 지표

양자 작업이 있는 경우 지표를 사용할 수 있습니다. CloudWatch 콘솔의 AWS/Braket/By Device 아래에 표시됩니다.

지표	설명
개수	양자 작업 수입니다.
Latency	이 지표는 양자 작업이 완료되면 생성됩니다. 양자 작업 초기화부터 완료까지의 총 시간을 나타냅니다.

양자 작업 지표의 차원

양자 작업 지표는 `deviceArn` 파라미터를 기반으로 하는 차원으로 게시되며, 형식은 arn:aws:braket:::device/xxx입니다.

CloudTrail을 사용한 양자 작업 로깅

Amazon Braket은 Amazon Braket AWS 서비스 에서 사용자 AWS CloudTrail, 역할 또는가 수행한 작업에 대한 레코드를 제공하는 서비스와 통합됩니다. CloudTrail은 Amazon Braket에 대한 모든 API 직접 호출을 이벤트로 캡처합니다. 캡처되는 직접 호출에는 Amazon Braket 콘솔로부터의 직접 호출과 Amazon Braket 연산에 대한 코드 직접 호출이 포함됩니다. 트레일을 생성하면 Amazon Braket 이벤트를 포함하여 CloudTrail 이벤트를 Amazon S3 버킷으로 지속적으로 전달하도록 설정할 수 있습니다.

트레일을 구성하지 않은 경우에도 CloudTrail 콘솔의 이벤트 기록에서 최신 이벤트를 볼 수 있습니다. CloudTrail에서 수집한 정보를 사용하여 Amazon Braket에 대해 수행된 요청, 요청이 수행된 IP 주소, 요청을 수행한 사람, 요청이 수행된 시간 및 추가 세부 정보를 확인할 수 있습니다.

CloudTrail에 대한 자세한 내용은 AWS CloudTrail 사용자 가이드를 참조하세요.

이 섹션의 내용:

- CloudTrail의 Amazon Braket 정보
- Amazon Braket 파일 항목 이해

CloudTrail의 Amazon Braket 정보

CloudTrail은 계정을 생성할 AWS 계정 때에서 활성화됩니다. Amazon Braket에서 활동이 발생하면 해당 활동이 이벤트 기록의 다른 이벤트와 함께 CloudTrail AWS 서비스 이벤트에 기록됩니다. 에서 최근 이벤트를 보고 검색하고 다운로드할 수 있습니다 AWS 계정. 자세한 설명은 CloudTrail 이벤트 기록으로 이벤트 보기를 참조하세요.

Amazon Braket에 대한 이벤트를 AWS 계정포함하여에 이벤트를 지속적으로 기록하려면 추적을 생성합니다. CloudTrail은 추적을 사용하여 Amazon S3 버킷으로 로그 파일을 전송할 수 있습니다. 콘솔에서 트레일을 생성하면 기본적으로 모든 AWS 리전에 트레일이 적용됩니다. 추적은 AWS 파티션의 모든 리전에서 이벤트를 로깅하고 지정한 Amazon S3 버킷으로 로그 파일을 전송합니다. 또한 CloudTrail 로그에서 수집된 이벤트 데이터를 추가로 분석하고 조치를 취 AWS 서비스 하도록 다른를 구성할 수 있습니다. 자세한 내용은 다음 자료를 참조하세요.

- 추적 생성 개요
- CloudTrail 지원 서비스 및 통합
- CloudTrail에서 Amazon SNS 알림 구성
- 여러 리전으로부터 CloudTrail 로그 파일 받기 및 여러 계정으로부터 CloudTrail 로그 파일 받기

모든 Amazon Braket 작업은 CloudTrail에 의해 로깅됩니다. 예를 들어, `GetQuantumTask` 또는 `GetDevice` 작업을 직접적으로 호출하면 CloudTrail 로그 파일에 항목이 생성됩니다.

모든 이벤트 또는 로그 항목에는 요청을 생성했던 사용자에 관한 정보가 포함됩니다. ID 정보는 다음 사항을 확인하는 데 도움이 됩니다.

- 역할 또는 페더레이션 사용자에 대한 임시 자격 증명을 사용하여 요청이 생성되었는지 여부
- 다른 AWS 서비스에서 요청했는지 여부

자세한 설명은 CloudTrail userIdentity 요소를 참조하세요.

Amazon Braket 파일 항목 이해

트레일이란 지정한 S3 버킷에 이벤트를 로그 파일로 입력할 수 있게 하는 구성입니다. CloudTrail 로그 파일에는 하나 이상의 로그 항목이 포함될 수 있습니다. 이벤트는 모든 소스로부터의 단일 요청을 나타내며 요청 작업, 작업 날짜와 시간, 요청 파라미터 등에 대한 정보가 들어 있습니다. CloudTrail 로그 파일은 공개 API 직접 호출에 대한 정렬된 스택 트레이스가 아니므로 특정 순서로 나타나지 않습니다.

