aws

開發人員指南

Amazon Braket

Amazon Braket: 開發人員指南

Amazon 的商標和商業外觀不得用於任何非 Amazon 的產品或服務，也不能以任何可能造成客戶混淆、任何貶低或使 Amazon 名譽受損的方式使用 Amazon 的商標和商業外觀。所有其他非 Amazon 擁有的商標均為其各自擁有者的財產，這些擁有者可能附屬於 Amazon，或與 Amazon 有合作關係，亦或受到 Amazon 贊助。

Table of Contents

什麼是 Amazon Braket？

> **Tip**
>
> 了解量子運算的基礎 AWS！註冊 Amazon Braket 數位學習計劃，在完成一系列的學習課程和數位評估後獲得自己的數位徽章。

Amazon Braket 是全受管 AWS 服務 的，可協助研究人員、科學家和開發人員開始使用量子運算。Quantum 運算有可能解決傳統電腦無法觸及的運算問題，因為它利用量子機制法以新方式處理資訊。

取得量子運算硬體的存取權可能既昂貴又不方便。有限的存取讓執行演算法、最佳化設計、評估目前的技術狀態，以及規劃何時投資您的資源以獲得最大利益變得困難。Braket 可協助您克服這些挑戰。

Braket 提供對各種量子運算技術的單一存取點。使用 Braket，您可以：

- 探索和設計量子和混合演算法。
- 在不同的量子電路模擬器上測試演算法。
- 在不同類型的量子電腦上執行演算法。
- 建立概念驗證應用程式。

定義量子問題和程式設計量子電腦來解決它們需要一組新的技能。為了協助您獲得這些技能，Raket 提供不同的環境來模擬和執行您的量子演算法。您可以找到最符合您需求的方法，並快速開始使用一組稱為筆記本的範例環境。

Braket 開發有三個階段：

- 建置 - Braket 提供全受管的 Jupyter 筆記本環境，可讓您直接開始使用。Braket 筆記本預先安裝了範例演算法、資源和開發人員工具，包括 Amazon Braket SDK。使用 Amazon Braket SDK，您可以建立量子演算法，然後透過變更單行程式碼，在不同的量子電腦和模擬器上測試和執行這些演算法。
- 測試 - Braket 可讓您存取全受管、高效能量子電路模擬器。您可以測試和驗證您的電路。Braket 會處理所有基礎軟體元件和 Amazon Elastic Compute Cloud (Amazon EC2) 叢集，以減輕傳統高效能運算 (HPC) 基礎設施上模擬量子電路的負擔。
- Run - Braket 提供不同類型的量子電腦的安全隨需存取。您可以從 IonQ、 IQM和 存取閘道型量子電腦Rigetti，以及從 QuEra 存取類比 Hamiltonian Simulator。您也不需要預先承諾，也不需要透過個別供應商取得存取權。

關於量子運算和 Braket

Quantum 運算處於早期開發階段。請務必了解目前沒有通用、容錯的量子電腦。因此，某些類型的量子硬體更適合每個使用案例，而且存取各種運算硬體至關重要。Braket 透過第三方供應商提供各種硬體。

現有的量子硬體因雜訊而受限，這會引入錯誤。產業處於雜訊中繼縮放量子 (NISQ) 時代。在 NISQ 時代，量子運算裝置過於吵雜，無法維持純量子演算法，例如 Shor 的演算法或 Grover 的演算法。直到有更好的量子錯誤校正可用之前，最實用的量子運算需要結合傳統（傳統）運算資源與量子電腦來建立混合演算法。Braket 可協助您使用混合量子演算法。

在混合量子演算法中，量子處理單元 (QPUs) 用作 CPUs 的共同處理器，從而加快傳統演算法中的特定計算速度。這些演算法使用反覆處理，其中運算會在傳統和量子電腦之間移動。例如，目前在化學、最佳化和機器學習中量子運算的應用程式是以變化量子演算法為基礎，這是一種混合量子演算法。在變化量子演算法中，傳統最佳化常式會反覆調整參數化量子電路的參數，就像根據機器學習訓練集中的錯誤反覆調整神經網路權重一樣。Braket 可讓您存取 PennyLane 開放原始碼軟體程式庫，該程式庫可協助您處理變化量子演算法。

Quantum 運算正在四個主要區域中取得運算的抓地力：

- 數字理論 – 包括因素和密碼編譯（例如，Shor 的演算法是數字理論運算的主要量子方法）
- 最佳化 - 包括限制滿意度、解決線性系統和機器學習
- 眼球運算 - 包括搜尋、隱藏子群組和訂單調查結果（例如，Grover 的演算法是眼球運算的主要量子方法）
- 模擬 — 包括直接模擬、綁定變異數和量子近似最佳化演算法 (QAOA) 應用程式

這些類別的運算應用程式可在金融服務、生物技術、製造和製藥中找到。Braket 提供功能和範例筆記本，除了某些實際問題之外，還可以套用至許多概念驗證問題。

在本節中：

Amazon Braket 的運作方式

> **Tip**
>
> 了解量子運算的基礎 AWS！註冊 Amazon Braket 數位學習計劃，在完成一系列的學習課程和數位評估後獲得自己的數位徽章。

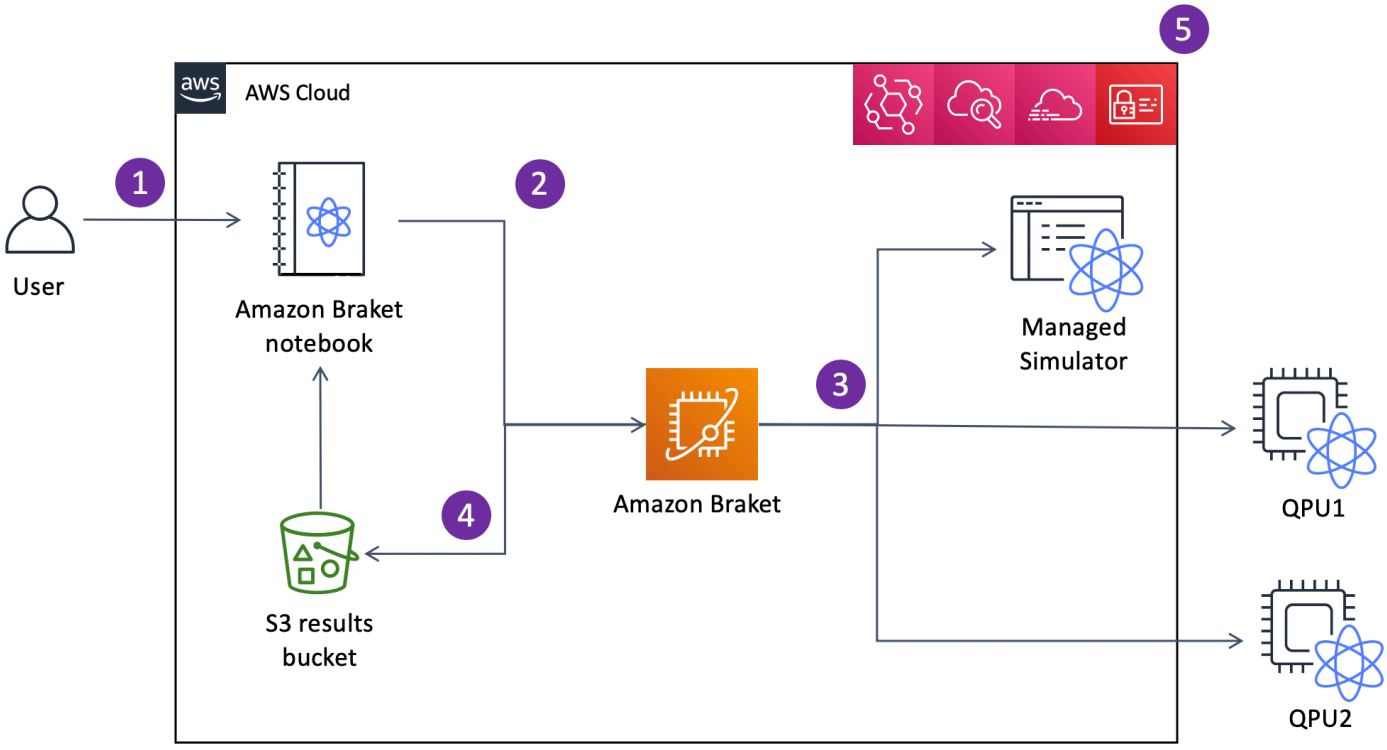
Amazon Braket 提供量子運算裝置的隨需存取，包括隨需電路模擬器和不同類型的 QPUs。在 Amazon Braket 中，對裝置的原子請求是量子任務。對於以閘道為基礎的 QC 裝置，此請求包含量子電路（包括測量指示和鏡頭數量）和其他請求中繼資料。對於類比 Hamiltonian Simulators，量子任務包含量子登錄的實體配置，以及操縱欄位的時間和空間相依性。

Braket Direct 是擴展如何探索量子運算 AWS、加速研究和創新的計畫。您可以在各種量子裝置上保留專用容量、直接與量子運算專家互動，以及提早存取新一代功能，包括來自 IonQ、Forte 的最新截獲離子裝置。

在本節中，我們將了解在 Amazon Braket 上執行量子任務的高階流程。

在本節中：

Amazon Braket 量子任務流程



使用Jupyter筆記本，您可以從 Amazon Braket 主控台或使用 Amazon Braket SDK 輕鬆定義、提交和監控您的量子任務。您可以直接在 SDK 中建置量子電路。不過，對於類比 Hamiltonian Simulators，您可以定義註冊配置和控制欄位。定義規定人數任務之後，您可以選擇要在其上執行的裝置，並將其提交至 Amazon Braket API (2)。根據您選擇的裝置，量子任務會排入佇列，直到裝置變成可用，並將任務傳送至 QPU 或模擬器以進行實作 (3)。Amazon Braket 可讓您存取不同類型的 QPUs (IonQ、IQM、QuEra、Rigetti)、三個隨需模擬器 (SV1、DM1、TN1)、兩個本機模擬器和一個內嵌模擬器。若要進一步了解，請參閱 Amazon Braket 支援的裝置。

處理您的量子任務後，AmazonRaket 會將結果傳回至 Amazon S3 儲存貯體，資料會存放在您的 AWS 帳戶 (4) 中。同時，軟體開發套件會在背景輪詢結果，並在量子任務完成時將其載入 Jupyter 筆記本。您也可以在 Amazon Braket 主控台的 Quantum 任務頁面上或使用 Amazon Braket `GetQuantumTask`的操作來檢視和管理量子任務API。

Amazon Braket 與 AWS Identity and Access Management (IAM)、Amazon CloudWatch AWS CloudTrail 和 Amazon EventBridge 整合，用於使用者存取管理、監控和記錄，以及事件型處理 (5)。

第三方資料處理

提交至 QPU 裝置的量子任務會在第三方供應商所操作設施中的量子電腦上處理。若要進一步了解 Amazon Braket 中的安全和第三方處理，請參閱 Amazon Braket 硬體供應商的安全性。

Amazon Braket 術語和概念

> **Tip**
>
> 了解量子運算的基礎 AWS！註冊 Amazon Braket 數位化學習計劃，在完成一系列的學習課程和數位評估後獲得自己的數位徽章。

下列術語和概念用於 Braket：

類比漢密爾頓模擬

類比漢密爾頓模擬 (AHS) 是一種獨特的量子運算範例，用於直接模擬多體系統的時間相依量子動態。在 AHS 中，使用者直接指定時間相依的漢米爾頓文，量子電腦的調校方式會直接模擬此漢米爾頓文下的持續時間演變。AHS 裝置通常是特殊用途的裝置，而不是通用量子電腦，例如以閘道為基礎的裝置。它們僅限於他們可以模擬的 Hamiltonian 類別。不過，由於這些 Hamiltonians 是自然實作在裝置上，因此 AHS 不會承受將演算法建構為電路和實作閘道操作所需的額外負荷。

剎車

我們在 bra-ket 表示法之後命名了 Braket 服務，這是量子力學中的標準表示法。它由 Paul Dirac 於 1939 年引入，用於描述量子系統的狀態，也稱為 Dirac 表示法。

Braket Direct

使用 Braket Direct，您可以預留對您選擇的不同量子裝置的專用存取權、與量子運算專家連線，以接收工作負載的指引，以及儘早存取新一代功能，例如可用性有限的新量子裝置。

Braket 混合任務

Amazon Braket 具有稱為 Amazon Braket 混合任務的功能，可提供混合演算法的全受管執行。Braket 混合任務包含三個元件：

1. 演算法的定義，可做為指令碼、Python 模組或 Docker 容器提供。
2. 混合式任務執行個體，以 Amazon EC2 為基礎，執行您的演算法。預設值為 ml.m5.xlarge 執行個體。

3. 要在其中執行屬於演算法之量子任務的量子裝置。單一混合任務通常包含許多量子任務的集合。

裝置

在 Amazon Braket 中，裝置是可以執行量子任務的後端。裝置可以是 QPU 或量子電路模擬器。若要進一步了解，請參閱 Amazon Braket 支援的裝置。

錯誤緩解

錯誤緩解包括執行多個實體電路，並結合其測量結果來改善結果。如需詳細資訊，請參閱錯誤緩解技術。

閘道式量子運算

在閘道式量子運算 (QC) 中，也稱為電路式 QC，運算會細分為基本操作 （閘道）。某些閘道集是通用的，這表示每個運算都可以以這些閘道的有限序列表示。Gates 是量子電路的建置區塊，類似於傳統數位電路的邏輯閘道。

Gateshot 限制

閘道限制是指每個拍攝的總閘道計數 （所有閘道類型的總和） 和每個任務的拍攝計數。在數學上，Gateshot 限制可以表示為：

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

漢密爾頓文

實體系統的量子動態由其 Hamiltonian 決定，其編碼有關系統元件與外源性驅動力之間互動的所有資訊。N-qubit 系統的 Hamiltonian 通常表示為傳統機器上複雜數字的 2^N x 2^N 矩陣。透過在量子裝置上執行類比 Hamiltonian 模擬，您可以避免這些指數資源需求。

脈衝

脈衝是傳輸到 qubit 的暫時性實體訊號。其描述方式是在做為電信業者訊號支援且繫結至硬體通道或連接埠的影格中播放的波形。客戶可以提供模擬信封來調節高頻率正弦波載波訊號，藉此設計自己的脈衝。影格由頻率和階段唯一描述，該頻率和階段通常被選擇為正進行振盪，並在 qubit 的 |0# 和 |1# 的能量層級之間進行能量分離。因此，閘道會以具有預定形狀和校正參數的脈波來制定，例如其振幅、頻率和持續時間。範本波形未涵蓋的使用案例將透過自訂波形啟用，並透過提供以固定的實體週期時間分隔的值清單，在單一範例解析度中指定。

Quantum 電路

量子電路是定義閘道式量子電腦上運算的說明集。量子電路是量子閘道的序列，這是qubit暫存器上的可逆轉換，以及測量指示。

Quantum 電路模擬器

量子電路模擬器是一種電腦程式，可在傳統電腦上執行，並計算量子電路的測量結果。對於一般電路，量子模擬的資源需求隨qubits要模擬的 數量呈指數增長。Braket 提供受管 （透過 Braket 存取 API) 和本機 (Amazon Braket SDK 的一部分） 量子電路模擬器的存取權。

Quantum 電腦

量子電腦是一種實體裝置，使用量子機械現象，例如疊加和糾結，來執行運算。量子運算 (QC) 有不同的範例，例如以閘道為基礎的 QC。

Quantum 處理單元 (QPU)

QPU 是一種實體量子運算裝置，可在量子任務上執行。QPUs可以根據不同的 QC 範例，例如以閘道為基礎的 QC。若要進一步了解，請參閱 Amazon Braket 支援的裝置。

QPU 原生閘道

QPU 原生閘道可以直接映射，以透過 QPU 控制系統控制脈衝。原生閘道可以在 QPU 裝置上執行，無需進一步編譯。QPU 支援的閘道子集。您可以在 Amazon Braket 主控台的裝置頁面上，以及透過 Braket SDK 找到裝置的原生閘道。

QPU 支援的閘道

QPU 支援的閘道是 QPU 裝置接受的閘道。這些閘道可能不會直接在 QPU 上執行，這表示它們可能需要分解為原生閘道。您可以在 Amazon Braket 主控台的裝置頁面上，以及透過 Amazon Braket SDK 找到裝置支援的閘道。

Quantum 任務

在 Braket 中，量子任務是對裝置的原子請求。對於以閘道為基礎的 QC 裝置，這包含量子電路（包括測量指示和 的數量shots) 和其他請求中繼資料。您可以透過 Amazon Braket SDK 或直接使用 CreateQuantumTaskAPI操作來建立量子任務。建立量子任務之後，它會排入佇列，直到請求的裝置變成可用為止。您可以在 Amazon Braket 主控台的 Quantum 任務頁面上或使用 GetQuantumTask或 SearchQuantumTasksAPI操作來檢視您的量子任務。

Qubit

量子電腦中的基本資訊單位稱為 qubit（量子位元），類似於傳統運算中的一些。qubit 是雙階量子系統，可透過不同的實體實作實現，例如超導電路或個別的離子和原子。其他qubit類型是根據光子、電子或核子旋轉，或更奇特的量子系統。

Queue depth

Queue depth 是指針對特定裝置排入佇列的量子任務和混合任務數量。裝置的量子任務和混合任務佇列計數可透過 Braket Software Development Kit (SDK)或 存取Amazon Braket Management Console。

1. 任務佇列深度是指等待以正常優先順序執行的量子任務總數。
2. 優先順序任務佇列深度是指等待透過 執行的已提交量子任務總數Amazon Braket Hybrid Jobs。混合式任務啟動後，這些任務會優先於獨立任務。
3. 混合任務佇列深度是指目前在裝置上排入佇列的混合任務總數。作為混合任務的一部分Quantum tasks提交 具有優先順序，並在 中彙總Priority Task Queue。

Queue position

Queue position 是指各自裝置佇列中量子任務或混合任務的目前位置。可以透過 或 取得量子任務Braket Software Development Kit (SDK)或混合任務Amazon Braket Management Console。

Shots

由於量子運算本質上具有機率，因此任何電路都需要評估多次，才能獲得準確的結果。單一電路執行和測量稱為鏡頭。根據結果所需的準確度來選擇電路的鏡頭 （ 重複執行 ） 數量。

AWS Amazon Braket 的術語和秘訣

IAM 政策

IAM 政策是允許或拒絕 AWS 服務 和 資源許可的文件。IAM 政策可讓您自訂使用者對 資源的存取層級。例如，您可以允許使用者存取 內的所有 Amazon S3 儲存貯體 AWS 帳戶，或只存取特定儲存貯體。

- 最佳實務：授予許可時，請遵循最低權限的安全原則。透過遵循此原則，您可以協助防止使用者或角色擁有比執行其量子任務所需的更多許可。例如，如果員工只需要存取特定儲存貯體，請在 IAM 政策中指定儲存貯體，而不是授予員工存取您 中所有儲存貯體的權限 AWS 帳戶。

IAM 角色

IAM 角色是您可以擔任的身分，以取得暫時存取許可。在使用者、應用程式或服務可以擔任 IAM 角色之前，必須授予他們切換到角色的許可。當某人擔任 IAM 角色時，他們會捨棄先前在先前角色下擁有的所有許可，並擔任新角色的許可。

- 最佳實務：IAM 角色非常適合需要暫時授予服務或資源存取權的情況，而非長期的情況。

Amazon S3 儲存貯體

Amazon Simple Storage Service (Amazon S3) 是 AWS 服務 ，可讓您將資料作為物件存放在儲存貯體中。Amazon S3 儲存貯體提供無限的儲存空間。Amazon S3 儲存貯體中物件的大小上限為 5 TB。您可以將任何類型的檔案資料上傳至 Amazon S3 儲存貯體，例如影像、影片、文字檔案、備份檔案、網站媒體檔案、封存文件，以及您的 Braket 量子任務結果。

- 最佳實務：您可以設定許可來控制對 S3 儲存貯體的存取。如需詳細資訊，請參閱 Amazon S3 文件中的儲存貯體政策。

成本追蹤和儲存

> **Tip**
>
> 了解量子運算的基礎 AWS！註冊 Amazon Braket 數位化學習計劃，在完成一系列的學習課程和數位評估後獲得自己的數位徽章。

透過 Amazon Braket，您可以隨需存取量子運算資源，而無需預先承諾。您僅需按實際用量付費。若要進一步了解定價，請造訪我們的定價頁面。

在本節中：

- 近乎即時的成本追蹤
- 節省成本的最佳實務

近乎即時的成本追蹤

Braket SDK 可讓您選擇將近乎即時的成本追蹤新增至量子工作負載。我們每個範例筆記本都包含成本追蹤程式碼，可為您提供 Braket 量子處理單元 (QPUs) 和隨需模擬器的最高成本預估。最高成本預估將以 USD 顯示，不包含任何點數或折扣。

> **Note**
>
> 顯示的費用是根據 Amazon Braket 模擬器和量子處理單元 (QPU) 任務用量估算。顯示的預估費用可能與您的實際費用不同。預估費用不會計入任何折扣或點數，而且您可能會因為使用 Amazon Elastic Compute Cloud (Amazon EC2) 等其他服務而產生額外費用。

SV1 的成本追蹤

為了示範如何使用成本追蹤函數，我們將建構 Bell State 電路，並在 SV1 模擬器上執行它。首先匯入 Braket SDK 模組、定義 Bell 狀態，並將 `Tracker()`函數新增至我們的電路：

```
#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})
```

當您執行筆記本時，您可以預期 Bell 狀態模擬有下列輸出。追蹤器函數會顯示傳送的鏡頭數量、完成的量子任務、執行持續時間、計費執行持續時間，以及以 USD 為單位的最高成本。每個模擬的執行時間可能有所不同。

```
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')
```

使用成本追蹤器設定最高成本

您可以使用成本追蹤器來設定程式的最高成本。對於您想要在指定程式上花費的金額，可能會有最大閾值。透過這種方式，您可以使用成本追蹤器在執行程式碼中建置成本控制邏輯。下列範例在 Rigetti QPU 上採用相同的電路，並將成本限制為 1 USD。在我們的程式碼中執行一次重複電路的成本為 0.30 USD。我們已將邏輯設定為重複反覆運算，直到總成本超過 1 USD；因此，程式碼片段將執行三次，直到下一次反覆運算超過 1 USD。一般而言，程式會繼續反覆運算，直到達到所需的最高成本為止，在此情況下為 - 三次反覆運算。

```
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**
>
> 成本追蹤器不會追蹤失敗TN1量子任務的持續時間。在TN1模擬期間，如果您的演練完成，但收縮步驟失敗，您的演練費用將不會顯示在成本追蹤器中。

節省成本的最佳實務

請考慮下列使用 Amazon Braket 的最佳實務。節省時間、降低成本，並避免常見的錯誤。

使用模擬器驗證

- 在 QPU 上執行之前，請使用模擬器驗證您的電路，因此您可以微調電路，而不會產生 QPU 使用費。
- 雖然在模擬器上執行電路的結果可能與在 QPU 上執行電路的結果不同，但您可以使用模擬器識別編碼錯誤或組態問題。

限制使用者存取特定裝置

- 您可以設定限制，防止未經授權的使用者在特定裝置上提交規定人數任務。限制存取的建議方法是使用 AWS IAM。如需如何執行此操作的詳細資訊，請參閱限制存取。

- 我們建議您不要使用管理員帳戶作為授予或限制使用者存取 Amazon Braket 裝置的方式。

設定帳單警示

- 您可以設定帳單警示，在帳單達到預設限制時通知您。設定警示的建議方法是透過 AWS Budgets。您可以設定自訂預算，並在成本或用量可能超過預算金額時收到提醒。如需相關資訊，請參閱 AWS Budgets。

低鏡頭計數的測試TN1量子任務

- 模擬器的成本低於 QPUs，但如果量子任務以高鏡頭計數執行，則某些模擬器可能會很昂貴。我們建議您使用低shot計數來測試TN1任務。 Shot計數不會影響 SV1和本機模擬器任務的成本。

檢查所有區域是否有量子任務

- 主控台只會針對您目前的 顯示量子任務 AWS 區域。尋找已提交的計費量子任務時，請務必檢查所有區域。
- 您可以在支援的裝置文件頁面上檢視裝置及其相關區域的清單。

Amazon Braket 的 API 參考和儲存庫

> **Tip**
>
> 了解量子運算的基礎 AWS！註冊 Amazon Braket 數位學習計劃，在完成一系列的學習課程和數位評估後獲得自己的數位徽章。

Amazon Braket 提供 APIs、 SDKs 和命令列界面，可用來建立和管理筆記本執行個體，以及訓練和部署模型。

- Amazon Braket Python SDK （建議）
- Amazon Braket API 參考
- AWS Command Line Interface
- 適用於 .NET 的 AWS SDK
- 適用於 C++ 的 AWS SDK

- 適用於 Go 的 AWS SDKAPI Reference
- 適用於 Java 的 AWS SDK
- 適用於 JavaScript 的 AWS SDK
- 適用於 PHP 的 AWS SDK
- AWS SDK for Python (Boto)
- 適用於 Ruby 的 AWS SDK

您也可以從 Amazon Braket 教學課程 GitHub 儲存庫取得程式碼範例。

- Braket 教學課程 GitHub

核心儲存庫

以下顯示核心儲存庫清單，其中包含用於 Braket 的金鑰套件：

- Braket Python SDK - 使用 Braket Python SDK 在Jupyter筆記本上以 Python 程式設計語言設定程式碼。設定Jupyter筆記本之後，您可以在 Braket 裝置和模擬器上執行程式碼
- Braket 結構描述 - Braket SDK 與 Braket 服務之間的合約。
- Braket 預設模擬器 - 我們所有適用於 Braket 的本機量子模擬器（狀態向量和密度矩陣）。

外掛程式

然後，有各種外掛程式與各種裝置和程式設計工具搭配使用。其中包括 Braket 支援的外掛程式，以及第三方支援的外掛程式，如下所示。

支援 Amazon Braket：

- Amazon Braket 演算法程式庫 - 以 Python 撰寫的預先建置量子演算法目錄。以原樣執行它們，或使用它們做為建置更複雜演算法的起點。
- Braket-PennyLane 外掛程式 - PennyLane用作 Braket 上的 QML 架構。

第三方 (Braket 團隊監控和貢獻）：

- Qiskit-Braket 提供者 - 使用 Qiskit SDK 存取 Braket 資源。
- Braket-Julia SDK - (EXPERIMENTAL) Braket SDK 的 Julia 原生版本

Amazon Braket 支援的區域和裝置

> **Tip**
>
> 了解量子運算的基礎 AWS！註冊 Amazon Braket 數位化學習計劃，在完成一系列的學習課程和數位評估後獲得自己的數位徽章。

在 Amazon Braket 中，裝置代表您可以呼叫以執行量子任務的 QPU 或模擬器。Amazon Braket 可讓您從 IonQ、QuEra、 IQM和 存取 QPU 裝置Rigetti、三個隨需模擬器、三個本機模擬器，以及一個內嵌模擬器。

如需支援量子硬體提供者的資訊，請參閱將量子任務提交至 QPUs。如需可用模擬器的資訊，請參閱將量子任務提交至模擬器。下表顯示可用裝置和模擬器的清單。

供應商	裝置名稱	範式	Type	裝置 ARN	區域
IonQ	Aria-1	以閘道為基礎的	QPU	arn：aws：braket：us-east-1：：device/qpu/ionq/Aria-1	us-east-1
IonQ	Aria-2	以閘道為基礎的	QPU	arn：aws：braket：us-east-1：：device/qpu/ionq/Aria-2	us-east-1
IonQ	Forte-1	以閘道為基礎的	QPU	arn：aws：braket：us-east-1：：device/qpu/ionq/Forte-1	us-east-1
IonQ	Forte-Enterprise-1	以閘道為基礎的	QPU	arn：aws：braket：us-east-1：：device/qpu/ionq/Forte-Enterprise-1	us-east-1
IQM	Garnet	以閘道為基礎的	QPU	arn：aws：braket：eu-north-1：：device/qpu/iqm/Garnet	eu-north-1

供應商	裝置名稱	範式	Type	裝置 ARN	區域
IQM	Emerald	以閘道為基礎的	QPU	arn：aws：braket：eu-north-1：：device/qpu/iqm/Emerald	eu-north-1
QuEra	Aquila	類比漢密爾頓模擬	QPU	arn：aws：braket：us-east-1：device/qpu/quera/Aquila	us-east-1
Rigetti	Ankaa-3	以閘道為基礎的	QPU	arn：aws：braket：us-west-1：：device/qpu/rigetti/Ankaa-3	us-west-1
AWS	braket_sv	以閘道為基礎的	本機模擬器	N/A（Braket SDK 中的本機模擬器）	N/A
AWS	braket_dm	以閘道為基礎的	本機模擬器	N/A（Braket SDK 中的本機模擬器）	N/A
AWS	braket_ahs	類比漢密爾頓模擬	本機模擬器	N/A（Braket SDK 中的本機模擬器）	N/A
AWS	SV1	以閘道為基礎的	隨需模擬器	arn：aws：braket：：device/quantum-simulator/amazon/sv1	us-east-1、us-west-1、us-west-2、eu-west-2
AWS	DM1	以閘道為基礎的	隨需模擬器	arn：aws：braket：：device/quantum-simulator/amazon/dm1	us-east-1、us-west-1、us-west-2、eu-west-2

供應商	裝置名稱	範式	Type	裝置 ARN	區域
AWS	TN1	以閘道為基礎的	隨需模擬器	arn：aws：braket：：device/quantum-simulator/amazon/tn1	us-east-1、us-west-2 和 eu-west-2

若要檢視可搭配 Amazon Braket 使用的 QPUs 的其他詳細資訊，請參閱 Amazon Braket Quantum Computers。

裝置屬性

對於所有裝置，您可以在 Amazon Braket 主控台的裝置索引標籤或 GetDevice API 上找到其他裝置屬性，例如裝置拓撲、校正資料和原生閘道集。使用模擬器建構電路時，Amazon Braket 會要求您使用連續的 qubits 或索引。使用 SDK 時，下列程式碼範例示範如何存取每個可用裝置和模擬器的裝置屬性。

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

device = AwsDevice('arn:aws:braket:::device/quantum-simulator/amazon/sv1')
 #SV1
# device = LocalSimulator()
 #Local State Vector Simulator
# device = LocalSimulator("default")
 #Local State Vector Simulator
# device = LocalSimulator(backend="default")
 #Local State Vector Simulator
# device = LocalSimulator(backend="braket_sv")
 #Local State Vector Simulator
# device = LocalSimulator(backend="braket_dm")
 #Local Density Matrix Simulator
# device = LocalSimulator(backend="braket_ahs")
 #Local Analog Hamiltonian Simulation
# device = AwsDevice('arn:aws:braket:::device/quantum-simulator/amazon/tn1')
 #TN1
# device = AwsDevice('arn:aws:braket:::device/quantum-simulator/amazon/dm1')
 #DM1
# device = AwsDevice('arn:aws:braket:us-east-1::device/qpu/ionq/Aria-1')
 #IonQ Aria-1
```

```
# device = AwsDevice('arn:aws:braket:us-east-1::device/qpu/ionq/Aria-2')
 #IonQ Aria-2
# device = AwsDevice('arn:aws:braket:us-east-1::device/qpu/ionq/Forte-1')
 #IonQ Forte-1
# device = AwsDevice('arn:aws:braket:us-east-1::device/qpu/ionq/Forte-Enterprise-1')
 #IonQ Forte-Enterprise-1
# device = AwsDevice('arn:aws:braket:eu-north-1::device/qpu/iqm/Garnet')
 #IQM Garnet
# device = AwsDevice('arn:aws:braket:eu-north-1::device/qpu/iqm/Emerald')
 #IQM Emerald
# device = AwsDevice('arn:aws:braket:us-east-1::device/qpu/quera/Aquila')
 #QuEra Aquila
# device = AwsDevice('arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3')
 #Rigetti Ankaa-3

# get device properties
device.properties
```

Amazon Braket 的區域和端點

如需區域和端點的完整清單，請參閱 AWS 一般參考。

在 QPU 裝置上執行的 Quantum 任務可以在該裝置區域的 Amazon Braket 主控台中檢視。使用 Amazon Braket SDK 時，您可以將量子任務提交至任何 QPU 裝置，無論您在哪個區域工作。SDK 會自動為指定的 QPU 建立 區域的工作階段。

Amazon Braket 可在下列內容中使用 AWS 區域：

區域名稱	區域	Braket 端點
美國東部 (維吉尼亞北部)	us-east-1	braket.us-east-1.amazonaws.com （僅限 IPv4） braket.us-east-1.api.aws （雙堆疊）
美國西部 (加利佛尼亞北部)	us-west-1	braket.us-west-1.amazonaws.com （僅限 IPv4） braket.us-west-1.api.aws （雙堆疊）

區域名稱	區域	Braket 端點
美國西部 2（奧勒岡）	us-west-2	braket.us-west-2.amazonaws.com（僅限 IPv4） braket.us-west-2.api.aws（雙堆疊）
歐洲北部 1 號（斯德哥爾摩）	eu-north-1	braket.eu-north-1.amazonaws.com（僅限 IPv4） braket.eu-north-1.api.aws（雙堆疊）
歐洲西部 2（倫敦）	eu-west-2	braket.eu-west-2.amazonaws.com（僅限 IPv4） braket.eu-west-2.api.aws（雙堆疊）

> **Note**
>
> Amazon Braket SDK 不支援IPv6-only的網路。

Amazon Braket 入門

> **Tip**
>
> 了解量子運算的基礎 AWS！註冊 Amazon Braket 數位化學習計劃，在完成一系列的學習課程和數位評估後獲得自己的數位徽章。

遵循啟用 Amazon Braket 中的指示之後，您就可以開始使用 Amazon Braket。

入門的步驟包括：

- 啟用 Amazon Braket
- 建立 Amazon Braket 筆記本執行個體
- 使用 建立 Braket 筆記本執行個體 AWS CloudFormation

啟用 Amazon Braket

> **Tip**
>
> 了解量子運算的基礎 AWS！註冊 Amazon Braket 數位學習計劃，在完成一系列的學習課程和數位評估後獲得自己的數位徽章。

您可以透過AWS 主控台在帳戶中啟用 Amazon Braket。

在本節中：

- 先決條件
- 啟用 Amazon Braket 的步驟

先決條件

若要啟用和執行 Amazon Braket，您必須擁有具有啟動 Amazon Braket 動作許可的使用者或角色。這些許可包含在 AmazonBraketFullAccess IAM 政策 (arn：aws：iam：：aws：policy/AmazonBraketFullAccess) 中。

> **Note**
>
> 如果您是管理員：
> 若要授予其他使用者 Amazon Braket 的存取權，請連接 AmazonBraketFullAccess 政策或連接您建立的自訂政策，以授予使用者許可。若要進一步了解使用 Amazon Braket 所需的許可，請參閱管理對 Amazon Braket 的存取。

啟用 Amazon Braket 的步驟

1. 使用您的 登入 Amazon Braket 主控台 AWS 帳戶。
2. 開啟 Amazon Braket 主控台。
3. 從 Braket 登陸頁面，按一下開始使用以移至服務儀表板頁面。服務儀表板頂端的提醒將引導您完成以下三個步驟：
   a. 建立服務連結角色 (SLR)
   b. 啟用對第三方量子電腦的存取
   c. 建立新的 Jupyter 筆記本執行個體

若要使用第三方量子裝置，您需要同意與您自己 AWS和這些裝置之間資料傳輸相關的特定條件。本協議的條款與條件提供於 Amazon Braket 主控台之許可和設定頁面的一般索引標籤。

> **Note**
>
> 不涉及任何第三方的 Quantum 裝置，例如 Braket 本機模擬器或隨需模擬器，可以在未同意啟用第三方裝置協議的情況下使用。
> 如果您存取第三方硬體，接受這些條款以啟用第三方裝置的使用，則每個帳戶只需要完成一次。

建立 Amazon Braket 筆記本執行個體

> **Tip**
>
> 了解量子運算的基礎 AWS！註冊 Amazon Braket 數位學習計劃，在完成一系列的學習課程和數位評估後獲得自己的數位徽章。

Amazon Braket 提供全受管 Jupyter 筆記本，讓您開始使用。Amazon Braket 筆記本執行個體是以 Amazon SageMaker AI 筆記本執行個體為基礎。下列步驟概述如何為新客戶和現有客戶建立新的筆記本執行個體。

Amazon Braket 新客戶：

1. 開啟 Amazon Braket 主控台，並導覽至左側窗格中的儀表板頁面。
2. 按一下儀表板頁面中心的歡迎使用 Amazon Braket 模態上的入門。提供筆記本名稱以建立預設的 Jupyter 筆記本。
   a. 建立筆記本可能需要幾分鐘的時間。
   b. 您的筆記本將列在筆記本頁面上，狀態為待定。
   c. 當您的筆記本執行個體可供使用時，狀態會變更為 InService。
   d. 重新整理頁面以顯示筆記本的更新狀態。

現有的 Amazon Braket 客戶：

1. 開啟 Amazon Braket 主控台，然後選取左側窗格中的筆記本。
2. 選取建立筆記本執行個體。
   a. 如果您有零個筆記本，請選取標準設定來建立預設的 Jupyter 筆記本。
3. 輸入筆記本執行個體名稱，僅使用英數字元和連字號字元，然後選取您偏好的視覺化模式。
4. 啟用或停用筆記本的筆記本閒置管理員。
   a. 如果啟用，請在重設筆記本之前選取所需的閒置持續時間。重設筆記本時，運算費用會停止產生，但儲存費用會繼續。
   b. 若要檢查筆記本執行個體剩餘的閒置時間，請導覽至命令列，選取 Braket 索引標籤，然後選取 Inactivity Manager 索引標籤。

> **Note**
>
> 若要儲存工作，請將 SageMaker AI 筆記本執行個體與 Git 儲存庫整合。或者，將您的工作移出 `/Braket Algorithms`和 `/Braket Examples` 資料夾，使其不會被筆記本執行個體重新啟動所覆寫。

5. （選用）透過進階設定，您可以建立具有存取許可、其他組態和網路存取設定的筆記本：
   a. 在筆記本組態中，選擇您的執行個體類型。

i. 預設會選擇符合成本效益的標準執行個體類型 ml.t3.medium。若要進一步了解執行個體定價，請參閱 Amazon SageMaker AI 定價。

b. 若要將公有 Github 儲存庫與您的筆記本執行個體建立關聯，請按一下 Git 儲存庫下拉式清單，然後從儲存庫下拉式功能表中選取從 URL 複製公有 Git 儲存庫。在 Git 儲存庫 URL 文字列中輸入儲存庫的 URL。

c. 在存取許可中，設定任何選用的 IAM 角色、根存取和加密金鑰。

d. 在網路存取中，設定Jupyter Notebook執行個體的自訂網路和存取設定。

6. 檢閱您的設定，並設定任何標籤來識別您的筆記本執行個體。按一下啟動。

> **Note**
>
> 在 Amazon Braket 和 Amazon SageMaker AI 主控台中檢視和管理 Amazon Braket 筆記本執行個體。其他 Amazon Braket 筆記本設定可透過 SageMaker 主控台取得。

如果您在 Amazon Braket SDK 的 Amazon Braket 主控台中工作 AWS ，且外掛程式會預先載入您建立的筆記本中。若要在您自己的機器上執行 ，請在執行 命令`pip install amazon-braket-sdk`或執行 PennyLane 外掛程式`pip install amazon-braket-pennylane-plugin`的 命令時安裝 SDK 和外掛程式。

使用 建立 Braket 筆記本執行個體 AWS CloudFormation

> **Tip**
>
> 了解量子運算的基礎 AWS！註冊 Amazon Braket 數位學習計劃，在完成一系列的學習課程和數位評估後獲得自己的數位徽章。

您可以使用 AWS CloudFormation 來管理 Amazon Braket 筆記本執行個體。Braket 筆記本執行個體是以 Amazon SageMaker AI 為基礎。使用 CloudFormation，您可以使用描述預期組態的範本檔案來佈建筆記本執行個體。範本檔案是以 JSON 或 YAML 格式撰寫。您可以以有序且可重複的方式建立、更新和刪除執行個體。當您在 中管理多個 Braket 筆記本執行個體時，您可能會發現這一點很有用 AWS 帳戶。

為 Braket 筆記本建立 CloudFormation 範本後，您可以使用 AWS CloudFormation 來部署資源。如需詳細資訊，請參閱《 AWS CloudFormation 使用者指南》中的在 AWS CloudFormation 主控台上建立堆疊。

若要使用 CloudFormation 建立 Braket 筆記本執行個體，請執行這三個步驟：

1. 建立 SageMaker AI 生命週期組態指令碼。
2. 建立由 SageMaker AI 擔任的 AWS Identity and Access Management (IAM) 角色。
3. 使用 字首建立 SageMaker AI 筆記本執行個體 **amazon-braket-**

您可以為您建立的所有 Braket 筆記本重複使用生命週期組態。您也可以為您指派相同執行許可的 Braket 筆記本重複使用 IAM 角色。

在本節中：

步驟 1：建立 SageMaker AI 生命週期組態指令碼

使用下列範本建立 SageMaker AI 生命週期組態指令碼。指令碼會自訂 Braket 的 SageMaker AI 筆記本執行個體。如需生命週期 CloudFormation 資源的組態選項，請參閱《 AWS CloudFormation 使用者指南》中的 AWS::SageMaker::NotebookInstanceLifecycleConfig。

```
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
```

步驟 2：建立 Amazon SageMaker AI 擔任的 IAM 角色

當您使用 Braket 筆記本執行個體時，SageMaker AI 會代表您執行操作。例如，假設您使用支援的裝置上的電路執行 Braket 筆記本。在筆記本執行個體中，SageMaker AI 會為您在 Braket 上執行 操作。筆記本執行角色會定義允許 SageMaker AI 代表您執行的確切操作。如需詳細資訊，請參閱《Amazon SageMaker AI 開發人員指南》中的 SageMaker AI 角色。 Amazon SageMaker

使用下列範例來建立具有所需許可的 Braket 筆記本執行角色。您可以根據您的需求修改政策。

> **Note**
>
> 請確定角色具有在字首為 的 Amazon S3 儲存貯體上 `s3:ListBucket`和 `s3:GetObject`操作的許可`braketnotebookcdk-`"。生命週期組態指令碼需要這些許可才能複製 Braket 筆記本安裝指令碼。

```
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: "*"
```

步驟 3：建立字首為 的 SageMaker AI 筆記本執行個體 **amazon-braket-**

使用 SageMaker AI 生命週期指令碼和步驟 1 和步驟 2 中建立的 IAM 角色來建立 SageMaker AI 筆記本執行個體。筆記本執行個體是針對 Braket 自訂的，可以透過 Amazon Braket 主控台存取。如需此 CloudFormation 資源組態選項的詳細資訊，請參閱AWS CloudFormation 《 使用者指南》中的 AWS::SageMaker::NotebookInstance。

```
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
```

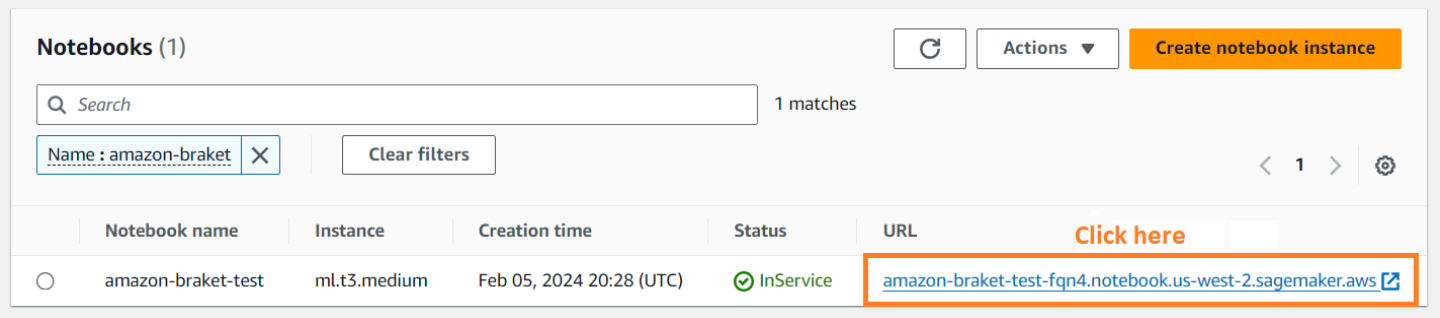
使用 Amazon Braket 建置您的量子任務

Braket 提供全受管的 Jupyter 筆記本環境，可讓您直接開始使用。Braket 筆記本預先安裝了範例演算法、資源和開發人員工具，包括 Amazon Braket SDK。使用 Amazon Braket SDK，您可以建置量子演算法，然後透過變更單一程式碼行，在不同的量子電腦和模擬器上測試和執行這些演算法。

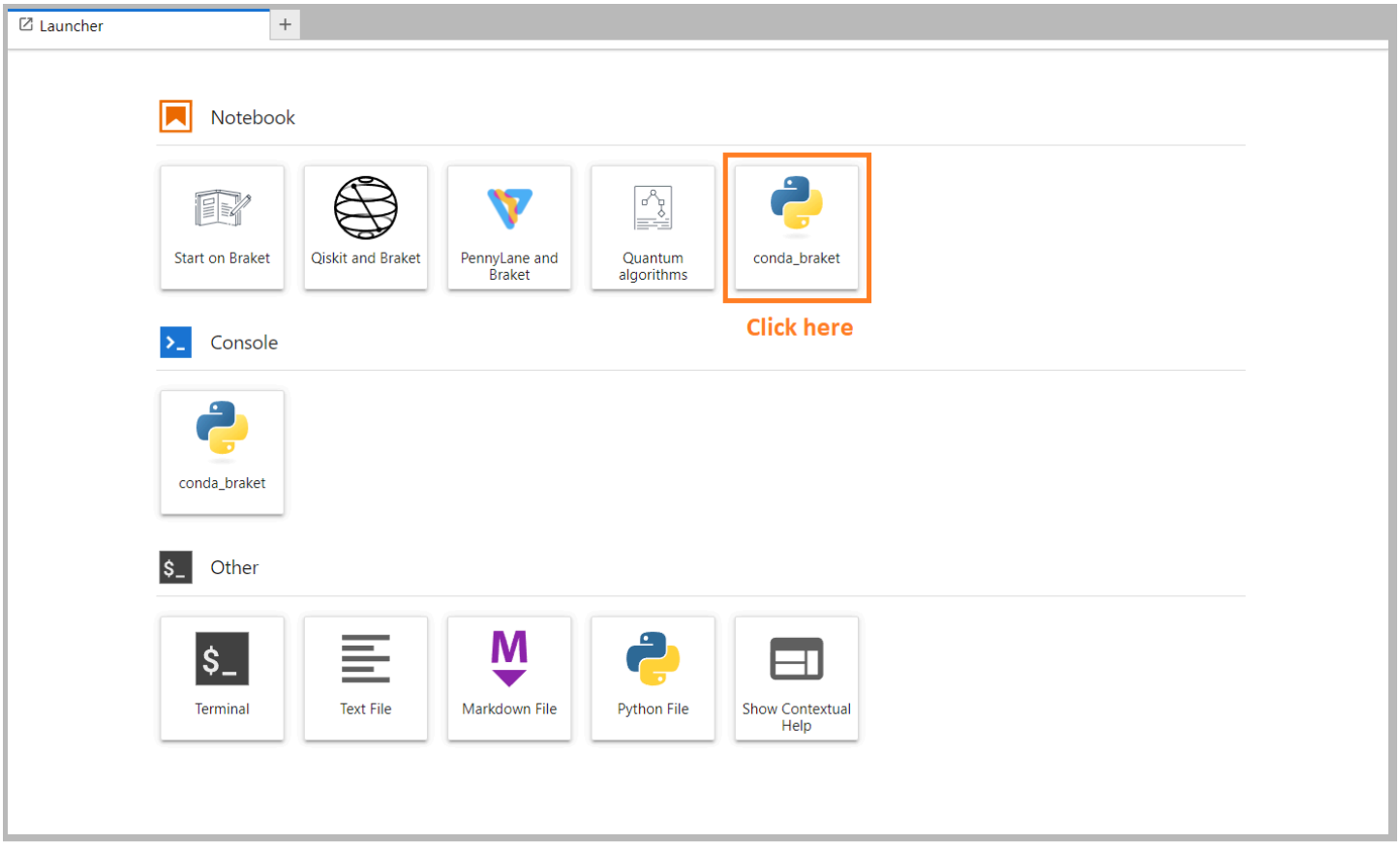
在本節中：

建置您的第一個電路

啟動筆記本執行個體之後，請選擇您剛建立的筆記本，以標準 Jupyter 介面開啟執行個體。



Amazon Braket 筆記本執行個體會預先安裝 Amazon Braket SDK 及其所有相依性。從使用`conda_braket`核心建立新的筆記本開始。