다음 예제는 양자 작업의 세부 정보를 가져오는 `GetQuantumTask` 작업에 대한 로그 항목입니다.


```
{
  "eventVersion": "1.05",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "foobar",
    "arn": "foobar",
    "accountId": "foobar",
    "accessKeyId": "foobar",
    "sessionContext": {
      "sessionIssuer": {
        "type": "Role",
        "principalId": "foobar",
        "arn": "foobar",
        "accountId": "foobar",
        "userName": "foobar"
      },
      "webIdFederationData": {},
      "attributes": {
        "mfaAuthenticated": "false",
        "creationDate": "2020-08-07T00:56:57Z"
      }
    }
  },
  "eventTime": "2020-08-07T01:00:08Z",
  "eventSource": "braket.amazonaws.com",
  "eventName": "GetQuantumTask",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "foobar",
  "userAgent": "aws-cli/1.18.110 Python/3.6.10
 Linux/4.9.184-0.1.ac.235.83.329.metal1.x86_64 botocore/1.17.33",
  "requestParameters": {
    "quantumTaskArn": "foobar"
```

```
  },
  "responseElements": null,
  "requestID": "20e8000c-29b8-4137-9cbc-af77d1dd12f7",
  "eventID": "4a2fdb22-a73d-414a-b30f-c0797c088f7c",
  "readOnly": true,
  "eventType": "AwsApiCall",
  "recipientAccountId": "foobar"
}
```

다음은 디바이스 이벤트의 세부 정보를 반환하는 GetDevice 작업에 대한 로그 항목을 보여줍니다.

```
{
  "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"
}
```

Amazon Braket을 사용한 고급 로깅

로거를 사용하여 전체 작업 처리 프로세스를 기록할 수 있습니다. 이러한 고급 로깅 기법을 사용하면 백그라운드 폴링을 확인하고 추후 디버깅을 위한 기록을 생성할 수 있습니다.

로거를 사용하려면 `poll_timeout_seconds` 및 `poll_interval_seconds` 파라미터를 변경하여 양자 작업을 장기 실행하고 양자 작업 상태를 지속적으로 로깅하며 결과를 파일에 저장하는 것이 좋습니다. 이 코드를 Jupyter Notebook 대신 Python 스크립트로 전송하여 스크립트를 백그라운드에서 프로세스로 실행할 수 있습니다.

로거 구성

먼저, 다음 예제 줄에서와 같이 모든 로그가 자동으로 텍스트 파일에 작성되도록 로거를 구성합니다.

```
# 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
```

회로 생성 및 실행

이제 회로를 생성하고 실행할 디바이스에 제출하여 이 예제에서 보여주는 것처럼 어떤 결과가 발생하는지 확인할 수 있습니다.

```
# 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
)
```

로그 파일 확인

다음 명령을 입력하여 파일에 작성되는 내용을 확인할 수 있습니다.

```
# 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})
```

로그 파일에서 ARN 가져오기

이전 예제와 같이 반환된 로그 파일 출력에서 ARN 정보를 얻을 수 있습니다. ARN ID를 사용하면 완료된 양자 작업의 결과를 검색할 수 있습니다.