您可以從簡單的「您好，世界！」開始 範例。首先，建構一個準備 Bell 狀態的電路，然後在不同的裝置上執行該電路以取得結果。

首先匯入 Start，匯入 Amazon Braket SDK 模組並定義 simpleBRAKETlong； SDK 模組，以及定義基本的 Bell 狀態電路。

```
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)
```

您可以使用此命令視覺化電路：

```
print(bell)
```

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

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

在本機模擬器上執行您的電路

接著，選擇要在其中執行電路的量子裝置。Amazon Braket SDK 隨附本機模擬器，用於快速原型設計和測試。我們建議將本機模擬器用於較小的電路，最多可達 25 個 qubits（取決於您的本機硬體）。

若要執行個體化本機模擬器：

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

並執行電路：

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

您應該會看到類似這樣的結果：

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

您準備的特定貝爾狀態是相等疊加的 |00# 和 |11#，以及大約相等（最高shot雜訊）分佈的 00 和 11 作為測量結果，如預期。

在隨需模擬器上執行您的電路

Amazon Braket 也提供隨需、高效能模擬器 的存取權，SV1以執行更大的電路。 SV1 是一種隨需狀態向量模擬器，允許模擬最多 34 個 的量子電路qubits。您可以在SV1支援的裝置區段和 AWS 主控台中找到有關 的詳細資訊。在 SV1（和 TN1或任何 QPU) 上執行量子任務時，量子任務的結果會存放在您帳戶中的 S3 儲存貯體中。如果您未指定儲存貯體，則 Braket SDK `amazon-braket-{region}-{accountID}`會為您建立預設儲存貯體。若要進一步了解，請參閱管理對 Amazon Braket 的存取。

> **Note**
>
> 填寫您實際的現有儲存貯體名稱，其中下列範例顯示amazon-braket-s3-demo-bucket為您的儲存貯體名稱。Amazon Braket 的儲存貯體名稱一律以 開頭，amazon-braket-後面接著您新增的其他識別字元。如果您需要如何設定 S3 儲存貯體的資訊，請參閱 Amazon S3 入門。

```
# 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)
```

若要在 上執行電路SV1，您必須提供先前在 .run() 呼叫中選取做為位置引數的 S3 儲存貯體位置。

```
# 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 主控台提供有關量子任務的進一步資訊。導覽至主控台中的 Quantum 任務索引標籤，您的量子任務應該位於清單頂端。或者，您可以使用唯一的量子任務 ID 或其他條件來搜尋量子任務。

> **Note**
>
> 90 天後，Amazon Braket 會自動移除與量子任務相關聯的所有量子任務 IDs 和其他中繼資料。如需詳細資訊，請參閱資料保留。

在 QPU 上執行

使用 Amazon Braket，您只需變更一行程式碼，即可在實體量子電腦上執行先前的量子電路範例。Amazon Braket 可讓您從 IonQ、QuEra、 IQM和 存取QPU裝置Rigetti。您可以在支援的裝置區段，以及裝置索引標籤下的 AWS 主控台中找到不同裝置和可用性時段的相關資訊。下列範例顯示如何執行個體化IQM裝置。

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

或者，選擇具有此程式碼IonQ的裝置：

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

選取裝置之後，在執行工作負載之前，您可以使用下列程式碼查詢裝置佇列深度，以判斷量子任務或混合任務的數量。此外，客戶可以在 的裝置頁面上檢視裝置特定的佇列深度Amazon Braket Management Console。

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

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

當您執行任務時，Amazon Braket SDK 會輪詢結果 （預設逾時為 5 天）。您可以修改 `.run()`命令中的 `poll_timeout_seconds` 參數來變更此預設值，如以下範例所示。請記住，如果您的輪詢逾時太短，則可能無法在輪詢時間內傳回結果，例如當 QPU 無法使用且傳回本機逾時錯誤時。您可以透過呼叫 `task.result()`函數來重新啟動輪詢。

```
# 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)
```

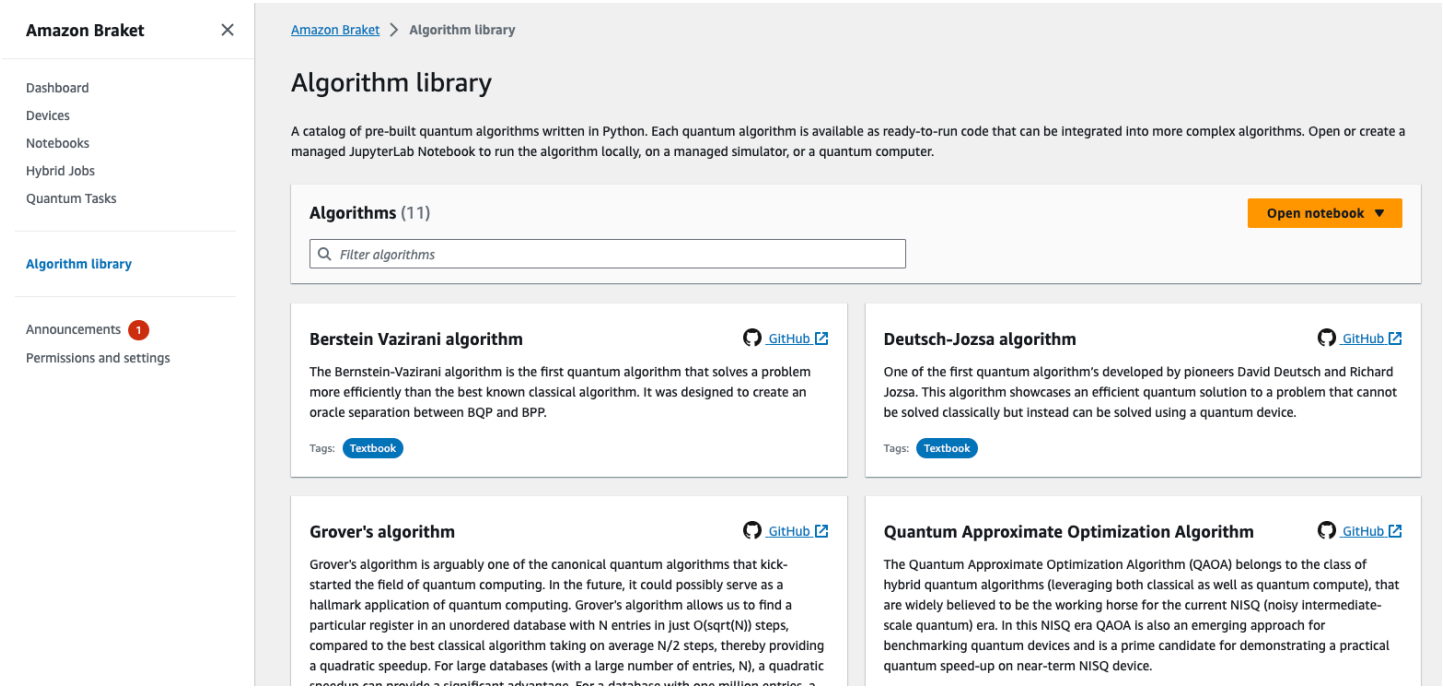
此外，在提交您的量子任務或混合任務之後，您可以呼叫 `queue_position()`函數來檢查您的佇列位置。

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

```
# '2'
```

建置您的第一個量子演算法

Amazon Braket 演算法程式庫是以 Python 撰寫的預先建置量子演算法目錄。依原樣執行這些演算法，或使用它們做為建置更複雜演算法的起點。您可以從 Braket 主控台存取演算法程式庫。如需詳細資訊，請參閱 Braket Github 演算法程式庫。



Braket 主控台提供演算法程式庫中每個可用演算法的說明。選擇 GitHub 連結以查看每個演算法的詳細資訊，或選擇開啟筆記本以開啟或建立包含所有可用演算法的筆記本。如果您選擇筆記本選項，您可以在筆記本的根資料夾中找到 Braket 演算法程式庫。

在 SDK 中建構電路

本節提供定義電路、檢視可用閘道、延伸電路和檢視每個裝置支援之閘道的範例。它還包含有關如何手動配置 的說明qubits、指示編譯器完全按照定義執行您的電路，以及使用雜訊模擬器建置雜訊電路。

您也可以使用 Braket 中的脈衝層級，處理具有特定 QPUs 的各種閘道。如需詳細資訊，請參閱 Amazon Braket 上的脈衝控制。

在本節中：

- 閘道和電路
- 程式集

- 部分測量
- 手動qubit配置
- 逐字編譯
- 雜訊模擬

閘道和電路

Quantum 閘道和電路在 Amazon Braket Python SDK 的 `braket.circuits`類別中定義。從 SDK，您可以透過呼叫 來執行個體化新的電路物件`Circuit()`。

範例：定義電路

此範例一開始會定義四個 qubits (標記 q0、q2、 q1和 q3) 的範例電路，其中包含標準、單一 1/4 位元 Hadamard 閘道和 2/4 位元 CNOT 閘道。您可以透過呼叫 `print`函數來視覺化此電路，如下列範例所示。

```
# 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     #
```

範例：定義參數化電路

在此範例中，我們定義一個具有閘道的電路，這些閘道依賴於可用參數。我們可以指定這些參數的值來建立新的電路，或在提交電路時，在特定裝置上做為量子任務執行。

```
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)
```

您可以從參數化電路建立新的非參數化電路，方法是提供單一 `float`（所有可用參數將採用的值）或關鍵字引數，指定每個參數的值給電路，如下所示。

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

請注意， `my_circuit` 未修改，因此您可以使用它來執行個體化許多具有固定參數值的新電路。

範例：修改電路中的閘道

下列範例定義具有使用控制和電源修改器之閘道的電路。您可以使用這些修改來建立新的閘道，例如控制Ry閘道。

```
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)
```

閘道修飾詞僅支援本機模擬器。

範例：查看所有可用的閘道

下列範例顯示如何在 Amazon Braket 中查看所有可用的閘道。

```
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)
```

此程式碼的輸出會列出所有閘道。

```
['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']
```

任何這些閘道都可以透過呼叫該類型電路的 方法附加到電路。例如，呼叫 `circ.h(0)`，將 Hadamard 閘道新增至第一個 qubit。

> **Note**
>
> 閘道會附加到適當位置，以下範例會將上一個範例中列出的所有閘道新增至相同的電路。

```
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)
```

除了預先定義的閘道集之外，您也可以將自我定義的單一閘道套用至電路。這些可以是單一 qubit 閘道（如下列原始碼所示）或套用到 `targets` 參數qubits所定義 的多 qubit 閘道。

```
import numpy as np

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

範例：擴展現有電路

您可以新增指示來擴展現有的電路。`Instruction` 是量子指令，描述在量子裝置上執行的量子任務。`Instruction`運算子`Gate`僅包含 類型的物件。

```
# 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})
```

範例：檢視每個裝置支援的閘道

模擬器支援 Braket SDK 中的所有閘道，但 QPU 裝置支援較小的子集。您可以在裝置屬性中找到裝置支援的閘道。以下顯示 IonQ 裝置的範例：

```
# 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']
```

支援的閘道可能需要編譯為原生閘道，才能在量子硬體上執行。當您提交電路時， Amazon Braket 會自動執行此編譯。

範例：以程式設計方式擷取裝置支援的原生閘道逼真度

您可以在 Braket 主控台的裝置頁面上檢視逼真度資訊。有時，以程式設計方式存取相同的資訊會很有幫助。下列程式碼示範如何在 QPU 的兩個qubit閘道之間擷取兩個閘道逼真度。

```
# 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}")
```

程式集

程式集可在單一量子任務中有效率地執行多個量子電路。在該任務中，您最多可以提交 100 個量子電路，或具有最多 100 個不同參數集的單一參數電路。此操作可將後續電路執行之間的時間降至最低，並減少量子任務處理額外負荷。目前，Amazon Braket Local Simulator和 IQM和 Rigetti 裝置支援程式集。

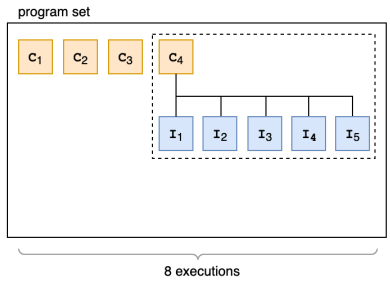
定義 ProgramSet

下列第一個程式碼範例示範如何使用參數化電路和不含參數的電路`ProgramSet`來建置 。

```
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,
])
```

此程式集包含四個唯一的程式：circ1、circ3、 circ2和 circuit_binding。此circuit_binding程式使用五個不同的參數繫結執行，建立五個可執行檔。其他三個無參數程式則各自建立一個可執行檔。這會產生總共八個可執行檔，如下圖所示。



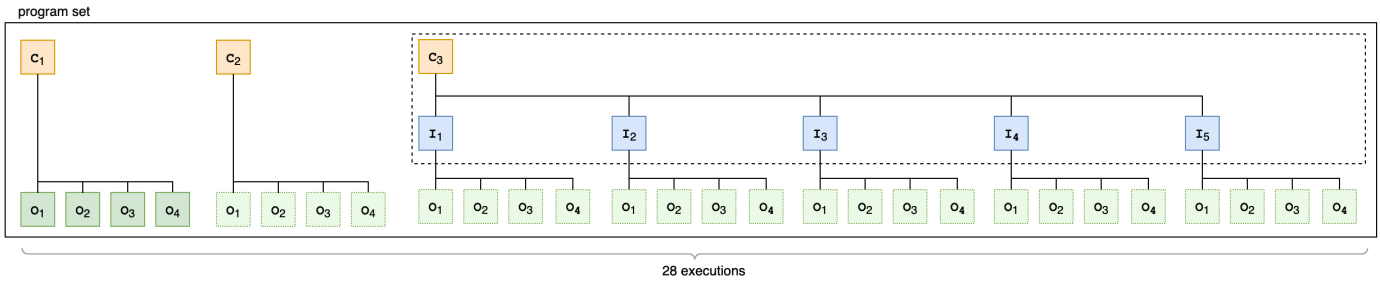

```
ProgramSet(
    [c1,
     c2,
     c3,
     CircuitBinding(c4, [i1, i2, i3, i4, i5])]
)
```

下列第二個程式碼範例示範如何使用 product()方法，將同一組可觀測值連接到程式集的每個可執行檔。

```
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
)
```

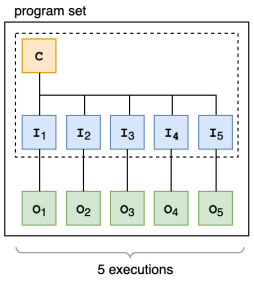
對於無參數程式，會針對每個電路測量每個可觀測項目。對於參數程式，會為每個輸入集測量每個可觀測值，如下圖所示。

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



下列第三個程式碼範例示範如何使用 zip()方法，將個別可觀測項目與 中的特定參數集配對ProgramSet。

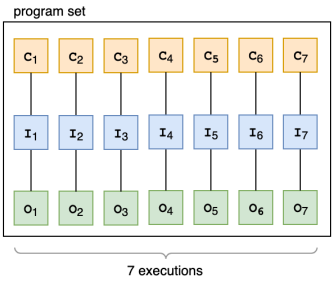
```
program_set_3 = ProgramSet.zip(
    circuits=circuit_binding,
    observables=observables + [Y(0) @ Y(1)]
)
```




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

您可以直接壓縮具有電路和輸入集清單的可觀測項目清單CircuitBinding()，而不是 。

```
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]
)
```

如需程式集的詳細資訊和範例，請參閱 amazon-braket-examples Github 中的程式集筆記本。

檢查並在裝置上執行程式集

程式集的可執行計數等於其唯一參數繫結電路的數量。使用以下程式碼範例計算電路可執行檔和鏡頭的總數。

```
# 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**
>
> 使用程式集，您可以根據程式集中所有電路的拍攝總數，支付每個任務的單一費用和每個拍攝費用。

若要執行程式集，請使用下列程式碼範例。

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

使用Rigetti裝置時，您的程式集可能會在任務部分完成且部分排入佇列時保持 RUNNING 狀態。如需更快的結果，請考慮將您的程式集提交為混合任務。

分析結果

執行下列程式碼來分析和測量 中可執行檔的結果ProgramSet。

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

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

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

部分測量

使用部分測量來測量個別的 qubit 或一部分的 qubit，而不是測量量子電路中的所有 qubit。

> **Note**
>
> 中電路測量和前饋操作等其他功能可作為實驗功能，請參閱存取 IQM 裝置上的動態電路。

範例：測量 qubit 的子集

下列程式碼範例透過在 Bell 狀態電路中僅測量 qubit 0 來示範部分測量。

```
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)
```

手動qubit配置

當您從 在量子電腦上執行量子電路時Rigetti，您可以選擇使用手動qubit配置來控制qubits用於演算法的 。Amazon Braket 主控台和 Amazon Braket SDK 可協助您檢查所選量子處理單元 (QPU) 裝置的最新校正資料，以便qubits為實驗選擇最佳校正資料。

手動qubit配置可讓您以更高的準確性執行電路，並調查個別qubit屬性。研究人員和進階使用者會根據最新的裝置校正資料來最佳化其電路設計，並可取得更準確的結果。

下列範例示範如何qubits明確配置 。

```
# 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)
```

如需詳細資訊，請參閱 GitHub 上的 Amazon Braket 範例，或更具體地參閱此筆記本：在 QPU 裝置上配置 Qubit。

逐字編譯

當您在以閘道為基礎的量子電腦上執行量子電路時，您可以指示編譯器完全按照定義執行電路，而不會進行任何修改。使用逐字編譯，您可以指定精確地保留整個電路，或僅保留其特定部分 (Rigetti僅 支援)。開發硬體基準測試或錯誤緩解通訊協定的演算法時，您可以選擇精確指定硬體上執行的閘道和電路配置。逐字編譯可讓您關閉特定最佳化步驟，直接控制編譯程序，進而確保電路完全按照設計執行。

Rigetti、 和 IQM 裝置支援逐字編譯IonQ，且需要使用原生閘道。使用逐字編譯時，建議您檢查裝置的拓撲，以確保在連接的 上呼叫閘道，qubits以及電路使用硬體上支援的原生閘道。下列範例顯示如何以程式設計方式存取裝置支援的原生閘道清單。

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

對於 Rigetti，必須透過設定 `disableQubitRewiring=True`與逐字編譯搭配使用來關閉qubit重新配線。如果在編譯中使用逐字方塊時設定 `disableQubitRewiring=False` ，則量子電路會驗證失敗且不會執行。

如果為電路啟用逐字編譯，並在不支援它的 QPU 上執行，則會產生錯誤，指出不支援的操作導致任務失敗。隨著更多量子硬體原生支援編譯器函數，此功能將擴展為包含這些裝置。使用下列程式碼查詢時，支援逐字編譯的裝置會將其納入支援的操作。

```
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
```

使用逐字編譯沒有相關的額外費用。根據 Amazon Braket 定價頁面上指定的目前費率，在 Braket QPU 裝置、筆記本執行個體和隨需模擬器上執行的量子任務會繼續向您收取費用。如需詳細資訊，請參閱 Verbatim 編譯範例筆記本。

> **Note**
>
> 如果您使用 OpenQASM 為IonQ裝置寫入電路，而且您希望將電路直接映射至實體 qubit，則需要使用 ，`#pragma braket verbatim`因為 OpenQASM 完全忽略`disableQubitRewiring`標記。

雜訊模擬

若要執行個體化本機雜訊模擬器，您可以變更後端，如下所示。

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

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

您可以透過兩種方式建置雜訊電路：

1. 從下而上建置雜訊電路。

2. 採用現有的無雜訊電路，並注入雜訊。

下列範例顯示使用具有去極化雜訊的基本電路和自訂 Kraus 頻道的方法。

```
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)
```

執行電路的使用者體驗與之前相同，如下列兩個範例所示。

範例 1

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

或

範例 2

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

如需更多範例，請參閱 Braket 簡介雜訊模擬器範例

檢查電路

Amazon Braket 中的 Quantum 電路具有稱為 的虛擬時間概念Moments。每個 qubit 都可以體驗每個的單一閘道Moment。Moments 的目的是讓電路及其閘道更容易解決並提供時間結構。

> **Note**
>
> 時刻通常不會對應到在 QPU 上執行閘道的即時時間。

電路的深度取決於該電路中的時刻總數。您可以檢視呼叫 方法的電路深度circuit.depth，如下列範例所示。

```
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
```

上述電路的總電路深度為 3 （顯示為時刻 0、 1和 2)。您可以檢查每個時刻的閘道操作。

Moments 函數做為索引鍵/值對的字典。

- 金鑰為 MomentsKey()，其中包含虛擬時間和qubit資訊。
- 值會以 的類型指派Instructions()。

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

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

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

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

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

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

您也可以透過 將閘道新增至電路Moments。

```
from braket.circuits import Instruction, Gate

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

```
                Instruction(Gate.H(), 1)
                ]

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

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

結果類型清單

使用 測量電路時，Amazon Braket 可以傳回不同類型的結果`ResultType`。電路可以傳回以下類型的結果。

- `AdjointGradient` 傳回所提供可觀測之預期值的漸層 （向量衍生性）。此可觀測值正在與使用並行差異化方法的指定參數相關的所提供目標上運作。您只能在 shots=0 時使用此方法。
- `Amplitude` 傳回輸出波函數中指定量子狀態的振幅。它僅適用於 SV1和本機模擬器。
- `Expectation` 傳回指定可觀測值的預期值，可使用本章稍後介紹的 `Observable`類別來指定。必須指定qubits用於測量可觀測的目標，且指定目標的數量必須等於可觀測行為qubits的 數量。如果未指定目標，則可觀測項目只能在 1 上運作，qubit並且會qubits平行套用至所有 。
- `Probability` 傳回測量運算基礎狀態的機率。如果未指定目標， 會`Probability`傳回測量所有基礎狀態的機率。如果指定了目標，qubits則只會傳回指定之基礎向量的邊際機率。受管模擬器和 QPUs 限制為最多 15 個 qubit，而本機模擬器限制為系統的記憶體大小。
- `Reduced density matrix` qubits 從 系統傳回指定目標之子系統的密度矩陣qubits。若要限制此結果類型的大小，Raket 會將目標的數量限制qubits為最多 8 個。
- `StateVector` 會傳回完整狀態向量。它可在本機模擬器上使用。
- `Sample` 會傳回指定目標qubit集和可觀察的測量計數。如果未指定目標，則可觀測項目只能在 1 上運作，qubit並qubits平行套用至所有 。如果指定了目標，則指定目標的數量必須等於可觀察行為qubits的 數量。

- `Variance` 傳回指定目標qubit集的變異數 (`mean([x-mean(x)]`^{2})，並可做為請求的結果類型觀察。如果未指定目標，則可觀測項目只能在 1 上運作，qubit並qubits平行套用至所有 。否則，指定的目標數量必須等於可觀測項目可套用qubits的 數量。

不同裝置支援的結果類型：

	本機 sim	SV1	DM1	TN1	Rigetti	IonQ	IQM
聯合漸層	N	Y	N	N	N	N	N
Amplitude	Y	Y	N	N	N	N	N
期望	Y	Y	Y	Y	Y	Y	Y
Probability (可能性)	Y	Y	Y	N	Y	Y	Y
密度降低矩陣	Y	N	Y	N	N	N	N
狀態向量	Y	N	N	N	N	N	N
樣本	Y	Y	Y	Y	Y	Y	Y
變異數	Y	Y	Y	Y	Y	Y	Y

您可以檢查裝置屬性來檢查支援的結果類型，如下列範例所示。

```
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
```

若要呼叫 `ResultType`，請將它附加到電路，如下列範例所示。

```
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**
>
> 不同的量子裝置會以各種格式提供結果。例如，Rigetti裝置會傳回測量，而IonQ裝置則提供機率。Amazon Braket SDK 為所有結果提供測量屬性。不過，對於傳回機率的裝置，這些測量是根據機率計算的，因為每個鏡頭的測量不可用。若要判斷結果是否已在計算後進行，請檢查結果物件`measurements_copied_from_device`上的 。此操作在 Amazon Braket SDK GitHub 儲存庫的 gate_model_quantum_task_result.py 檔案中詳細說明。

可觀測項目

Amazon Braket 的`Observable`類別可讓您測量特定的可觀測值。

您只能將一個可觀測的唯一非身分套用至每個 qubit。如果您指定兩個或多個不同的非身分可觀測到相同的 ，則會發生錯誤qubit。為此，張量產品的每個因素都算作個別可觀測。這表示您可以在相同的 上擁有多個張量產品qubit，只要作用於 的因素qubit保持不變。

可觀測項目可以擴展並新增其他可觀測項目（無論是否擴展）。這會建立Sum可用於AdjointGradient結果類型的 。

Observable 類別包含下列可觀測項目。

```
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)))
```

參數

電路可以包含免費參數。這些免費參數只需要建構一次即可執行多次，而且可用於運算漸層。

每個可用參數都使用字串編碼名稱，用於：

- 設定參數值
- 識別要使用的參數

```
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)
```

聯結漸層

SV1 裝置會計算可觀測預期值的相鄰梯度，包括多期間 Hamiltonian。若要區分參數，請指定其名稱（以字串格式）或直接參考。

```
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])
```

將固定參數值做為引數傳遞至參數化電路，將會移除可用參數。使用 執行此電路`AdjointGradient`會產生錯誤，因為可用參數不再存在。下列程式碼範例示範正確和不正確的使用：

```
# 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})
```

取得專家建議

直接在 Braket 管理主控台中與量子運算專家連線，以取得工作負載的其他指引。

若要透過 Braket Direct 探索專家建議選項，請開啟 Braket 主控台，選擇左側窗格中的 Braket Direct，然後導覽至專家建議區段。下列專家建議選項可供使用：

- 剎車上班時間：剎車上班時間是 1：1 工作階段，先到先得，每月舉行。每個可用的辦公時間時段為 30 分鐘且免費。與 Braket 專家交談可協助您更快地從構想到執行，方法為探索use-case-to-device擬合、找出最適合演算法使用 Braket 的選項，以及取得如何使用特定 Braket 功能的建議，例如 Amazon Braket Hybrid Jobs、Raket Pulse 或類比 Hamiltonian Simulation。
  - 若要註冊 Braket 上班時間，請選取註冊並填寫聯絡資訊、工作負載詳細資訊，以及您想要的討論主題。
  - 您將透過電子郵件收到下一個可用位置的行事曆邀請。

  > **Note**
  >
  > 對於緊急問題或快速故障診斷問題，我們建議您聯絡 AWS Support。對於非緊急問題，您也可以使用 AWS re：Post 論壇或 Quantum Computing Stack Exchange，您可以在其中瀏覽先前回答的問題並詢問新的問題。

- Quantum 硬體供應商產品：IonQ、 QuEra和 Rigetti各透過 提供專業服務產品 AWS Marketplace。
  - 若要探索其方案，請選取連線並瀏覽其清單。
  - 若要進一步了解 上的專業服務方案 AWS Marketplace，請參閱 專業服務產品。
- Amazon Quantum Solutions Lab (QSL)：QSL 是一個協作研究和專業服務團隊，配備量子運算專家，可協助您有效地探索量子運算並評估此技術目前的效能。
  - 若要聯絡 QSL，請選取連線，然後填寫聯絡資訊和使用案例詳細資訊。
  - QSL 團隊將透過電子郵件與您聯絡，並提供後續步驟。

Amazon Braket 混合任務入門

本節提供有關如何在 Amazon Braket 中設定混合任務的元件和指示的資訊。

您可以使用下列方式存取 Braket 中的混合式任務：

- Amazon Braket Python SDK。
- Amazon Braket 主控台。
- Amazon Braket API。

在本節中：

什麼是混合任務？

Amazon Braket Hybrid Jobs 可讓您執行混合量子傳統演算法，同時需要傳統 AWS 資源和量子處理單元 (QPUs)。混合任務旨在啟動請求的傳統資源、執行演算法，並在完成後釋放執行個體，因此您只需支付使用的費用。

混合任務非常適合長時間執行的反覆運算演算法，這些演算法涉及使用傳統運算資源和量子運算資源。透過混合任務，在提交演算法以執行之後，Raket 會在可擴展的容器化環境中執行您的演算法。演算法完成後，您就可以擷取結果。

此外，從混合任務建立的量子任務受益於目標 QPU 裝置的較高優先順序佇列。此優先順序可確保在佇列中等待的其他任務之前處理和執行您的量子運算。這對於疊代混合演算法特別有利，其中一個量子任務的結果取決於先前量子任務的結果。此類演算法的範例包括量子近似最佳化演算法 (QAOA)、變化量子 eigensolver 或量子機器學習。您也可以近乎即時地監控演算法進度，讓您追蹤成本、預算或自訂指標，例如訓練損失或期望值。

何時使用 Amazon Braket 混合任務

Amazon Braket 混合任務可讓您執行混合量子傳統演算法，例如變量量子 Eigensolver (VQE) 和量子近似最佳化演算法 (QAOA)，這些演算法結合傳統運算資源與量子運算裝置，以最佳化現今量子系統的效能。Amazon Braket Hybrid Jobs 提供三個主要優點：

1. 效能：Amazon Braket 混合任務的效能優於從您自己的環境執行混合演算法。當您的任務執行時，它會優先存取選取的目標 QPU。任務中的任務會在裝置上排入佇列的其他任務之前執行。這會導致混合演算法的執行時間較短且更可預測。Amazon Braket Hybrid Jobs 也支援參數編譯。您可以使用免費參數提交電路，而 Braket 會編譯一次電路，而不需要重新編譯以對相同電路進行後續參數更新，進而加快執行時間。
2. 便利性：Amazon Braket Hybrid Jobs 可簡化設定和管理運算環境，並在混合式演算法執行時保持執行狀態。您只需提供演算法指令碼，然後選取要在其中執行的量子裝置（量子處理單元或模擬器）。Amazon Braket 會等待目標裝置變成可用、啟動傳統資源、在預先建置的容器環境中執行工作負載、將結果傳回 Amazon Simple Storage Service (Amazon S3)，並釋出運算資源。
3. 指標：Amazon Braket Hybrid Jobs 提供on-the-fly洞見，並近乎即時地將可自訂的演算法指標交付給 Amazon CloudWatch 和 Amazon Braket 主控台，讓您可以追蹤演算法的進度。

輸入、輸出、環境變數和協助程式函數

除了構成完整演算法指令碼的檔案之外，您的混合任務還可以有額外的輸入和輸出。當您的混合任務開始時，AmazonRaket 會將作為混合任務建立一部分而提供的輸入複製到執行演算法指令碼的容器中。當混合任務完成時，演算法期間定義的所有輸出都會複製到指定的 Amazon S3 位置。

> **Note**
>
> 演算法指標會即時回報，且不遵循此輸出程序。

Amazon Braket 也提供數個環境變數和協助程式函數，以簡化與容器輸入和輸出的互動。

本節說明 Amazon Braket Python SDK 所提供`AwsQuantumJob.create`函數的重要概念，及其與容器檔案結構的映射。

在本節中：

- 輸入
- 輸出

- 環境變數
- 協助程式函數

輸入

輸入資料：使用 input_data引數指定設定為字典的輸入資料檔案，即可將輸入資料提供給混合式演算法。使用者在 SDK 的 AwsQuantumJob.create函數中定義input_data引數。這會將輸入資料複製到環境變數 提供的位置的容器檔案系統"AMZN_BRAKET_INPUT_DIR"。如需如何在混合式演算法中使用輸入資料的幾個範例，請參閱 Amazon Braket Hybrid Jobs 中的 QAOA 和 Amazon Braket Hybrid Jobs Jupyter 筆記本中的 PennyLane 和 Quantum Machine Learning。 Amazon Braket

> **Note**
>
> 當輸入資料很大 (>1GB) 時，提交混合任務之前會有很長的等待時間。這是因為本機輸入資料會先上傳至 S3 儲存貯體，然後將 S3 路徑新增至混合式任務請求，最後將混合式任務請求提交至 Braket 服務。

超參數：如果您傳入 hyperparameters，它們可在環境變數 下使用"AMZN_BRAKET_HP_FILE"。

> **Note**
>
> 如需如何建立超參數和輸入資料，然後將此資訊傳遞至混合任務指令碼的詳細資訊，請參閱使用超參數一節和此 github 頁面。

檢查點：若要指定您想要在新混合任務中使用的job-arn檢查點，請使用 copy_checkpoints_from_job命令。此命令會透過檢查點資料複製到新混合任務checkpoint_configs3Uri的 ，使其可在任務執行AMZN_BRAKET_CHECKPOINT_DIR時環境變數指定的路徑中使用。預設值為 None，表示來自另一個混合任務的檢查點資料不會用於新的混合任務。

輸出

Quantum 任務：Quantum 任務結果存放在 S3 位置 s3://amazon-braket-<region>-<accountID>/jobs/<job-name>/tasks。

任務結果：您的演算法指令碼儲存至環境變數所提供目錄的所有內容"AMZN_BRAKET_JOB_RESULTS_DIR"都會複製到 中指定的 S3 位置output_data_config。如

果未指定值，則預設為 s3://amazon-braket-<region>-<accountID>/jobs/<job-name>/<timestamp>/data。我們提供 SDK 協助程式函數 **save_job_result** ，當您從演算法指令碼呼叫 時，您可以使用它以字典的形式方便地存放結果。

檢查點：如果您想要使用檢查點，您可以將它們儲存在環境變數 提供的目錄中"AMZN_BRAKET_CHECKPOINT_DIR"。您也可以save_job_checkpoint改用 SDK 協助程式函數。

演算法指標：您可以將演算法指標定義為演算法指令碼的一部分，該指令碼會在混合任務執行時即時傳送到 Amazon CloudWatch，並在 Amazon Braket 主控台中顯示。如需如何使用演算法指標的範例，請參閱使用 Amazon Braket 混合任務來執行 QAOA 演算法。

環境變數

Amazon Braket 提供數個環境變數，以簡化與容器輸入和輸出的互動。以下程式碼列出 Braket 使用的環境變數。

- AMZN_BRAKET_INPUT_DIR – 輸入資料目錄 opt/braket/input/data。
- AMZN_BRAKET_JOB_RESULTS_DIR – 要寫入任務結果的輸出目錄 opt/braket/model。
- AMZN_BRAKET_JOB_NAME – 任務的名稱。
- AMZN_BRAKET_CHECKPOINT_DIR – 檢查點目錄。
- AMZN_BRAKET_HP_FILE – 包含超參數的檔案。
- AMZN_BRAKET_DEVICE_ARN – 裝置 ARN (AWS 資源名稱）。
- AMZN_BRAKET_OUT_S3_BUCKET – 輸出 Amazon S3 儲存貯體，如CreateJob請求的 中所指定OutputDataConfig。
- AMZN_BRAKET_SCRIPT_ENTRY_POINT – CreateJob請求 中指定的進入點ScriptModeConfig。
- AMZN_BRAKET_SCRIPT_COMPRESSION_TYPE – CreateJob請求的 中指定的壓縮類型ScriptModeConfig。
- AMZN_BRAKET_SCRIPT_S3_URI – 使用者指令碼的 Amazon S3 位置，如CreateJob請求的 中所指定ScriptModeConfig。
- AMZN_BRAKET_TASK_RESULTS_S3_URI – 軟體開發套件預設會存放任務量子任務結果的 Amazon S3 位置。
- AMZN_BRAKET_JOB_RESULTS_S3_PATH – 存放任務結果的 Amazon S3 位置，如CreateJob請求的 中所指定OutputDataConfig。

- AMZN_BRAKET_JOB_TOKEN – 對於在任務容器中建立的量子任務，應該傳遞至 CreateQuantumTask的 jobToken 參數的字串。

協助程式函數

Amazon Braket 提供多種協助程式函數，可簡化與容器輸入和輸出的互動。這些協助程式函數會從用來執行混合任務的演算法指令碼中呼叫。下列範例示範如何使用它們。

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

定義演算法指令碼的環境

Amazon Braket 為您的演算法指令碼支援容器定義的三個環境：

- 基礎容器 （預設，如果未指定image_uri)
- 具有 Tensorflow 和 PennyLane 的容器
- 具有 PyTorch 和 PennyLane 的容器

下表提供有關容器及其包含的程式庫的詳細資訊。

Amazon Braket 容器

Type	PennyLane 搭配 TensorFlow	PennyLane 搭配 PyTorch	潘尼蘭
Base	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	292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-base-jobs:latest

Type	PennyLane 搭配 TensorFlow	PennyLane 搭配 PyTorch	潘尼蘭
繼承的程式庫	• awscli • numpy • pandas • scipy	• awscli • numpy • pandas • scipy	
其他程式庫	• amazon-braket-default-simulator • amazon-braket-pennylane-plugin • amazon-braket-schemas • amazon-braket-sdk • ipykernel • 角 • 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 • 角 • 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 • awscli • boto3 • ipykernel • matplotlib • networkx • numpy • Openbabel • pandas • PennyLane • protobuf • psi4 • rsa • scipy

您可以在 aws/amazon-braket-containers 中檢視和存取開放原始碼容器定義。選擇最符合您使用案例的容器。容器必須位於您調用混合任務 AWS 區域 的 中。您可以在建立混合任務時指定容器映像，方法是將下列三個引數之一新增至混合任務指令碼中的`create(…)` 呼叫。您可以將其他相依性安裝到

您在執行時間選擇的容器（啟動或執行時間的成本），因為 Amazon Braket 容器具有網際網路連線能力。下列範例適用於 us-west-2 區域。

- 基礎映像 image_uri="292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-base-jobs:1.0-cpu-py39-ubuntu22.04"
- Tensorflow 影像 image_uri="292282985366.dkr.ecr.us-east-1.amazonaws.com/amazon-braket-tensorflow-jobs:2.11.0-gpu-py39-cu112-ubuntu20.04"
- PyTorch image_uri="292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-pytorch-jobs:1.13.1-gpu-py39-cu117-ubuntu20.04"

`image-uris` 您也可以使用 Amazon Braket SDK 中的 `retrieve_image()`函數來擷取 。下列範例示範如何從 us-west-2 擷取它們 AWS 區域。

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

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

使用超參數

您可以在建立混合任務時定義演算法所需的超參數，例如學習率或步進大小。超參數值通常用於控制演算法的各個層面，並且通常可以進行調校以最佳化演算法的效能。若要在 Braket 混合任務中使用超參數，您需要將其名稱和值明確指定為字典。請注意，這些值必須是字串資料類型。您可以在搜尋最佳值集時指定要測試的超參數值。使用超參數的第一步是將超參數設定為字典並將其定義為字典，如下列程式碼所示：

```
# Defining the number of qubits used
n_qubits = 8
# Defining the number of layers used
n_layers = 10
# Defining the number of iterations used for your optimization algorithm
n_iterations = 10

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

然後，您將傳遞上述程式碼片段中定義的超參數，以便在您選擇的演算法中使用，如下所示：

```
import time
from braket.aws import AwsQuantumJob

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

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

> **Note**
>
> 若要進一步了解輸入資料，請參閱輸入區段。

然後，會使用下列程式碼將超參數載入混合任務指令碼：

```
import json
import os

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

> **Note**
>
> 如需如何將輸入資料和裝置等資訊傳遞至混合式任務指令碼的詳細資訊，請參閱此 github 頁面。

在 Amazon Braket Hybrid Jobs 教學課程中，QAOA 會針對如何使用超參數，以及 PennyLane 和 Quantum Machine Learning Amazon Braket 提供幾個非常實用的指南。

搭配 Amazon Braket 使用 PennyLane

混合演算法是同時包含傳統和量子指示的演算法。傳統說明是在傳統硬體 (EC2 執行個體或筆記型電腦） 上執行，而量子說明是在模擬器或量子電腦上執行。我們建議您使用混合任務功能執行混合演算法。如需詳細資訊，請參閱何時使用 Amazon Braket 任務。

Amazon Braket 可讓您在 Amazon Braket PennyLane 外掛程式或 Amazon Braket Python SDK 和範例筆記本儲存庫的協助下，設定和執行混合式量子演算法。Amazon Braket 範例筆記本以 SDK 為基礎，可讓您在沒有 PennyLane 外掛程式的情況下設定和執行特定混合式演算法。不過，我們建議使用 PennyLane，因為它提供更豐富的體驗。

關於混合量子演算法

混合量子演算法對現今的產業來說很重要，因為現代量子運算裝置通常會產生雜訊，因此會產生錯誤。新增至運算的每個量子閘道都會提高增加雜訊的機會；因此，長時間執行的演算法可能會因為雜訊而不堪重負，從而導致運算錯誤。

純量子演算法，例如 Shor 的 （量子階段估算範例） 或 Grover 的 (Grover 範例），需要數千或數百萬個操作。因此，它們對於現有的量子裝置可能不切實際，通常稱為雜訊中階量子 (NISQ) 裝置。

在混合量子演算法中，量子處理單元 (QPUs) 可做為傳統 CPUs 的共同處理器，特別是在傳統演算法中加速特定計算。在現今裝置的功能範圍內，電路執行會縮短許多。

在本節中：

- Amazon Braket 搭配 PennyLane
- Amazon Braket 範例筆記本中的混合演算法
- 具有內嵌 PennyLane 模擬器的混合演算法
- PennyLane 上的聯結漸層搭配 Amazon Braket 模擬器

- 使用混合任務和 PennyLane 執行 QAOA 演算法
- 使用 PennyLane 內嵌模擬器執行混合工作負載

Amazon Braket 搭配 PennyLane

Amazon Braket 支援 PennyLane，這是一種以量子可區分程式設計概念為基礎的開放原始碼軟體架構。您可以使用此架構，以訓練神經網路的方式訓練量子電路，以尋找量子化學、量子機器學習和最佳化中運算問題的解決方案。

PennyLane 程式庫提供熟悉的機器學習工具介面，包括 PyTorch 和 TensorFlow，讓訓練量子電路快速且直覺。

- PennyLane 程式庫 – PennyLane 已預先安裝在 Amazon Braket 筆記本中。若要從 PennyLane 存取 Amazon Braket 裝置，請開啟筆記本並使用下列命令匯入 PennyLane 程式庫。

```
import pennylane as qml
```

教學筆記本可協助您快速入門。或者，您可以從您選擇的 IDE 在 Amazon Braket 上使用 PennyLane。

- Amazon Braket PennyLane 外掛程式 - 若要使用您自己的 IDE，您可以手動安裝 Amazon Braket PennyLane 外掛程式。外掛程式會將 PennyLane 與 Amazon Braket Python SDK 連線，因此您可以在 Amazon Braket 裝置上在 PennyLane 中執行電路。若要安裝 PennyLane 外掛程式，請使用下列命令。

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

下列範例示範如何在 PennyLane 中設定對 Amazon Braket 裝置的存取權：

```
# 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)
```

如需教學課程範例和 PennyLane 的詳細資訊，請參閱 Amazon Braket 範例儲存庫。

Amazon Braket PennyLane 外掛程式可讓您使用單行程式碼在 PennyLane 中的 Amazon Braket QPU 和內嵌模擬器裝置之間切換。它提供兩個用於 PennyLane 的 Amazon Braket 量子裝置：

- `braket.aws.qubit` 使用 Amazon Braket 服務的量子裝置執行，包括 QPUs 和模擬器
- `braket.local.qubit` 使用 Amazon Braket SDK 的本機模擬器執行

Amazon Braket PennyLane 外掛程式是開放原始碼。您可以從 PennyLane 外掛程式 GitHub 儲存庫安裝它。

如需 PennyLane 的詳細資訊，請參閱 PennyLane 網站上的文件。

Amazon Braket 範例筆記本中的混合演算法

Amazon Braket 確實提供各種範例筆記本，這些筆記本不依賴 PennyLane 外掛程式來執行混合式演算法。您可以開始使用任何說明變化方法的 Amazon Braket 混合式範例筆記本，例如 Quantum 近似最佳化演算法 (QAOA) 或變化量 Eigensolver (VQE)。

Amazon Braket 範例筆記本依賴 Amazon Braket Python SDK。開發套件提供框架，可透過 Amazon Braket 與量子運算硬體裝置互動。它是一種開放原始碼程式庫，旨在協助您處理混合工作流程的量子部分。

您可以使用我們的範例筆記本進一步探索 Amazon Braket。

具有內嵌 PennyLane 模擬器的混合演算法

Amazon Braket 混合任務現在隨附 PennyLane 的高效能 CPU 和 GPU 型內嵌模擬器。此系列的內嵌模擬器可以直接內嵌在您的混合任務容器中，並包含快速狀態向量`lightning.qubit`模擬器、使用 NVIDIA 的 cuQuantum 程式庫加速的`lightning.gpu`模擬器等。這些內嵌模擬器非常適合各種演

算法，例如量子機器學習，這些演算法可以從輔助差異化方法等進階方法中受益。您可以在一或多個 CPU 或 GPU 執行個體上執行這些內嵌模擬器。

使用混合任務，您現在可以使用傳統協同處理器和 QPU 的組合、例如 的 Amazon Braket 隨需模擬器 SV1，或直接從 PennyLane 使用內嵌模擬器來執行變化演算法程式碼。

內嵌模擬器已與混合任務容器搭配使用，您需要使用@hybrid_job裝飾器裝飾您的主要 Python 函數。若要使用 PennyLane `lightning.gpu` 模擬器，您也需要在 中指定 GPU 執行個體`InstanceConfig`，如下列程式碼片段所示：

```
import pennylane as qml
from braket.jobs import hybrid_job
from braket.jobs.config import InstanceConfig


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

請參閱範例筆記本，以開始使用 PennyLane 內嵌模擬器搭配混合任務。

PennyLane 上的聯結漸層搭配 Amazon Braket 模擬器

使用適用於 Amazon Braket 的PennyLane外掛程式，您可以在本機狀態向量模擬器或 SV1 上執行時，使用聯結差異化方法來計算漸層。

注意：若要使用聯結差異化方法，您必須在 `diff_method='device'`中指定 qnode，而不是 `diff_method='adjoint'`。請參閱以下範例。

```
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**
>
> 目前，PennyLane會運算 QAOA Hamiltonians 的分組索引，並使用它們將 Hamiltonian 分割為多個預期值。如果您想要在從 執行 QAOA 時使用 SV1 的聯結差異化功能 PennyLane，則需要移除分組索引來重建 Hamiltonian 成本，如下所示： `cost_h, mixer_h = qml.qaoa.max_clique(g, constrained=False) cost_h = qml.Hamiltonian(cost_h.coeffs, cost_h.ops)`

使用混合任務和 PennyLane 執行 QAOA 演算法

在本節中，您將使用學到的知識，使用 PennyLane 搭配參數編譯來撰寫實際的混合式程式。您可以使用演算法指令碼來解決 Quantum 近似最佳化演算法 (QAOA) 問題。程式會建立與傳統 Max Cut 最佳化問題對應的成本函數，指定參數化量子電路，並使用梯度下降方法來最佳化參數，以將成本函數降至最低。在此範例中，為了簡單起見，我們在演算法指令碼中產生問題圖表，但對於更典型的使用案例，最佳實務是透過輸入資料組態中的專用管道提供問題規格。旗標`parametrize_differentiable`預設為 `True`，因此您可以從支援的 QPUs 上的參數編譯中自動獲得改善執行時間效能的優勢。

```
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**
>
> 除了脈衝層級程式Rigetti Computing之外， 的所有超執行、以閘道為基礎的 QPUs都支援參數編譯。

使用 PennyLane 內嵌模擬器執行混合工作負載

讓我們來看看如何在 Amazon Braket 混合任務上使用來自 PennyLane 的內嵌模擬器來執行混合工作負載。Pennylane 的 GPU 型內嵌模擬器 `lightning.gpu`使用 Nvidia cuQuantum 程式庫來加速電路模擬。內嵌 GPU 模擬器已預先在所有 Braket 任務容器中設定，使用者可以立即使用。在此頁面上，我們會示範如何使用 `lightning.gpu` 來加速混合式工作負載。

將 **`lightning.gpu`**用於 QAOA 工作負載

考慮此筆記本中的 Quantum 近似最佳化演算法 (QAOA) 範例。若要選取內嵌模擬器，請將`device`引數指定為格式的字串：`"local:<provider>/<simulator_name>"`。例如，您會`"local:pennylane/lightning.gpu"`為 設定 `lightning.gpu`。您在啟動時提供給混合任務的裝置字串會以環境變數 的形式傳遞給任務`"AMZN_BRAKET_DEVICE_ARN"`。

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

在此頁面上，比較兩個內嵌 PennyLane 狀態向量模擬器 `lightning.qubit`（以 CPU 為基礎） 和 `lightning.gpu`（以 GPU 為基礎）。為模擬器提供自訂閘道分解，以計算各種漸層。

現在您已準備好準備混合任務啟動指令碼。使用兩種執行個體類型執行 QAOA 演算法：`m5.2xlarge`和 `p3.2xlarge`。`m5.2xlarge` 執行個體類型與標準開發人員筆記型電腦相當。`p3.2xlarge` 是具有單一 NVIDIA Volta GPU 和 16GB 記憶體的加速運算執行個體。

所有混合任務`hyperparameters`的 將相同。您只需依照下列方式變更兩行，即可嘗試不同的執行個體和模擬器。

```
# Specify device that the hybrid job will primarily be targeting
device = "local:pennylane/lightning.qubit"
```

```
# Run on a CPU based instance with about as much power as a laptop
instance_config = InstanceConfig(instanceType='ml.m5.2xlarge')
```