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

# with this arn you can restore again task from unique arn
task_load = AwsQuantumTask(arn=arn, aws_session=AwsSession())

# get results of task
result = task_load.result()
```

Amazon Braket 할당량

다음 표에는 Amazon Braket의 서비스 할당량이 나와 있습니다. 한도라고도 하는 서비스 할당량은 AWS 계정에 대한 서비스 리소스 또는 작업의 최대 수입니다.

일부 할당량은 늘릴 수 있습니다. 자세한 내용은 AWS 서비스할당량을 참조하세요.

- 버스트 속도 할당량은 늘릴 수 없습니다.
- 조정 가능한 할당량의 최대 속도 증가(조정이 불가능한 버스트 속도 제외)는 지정된 기본 속도 제한의 2배입니다. 예를 들어 기본 할당량 60은 최대 120까지 조정할 수 있습니다.
- 동시 SV1(DM1) 양자 작업에 대한 조정 가능한 할당량은AWS 리전당 최대 60개까지 허용합니다.
- 하이브리드 작업에 허용되는 최대 컴퓨팅 인스턴스 수는 1개이며 할당량은 조정할 수 있습니다.

Resource	설명	한도	조정 가능
API 요청 속도	현재 리전의 이 계정에서 보낼 수 있는 초당 요청의 최대 수입니다.	140	예
API 요청 버스트 속도	현재 리전의 이 계정에서 한 번의 버스트로 보낼 수 있는 초당 최대 추가 요청 수(RPS)입니다.	600	아니요
`CreateQuantumTask` 요청 속도	리전별로 이 계정에서 초당 보낼 수 있는 `CreateQuantumTask` 요청의 최대 수입니다.	초당 20	예
`CreateQuantumTask` 요청 버스트 속도	현재 리전의 이 계정에서 한 번의 버스트로 보낼 수 있는 초당 최대 추가 `CreateQua`	40	아니요

Resource	설명	한도	조정 가능
	ntumTask 요청 수(RPS)입니다.		
SearchQuantumTasks 요청 속도	리전별로 이 계정에서 초당 보낼 수 있는 SearchQuantumTasks 요청의 최대 수입니다.	초당 5	예
SearchQuantumTasks 요청 버스트 속도	현재 리전의 이 계정에서 한 번의 버스트로 보낼 수 있는 초당 최대 추가 SearchQuantumTasks 요청 수(RPS)입니다.	50	아니요
GetQuantumTask 요청 속도	리전별로 이 계정에서 초당 보낼 수 있는 GetQuantumTask 요청의 최대 수입니다.	100/초	예
GetQuantumTask 요청 버스트 속도	현재 리전의 이 계정에서 한 번의 버스트로 보낼 수 있는 초당 최대 추가 GetQuantumTask 요청 수(RPS)입니다.	500	아니요
CancelQuantumTask 요청 속도	리전별로 이 계정에서 초당 보낼 수 있는 CancelQuantumTask 요청의 최대 수입니다.	2/초	예

Resource	설명	한도	조정 가능
CancelQuantumTask 요청 버스트 속도	현재 리전의 이 계정에서 한 번의 버스트로 보낼 수 있는 초당 최대 추가 CancelQuantumTask 요청 수(RPS)입니다.	20	아니요
GetDevice 요청 속도	리전별로 이 계정에서 초당 보낼 수 있는 GetDevice 요청의 최대 수입니다.	초당 5	예
GetDevice 요청 버스트 속도	현재 리전의 이 계정에서 한 번의 버스트로 보낼 수 있는 초당 최대 추가 GetDevice 요청 수(RPS)입니다.	50	아니요
SearchDevices 요청 속도	리전별로 이 계정에서 초당 보낼 수 있는 SearchDevices 요청의 최대 수입니다.	초당 5	예
SearchDevices 요청 버스트 속도	현재 리전의 이 계정에서 한 번의 버스트로 보낼 수 있는 초당 최대 추가 SearchDevices 요청 수(RPS)입니다.	50	아니요
CreateJob 요청 속도	리전별로 이 계정에서 초당 보낼 수 있는 CreateJob 요청의 최대 수입니다.	초당 1	예

Resource	설명	한도	조정 가능
CreateJob 요청 버스트 속도	현재 리전의 이 계정에서 한 번의 버스트로 보낼 수 있는 초당 최대 추가 CreateJob 요청 수(RPS)입니다.	5	아니요
SearchJobs 요청 속도	리전별로 이 계정에서 초당 보낼 수 있는 SearchJob 요청의 최대 수입니다.	초당 5	예
SearchJobs 요청 버스트 속도	현재 리전의 이 계정에서 한 번의 버스트로 보낼 수 있는 초당 최대 추가 SearchJob 요청 수(RPS)입니다.	50	아니요
GetJob 요청 속도	리전별로 이 계정에서 초당 보낼 수 있는 GetJob 요청의 최대 수입니다.	초당 5	예
GetJob 요청 버스트 속도	현재 리전의 이 계정에서 한 번의 버스트로 보낼 수 있는 초당 최대 추가 GetJob 요청 수(RPS)입니다.	25	아니요
CancelJob 요청 속도	리전별로 이 계정에서 초당 보낼 수 있는 CancelJob 요청의 최대 수입니다.	2/초	예

Resource	설명	한도	조정 가능
`CancelJob` 요청 버스트 속도	현재 리전의 이 계정에서 한 번의 버스트로 보낼 수 있는 초당 최대 추가 `CancelJob` 요청 수(RPS)입니다.	5	아니요
동시 SV1 양자 작업 수	현재 리전의 상태 벡터 시뮬레이터(SV1)에서 동시에 실행되는 양자 작업의 최대 수입니다.	us-east-1 100개 us-west-1 50개 us-west-2 100개 eu-west-2 50개	아니요
동시 DM1 양자 작업 수	현재 리전의 밀도 행렬 시뮬레이터(DM1)에서 동시에 실행되는 양자 작업의 최대 수입니다.	us-east-1 100개 us-west-1 50개 us-west-2 100개 eu-west-2 50개	아니요
동시 TN1 양자 작업 수	현재 리전의 텐서 네트워크 시뮬레이터(TN1)에서 동시에 실행되는 양자 작업의 최대 수입니다.	us-east-1 10개 us-west-2 10개 eu-west-2 5개	예