或：

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

> **Note**
>
> 如果您將 指定`instance_config`為使用 GPU 型執行個體，但選擇 `device`做為內嵌的 CPU 型模擬器 (`lightning.qubit`)，則不會使用 GPU。如果您想要以 GPU 為目標，請務必使用嵌入式 GPU 型模擬器！

首先，您可以建立兩個混合任務，並在具有 18 個頂點的圖形上使用 QAOA 解決 Max-Cut。這會轉換為 18 qubit 電路，相對較小且可行，可在您的筆記型電腦或`m5.2xlarge`執行個體上快速執行。

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

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

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

m5.2xlarge 執行個體的平均反覆運算時間約為 25 秒，而p3.2xlarge執行個體的反覆運算時間約為 12 秒。對於此 18 qubit 工作流程，GPU 執行個體可提供我們 2 倍的速度。如果您查看 Amazon Braket Hybrid Jobs 定價頁面，您可以看到m5.2xlarge執行個體的每分鐘成本為 0.00768 USD，而p3.2xlarge執行個體則為 0.06375 USD。若要執行總共 5 次反覆運算，使用 CPU 執行個體的費用為 0.016 USD，或使用 GPU 執行個體的費用為 0.06375 USD，兩者都相當便宜！

現在解決了 24 個頂點圖形上的 Max-Cut 問題，這會轉換為 24 個 qubit。在相同的兩個執行個體上再次執行混合任務，並比較成本。

> **Note**
>
> 在 CPU 執行個體上執行此混合任務的時間約為 5 小時。

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

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

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

m5.2xlarge 執行個體的平均反覆運算時間約為一小時，而p3.2xlarge執行個體約為兩分鐘。對於這個較大的問題，GPU 執行個體的順序會更快。您只需變更兩行程式碼，替換執行個體類型和使用的本機模擬器，即可從此加速中獲益。若要執行總共 5 次反覆運算，如此處所述，使用 CPU 執行個體的費用約為 2.27072 USD，或使用 GPU 執行個體的費用約為 0.775625 USD。CPU 用量不僅更昂貴，還需要更多時間來執行。使用由 NVIDIA CuQuantum 支援的 PennyLane 內嵌模擬器 AWS，透過可用

的 GPU 執行個體加速此工作流程，可讓您以更低的總成本和更短的時間執行具有中繼 qubit 計數（介於 20 到 30 之間）的工作流程。這表示即使問題太大而無法在筆記型電腦或類似大小的執行個體上快速執行，您也可以實驗量子運算。

Quantum 機器學習和資料平行處理

如果您的工作負載類型是在資料集上訓練的量子機器學習 (QML)，您可以使用資料平行處理進一步加速工作負載。在 QML 中，模型包含一或多個量子電路。模型可能也可能不包含傳統神經網路。使用資料集訓練模型時，模型中的參數會更新，以將損失函數降至最低。損失函數通常針對單一資料點定義，以及整個資料集平均損失的總損失。在 QML 中，損失通常會先以序列方式計算，再平均至梯度運算的總損失。此程序耗時，特別是有數百個資料點時。

由於某個資料點的損失不取決於其他資料點，因此可以平行評估損失！您可以同時評估與不同資料點相關聯的損失和梯度。這稱為資料平行處理。使用 SageMaker 的分散式資料平行程式庫，Amazon Braket Hybrid Jobs 可讓您更輕鬆地使用資料平行處理來加速訓練。

請考慮下列資料平行處理的 QML 工作負載，其使用來自已知 UCI 儲存庫的 Sonar 資料集作為二進位分類的範例。聲納資料集各有 208 個資料點，其中包含 60 個從聲納訊號彈射材料收集的功能。每個資料點都標記為「M」表示礦區，或「R」表示石頭。我們的 QML 模型包含輸入層、做為隱藏層的量子電路，以及輸出層。輸入和輸出層是在 PyTorch 中實作的傳統神經網路。量子電路使用 PennyLane 的 qml.qnn 模組與 PyTorch 神經網路整合。如需工作負載的詳細資訊，請參閱我們的範例筆記本。如同上述的 QAOA 範例，您可以使用 PennyLane 等嵌入式 GPU 型模擬器來利用 GPU 的強大功能`lightning.gpu`，以改善與嵌入式 CPU 型模擬器相比的效能。

若要建立混合任務，您可以透過其關鍵字引數呼叫`AwsQuantumJob.create`並指定演算法指令碼、裝置和其他組態。

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

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

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

```
)
```

若要使用資料平行處理，您需要修改 SageMaker 分散式程式庫演算法指令碼中的幾行程式碼，以正確平行化訓練。首先，您可以匯入 套件，該smdistributed套件會處理將工作負載分散到多個 GPUs 和多個執行個體的大部分繁重工作。此套件已在 Braket PyTorch 和 TensorFlow 容器中預先設定。dist 模組會告知我們的演算法指令碼，訓練的 GPUs 總數 (world_size) 以及 GPU 核心local_rank的 rank和 。 rank 是 GPU 在所有執行個體的絕對索引，而 local_rank是執行個體內 GPU 的索引。例如，如果有四個執行個體，每個都有八個 GPUs 配置給訓練，rank範圍從 0 到 31，local_rank範圍從 0 到 7。

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

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

接著，您可以DistributedSampler根據 定義 world_sizerank，然後將它傳遞至資料載入器。此取樣器會避免 GPUs 存取資料集的相同配量。

```
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,
)
```

接著，您可以使用 DistributedDataParallel類別來啟用資料平行處理。

```
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"])
```

以上是使用資料平行處理所需的變更。在 QML 中，您通常想要儲存結果並列印訓練進度。如果每個 GPU 執行儲存和列印命令，日誌會充滿重複的資訊，而結果會互相覆寫。若要避免這種情況，您只能從具有 0 `rank` 的 GPU 儲存和列印。

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

Amazon Braket Hybrid Jobs 支援 SageMaker 分散式資料平行程式庫的`ml.p3.16xlarge`執行個體類型。您可以透過混合任務中的 `InstanceConfig` 引數來設定執行個體類型。若要讓 SageMaker 分散式資料平行程式庫知道已啟用資料平行處理，您需要新增兩個額外的超參數，將 `"sagemaker_distributed_dataparallel_enabled"`設定為 `"true"` ，並將 `"sagemaker_instance_type"`設定為您正在使用的執行個體類型。`smdistributed` 套件會使用這兩個超參數。您的演算法指令碼不需要明確使用它們。在 Amazon Braket SDK 中，它提供方便的關鍵字引數 `distribution`。在建立混合式任務`distribution="data_parallel"`時，Amazon Braket SDK 會自動為您插入兩個超參數。如果您使用 Amazon Braket API，則需要包含這兩個超參數。

設定執行個體和資料平行處理後，您現在可以提交混合任務。`ml.p3.16xlarge` 執行個體中有 8 個 GPUs。當您設定 `instanceCount=1` 時，工作負載會分散到執行個體中的 8 個 GPUs。當您設定`instanceCount`大於一個時，工作負載會分散到所有執行個體中可用的 GPUs。使用多個執行個體時，每個執行個體都會根據您使用的時間而產生費用。例如，當您使用四個執行個體時，計費時間為每個執行個體執行時間的四倍，因為有四個執行個體同時執行您的工作負載。

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

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

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

> **Note**
>
> 在上述混合任務建立中，`train_dp.py`是使用資料平行處理的修改演算法指令碼。請注意，只有在您根據上述章節修改演算法指令碼時，資料平行處理才能正常運作。如果資料平行處理選項在未正確修改演算法指令碼的情況下啟用，混合任務可能會擲回錯誤，或者每個 GPU 可能會重複處理相同的資料配量，這並不有效。

比較上述二進位分類問題使用 26 分位元量子電路訓練模型時，範例中的執行時間和成本。此範例中使用的`ml.p3.16xlarge`執行個體每分鐘花費 0.4692 美元。如果沒有資料平行處理，模擬器大約需要 45 分鐘來訓練 1 個 epoch（即超過 208 個資料點）的模型，而且成本約為 20 美元。透過跨 1 個執行個體和 4 個執行個體的資料平行處理，分別只需要 6 分鐘和 1.5 分鐘，這兩者的轉換約為 2.8 USD。透過在 4 個執行個體間使用資料平行處理，您不僅能將執行時間提升 30 倍，還能將成本大幅降低！

搭配 Amazon Braket 使用 CUDA-Q

NVIDIA's CUDA-Q 是一種軟體程式庫，專為結合 CPUs、GPUs 和 Quantum 處理單元 (QPUs) 的程式設計混合量子演算法而設計。它提供統一的程式設計模型，允許開發人員在單一程式中同時表達傳統和量子指令，簡化工作流程。 使用其內建的 CPU 和 GPU 模擬器CUDA-Q加速量子程式模擬和執行時間。

在 Amazon Braket 混合任務CUDA-Q上使用 可提供彈性的隨需運算環境。運算執行個體只會在工作負載期間執行，確保您只需支付使用量的費用。Amazon Braket Hybrid Jobs 也提供可擴展的體驗。使用者可以從用於原型設計和測試的較小執行個體開始，然後擴展到能夠處理更大工作負載的大型執行個體，以進行完整的實驗。

Amazon Braket 混合任務支援 GPUs，對於最大化 CUDA-Q的潛力至關重要。與 CPU 型模擬器相比，GPUs 可大幅加速量子程式模擬，特別是在使用高 qubit 計數電路時。在 Amazon Braket 混合任

務CUDA-Q上使用 時，平行化會變得直接。混合任務可簡化多個運算節點之間電路取樣和可觀測評估的分佈。這種無縫的CUDA-Q工作負載平行化可讓使用者更專注於開發工作負載，而不是為大規模實驗設定基礎設施。

若要開始使用，請參閱 Amazon Braket 範例 Github 上的CUDA-Q入門範例，CUDA-Q透過自有容器 (BYOC) 建立 支援的任務容器。請確定您擁有適當的 IAM 許可，以建置容器並將其發佈CUDA-Q至 Amazon ECR 儲存庫。

下列程式碼片段是使用 Amazon Braket Hybrid Jobs 執行CUDA-Q程式`hello-world`的範例。

```
image_uri = "<ecr-image-uri>"

@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
```

上述範例模擬 CPU 模擬器上的 Bell 電路。此範例會在您的筆記型電腦或 Braket Jupyter 筆記本上執行。由於 `local=True`設定，當您執行此指令碼時，容器會在您的本機環境中啟動，以執行 CUDA-Q 程式來測試和偵錯。完成測試後，您可以移除 `local=True`旗標並執行任務 AWS。若要進一步了解，請參閱 Amazon Braket 混合任務入門。

如果您的工作負載具有高 qubit 計數、大量電路或大量反覆運算，您可以透過指定 `instance_config`設定來使用更強大的 CPU 運算資源。下列程式碼片段示範如何在`hybrid_job`裝飾項目中設定 `instance_config`設定。如需支援執行個體類型的詳細資訊，請參閱設定混合任務執行個體以執行指令碼。如需執行個體類型的清單，請參閱 Amazon EC2 執行個體類型。

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

對於要求更高的工作負載，您可以在 CUDA-Q GPU 模擬器上執行工作負載。若要啟用 GPU 模擬器，請使用後端名稱 nvidia。nvidia 後端以 CUDA-Q GPU 模擬器的形式運作。接著，選取支援 NVIDIA GPU 的 Amazon EC2 執行個體類型。下列程式碼片段顯示 GPU 設定的hybrid_job裝飾項目。

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

Amazon Braket Hybrid Jobs 支援使用 的平行 GPU 模擬CUDA-Q。您可以平行評估多個可觀測項目或多個電路，以提升工作負載的效能。若要平行處理多個可觀測項目，請對演算法指令碼進行下列變更。

設定nvidia後端mgpu的選項。這是平行化可觀測項目的必要項目。平行處理使用 MPI 在 GPUs 之間進行通訊，因此 MPI 需要在執行之前初始化，並在之後完成。

接著，透過設定 來指定執行模式execution=cudaq.parallel.mpi。下列程式碼片段顯示這些變更。

```
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()
```

在hybrid_job裝飾項目中，指定託管多個 GPUs 的執行個體類型，如下列程式碼片段所示。

```
@hybrid_job(
    device="local:nvidia/nvidia-mqpu",
```

```
    instance_config=InstanceConfig(instanceType="ml.p3.8xlarge", instanceCount=1),
    image_uri=image_uri,
)
def parallel_observables_gpu_job(sagemaker_mpi_enabled=True):
    ...
```

Amazon Braket 範例 Github 中的平行模擬筆記本提供end-to-end範例，示範如何在 GPU 後端上執行量子程式模擬，以及對可觀測項目和電路批次執行平行模擬。

在量子電腦上執行工作負載

完成模擬器測試後，您可以轉換至在 QPUs 上執行實驗。只要將目標切換到 Amazon Braket QPU，例如 IQM、 IonQ或 Rigetti 裝置。下列程式碼片段說明如何將目標設定為IQM Garnet裝置。如需可用QPUs的清單，請參閱 Amazon Braket 主控台。

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

如需 Amazon Braket 混合任務的詳細資訊，請參閱開發人員指南中的使用 Amazon Braket 混合任務。若要進一步了解 CUDA-Q，請參閱 CUDA-Q 文件。

使用 OpenQASM 3.0 執行您的電路

Amazon Braket 現在支援閘道式量子裝置和模擬器的 OpenQASM 3.0。本使用者指南提供有關 Braket 支援的 OpenQASM 3.0 子集的資訊。Braket 客戶現在可以選擇使用 SDK 提交 Braket 電路，或使用 Amazon Braket API 和 Amazon Braket Python SDK 直接提供 OpenQASM 3.0 字串給所有閘道型裝置。

本指南中的主題會逐步解說如何完成下列量子任務的各種範例。

- 在不同的 Braket 裝置上建立和提交 OpenQASM 量子任務
- 存取支援的操作和結果類型
- 使用 OpenQASM 模擬雜訊
- 使用逐字編譯搭配 OpenQASM
- 故障診斷 OpenQASM 問題

本指南也提供特定硬體特定功能的簡介，這些功能可在 Braket 上使用 OpenQASM 3.0 實作，以及進一步資源的連結。

在本節中：

什麼是 OpenQASM 3.0？

Open Quantum Assembly Language (OpenQASM) 是量子指示的中繼表示法。OpenQASM 是一種開放原始碼架構，廣泛用於閘道型裝置的量子程式規格。使用 OpenQASM，使用者可以程式設計構成量子運算建置區塊的量子閘道和測量操作。舊版 OpenQASM (2.0) 已由多個量子程式設計程式庫用來描述基本程式。

新版本的 OpenQASM (3.0) 擴展先前的版本以包含更多功能，例如脈衝層級控制、閘道計時和傳統控制流程，以彌補最終使用者界面和硬體描述語言之間的差距。GitHub OpenQASM 3.x Live Specification 提供目前 3.0 版的詳細資訊和規格。OpenQASM 的未來開發由 OpenQASM 3.0 技術指導委員會管理，該委員會 AWS 是 IBM、Microsoft 和 University of Innsbruck 的成員。

何時使用 OpenQASM 3.0

OpenQASM 提供表達式架構，透過非架構特定性的低階控制項來指定量子程式，因此非常適合做為多個閘道型裝置的表示。對 OpenQASM 的 Braket 支援進一步將其採用為開發閘道式量子演算法的一致方法，減少使用者在多個架構中學習和維護程式庫的需求。

如果您在 OpenQASM 3.0 中有現有的程式庫，您可以調整它們以與 Braket 搭配使用，而不是完全重寫這些電路。研究人員和開發人員也應受益於越來越多的可用第三方程式庫，並支援 OpenQASM 中的演算法開發。

OpenQASM 3.0 的運作方式

從 Braket 支援 OpenQASM 3.0，可提供與目前中繼表示法相同的功能。這表示您現在可以在硬體裝置和隨需模擬器上使用 Braket 執行的任何操作，您可以使用 Braket 使用 OpenQASM 執行API。您可以透過直接提供 OpenQASM 字串給所有以閘道為基礎的裝置來執行 OpenQASM 3.0 程式，其方式類似於目前將電路提供給 Braket 上的裝置的方式。 OpenQASM Braket 使用者也可以整合支援 OpenQASM 3.0 的第三方程式庫。本指南的其餘部分會詳細說明如何開發 OpenQASM 表示法以搭配 Braket 使用。

先決條件

若要在 Amazon Braket 上使用 OpenQASM 3.0，您必須擁有 Amazon Braket Python Schemas 1.8.0 版和 Amazon Braket Python SDK 1.17.0 版或更新版本。

如果您是第一次使用 Amazon Braket，則需要啟用 Amazon Braket。如需說明，請參閱啟用 Amazon Braket。

Braket 支援哪些 OpenQASM 功能？

下一節列出 Braket 支援的 OpenQASM 3.0 資料類型、陳述式和 pragma 指示。

在本節中：

- 支援的 OpenQASM 資料類型
- 支援的 OpenQASM 陳述式
- Braket OpenQASM pragmas
- Local Simulator 上 OpenQASM 的進階功能支援
- 使用 OpenPulse 支援的操作和文法

支援的 OpenQASM 資料類型

Amazon Braket 支援下列 OpenQASM 資料類型。

- 非負整數用於（虛擬和實體） qubit 索引：

  - `cnot q[0], q[1];`
  - `h $0;`
- 浮點數或常數可用於閘道旋轉角度：
  - `rx(-0.314) $0;`
  - `rx(pi/4) $0;`

> **Note**
>
> pi 是 OpenQASM 中的內建常數，無法用作參數名稱。

- 複雜數字陣列（具有虛構部分的 OpenQASM `im`表示法）允許使用結果類型 pragmas 來定義一般疱疱可觀測值，並以單一 pragmas 表示：
  - `#pragma braket unitary [[0, -1im], [1im, 0]] q[0]`
  - `#pragma braket result expectation hermitian([[0, -1im], [1im, 0]]) q[0]`

支援的 OpenQASM 陳述式

Amazon Braket 支援下列 OpenQASM 陳述式。

- `Header: OPENQASM 3;`
- 傳統位元宣告：
  - `bit b1;`（相當於 `creg b1;`）
  - `bit[10] b2;`（相當於 `creg b2[10];`）
- Qubit 宣告：
  - `qubit b1;`（相當於 `qreg b1;`）
  - `qubit[10] b2;`（相當於 `qreg b2[10];`）
- 陣列內的索引：`q[0]`
- 輸入：`input float alpha;`
- 實體 的規格qubits：`$0`
- 裝置上支援的閘道和操作：
  - `h $0;`

- `iswap q[0], q[1];`

> **Note**
>
> 您可以在 OpenQASM 動作的裝置屬性中找到裝置支援的閘道；使用這些閘道不需要閘道定義。

- 逐字方塊陳述式。目前，我們不支援方塊持續時間表示法。逐字方塊中qubits需要原生閘道和實體。

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

- qubits 或整個qubit暫存器上的測量和測量指派。
  - `measure $0;`
  - `measure q;`
  - `measure q[0];`
  - `b = measure q;`
  - `measure q # b;`

Braket OpenQASM pragmas

Amazon Braket 支援下列 OpenQASM pragma 說明。

- 雜訊小數
  - `#pragma braket noise bit_flip(0.2) q[0]`
  - `#pragma braket noise phase_flip(0.1) q[0]`
  - `#pragma braket noise pauli_channel`
- 逐字 pragmas
  - `#pragma braket verbatim`
- 結果類型 pragmas
  - 基礎不可變的結果類型：

  - 狀態向量： `#pragma braket result state_vector`
  - 密度矩陣： `#pragma braket result density_matrix`
- 梯度運算 pragmas：
  - 聯結漸層： `#pragma braket result adjoint_gradient expectation(2.2 * x[0] @ x[1]) all`
- Z 基礎結果類型：
  - 振幅： `#pragma braket result amplitude "01"`
  - 機率： `#pragma braket result probability q[0], q[1]`
- 基底輪換的結果類型
  - 期望： `#pragma braket result expectation x(q[0]) @ y([q1])`
  - 變異數： `#pragma braket result variance hermitian([[0, -1im], [1im, 0]]) $0`
  - 範例： `#pragma braket result sample h($1)`

> **Note**
>
> OpenQASM 3.0 與 OpenQASM 2.0 回溯相容，因此使用 2.0 編寫的程式可以在 Braket 上執行。不過，Raket 支援的 OpenQASM 3.0 功能有一些次要語法差異，例如 `qreg` vs `creg`和 `qubit` vs `bit`。測量語法也有差異，因此需要以正確的語法支援這些語法。

Local Simulator 上 OpenQASM 的進階功能支援

`LocalSimulator` 支援進階 OpenQASM 功能，這些功能不會作為 Braket 的 QPU 或隨需模擬器的一部分提供。下列功能清單僅支援 `LocalSimulator`：

- 閘道修飾詞
- OpenQASM 內建閘道
- 傳統變數
- 傳統操作
- 自訂閘道
- 傳統控制
- QASM 檔案

- 子例行程序

如需每個進階功能的範例，請參閱此範例筆記本。如需完整的 OpenQASM 規格，請參閱 OpenQASM 網站。

使用 OpenPulse 支援的操作和文法

支援的 OpenPulse 資料類型

Cal 區塊：

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

Defcal 區塊：

```
// 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 {
...
}
```

影格：

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

波形：

```
// 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};
```

自訂閘道校正範例：

```
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;
```

任意脈衝範例：

```
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);
}
```

建立並提交範例 OpenQASM 3.0 量子任務

您可以使用 Amazon Braket Python SDK、Boto3 或 AWS CLI ，將 OpenQASM 3.0 量子任務提交至 Amazon Braket 裝置。

在本節中：

- OpenQASM 3.0 程式範例
- 使用 Python SDK 建立 OpenQASM 3.0 量子任務
- 使用 Boto3 建立 OpenQASM 3.0 量子任務
- 使用 AWS CLI 建立 OpenQASM 3.0 任務

OpenQASM 3.0 程式範例

若要建立 OpenQASM 3.0 任務，您可以從準備 GHZ 狀態的基本 OpenQASM 3.0 程式 (ghz.qasm) 開始，如下列範例所示。

```
// 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;
```

使用 Python SDK 建立 OpenQASM 3.0 量子任務

您可以使用 Amazon Braket Python SDK，將此程式提交至具有下列程式碼的 Amazon Braket 裝置。請務必將範例 Amazon S3 儲存貯體位置「amzn-s3-demo-bucket」取代為您自己的 Amazon S3 儲存貯體名稱。

```
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,
)
```

使用 Boto3 建立 OpenQASM 3.0 量子任務

您也可以使用 AWS Python SDK for Braket (Boto3)，使用 OpenQASM 3.0 字串建立量子任務，如下列範例所示。下列程式碼片段參考 ghz.qasm，以準備 GHZ 狀態，如上所示。

```
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,
)
```

使用 AWS CLI 建立 OpenQASM 3.0 任務

AWS Command Line Interface (CLI) 也可用來提交 OpenQASM 3.0 程式，如下列範例所示。

```
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)
    }'
```

支援不同 Braket 裝置上的 OpenQASM

對於支援 OpenQASM 3.0 的裝置， `action` 欄位支援透過 `GetDevice` 回應的新動作，如下列 Rigetti 和 IonQ 裝置範例所示。

```
//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"
            ],
            ….
        }
    }
}
```

對於支援脈衝控制的裝置， pulse 欄位會顯示在GetDevice回應中。下列範例顯示Rigetti裝置的pulse此欄位。

```
// 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
    }
  }
}
```

上述欄位詳細說明下列項目：

連接埠：

除了指定連接埠的關聯屬性之外，說明 QPU 上宣告的預先製作外部 (extern) 裝置連接埠。此結構中列出的所有連接埠都會在使用者提交的OpenQASM 3.0程式中預先宣告為有效的識別符。連接埠的其他屬性包括：

- 連接埠 ID (portId)
  - 在 OpenQASM 3.0 中宣告為識別符的連接埠名稱。
- 方向 （方向）
  - 連接埠的方向。磁碟機連接埠傳輸脈衝 （方向 “tx”），而測量連接埠接收脈衝 （方向 “rx”）。
- 連接埠類型 (portType)
  - 此連接埠負責的動作類型 （例如，磁碟機、擷取或 ff - 快速流量）。
- Dt (dt)
  - 代表指定連接埠上單一範例時間步驟的時間，以秒為單位。
- Qubit 映射 (qubitMappings)
  - 與指定連接埠相關聯的 qubit。
- 中心頻率 (centerFrequencies)
  - 連接埠上所有預先宣告或使用者定義影格的關聯中心頻率清單。如需詳細資訊，請參閱影格。
- QHP 特定屬性 (qhpSpecificProperties)
  - 選用的映射，詳細說明有關 QHP 特定連接埠的現有屬性。