동시 하이브리드 작업 수	현재 리전의 동시 하이브리드 작업 최대 수입니다.	3	예
하이브리드 작업 런타임 제한	하이브리드 작업을 실행할 수 있는 최대 시간(일 단위)입니다.	5	아니요

다음은 하이브리드 작업에 대한 기본 고전 컴퓨팅 인스턴스 할당량입니다. 이러한 할당량을 늘리려면 지원에 문의하세요. 또한 각 인스턴스에 대해 사용 가능한 리전이 지정됩니다.

Resource	설명	한도	조정 가능	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
하이브리드 작업에 대한 ml.c4.xlarge의 최대 인스턴스 수	이 계정과 리전의 모든 Amazon Braket Hybrid Jobs에 허용되는 ml.c4.xlarge 유형의 최대 인스턴스 수입니다.	5	예	예	예	예	예	아니요
하이브리드 작업에 대한 ml.c4.2xlarge의 최대 인스턴스 수	이 계정과 리전의 모든 Amazon Braket Hybrid Jobs에 허용되는 ml.c4.2xlarge 유형의 최대 인스턴스 수입니다.	5	예	예	예	예	예	아니요

Resource	설명	한도	조정 가능	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
하이브리드 작업에 대한 ml.c4.4xlarge의 최대 인스턴스 수	이 계정과 리전의 모든 Amazon Braket Hybrid Jobs에 허용되는 ml.c4.4xlarge 유형의 최대 인스턴스 수입니다.	5	예	예	예	예	예	아니요
하이브리드 작업에 대한 ml.c4.8xlarge의 최대 인스턴스 수	이 계정과 리전의 모든 Amazon Braket Hybrid Jobs에 허용되는 ml.c4.8xlarge 유형의 최대 인스턴스 수입니다.	5	예	예	예	예	아니요	아니요

Resource	설명	한도	조정 가능	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
하이브리드 작업에 대한 ml.c5.xlarge의 최대 인스턴스 수	이 계정과 리전의 모든 Amazon Braket Hybrid Jobs에 허용되는 ml.c5.xlarge 유형의 최대 인스턴스 수입니다.	5	예	예	예	예	예	예
하이브리드 작업에 대한 ml.c5.2xlarge의 최대 인스턴스 수	이 계정과 리전의 모든 Amazon Braket Hybrid Jobs에 허용되는 ml.c5.2xlarge 유형의 최대 인스턴스 수입니다.	5	예	예	예	예	예	예

Resource	설명	한도	조정 가능	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
하이브리드 작업에 대한 ml.c5.4xlarge의 최대 인스턴스 수	이 계정과 리전의 모든 Amazon Braket Hybrid Jobs에 허용되는 ml.c5.4xlarge 유형의 최대 인스턴스 수입니다.	1	예	예	예	예	예	예
하이브리드 작업에 대한 ml.c5.9xlarge의 최대 인스턴스 수	이 계정과 리전의 모든 Amazon Braket Hybrid Jobs에 허용되는 ml.c5.9xlarge 유형의 최대 인스턴스 수입니다.	1	예	예	예	예	예	예

Resource	설명	한도	조정 가능	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
하이브리드 작업에 대한 ml.c5.18xlarge의 최대 인스턴스 수	이 계정과 리전의 모든 Amazon Braket Hybrid Jobs에 허용되는 ml.c5.18xlarge 유형의 최대 인스턴스 수입니다.	0	예	예	예	예	예	예
하이브리드 작업에 대한 ml.c5n.xlarge의 최대 인스턴스 수	이 계정과 리전의 모든 Amazon Braket Hybrid Jobs에 허용되는 ml.c5n.xlarge 유형의 최대 인스턴스 수입니다.	0	예	예	예	예	아니요	아니요

Resource	설명	한도	조정 가능	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
하이브리드 작업에 대한 ml.c5n.2xlarge의 최대 인스턴스 수	이 계정과 리전의 모든 Amazon Braket Hybrid Jobs에 허용되는 ml.c5n.2xlarge 유형의 최대 인스턴스 수입니다.	0	예	예	예	예	아니요	아니요
하이브리드 작업에 대한 ml.c5n.4xlarge의 최대 인스턴스 수	이 계정과 리전의 모든 Amazon Braket Hybrid Jobs에 허용되는 ml.c5n.4xlarge 유형의 최대 인스턴스 수입니다.	0	예	예	예	예	아니요	아니요

Resource	설명	한도	조정 가능	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
하이브리드 작업에 대한 ml.c5n.9xlarge의 최대 인스턴스 수	이 계정과 리전의 모든 Amazon Braket Hybrid Jobs에 허용되는 ml.c5n.9xlarge 유형의 최대 인스턴스 수입니다.	0	예	예	예	예	아니요	아니요
하이브리드 작업에 대한 ml.c5n.18xlarge의 최대 인스턴스 수	이 계정과 리전의 모든 Amazon Braket Hybrid Jobs에 허용되는 ml.c5n.18xlarge 유형의 최대 인스턴스 수입니다.	0	예	예	예	예	아니요	아니요

Resource	설명	한도	조정 가능	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
하이브리드 작업에 대한 ml.g4dn.xlarge의 최대 인스턴스 수	이 계정과 리전의 모든 Amazon Braket Hybrid Jobs에 허용되는 ml.g4dn.xlarge 유형의 최대 인스턴스 수입니다.	0	예	예	예	예	예	예
하이브리드 작업에 대한 ml.g4dn.2xlarge의 최대 인스턴스 수	이 계정과 리전의 모든 Amazon Braket Hybrid Jobs에 허용되는 ml.g4dn.2xlarge 유형의 최대 인스턴스 수입니다.	0	예	예	예	예	예	예

Resource	설명	한도	조정 가능	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
하이브리드 작업에 대한 ml.g4dn.4xlarge의 최대 인스턴스 수	이 계정과 리전의 모든 Amazon Braket Hybrid Jobs에 허용되는 ml.g4dn.4xlarge 유형의 최대 인스턴스 수입니다.	0	예	예	예	예	예	예
하이브리드 작업에 대한 ml.g4dn.8xlarge의 최대 인스턴스 수	이 계정과 리전의 모든 Amazon Braket Hybrid Jobs에 허용되는 ml.g4dn.8xlarge 유형의 최대 인스턴스 수입니다.	0	예	예	예	예	예	예

Resource	설명	한도	조정 가능	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
하이브리드 작업에 대한 ml.g4dn.12xlarge의 최대 인스턴스 수	이 계정과 리전의 모든 Amazon Braket Hybrid Jobs에 허용되는 ml.g4dn.12xlarge 유형의 최대 인스턴스 수입니다.	0	예	예	예	예	예	예
하이브리드 작업에 대한 ml.g4dn.16xlarge의 최대 인스턴스 수	이 계정과 리전의 모든 Amazon Braket Hybrid Jobs에 허용되는 ml.g4dn.16xlarge 유형의 최대 인스턴스 수입니다.	0	예	예	예	예	예	예

Resource	설명	한도	조정 가능	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
하이브리드 작업에 대한 ml.m4.xlarge의 최대 인스턴스 수	이 계정과 리전의 모든 Amazon Braket Hybrid Jobs에 허용되는 ml.m4.xlarge 유형의 최대 인스턴스 수입니다.	5	예	예	예	예	예	아니요
하이브리드 작업에 대한 ml.m4.2xlarge의 최대 인스턴스 수	이 계정과 리전의 모든 Amazon Braket Hybrid Jobs에 허용되는 m4.2xlarge 유형의 최대 인스턴스 수입니다.	5	예	예	예	예	예	아니요

Resource	설명	한도	조정 가능	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
하이브리드 작업에 대한 ml.m4.4xlarge의 최대 인스턴스 수	이 계정과 리전의 모든 Amazon Braket Hybrid Jobs에 허용되는 ml.m4.4xlarge 유형의 최대 인스턴스 수입니다.	2	예	예	예	예	예	아니요
하이브리드 작업에 대한 ml.m4.10xlarge의 최대 인스턴스 수	이 계정과 리전의 모든 Amazon Braket Hybrid Jobs에 허용되는 m4.10xlarge 유형의 최대 인스턴스 수입니다.	0	예	예	예	예	예	아니요

Resource	설명	한도	조정 가능	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
하이브리드 작업에 대한 ml.m4.16› large의 최대 인스턴스 수	이 계정과 리전의 모든 Amazon Braket Hybrid Jobs에 허용되는 ml.m4.16› large 유형의 최대 인스턴스 수입니다.	0	예	예	예	예	예	아니요
하이브리드 작업에 대한 ml.m5.large의 최대 인스턴스 수	이 계정과 리전의 모든 Amazon Braket Hybrid Jobs에 허용되는 ml.m5.large 유형의 최대 인스턴스 수입니다.	5	예	예	예	예	예	예

Resource	설명	한도	조정 가능	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
하이브리드 작업에 대한 ml.m5.xlarge의 최대 인스턴스 수	이 계정과 리전의 모든 Amazon Braket Hybrid Jobs에 허용되는 ml.m5.xlarge 유형의 최대 인스턴스 수입니다.	5	예	예	예	예	예	예
하이브리드 작업에 대한 ml.m5.2xlarge의 최대 인스턴스 수	이 계정과 리전의 모든 Amazon Braket Hybrid Jobs에 허용되는 ml.m5.2xlarge 유형의 최대 인스턴스 수입니다.	5	예	예	예	예	예	예

Resource	설명	한도	조정 가능	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
하이브리드 작업에 대한 ml.m5.4xlarge의 최대 인스턴스 수	이 계정과 리전의 모든 Amazon Braket Hybrid Jobs에 허용되는 ml.m5.4xlarge 유형의 최대 인스턴스 수입니다.	5	예	예	예	예	예	예
하이브리드 작업에 대한 ml.m5.12xlarge의 최대 인스턴스 수	이 계정과 리전의 모든 Amazon Braket Hybrid Jobs에 허용되는 ml.m5.12xlarge 유형의 최대 인스턴스 수입니다.	0	예	예	예	예	예	예

Resource	설명	한도	조정 가능	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
하이브리드 작업에 대한 ml.m5.24› large의 최대 인스턴스 수	이 계정과 리전의 모든 Amazon Braket Hybrid Jobs에 허용되는 ml.m5.24› large 유형의 최대 인스턴스 수입니다.	0	예	예	예	예	예	예
하이브리드 작업에 대한 ml.p2.xlarge의 최대 인스턴스 수	이 계정과 리전의 모든 Amazon Braket Hybrid Jobs에 허용되는 ml.p2.xlarge 유형의 최대 인스턴스 수입니다.	0	예	예	아니요	예	아니요	아니요

Resource	설명	한도	조정 가능	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
하이브리드 작업에 대한 ml.p2.8xlarge의 최대 인스턴스 수	이 계정과 리전의 모든 Amazon Braket Hybrid Jobs에 허용되는 ml.p2.8xlarge 유형의 최대 인스턴스 수입니다.	0	예	예	아니요	예	아니요	아니요
하이브리드 작업에 대한 ml.p2.16xlarge의 최대 인스턴스 수	이 계정과 리전의 모든 Amazon Braket Hybrid Jobs에 허용되는 ml.p2.16xlarge 유형의 최대 인스턴스 수입니다.	0	예	예	아니요	예	아니요	아니요