影格：

描述在 QPU 上宣告的預先製作外部影格，以及影格的相關屬性。此結構中列出的所有影格都會在使用者提交的OpenQASM 3.0程式中預先宣告為有效的識別符。影格的其他屬性包括：

- 框架 ID (frameId)
  - 在 OpenQASM 3.0 中宣告為識別符的影格名稱。
- 連接埠 ID (portId)
  - 影格的相關硬體連接埠。
- 頻率 （頻率）
  - 影格的預設初始頻率。
- 中心頻率 (centerFrequency)
  - 影格的頻率頻寬中心。一般而言，影格只能調整為圍繞中心頻率的特定頻寬。因此，頻率調整應保持在中心頻率的特定差異內。您可以在驗證參數中找到頻寬值。

- 階段 （ 階段 ）
  - 影格的預設初始階段。
- 關聯的閘道 (associatedGate)
  - 與指定影格相關聯的閘道。
- Qubit 映射 (qubitMappings)
  - 與指定影格相關聯的 qubit。
- QHP 特定屬性 (qhpSpecificProperties)
  - 選用地圖，詳細說明 QHP 特定影格的現有屬性。

SupportsDynamicFrames：

描述是否可以透過 OpenPulse`newframe`函數在 `cal`或 `defcal`區塊中宣告影格。如果這是 false，則只能在程式中使用影格結構中列出的影格。

SupportedFunctions：

除了指定OpenPulse函數的相關聯引數、引數類型和傳回類型之外，說明裝置支援的函數。若要查看使用 OpenPulse函數的範例，請參閱 OpenPulse 規格。目前，Raket 支援：

- shift_phase
  - 將影格的階段轉移為指定的值
- set_phase
  - 將影格的階段設定為指定的值
- swap_phases
  - 交換兩個影格之間的階段。
- shift_frequency
  - 將影格的頻率轉移為指定的值
- set_frequency
  - 將影格的頻率設定為指定的值
- 播放
  - 排程波形
- capture_v0
  - 將擷取影格上的值傳回位元登錄

SupportedQhpTemplateWaveforms：

描述裝置上可用的預先建置波形函數，以及相關聯的引數和類型。根據預設，Raket Pulse 會在所有裝置上提供預先建置的波形常式，包括：

常數

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

τ 是波形的長度，而 iq 是複雜的數字。

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

高斯文

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

τ 是波形的長度， σ是高斯的寬度， A是振幅。如果ZaE將 設定為 True，高斯會偏移並重新調整規模，以便在波形的開始和結束時等於零，並達到A最大值。

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

DRAG 高斯文

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

τ 是波形的長度， σ 是高斯的寬度， β 是免費參數， A是振幅。如果ZaE將 設定為 True，則由 Adiabatic Gate (DRAG) Gaussian 進行的衍生移除會偏移和重新調整規模，以便在波形的開始和結束時等於零，且實際部分達到A最大值。如需 DRAG 波形的詳細資訊，請參閱白皮書 Simple Pulses for 為避免微弱非線性 Qubits 中的洩漏。

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

歐夫廣場

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

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

其中 L是長度， W是波形的寬度， σ定義邊緣上升和下降的速度，t_1=(L-W)/2 而 $t_2$2=(L+W)/2A是振幅。如果ZaE將 設定為 True，高斯會偏移並重新調整規模，使其在波形的開始和結束時等於零，並達到A最大值。下列方程式是重新調整規模的波形版本。

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

a=erf(W/2σ)和 的位置b=erf(-t_1/σ)/2+erf(t $_2$/σ)/2 。

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

SupportsLocalPulseElements：

描述是否可以在defcal區塊中本機定義脈衝元素，例如連接埠、影格和波形。如果值為 false，則必須在cal區塊中定義元素。

SupportsNonNativeGatesWithPulses：

描述我們是否可以或不能將非原生閘道與脈衝程式結合使用。例如，如果沒有先defcal透過 為使用的 qubit 定義H閘道，您就無法在程式中使用非原生閘道，例如閘道。您可以在裝置功能下找到原生閘道nativeGateSet金鑰的清單。

ValidationParameters：

描述脈衝元素驗證界限，包括：

- 波形的最大縮放/最大振幅值 （任意和預先建置）
- 提供中心頻率的最大頻率頻寬，以 Hz 為單位
- 最小脈衝長度/持續時間，以秒為單位
- 脈衝長度/持續時間上限，以秒為單位

使用 OpenQASM 支援的操作、結果和結果類型

若要了解每個裝置支援哪些 OpenQASM 3.0 功能，您可以在裝置功能輸出的 `action` 欄位中參考 `braket.ir.openqasm.program`金鑰。例如，下列是 Braket State Vector 模擬器 可用的支援操作和結果類型SV1。

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

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

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

使用 OpenQASM 3.0 模擬雜訊

若要使用 OpenQASM3 模擬雜訊，您可以使用 pragma 指示來新增雜訊運算子。例如，若要模擬先前提供的 GHZ 程式的雜訊版本，您可以提交下列 OpenQASM 程式。

```
// 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;
```

下列清單提供所有支援的 pragma 雜訊運算子規格。

```
#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 運算子

若要產生 Kraus 運算子，您可以逐一查看矩陣清單，將矩陣的每個元素列印為複雜表達式。

使用 Kraus 運算子時，請記住下列事項：

- 的數量qubits不得超過 2。結構描述中的目前定義會設定此限制。
- 引數清單的長度必須是 8 的倍數。這表示它只能由 2x2 矩陣組成。
- 總長度不超過 2^{2*num_qubits} 矩陣。這表示 1 有 4 個矩陣qubit，2 有 16 個矩陣qubits。
- 所有提供的矩陣都是完全正面的追蹤保留 (CPTP)。
- Kraus 運算子的產品及其轉置共軛需要加到身分矩陣。

Qubit 使用 OpenQASM 3.0 重新配線

Amazon Braket 支援Rigetti裝置上 OpenQASM 內的實體qubit表示法（如需進一步了解，請參閱此頁面)。將實體 qubits與原生重新配線策略搭配使用時，請確保qubits在選取的裝置上連接 。或者，如果改為使用qubit註冊，Rigetti則裝置預設會啟用 PARTIAL rewiring 策略。

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

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

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

使用 OpenQASM 3.0 逐字編譯

當您在 Rigetti、 和 等廠商提供的量子電腦上執行量子電路時IonQ，您可以指示編譯器完全依照定義執行您的電路，無需進行任何修改。此功能稱為逐字編譯。使用 Rigetti 裝置，您可以精確指定保留的項目，無論是整個電路，或只保留其中的特定部分。若要僅保留電路的特定部分，您需要在保留區域內使用原生閘道。目前， IonQ 僅支援整個電路的逐字編譯，因此電路中的每個指令都需要用逐字方塊括住。

使用 OpenQASM，您可以明確地在程式碼方塊周圍指定一字不差的 pragma，然後不受硬體的低階編譯常式最佳化。下列程式碼範例示範如何使用 `#pragma braket verbatim`指令來達成此目標。

```
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;
```

如需有關逐字編譯程序的詳細資訊，包括範例和最佳實務，請參閱 amazon-braket-examples github 儲存庫中提供的逐字編譯範例筆記本。

Braket 主控台

OpenQASM 3.0 任務可供使用，並且可以在 Amazon Braket 主控台中管理。在 主控台上，您在 OpenQASM 3.0 中提交量子任務的經驗與提交現有量子任務的經驗相同。

其他資源

OpenQASM 適用於所有 Amazon Braket 區域。

如需在 Amazon Braket 上開始使用 OpenQASM 的範例筆記本，請參閱 Braket 教學課程 GitHub。

使用 OpenQASM 3.0 計算梯度

在 `shots=0`（確切）模式下執行時，Amazon Braket 支援隨需和本機模擬器上的漸層運算。這是透過使用聯結差異化方法來實現的。若要指定要運算的漸層，您可以提供適當的 pragma，如下列範例中程式碼所示。

```
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
```

您也可以在 pragma 中指定`all`關鍵字，而不是明確列出所有個別參數。這將根據列出的所有`input`參數計算梯度，當參數數量非常大時，這可能是方便的選項。在此情況下，pragma 看起來會像下列範例中的程式碼。

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

Amazon Braket 的 OpenQASM 3.0 實作支援所有可觀測類型，包括個別運算子、張量產品、Hermitian 可觀測項目和可`Sum`觀測項目。運算漸層時您想要使用的特定運算子必須包裝在 `expectation()`函數內，而且必須明確指定可觀測的每個字詞對其執行動作的 qubit。

使用 OpenQASM 3.0 測量特定 qubit

Amazon Braket 提供的本機狀態向量模擬器和本機密度矩陣模擬器支援提交OpenQASM程式，其中可以選擇性地測量電路的 qubit 子集。此功能通常稱為部分測量，可讓量子運算更具目標性和效率。例如，在以下程式碼片段中，您可以建立雙 qubit 電路，並選擇只測量第一個 qubit，同時不測量第二個 qubit。

```
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];
"""
```

在此範例中，我們有一個具有兩個 qubit `q[0]`和 的量子電路`q[1]`，但我們只有興趣測量第一個 qubit 的狀態。這是由行 所達成`b[0] = measure q[0]`，該行會測量 qubit【0】 的狀態，並將結果存放在傳統位元 b【0】 中。若要執行此部分測量案例，我們可以在 Amazon Braket 提供的本機狀態向量模擬器上執行下列程式碼。

```
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)
```

您可以檢查裝置是否支援部分測量，方法是檢查其動作屬性中的 `requiresAllQubitsMeasurement` 欄位；如果是 `False`，則支援部分測量。

```
from braket.devices import Devices

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

在這裡， `requiresAllQubitsMeasurement`是 `False`，表示並非所有 qubit 都必須測量。

探索實驗功能

實驗功能提供對硬體的存取，其可用性有限且新功能的出現。對於 QuEra Aquila 的功能，您必須直接在 Braket 主控台中請求存取可用的實驗性功能。

若要請求存取 QuEra Aquila 的實驗功能：

1. 導覽至 Amazon Braket 主控台，然後選取左側選單中的 Braket Direct，然後導覽至實驗功能區段。
2. 選擇取得存取權並填寫請求的資訊。
3. 提供工作負載的詳細資訊，以及您計劃使用此功能的位置。

在本節中：

存取 QuEra Aquila 上的本機微調

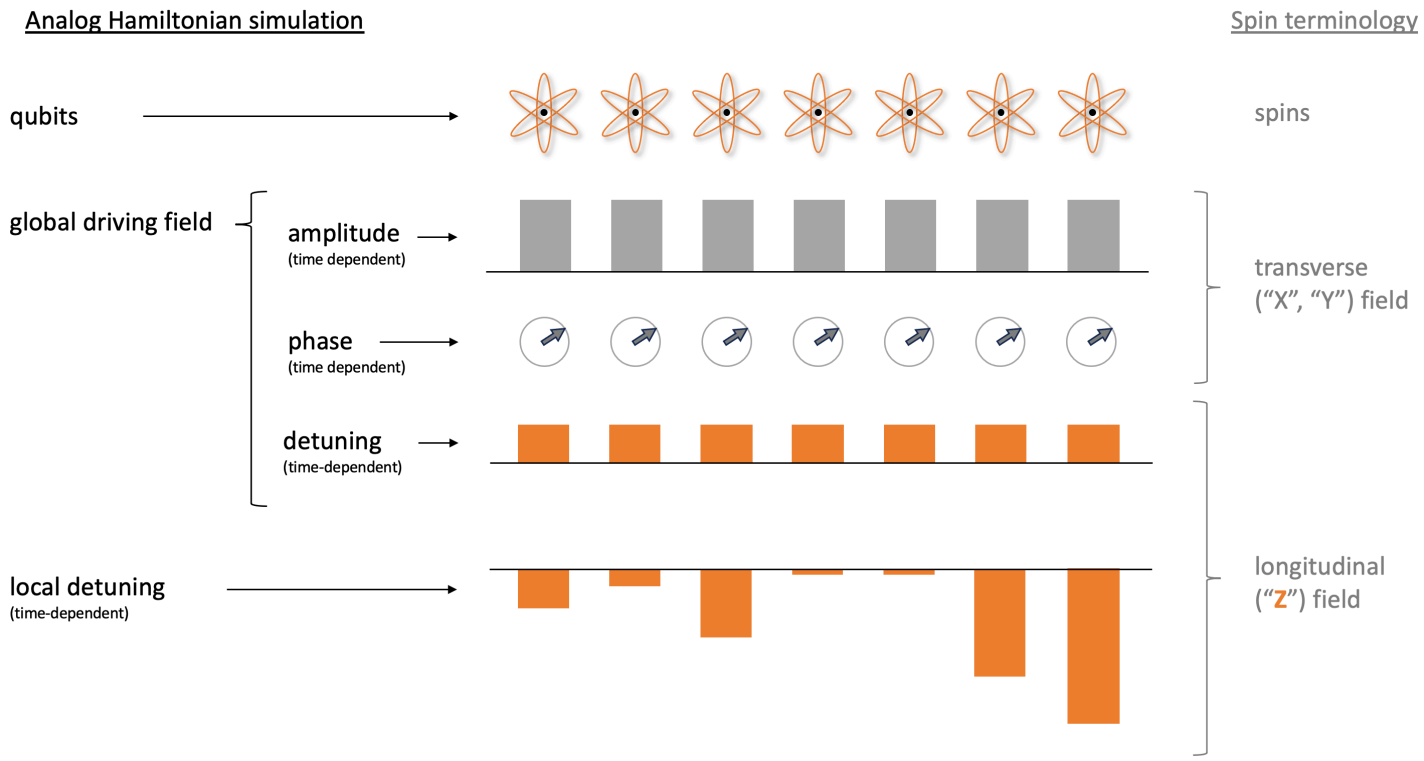
Local detuning (LD) 是一種新的時間相依控制欄位，具有可自訂的空間模式。LD 欄位會根據可自訂的空間模式影響 qubits，除了統一駕駛欄位和 Rydberg-Rydberg 互動可以建立的內容之外，為不同的 qubit 實現不同的 Hamiltonians。

限制條件：

本機調整欄位的空間模式可為每個 AHS 程式自訂，但在程式過程中是固定的。本機調整欄位的時間序列必須以零開始和結束，且所有值都小於或等於零。此外，本機調整欄位的參數會受到數值限制，可透過特定裝置屬性 - 中的 Braket SDK 檢視`aquila_device.properties.paradigm.rydberg.rydbergLocal`。

限制：

執行使用本機調整欄位的量子程式時（即使其大小在漢密爾頓中設定為常數零），裝置會體驗比 Aquila 屬性效能區段中列出的 T2 時間更快的一致性。如果不需要，最佳實務是從 AHS 計劃的漢密爾頓省略本機位移欄位。



範例：

1. 模擬旋轉系統中不均勻縱向磁場的效果

   雖然驅動欄位的振幅和階段對 qubits 的影響與旋轉時的橫向磁場相同，但驅動欄位的調校和局部調校的總和對 qubits 的影響與旋轉時的縱向欄位相同。透過對本機調整欄位的空間控制，可以模擬更複雜的旋轉系統。

2. 準備非平衡初始狀態

範例筆記本使用 Rydberg 原子模擬格線圖理論，示範如何在將系統退火至 Z2 排序階段時，抑制 9-atom 線性排列的中央原子被激動。在準備步驟之後，本機調整欄位會緩慢下降，而且 AHS 程式會繼續模擬系統從此特定非平衡狀態開始的時間演變。

3. 解決加權最佳化問題

   筆記本範例 最大權重獨立集 (MWIS) 顯示如何在 Aquila 上解決 MWIS 問題。本機調整欄位用於定義單位磁碟圖表節點上的權重，其邊緣由 Rybderg-blockage 效果實現。從統一的地面狀態開始，並逐漸擴大本機調整欄位，可讓系統轉換為 MWIS Hamiltonian 的地面狀態，以尋找問題的解決方案。

存取 QuEra Aquila 上的高幾何

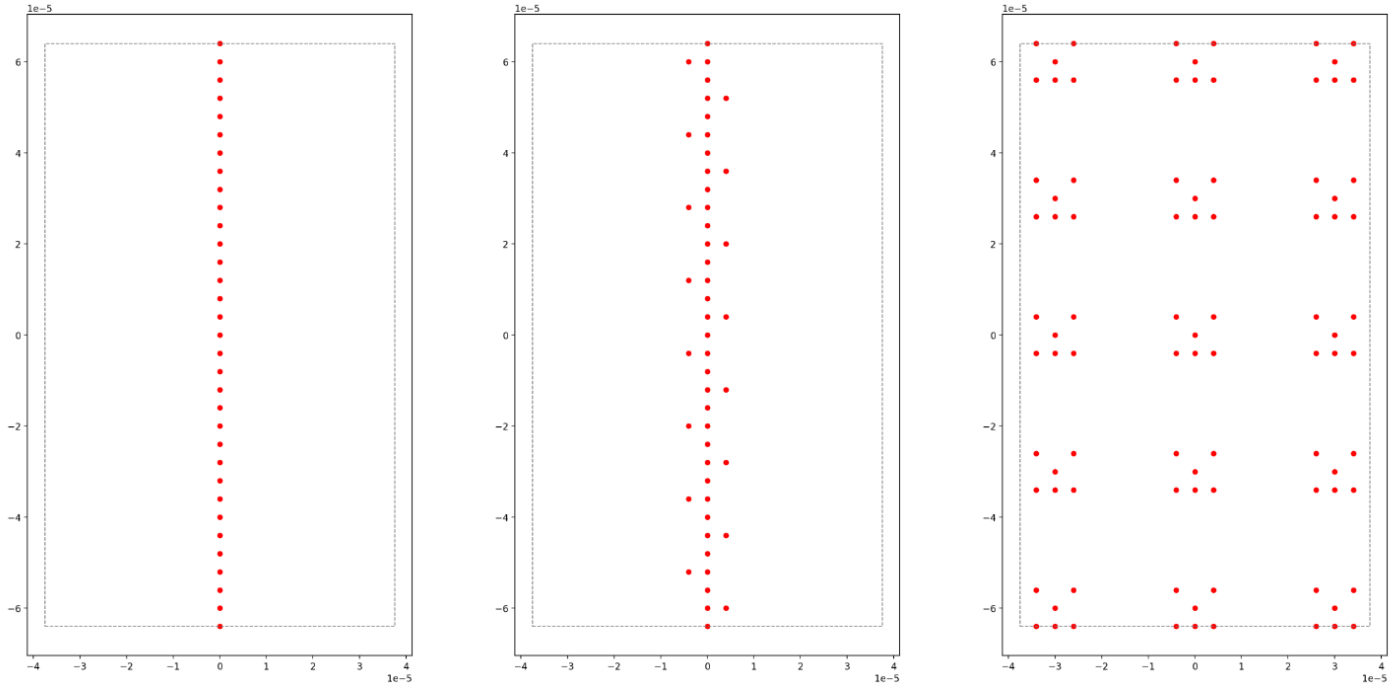
較高的幾何體功能可讓您指定高度較高的幾何體。有了此功能，您的 AHS 程式的原子排列可以在 Aquila 一般功能以外的 y 方向上跨越額外的長度。

限制條件：

高幾何體的最大高度為 0.000128 公尺 (128 um)。

限制：

當您的帳戶啟用此實驗性功能時，裝置屬性頁面上顯示的功能和`GetDevice`呼叫會繼續反映高度的一般下限。當 AHS 程式使用超出一般功能的原子排列時，填充錯誤預期會增加。在任務結果`pre_sequence`的部分中，您會發現非預期的 0s 數量增加，進而降低取得完美初始化安排的機會。此效果在具有許多原子的資料列中最強大。



範例：

1. 較大的 1d 和 quasi-1d 排列

   原子鏈和梯形排列可以擴展到更高的原子數。透過將長方向平行於 y，允許程式設計這些模型的較長執行個體。

2. 使用小型幾何體多工執行任務的更多空間

   Aquila 上的範例筆記本平行量子任務顯示如何充分利用可用區域：透過在一個原子排列中放置有問題的幾何的多工副本。使用更多可用區域，可以放置更多副本。

存取 QuEra Aquila 上的緊密幾何

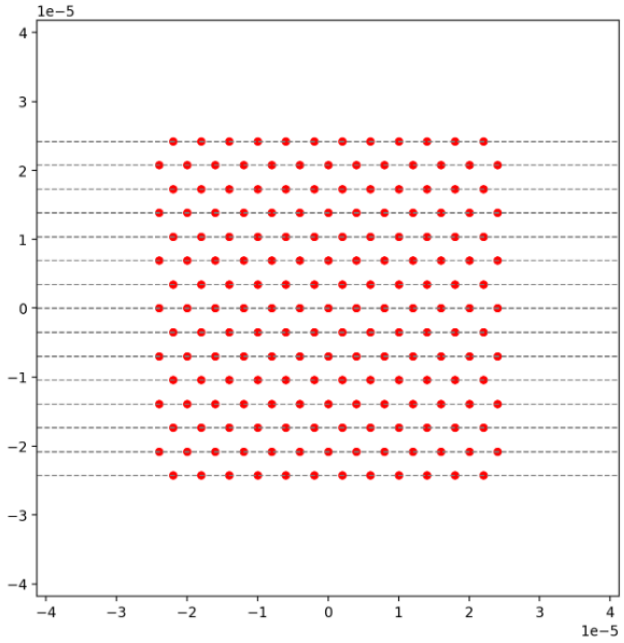
緊密幾何功能可讓您指定相鄰資料列之間間距較短的幾何。在 AHS 程式中，原子以資料列排列，並以最小的垂直間距分隔。任何兩個原子位點的 y 座標必須為零（相同資料列），或相差大於最小資料列間距（不同資料列）。藉助緊密的幾何體功能，最小的資料列間距會減少，從而建立更緊密的原子排列。雖然此延伸不會變更原子之間的最小歐幾里得距離需求，但它允許建立格線，其中遠端原子佔用彼此更接近的相鄰列，但值得注意的範例是三角形格線。

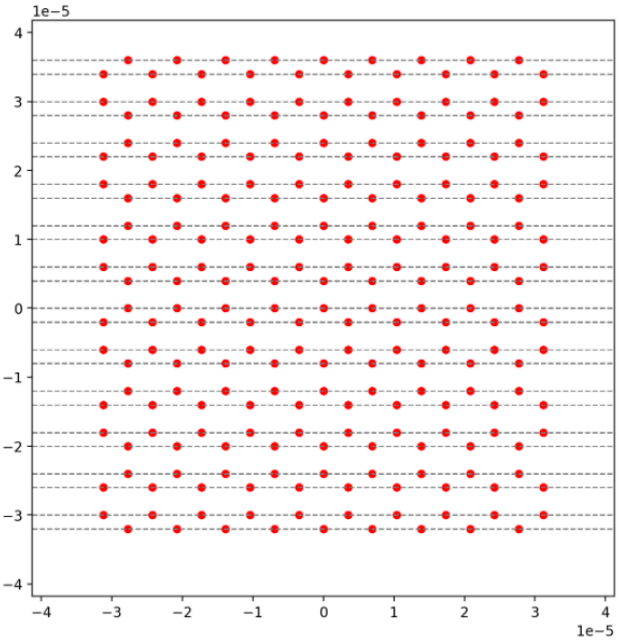
限制條件：

緊密幾何體的最小資料列間距為 0.00002 公尺 (2 um)。

限制：

當您的帳戶啟用此實驗性功能時，裝置屬性頁面上顯示的功能和GetDevice呼叫會繼續反映高度的一般下限。當 AHS 程式使用超出一般功能的原子排列時，填充錯誤預期會增加。客戶會在任務結果pre_sequence的部分中找到更多非預期的 0，進而降低取得完美初始化安排的機會。此效果在具有許多原子的資料列中最強大。





範例：

1. 具有小格子常數的非矩形格子

   更緊密的資料列間距允許建立格線，其中最接近某些原子的相鄰處位於對角方向。值得注意的範例是三角形、六邊形和 Kagome 格狀，以及一些半晶形。

2. 可調整的格子系列

   在 AHS 程式中，透過調整原子對之間的距離來調整互動。更緊密的資料列間距允許更自由地調整不同原子對相對於彼此的互動，因為定義原子結構的角度和距離不受最小資料列間距限制條件的限制。值得注意的範例是具有不同鍵長度的 Shastry-Sutherland 格子系列。

IQM 裝置上的動態電路

IQM 裝置上的動態電路可啟用中電路測量 (MCM) 和向前饋送操作。這些功能可讓量子研究人員和開發人員實作具有條件式邏輯和量子重複使用功能的進階量子演算法。此實驗性功能有助於探索量子演算法，改善資源效率，以及研究量子錯誤緩解和錯誤修正機制。