Resource	설명	한도	조정 가능	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
하이브리드 작업에 대한 ml.p4d.24xlarge의 최대 인스턴스 수	이 계정과 리전의 모든 Amazon Braket Hybrid Jobs에 허용되는 ml.p4d.24xlarge 유형의 최대 인스턴스 수입니다.	0	예	예	아니요	예	아니요	아니요

한도 업데이트 요청

인스턴스 유형에 대해 ServiceQuotaExceeded 예외가 수신되고 인스턴스에 사용할 수 있는 인스턴스가 충분하지 않은 경우AWS콘솔의 Service Quotas 페이지에서 한도 증가를 요청하고AWS서비스에서 Amazon Braket를 검색할 수 있습니다.

> **Note**
>
> 하이브리드 작업이 요청된 ML 컴퓨팅 용량을 프로비저닝할 수 없는 경우 다른 리전을 사용합니다. 또한 표에 인스턴스가 표시되지 않는 경우 하이브리드 작업에 사용할 수 없습니다.

추가 할당량 및 한도

- Amazon Braket 양자 작업 관련 작업은 크기가 3MB로 제한됩니다.

- SV1의 경우 최대 실행 시간은 31큐비트 이하 회로의 경우 3시간이고, 31큐비트 이상 회로의 경우 11시간입니다.
- SV1, DM1및 Rigetti 디바이스에 허용되는 작업당 최대 샷 수는 50,000개입니다.
- TN1에 허용되는 작업당 최대 샷 수는 1,000개입니다.
- AQT의 IBEX-Q1 디바이스의 경우 최대 샷 수는 작업당 2,000회입니다.
- 모든 디바이스IonQ의 경우: 작업당 최소 샷 수는 100입니다. 온디맨드 모델을 사용하는 경우 게이트 샷 한도는 1백만 개이고 오류 완화 작업의 경우 최소 2,500회입니다. 직접 예약의 경우 게이트샷 제한이 없으며 오류 완화 작업을 위한 최소 샷 수는 500개입니다.
- QuEra의 Aquila 디바이스의 경우 작업당 최대 샷 수는 1,000개입니다.
- IQM의 Garnet 및 Emerald 디바이스의 경우 최대 샷 수는 작업당 20,000회입니다.
- TN1 및 QPU 디바이스의 경우 작업당 샷 수는 > 0이어야 합니다.

Amazon Braket 개발자 안내서의 문서 기록

다음 표에서는 Amazon Braket에 대한 설명서 릴리스를 설명합니다.

- 최종 API 참조 업데이트: 2025년 11월 20일
- 최종 설명서 업데이트: 2026년 1월 14일

변경 사항	설명	날짜
"예약 작업" 페이지 업데이트	"예약 작업" 페이지의 명확성 개선	2026년 2월 3일
Braket 노트북 및 관리형 컨테이너에 대한 Python 버전 3.12 지원	Amazon Braket 노트북 및 관리형 컨테이너(기본, CUDA-Q, PennyLane 및 Tensorflow)에 대한 Python 버전 3.12 지원이 추가되었습니다. Python 3.12 업그레이드에 대한 문제 해결 안내서가 포함되어 있습니다.	2026년 1월 21일
P3 예제 제거	SageMaker는 `ml.p3` 인스턴스 패밀리를 사용 중지하고 있습니다. 예제를 권장 `ml.g4dn` 인스턴스 패밀리로 대체했습니다.	2025년 12월 19일
새로운 지출 한도 기능	구성된 지출 임계값을 초과하는 작업을 자동으로 검증하고 거부하는 개별 QPUs에 선택적 예산 한도를 설정할 수 있는 Amazon Braket 지출 한도 기능에 대한 지원이 추가되었습니다.	2025년 11월 20일
새 Braket 디바이스 AQT IBEX-Q1	AQT IBEX-Q1 디바이스에 대한 지원이 추가되었습니다. 이 디바이스는 초고진공실에 배치	2025년 11월 18일

	된 거시적 무선 주파수 트랩에서 $^{40}Ca^{+}$의 수정을 기반으로 합니다.	
Amazon Braket NBI에서의 CUDA-Q에 대한 새로운 기본 지원	Amazon Braket 노트북 인스턴스에서 CUDA-Q에 대한 기본 지원이 추가되었습니다. 자세한 내용은 NBI에서의 CUDA-Q를 참조하세요.	2025년 11월 10일
IonQ Aria-2 디바이스 사용 중지	IonQ Aria-2 디바이스에 대한 지원이 제거되었습니다.	2025년 10월 27일
통합 하이브리드 작업 설명서	Amazon Braket Hybrid Jobs 작업 아래에 표시되는 하이브리드 작업 섹션을 통합했습니다.	2025년 10월 21일
새 Braket, CUDA-Q 컨테이너 제공	제공된 CUDA-Q 하이브리드 작업 컨테이너에 대한 지원이 추가되었습니다. 자세한 내용은 알고리즘 스크립트의 환경 정의를 참조하세요.	2025년 9월 2일
새로운 로컬 디바이스 에뮬레이터 기능	양자 디바이스에 제출하기 전에 축어적 프로그램을 에뮬레이션하는 로컬 양자 디바이스 에뮬레이터 도구에 대한 지원이 추가되었습니다.	2025년 8월 25일
빌드 섹션 아래의 Pennylane 및 CUDA-Q 페이지 이동	Pennylane 및 CUDA-Q 페이지가 목차의 빌드 섹션 아래에 표시되도록 이동했습니다.	2025년 8월 15일
새로운 `ProgramSet` 기능	단일 양자 작업에서 여러 양자 회로를 실행하는 연산인 프로그램 세트에 대한 지원이 추가되었습니다.	2025년 8월 14일

새 디바이스 IQM Emerald	IQM Emerald 디바이스에 대한 지원이 추가되었습니다. 정사각형(결정) 격자 토폴로지를 갖는 54큐비트 디바이스입니다.	2025년 7월 21일
AmazonBraketServiceRolePolicy 정책 업데이트	AmazonBraketServiceRolePolicy는 이제 aws:PrincipalAccount에 s3:* 및 log:* 작업만 제공합니다. 따라서 요청자의 버킷 및 로그 그룹으로만 액세스가 제한됩니다.	2025년 7월 11일
새로운 실험 기능 특성: 동적 회로	중간 회로 측정 및 피드포워드 연산은 실험 기능으로 사용할 수 있습니다. IQM 디바이스의 동적 회로에 대한 액세스를 참조하세요.	2025년 6월 26일
AmazonBraketFullAccess 정책 업데이트	AmazonBraketFullAccess에 이제 하드웨어 비용을 콘솔에 표시하는 pricing:GetProducts가 포함됩니다.	2025년 4월 14일