金鑰指示：

- `measure_ff`：實作前饋控制的測量，測量 qubit，並使用意見回饋金鑰儲存結果。
- `cc_prx`：實作傳統控制的輪換，只有在與指定意見回饋金鑰相關聯的結果測量 |1# 狀態時才適用。

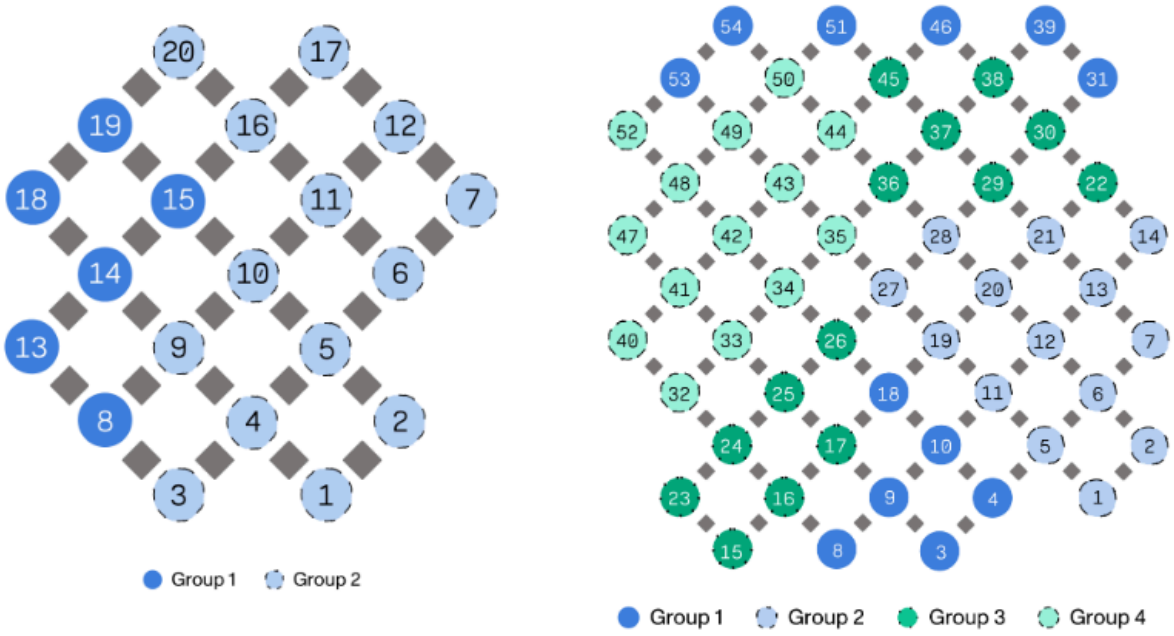
Amazon Braket 透過 OpenQASM、 Amazon Braket SDK和 支援動態電路Amazon Braket Qiskit Provider。

限制條件：

1. `measure_ff` 指示中的意見回饋金鑰必須是唯一的。
2. `cc_prx` 必須使用相同的意見回饋金鑰`measure_ff`在 之後發生。
3. 在單一電路中，對 qubit 的向前饋送只能由一個 qubit 控制，無論是單獨或由另一個 qubit 控制。在不同的電路中，您可以有不同的控制對。
   a. 例如，如果 qubit 1 由 qubit 2 控制，則無法由相同電路中的 qubit 3 控制。在 qubit 1 和 qubit 2 之間套用控制項的次數沒有限制。Qubit 2 可以透過 qubit 3 （或 qubit 1) 控制，除非在 qubit 2 上執行主動重設。
4. 控制項只能套用至相同群組中的 qubit。IQM Garnet 和 Emerald 裝置的 qubit 群組位於下列影像中。
5. 具有這些功能的程式必須以逐字方式提交。若要進一步了解逐字程式，請參閱使用 OpenQASM 3.0 進行逐字編譯。

限制：

MCM 只能用於程式中的前饋控制。MCM 結果 (0 或 1) 不會在任務結果中傳回。



這些影像會顯示兩個IQM裝置的 qubit 分組。Garnet 20 qubit 裝置包含 2 個 qubit 群組，而 Emerald 54 qubit 裝置包含 4 個 qubit 群組。

範例：

1. 透過主動重設進行 Qubit 重複使用

   具有條件式重設操作的 MCM 可在單一電路執行中啟用 qubit 重複使用。這可減少電路深度需求，並改善量子裝置資源使用率。

2. 主動位元翻轉保護

   動態電路可偵測位元翻轉錯誤，並根據測量結果套用修正操作。此實作可做為量子錯誤偵測實驗。

3. 電信實驗

   狀態電信使用本機量子操作和 MCMs 的傳統資訊來傳輸 qubit 狀態。Gate Teleportation 會在 qubit 之間實作閘道，而不需要直接 quantum 操作。這些實驗在三個關鍵領域中示範基礎子常式：量子錯誤校正、以測量為基礎的量子運算和量子通訊。

4. 開放式量子系統模擬

   動態電路會透過資料量位元和環境糾結，以及環境測量來模擬量子系統中的雜訊。此方法使用特定 qubit 來表示資料和環境元素。雜訊通道可由套用至環境的閘道和測量設計。

如需使用動態電路的詳細資訊，請參閱 Amazon Braket 筆記本儲存庫中的其他範例。

Amazon Braket 上的脈衝控制

脈衝是控制量子電腦中 qubit 的類比訊號。透過 Amazon Braket 上的特定裝置，您可以存取脈衝控制功能，以使用脈衝提交電路。您可以透過 Braket SDK、使用 OpenQASM 3.0 或直接透過 Braket APIs 存取脈衝控制。首先，介紹在 Braket 中用於脈衝控制的一些關鍵概念。

在本節中：

- 影格
- 連接埠
- 波形
- 影格和連接埠的角色
- 使用 Hello Pulse
- 使用脈衝存取原生閘道

影格

框架是一種軟體抽象，可同時做為量子程式和階段中的時鐘。時鐘時間會在每次使用量和由頻率定義的具狀態電信業者訊號上遞增。將訊號傳輸到 qubit 時，影格會決定 qubit 的載波頻率、階段位移，以及發出波形信封的時間。在 Braket Pulse 中，建構影格取決於裝置、頻率和階段。視裝置而定，您可以選擇預先定義的影格，或提供連接埠來執行個體化新的影格。

```
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)
```

連接埠

連接埠是一種軟體抽象，代表控制 qubit 的任何輸入/輸出硬體元件。它有助於硬體供應商提供使用者可以與之互動的界面，以操作和觀察 qubit。連接埠的特徵是代表連接器名稱的單一字串。此字串也會公開最小時間增量，指定我們可以定義波形的精細程度。

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

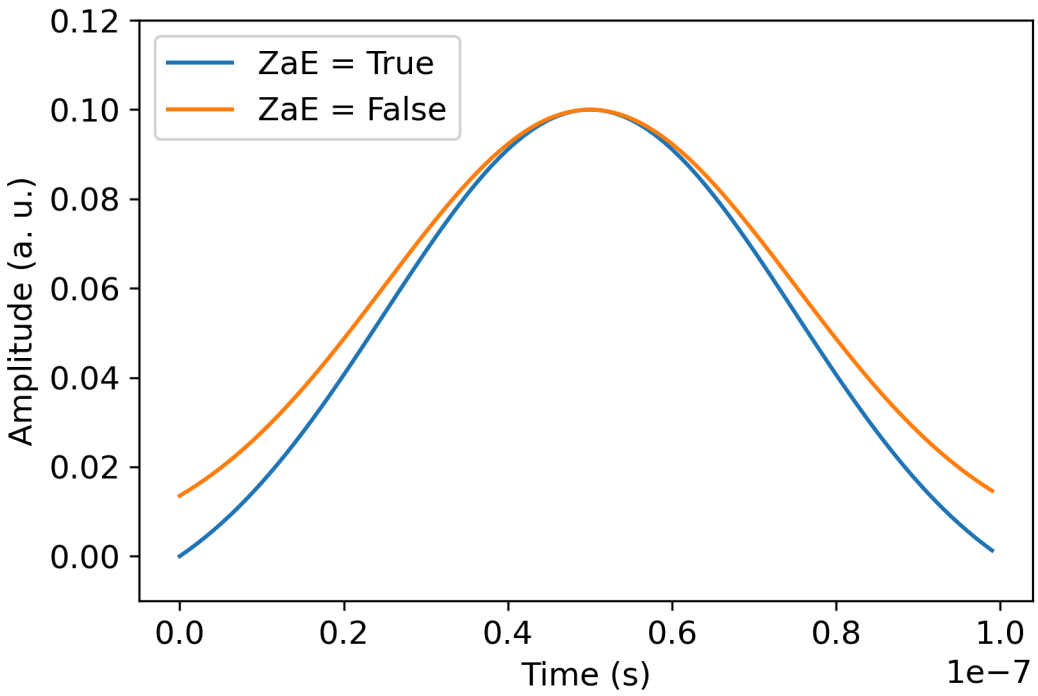
波形

波形是一種時間相依的信封，可用來在輸出連接埠上發出訊號，或透過輸入連接埠擷取訊號。您可以透過複雜數字清單或使用波形範本從硬體提供者產生清單，直接指定您的波形。

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

Braket Pulse 提供標準的波形程式庫，包括常數波形、高斯波形，以及依二軸閘道 (DRAG) 波形的衍生移除。您可以透過 `sample`函數擷取波形資料，以繪製波形的形狀，如下列範例所示。

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



上圖描述從 建立的高斯波形`GaussianWaveform`。我們選擇 100 ns 的脈衝長度、25 ns 的寬度，以及 0.1 的振幅 （任意單位）。波形以脈衝視窗為中心。 `GaussianWaveform`接受布林引數 `zero_at_edges`（圖例中的 ZaE)。設為 時`True`，此引數會偏移高斯波形，讓 t=0 和 t=`length` 的點位於零，並重新調整其振幅，讓最大值對應至`amplitude`引數。

影格和連接埠的角色

本節說明每個裝置可用的預先定義影格和連接埠。我們也將簡短討論在特定影格上播放脈衝時涉及的機制。

Rigetti 影格

Rigetti 裝置支援預先定義的影格，其頻率和階段已校正為與相關聯 qubit 進行共振。命名慣例是指 `q{i}[_q{j}]_{role}_frame` `{i}` 是指第一個 qubit 數字，`{j}`是指第二個 qubit 數字，以防影格用於啟用兩個 qubit 互動，並`{role}`是指影格的角色。角色如下所示：

- `rf` 是驅動 qubit 0-1 轉換的框架。脈衝會以先前透過 `set`和 `shift`函數提供的頻率和階段的微波暫時性訊號傳輸。訊號的時間相依振幅是由影格上播放的波形提供。框架會插入單一四位元的對角線外互動。如需詳細資訊，請參閱 Krantz et al. 和 Rahamim et al.。
- `rf_f12` 類似於 ，`rf`其參數以 1-2 轉換為目標。

- `ro_rx` 用於透過耦合共平面波導實現量子的分散讀取。讀取波形的頻率、階段和完整參數集會預先校正。它透過 使用`capture_v0`，除了影格識別符之外，不需要任何引數。
- `ro_tx` 用於從振盪器傳輸訊號。它目前未使用。
- `cz` 是經過校正的影格，可啟用雙 qubit `cz`閘道。與`ff`與連接埠相關聯的所有影格一樣，它會透過通量線來開啟串連互動，方法是在與其鄰接的振盪中調節配對的可調校 qubit。如需繫結機制的詳細資訊，請參閱 Reagor et al.、Caldwell et al. 和 Didier et al.。
- `cphase` 是經過校正的影格，可啟用雙 qubit `cphaseshift`閘道，並連結至 `ff` 連接埠。如需繫結機制的詳細資訊，請參閱影`cz`格的說明。
- `xy` 是經過校正的影格，可啟用雙位元 XY(θ) 閘道，並連結至 `ff` 連接埠。如需有關串連機制以及如何實現 XY 閘道的詳細資訊，請參閱`cz`框架的描述和 Abrams 等。

當以`ff`連接埠為基礎的影格轉移可調校 qubit 的頻率時，所有其他與 qubit 相關的驅動影格都會以與振幅和頻率轉移持續時間相關的量進行分裂。因此，您必須透過將對應的相移新增至相鄰 qubit 的影格來補償此效果。

連接埠

Rigetti 裝置提供可透過裝置功能檢查的連接埠清單。連接埠名稱遵循慣例，`q{i}_{type}`其中 `{i}`是指 qubit 號碼，而 `{type}`是指連接埠的類型。請注意，並非所有 qubit 都有一組完整的連接埠。連接埠的類型如下所示：

- `rf` 代表驅動單一 qubit 轉換的主要介面。它與 `rf`和 `rf_f12`影格相關聯。它以電容方式與 qubit 耦合，允許微波在 gigahertz 範圍內行駛。
- `ro_tx` 用於將訊號傳輸到以電容方式耦合到 qubit 的讀取共振器。讀取訊號交付是八邊形八倍的倍數。
- `ro_rx` 用於接收來自耦合至 qubit 之讀取振盪器的訊號。
- `ff` 代表感應耦合至 qubit 的快速通量線。我們可以用它來調整轉子的頻率。只有設計為可高度調校的 qubits 才具有`ff`連接埠。此連接埠可用來啟用 qubit-qubit 互動，因為每對相鄰的轉子之間都有靜態的電容耦合。

如需架構的詳細資訊，請參閱 Valery 等。

使用 Hello Pulse

在本節中，您將了解如何直接使用Rigetti裝置上的脈衝來描述和建構單一 qubit 閘道。將電磁欄位套用至 qubit 會導致 Rabi 振盪，並在其 0 狀態和 1 狀態之間切換 qubit。透過已校正的脈衝長度和階

段，Rabi 振盪可以計算單一 qubit 閘道。在這裡，我們將決定測量 pi/2 脈衝的最佳脈衝長度，這是用來建置更複雜脈衝序列的基本區塊。

首先，若要建置脈衝序列，請匯入 PulseSequence類別。

```
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
```

接著，使用 QPU 的 Amazon Resource Name(ARN) 執行個體化新的 Braket 裝置。下列程式碼區塊使用 Rigetti Ankaa-3。

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

下列脈衝序列包含兩個元件：播放波形和測量 qubit。脈衝序列通常可以套用至影格。除了一些例外狀況，例如障礙和延遲，這些例外狀況可以套用至 qubit。在建構脈衝序列之前，您必須擷取可用的影格。驅動框架用於套用 Rabi 振盪的脈衝，而讀取框架用於測量 qubit 狀態。此範例使用 qubit 25 的影格。如需影格的詳細資訊，請參閱影格和連接埠的角色。

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

現在，建立將在驅動影格中播放的波形。目標是描述不同脈衝長度的 qubit 行為。您每次都會播放不同長度的波形。使用脈衝序列FreeParameter中支援的 Braket，而不是每次執行個體化新波形。您可以使用免費參數建立波形和脈衝序列一次，然後使用不同的輸入值執行相同的脈衝序列。

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

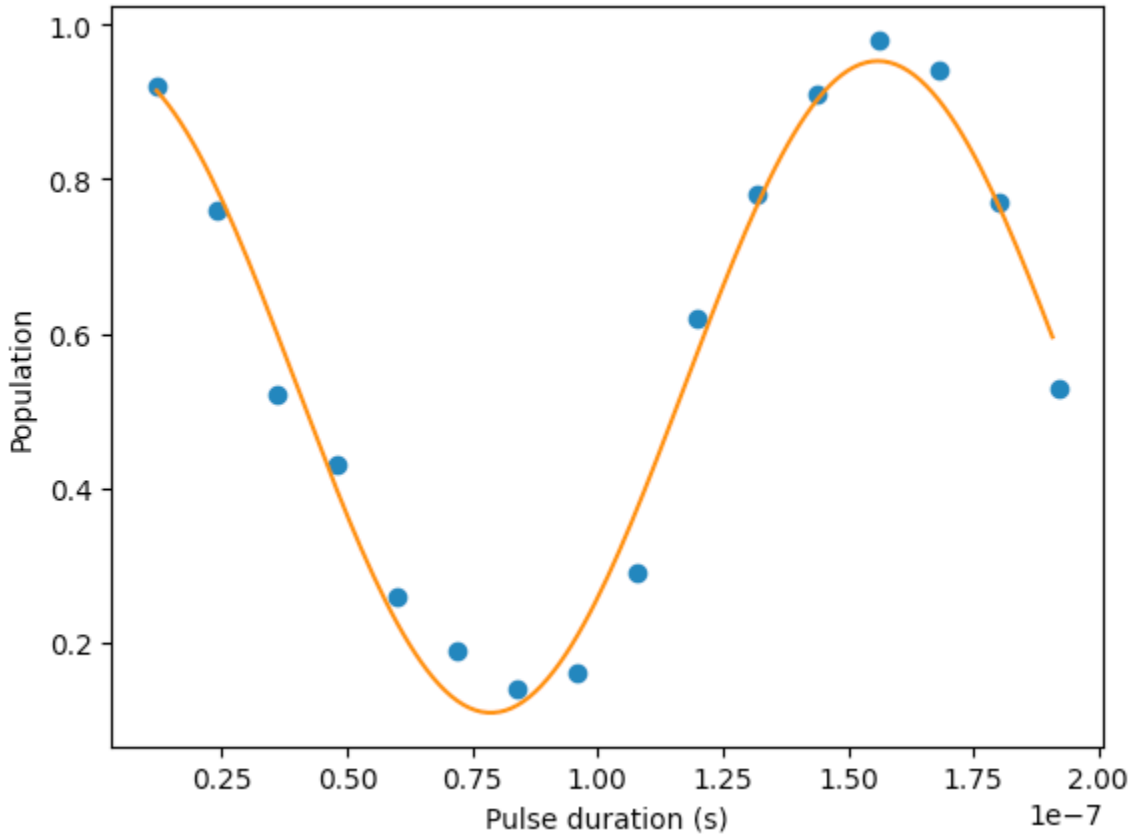
最後，將它們放在一起做為脈衝序列。在脈衝序列中， 會在驅動影格上play播放指定的波形，並從讀取影格capture_v0測量狀態。

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

掃描各種脈衝長度，並將其提交至 QPU。

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

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

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

qubit 測量的統計資料會顯示在 0 狀態和 1 狀態之間振盪的 qubit 振盪動態。從測量資料中，您可以擷取 Rabi 頻率，並微調脈波的長度，以實作特定的 1-qubit 閘道。例如，從下圖中的資料中，週期性約為 154 ns。因此，pi/2 輪換閘道會對應至長度 = 38.5ns 的脈衝序列。



使用 OpenPulse 的 Hello Pulse

OpenPulse 是一種用於指定一般量子裝置的脈衝層級控制的語言，也是 OpenQASM 3.0 規格的一部分。Amazon Braket 支援使用 OpenQASM 3.0 表示法OpenPulse直接程式設計脈衝。

Braket 使用 OpenPulse做為基礎中繼表示法，以在原生指示中表達脈衝。 OpenPulse支援以 `defcal`(「定義校正」的簡稱）宣告形式新增指令校正。透過這些宣告，您可以在較低層級的控制文法中指定閘道指令的實作。

您可以使用`PulseSequence`下列命令檢視 Braket 的 OpenPulse 程式。

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

您也可以直接建構 OpenPulse 程式。

```
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);
}
"""
```

使用指令碼建立`Program`物件。然後，將程式提交至 QPU。

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

program = Program(source=openpulse_script)

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

使用脈衝存取原生閘道

研究人員通常需要確切知道特定 QPU 支援的原生閘道如何實作為脈衝。脈衝序列由硬體提供者仔細校正，但存取它們可讓研究人員有機會設計更好的閘道，或探索錯誤緩解的通訊協定，例如透過延伸特定閘道的脈衝進行零雜訊外推。

Amazon Braket 支援從 Rigetti 以程式設計方式存取原生閘道。

```
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**
>
> 硬體供應商會定期校正 QPU，通常每天不只一次。Braket SDK 可讓您取得最新的閘道校正。

```
device.refresh_gate_calibrations()
```

若要擷取指定的原生閘道，例如 RX 或 XY 閘道，您需要傳遞`Gate`物件和感興趣的 qubit。例如，您可以檢查在 0 上套用的 RX(π/2) qubit 的脈衝實作。

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

pulse_sequence_rx_pi_2_q0 = calibrations.pulse_sequences[rx_pi_2_q0]
```

您可以使用 `filter`函數建立一組篩選後的校正。您可以傳遞閘道清單或 清單`QubitSet`。下列程式碼會建立兩組，其中包含 RX(π/2) 和 0 qubit 的所有校正。

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

現在，您可以透過連接自訂校正集來提供或修改原生閘道的動作。例如，請考慮下列電路。

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

您可以透過將PulseSequence物件字典傳遞至gate_definitions關鍵字引數qubit 0，以閘道 的自訂rx閘道校正來執行它。您可以從 GateCalibrations 物件pulse_sequences的 屬性建構字典。所有未指定的閘道都會以量子硬體提供者的脈衝校正取代。

```
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)
```

類比漢密爾頓模擬

類比漢密爾頓模擬 (AHS) 是量子運算的新興範例，與傳統的量子電路模型有很大的差異。而不是一連串的閘道，其中每個電路一次只能作用於幾個 qubit。AHS 程式是由有問題的漢密爾頓時間相依和空間相依參數所定義。系統的 Hamiltonian 會編碼其能源水準和外部力的影響，這些作用力共同管理其狀態的時間演變。對於 N-qubit 系統，Hamiltonian 可由複雜數字的 $2^N X 2^N$ 平方矩陣表示。

能夠執行 AHS 的 Quantum 裝置旨在透過仔細調校其內部控制參數，在自訂 Hamiltonian 下接近量子系統的時間演變。例如，調整一致駕駛欄位的振幅和調整參數。AHS 範例非常適合使用許多互動粒子來模擬量子系統的靜態和動態屬性，例如在壓縮物質物理或量子化學中。專門建置的量子處理單元 (QPUs)，例如來自 的 Aquila 裝置QuEra，已開發為使用 AHS 的強大功能，並以創新方式解決傳統數位量子運算方法無法觸及的問題。

在本節中：

- AHS 您好：執行您的第一個類比 Hamiltonian 模擬
- 使用 QuEra Aquila 提交類比程式

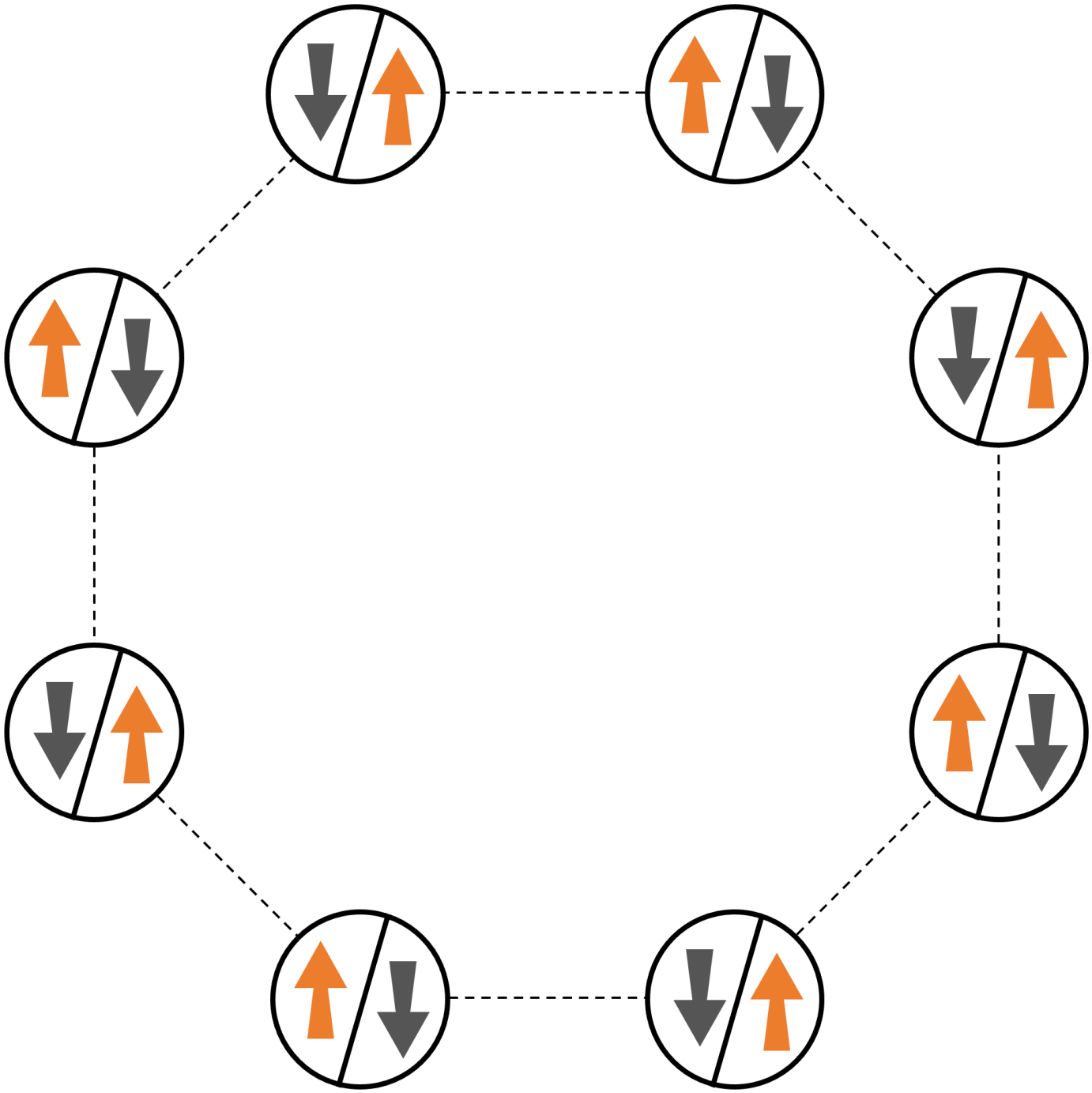
AHS 您好：執行您的第一個類比 Hamiltonian 模擬

本節提供有關執行第一個類比 Hamiltonian 模擬的資訊。

在本節中：

互動旋轉鏈

對於許多互動粒子系統的正式範例，讓我們考慮一個八次旋轉的環（每個旋轉環可以是「向上」|↑# 和「向下」|↓# 狀態）。雖然很小，但此模型系統已展現一些自然產生磁性材料的有趣現象。在此範例中，我們將示範如何準備所謂的抗鐵磁順序，其中連續旋轉指向相反方向。