새 디바이스 IonQ Forte-Enterprise-1	IonQ Forte-Enterprise-1 디바이스에 대한 지원이 추가되었습니다. 트랩된 이온 기술을 활용하는 36큐비트 디바이스입니다.	2025년 3월 17일
S3 조건 권한 개선	보안을 개선하기 위해 `AmazonBraketFullAccess` 는 이제 `aws:PrincipalAccount` 에만 `s3:*` 작업을 제공합니다. 따라서 요청자의 자체 버킷으로만 액세스가 제한됩니다.	2025년 3월 7일

새 디바이스 Rigetti Ankaa-3	Rigetti Ankaa-3 디바이스에 대한 지원이 추가되었습니다. 확장 가능한 멀티칩 기술을 활용하는 84큐비트 디바이스입니다.	2025년 1월 14일
Rigetti Ankaa-2 디바이스 사용 중지	Rigetti Ankaa-2 디바이스에 대한 지원이 제거되었습니다.	2025년 1월 14일
IPv6 트래픽 지원	Amazon Braket은 이제 듀얼 스택 엔드포인트 `braket.{region}.api.aws` 를 사용하여 IPv6 트래픽을 지원합니다.	2024년 12월 12일
Amazon Braket에서 NVIDIA's CUDA-Q 지원	이제 고객은 Amazon Braket에서 NVIDIA's CUDA-Q 개발자 프레임워크를 사용하여 양자 프로그램을 실행할 수 있습니다.	2024년 12월 6일
IonQ Forte-1 디바이스 즉시 사용 가능	IonQ Forte-1 디바이스는 더 이상 예약 전용이 아니며 이제 고객은 즉시 사용할 수 있습니다.	2024년 11월 22일
Rigetti Aspen-M-3 디바이스 사용 중지	Rigetti Aspen-M-3 디바이스에 대한 지원이 제거되었습니다.	2024년 9월 27일
IonQ Harmony 디바이스 사용 중지	IonQ Harmony 디바이스에 대한 지원이 제거되었습니다.	2024년 8월 29일
새 디바이스 Rigetti Ankaa-2	Rigetti Ankaa-2 디바이스에 대한 지원이 추가되었습니다. 확장 가능한 멀티칩 기술을 활용하는 84큐비트 디바이스입니다.	2024년 8월 26일

개발자 안내서 재구성	새 개발자 안내서는 기존의 빌드, 테스트, 실행 고객 여정을 살펴보고 Amazon Braket을 통해 이 경로를 따라 사용자를 안내합니다.	2024년 8월 23일
OQC Lucy 디바이스 사용 중지	OQC Lucy 디바이스에 대한 지원이 제거되었습니다.	2024년 6월 28일
새 디바이스 IQM Garnet 및 리전 Europe North 1	IQM Garnet 디바이스에 대한 지원이 추가되었습니다. 정사각형 격자 토폴로지를 갖춘 20큐비트 디바이스입니다. Braket 지원 리전을 Europe North 1(스톡홀름)로 확장했습니다.	2024년 5월 22일
로컬 디튜닝 릴리스	실험 기능에 이제 QuEra의 Aquila QPU의 로컬 디튜닝 기능이 포함됩니다.	2024년 4월 11일
노트북 비활성 관리자 릴리스	노트북 인스턴스를 생성할 때 비활성 관리자를 활성화하고 유휴 지속 시간을 설정하여 Braket 노트북 인스턴스를 자동으로 초기화합니다.	2024년 3월 27일
목차 재작업	AWS 스타일 가이드 요구 사항을 준수하고 고객 경험을 위한 콘텐츠 흐름을 개선하기 위해 Amazon Braket 목차를 재구성했습니다.	2023년 12월 12일

Braket Direct 릴리스	다음을 포함한 Braket Direct 기능에 대한 지원이 추가되었습니다. • 예약 작업 • 전문가 조언 받기 • 실험 기능 살펴보기	2023년 11월 27일
Amazon Braket 노트북 인스턴스 생성 업데이트	신규 및 기존 Amazon Braket 고객을 위한 노트북 인스턴스를 생성하는 정보를 추가하도록 설명서를 업데이트했습니다.	2023년 11월 27일
자체 컨테이너 사용(Bring Your Own Container, BYOC) 업데이트	BYOC 시기, BYOC에 대한 레시피, 컨테이너에서 Braket Hybrid Jobs 실행에 대한 정보를 추가하도록 설명서를 업데이트했습니다.	2023년 10월 18일
하이브리드 작업 데코레이터 릴리스	로컬 코드를 하이브리드 작업으로 실행 페이지가 추가되었습니다. 다음 예제를 포함합니다. • 로컬 Python 코드에서 하이브리드 작업 생성 • 추가 Python 패키지 및 소스 코드 설치 • 하이브리드 작업 인스턴스에 데이터 저장 및 로드 • 하이브리드 작업 데코레이터 모범 사례	2023년 10월 16일

대기열 가시성 추가	queue depth 및 queue position을 포함하도록 개발자 안내서 설명서를 업데이트했습니다. 대기열 가시성을 위한 새 API 변경 사항을 반영하도록 API 문서를 업데이트했습니다.	2023년 9월 25일
설명서의 이름 지정 표준화	"작업"(job)의 인스턴스를 "하이브리드 작업"으로 변경하고 "작업"(task)을 "양자 작업"으로 변경하도록 설명서를 업데이트했습니다.	2023년 9월 11일
새 디바이스 IonQ Aria 2	IonQ Aria 2 디바이스에 대한 지원 추가	2023년 9월 8일
네이티브 게이트 업데이트	Rigetti의 네이티브 게이트에 대한 프로그래밍 방식 액세스에 대한 정보를 추가하도록 설명서를 업데이트했습니다.	2023년 8월 16일
Xanadu 출발	모든 Xanadu 디바이스를 제거하도록 설명서 업데이트	2023년 6월 2일
새 디바이스 IonQ Aria	IonQ Aria 디바이스에 대한 지원 추가	2023년 5월 16일
Rigetti 디바이스 사용 중지	Rigetti Aspen-M-2에 대한 지원 중단	2023년 5월 2일
AmazonBraketFullAccess 정책 정보 업데이트	servicequotas:GetServiceQuota 및 cloudwatch:GetMetricData 작업을 비롯해 할당량과 관련된 제한에 대한 정보를 포함하도록 AmazonBraketFullAccess 정책의 내용을 정의하는 스크립트를 업데이트했습니다.	2023년 4월 19일

안내형 여정 출시	Braket 온보딩에 대한 최신 및 간소화된 방법을 반영하도록 설명서를 변경했습니다.	2023년 4월 5일
새 디바이스 Rigetti Aspen-M-3	Rigetti Aspen-M-3 디바이스에 대한 지원 추가	2023년 1월 17일
새로운 수반 기울기 기능	SV1에서 제공하는 수반 기울기 기능에 대한 정보 추가	2022년 12월 7일
새로운 알고리즘 라이브러리 기능	사전 빌드된 양자 알고리즘의 카탈로그를 제공하는 Braket 알고리즘 라이브러리에 대한 정보가 추가되었습니다.	2022년 11월 28일
D-Wave 출발	모든 D-Wave 디바이스 제거를 수용하도록 설명서 업데이트	2022년 11월 17일
새 디바이스 QuEra Aquila	QuEra Aquila 디바이스에 대한 지원 추가	2022년 10월 31일
Braket Pulse 지원	Rigetti 및 OQC 디바이스에서 펄스 제어를 사용할 수 있게 하는 Braket Pulse에 대한 지원이 추가되었습니다.	2022년 10월 20일
IonQ 네이티브 게이트 지원	IonQ 디바이스에서 제공하는 네이티브 게이트 세트에 대한 지원 추가	2022년 9월 13일
새 인스턴스 할당량	하이브리드 작업과 연결된 기본 고전 컴퓨팅 인스턴스 할당량 업데이트	2022년 8월 22일
새 서비스 대시보드	서비스 대시보드를 포함하도록 콘솔 스크린샷 업데이트	2022년 8월 17일
새 디바이스 Rigetti Aspen-M-2	Rigetti Aspen-M-2 디바이스에 대한 지원 추가	2022년 8월 12일

새로운 OpenQASM 기능	로컬 시뮬레이터(braket_sv 및 Braket_dm)에 대한 OpenQASM 기능 지원 추가	2022년 8월 4일
새로운 비용 추적 절차	시뮬레이터 및 하드웨어 워크로드의 최대 비용을 거의 실시간으로 추정하는 방법 추가	2022년 7월 18일
새 Xanadu Borealis 디바이스	Xanadu Borealis 디바이스에 대한 지원 추가	2022년 6월 2일
새로운 온보딩 간소화 절차	신규 및 간소화된 온보딩 절차의 작동 방식에 대한 정보 추가	2022년 5월 16일
새 디바이스 D-Wave Advantage_system6.1	D-Wave Advantage_system6.1 디바이스에 대한 지원 추가	2022년 5월 12일
임베디드 시뮬레이터 지원	하이브리드 작업으로 임베디드 시뮬레이션을 실행하는 방법과 PennyLane 라이트닝 시뮬레이터를 사용하는 방법 추가	2022년 5월 4일
AmazonBraketFullAccess - Amazon Braket에 대한 전체 액세스 정책	사용자가 Amazon Braket에 생성 및 사용되는 버킷을 보고 검사할 수 있도록 s3:ListAllMyBuckets 권한 추가	2022년 3월 31일
OpenQASM 지원	게이트 기반 양자 디바이스 및 시뮬레이터에 대한 OpenQASM 3.0 지원 추가	2022년 3월 7일
새로운 양자 하드웨어 공급자 Oxford Quantum Circuits 및 새로운 리전 eu-west-2	OQC 및 eu-west-2에 대한 지원 추가	2022년 2월 28일
새 Rigetti 디바이스	Rigetti Aspen M-1에 대한 지원 추가	2022년 2월 15일

새로운 리소스 제한	동시 DM1 및 SV1 작업의 최대 수를 55개에서 100개로 늘림	2022년 1월 5일
새 Rigetti 디바이스	Rigetti Aspen-11에 대한 지원 추가	2021년 12월 20일
Rigetti 디바이스 사용 중지	Rigetti Aspen-10 디바이스에 대한 지원 중단	2021년 12월 20일
새 결과 유형	로컬 밀도 행렬 시뮬레이터 및 DM1 디바이스에 의해 지원되는 축약 밀도 행렬 결과 유형	2021년 12월 20일
정책 설명 업데이트	Amazon Braket이 `servicerole/` 경로를 포함하도록 역할 ARN을 업데이트했습니다. 정책 업데이트에 대한 자세한 내용은 AWS 관리형 정책에 대한 Amazon Braket 업데이트 표를 참조하세요.	2021년 11월 29일
Amazon Braket 작업	Amazon Braket Hybrid Jobs 사용자 가이드 및 API 추가	2021년 11월 29일
새 Rigetti 디바이스	Rigetti Aspen-10에 대한 지원 추가	2021년 11월 20일
D-Wave 디바이스 사용 중지	D-Wave QPU, Advantage_system1에 대한 지원 중단	2021년 11월 4일
새 D-Wave 디바이스	추가 D-Wave QPU, Advantage_system4에 대한 지원 추가	2021년 10월 5일
새로운 노이즈 시뮬레이터	최대 17qubits 회로와 로컬 노이즈 시뮬레이터 Braket_dm을 시뮬레이션할 수 있는 밀도 행렬 시뮬레이터(DM1)에 대한 지원 추가	2021년 5월 25일

PennyLane 지원	Amazon Braket에서의 PennyLane에 대한 지원 추가	2020년 12월 8일
새 시뮬레이터	대규모 회로를 허용하는 Tensor Network Simulator (TN1)에 대한 지원 추가	2020년 12월 8일
작업 일괄 처리	Braket, 고객 작업 일괄 처리 지원	2020년 11월 24일
수동 qubit 할당	Braket, Rigetti 디바이스에 대한 수동 qubit 할당 지원	2020년 11월 24일
조정 가능한 할당량	Braket, 작업 리소스에 대한 셀프 서비스 조정 가능한 할당량 지원	2020년 10월 30일
PrivateLink 지원	Braket 작업에 대한 비공개 VPC 엔드포인트 설정 가능	2020년 10월 30일
태그 지원	Braket, quantum-task 리소스에 대한 API기반 태그 지원	2020년 10월 30일
새 D-Wave 디바이스	추가 D-Wave QPU, Advantage_system1에 대한 지원 추가	2020년 9월 29일
초기 릴리스	Amazon Braket 설명서의 최초 릴리스	2020년 8월 12일

기계 번역으로 제공되는 번역입니다. 제공된 번역과 원본 영어의 내용이 상충하는 경우에는 영어 버전이 우선합니다.