安排

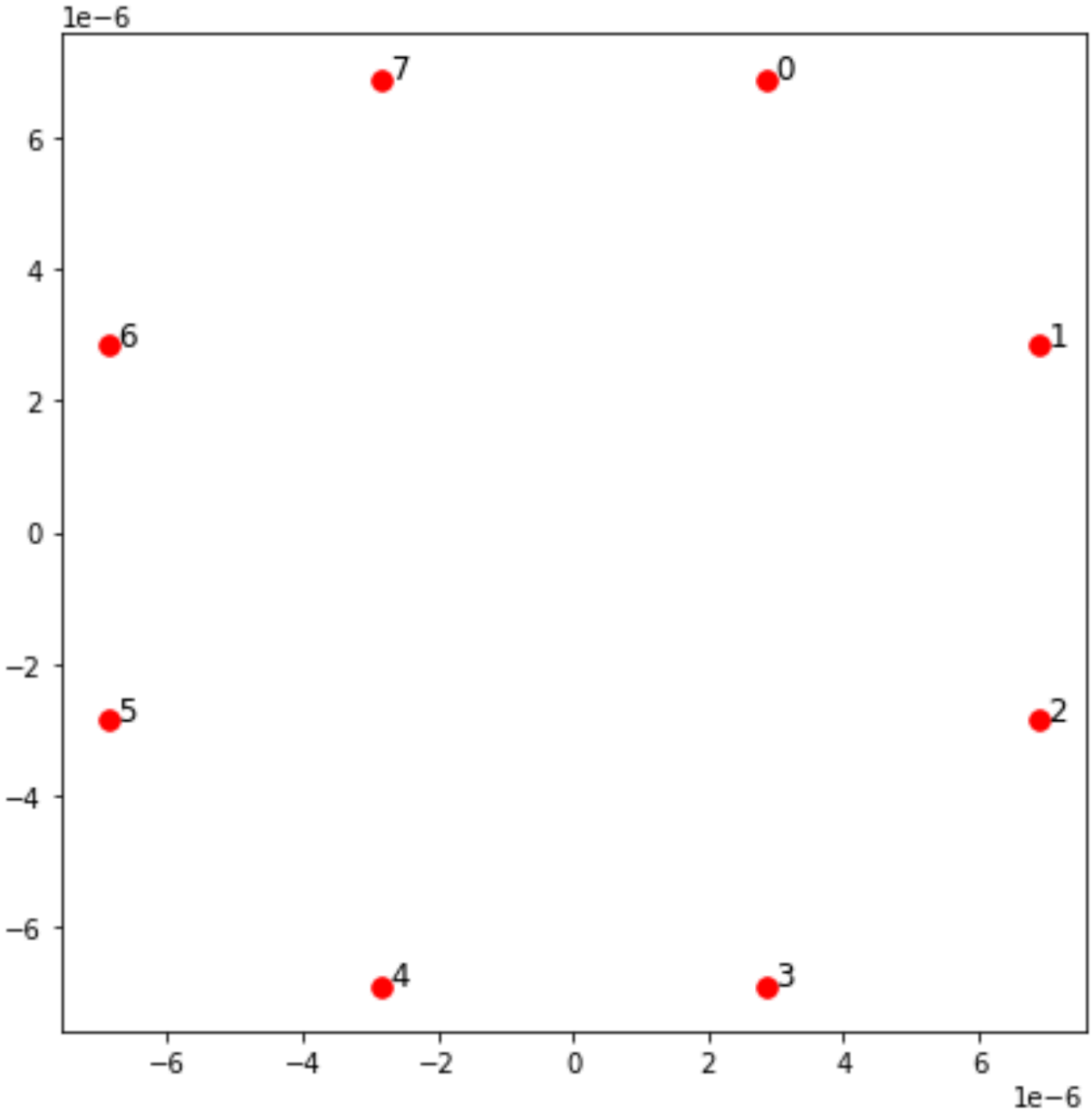
我們將使用一個中性原子來代表每次旋轉，而「向上」和「向下」旋轉狀態將分別以激磁的 Rydberg 狀態和原子的接地狀態進行編碼。首先，我們建立 2 天排列。我們可以使用下列程式碼來編寫上述旋轉環的程式。

先決條件：您需要 pip 安裝 Braket SDK。（如果您使用的是 Braket 託管筆記本執行個體，此 SDK 會預先安裝筆記本。）若要重現圖，您也需要使用 shell 命令 單獨安裝 matplotlibpip install matplotlib。

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

   from braket.ahs.atom_arrangement import AtomArrangement

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

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

我們也可以使用 繪製

```
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
```



互動

若要準備抗鐵磁階段，我們需要在鄰近旋轉之間觸發互動。我們對此使用 van der Waals 互動，這是由中性原子裝置（例如來自 Aquila的裝置QuEra) 原生實作。使用旋轉代表法，此互動的 Hamiltonian 術語可以用總和表示，超過所有旋轉對 (j，k)。

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

在這裡，nj =$_{jj}$|↑##↑| 是運算子，只有在旋轉 j 處於「上升」狀態時，才會採用 1 的值，否則為 0。強度為 $V_{j,k}=C_6/(d_{j,k})^6$，其中 $_6$Cis 是固定係數，而 dis $_{j,k}$是旋轉 j 和 k 之間的歐幾里得距離。此互動術語的立即效果是，旋轉 j 和旋轉 k 都是「向上」的任何狀態都具有更高的能量（依數量 V 計算$_{j,k}$)。透過仔細設計 AHS 計劃的其餘部分，此互動將防止相鄰旋轉同時處於「上」狀態，這是一種通常稱為「Rydberg 封鎖」的效果。

駕駛欄位

在 AHS 程式開始時，所有旋轉（預設）都會以「向下」狀態開始，處於所謂的鐵磁階段。密切關注我們準備抗鐵磁階段的目標，我們指定一個時間相依的連貫駕駛欄位，可將旋轉從此狀態順暢地轉換為多體狀態，其中「上升」狀態為偏好狀態。對應的 Hamiltonian 可以寫入為

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

其中 DLR(t)、φ(t)、Δ(t) 是時間相依的全域振幅（也稱為 Rabi 頻率)、階段，以及影響所有旋轉的駕駛欄位的調整。此處的 $S_{-,k}$=$_{kk}$|↓##↑|and $S_{+,k}=(S_{-,k})^{\dagger}$=$_{kk}$|↑##↓| 分別是旋轉 k 的降低和提高運算子，而 n_k=$_{kk}$|↑��##↑| 是與之前相同的運算子。駕駛欄位的 Vue 部分一致地聯結所有旋轉的「向下」和「向上」狀態，而 Δ 部分控制「向上」狀態的能源獎勵。

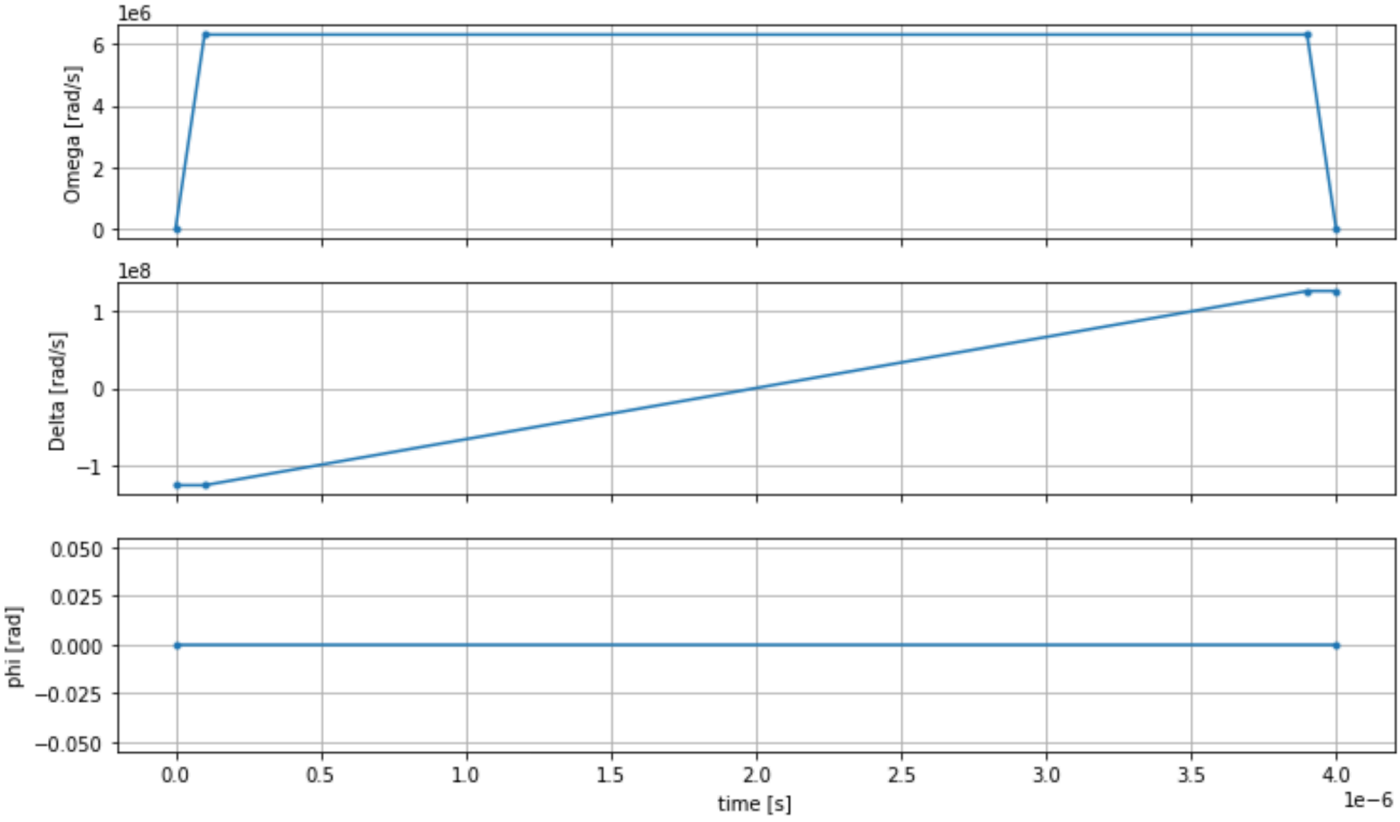
若要程式設計從鐵磁階段順利轉換為抗鐵磁階段，請使用下列程式碼指定驅動欄位。

```
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
    )
```

我們可以使用以下指令碼視覺化駕駛欄位的時間序列。

```
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 計劃

註冊、駕駛欄位（以及隱含 van der Waals 互動）組成了類比 Hamiltonian 模擬程式 `ahs_program`。

```
from braket.ahs.analog_hamiltonian_simulation import AnalogHamiltonianSimulation

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

在本機模擬器上執行

由於此範例很小（少於 15 次旋轉），因此在 AHS 相容 QPU 上執行之前，我們可以在隨附於 Braket SDK 的本機 AHS 模擬器上執行。由於本機模擬器可透過 Braket SDK 免費取得，因此最佳實務是確保程式碼可以正確執行。

在這裡，我們可以將鏡頭數量設定為高值（例如 100 萬），因為本機模擬器會追蹤量子狀態的時間演變，並從最終狀態提取樣本；因此，增加鏡頭數量，同時只稍微增加總執行時間。

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

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

分析模擬器結果

我們可以將拍攝結果彙總為下列函數，以推斷每次旋轉的狀態（可能是「d」表示「down」、「u」表示「up」或「e」表示空的網站），並計算每個組態在拍攝中發生的次數。

```
from collections import Counter

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

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

       Args:
          result
 (braket.tasks.analog_hamiltonian_simulation_quantum_task_result.AnalogHamiltonianSimulationQua

       Returns
           dict: number of times each state configuration is measured

       """
       state_counts = Counter()
       states = ['e', 'u', 'd']
       for shot in result.measurements:
           pre = shot.pre_sequence
           post = shot.post_sequence
           state_idx = np.array(pre) * (1 + np.array(post))
           state = "".join(map(lambda s_idx: states[s_idx], state_idx))
           state_counts.update((state,))
       return dict(state_counts)

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

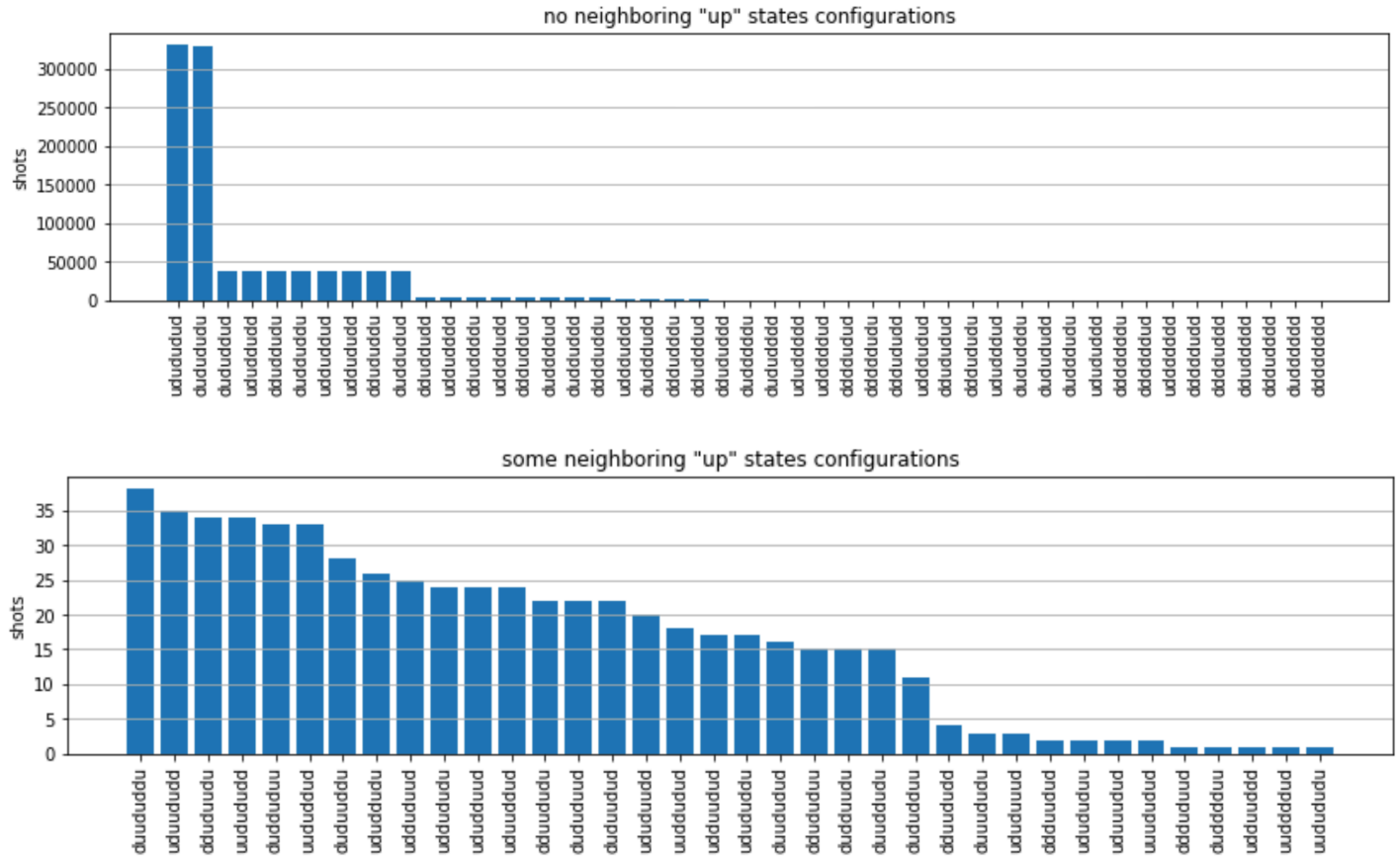
```
print(counts_simulator)
```

```
{'udududud': 330944, 'dudududu': 329576, 'dududdud': 38033, ...}
```

counts 以下是字典，用於計算跨鏡頭觀察每個狀態組態的次數。我們也可以使用以下程式碼將其視覺化。

```
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)
```



從圖中，我們可以讀取下列觀察，確認已成功準備抗鐵磁階段。

1. 一般而言，非封鎖狀態（其中沒有兩個相鄰旋轉處於「上」狀態）比至少有一個相鄰旋轉處於「上」狀態的狀態更常見。
2. 一般而言，除非組態遭到封鎖，否則具有更多 "up" 激發的狀態是有利的。
3. 最常見的狀態確實是完美的抗鐵磁狀態 "dudududu"和 "udududud"。
4. 第二個最常見的狀態是其中只有 3 個「向上」激發，連續分隔為 1、2、2。這顯示 van der Waals 互動也會影響（雖然更小）最近鄰。

在 QuEra 的 Aquila QPU 上執行

先決條件：除了 pip 安裝 Braket SDK 之外，如果您是 Amazon Braket 的新手，請確定您已完成必要的入門步驟。

> **Note**
>
> 如果您使用的是 Braket 託管筆記本執行個體，則 Braket SDK 會預先安裝執行個體。

安裝所有相依性後，我們可以連線到 Aquila QPU。

```
from braket.aws import AwsDevice

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

為了讓 AHS 程式適合QuEra機器，我們需要四捨五入所有值，以符合 Aquila QPU 允許的精確度層級。（這些要求是由名稱中具有「解析度」的裝置參數所管理。 我們可以透過在筆記本`aquila_qpu.properties.dict()`中執行來查看它們。 如需 Aquila 功能和需求的詳細資訊，請參閱 Aquila 筆記本簡介。）我們可以呼叫 `discretize`方法來執行此操作。

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

現在，我們可以在 Aquila QPU 上執行程式（目前只執行 100 個鏡頭）。

> **Note**
>
> 在Aquila處理器上執行此程式會產生費用。Amazon Braket SDK 包含成本追蹤器，可讓客戶設定成本限制，並近乎即時地追蹤其成本。

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

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

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

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

由於量子任務可能需要多長時間才能執行的很大差異（取決於可用時段和 QPU 使用率），最好記下量子任務 ARN，以便我們稍後可以使用下列程式碼片段來檢查其狀態。

```
# Optionally, in a new python session

   from braket.aws import AwsQuantumTask

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

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

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

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

一旦狀態為 COMPLETED（也可以從 Amazon Braket 主控台的量子任務頁面進行檢查），我們可以使用下列方式查詢結果：

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

分析 QPU 結果

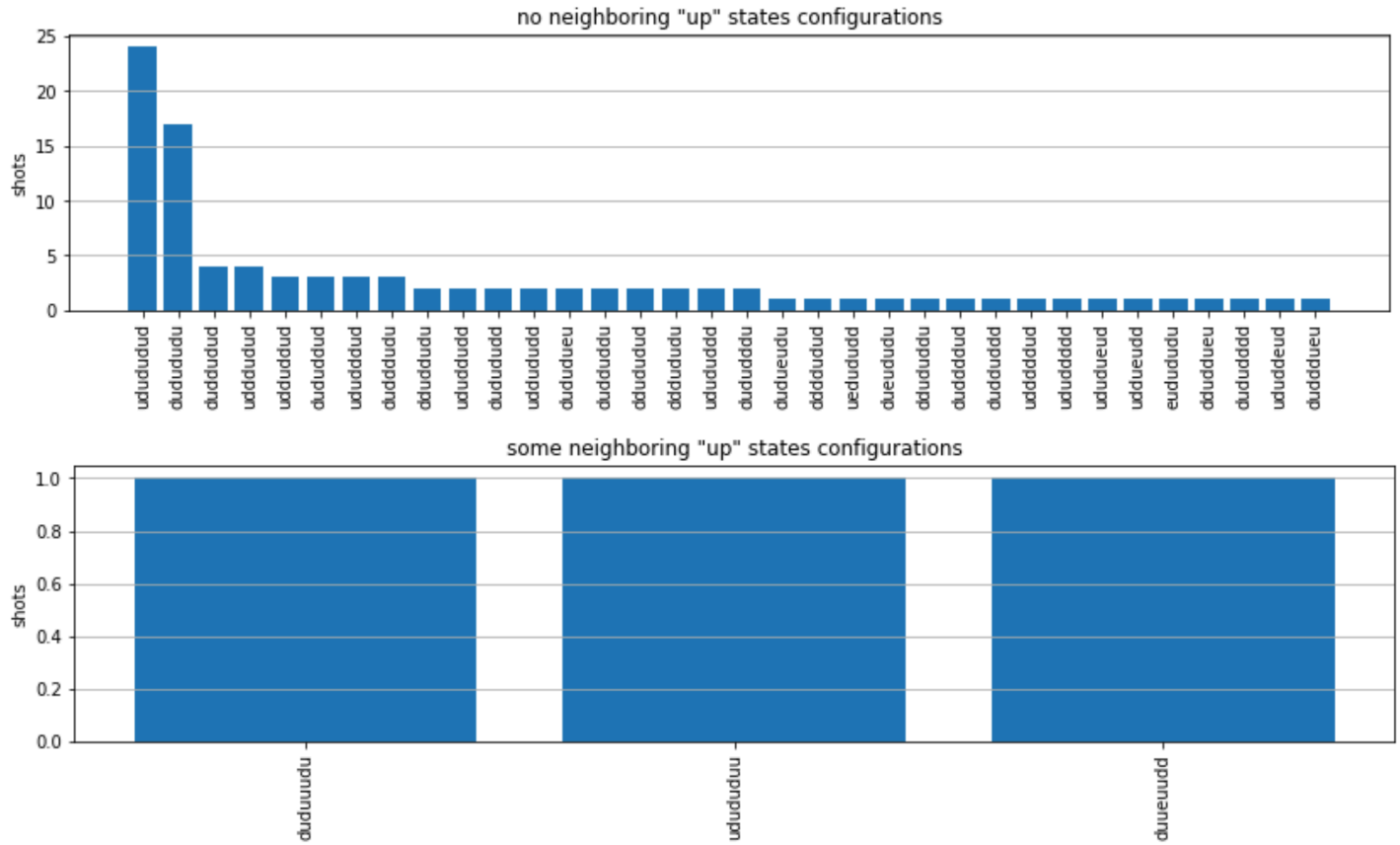
使用與之前相同的`get_counts`函數，我們可以計算計數：

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

```
*[Output]*
   {'udududud': 24, 'dudududu': 17, 'dududdud': 3, ...}
```

並使用 繪製它們`plot_counts`：

```
plot_counts(counts_aquila)
```



請注意，一小部分的鏡頭具有空的網站（以 “e” 標記）。這是因為 Aquila QPU 的每個原子準備瑕疵為 1-2%。除此之外，由於鏡頭數量較少，結果與模擬在預期的統計波動內相符。

後續步驟

恭喜，您現在已使用本機 AHS 模擬器和 Aquila QPU 在 Amazon Braket 上執行第一個 AHS 工作負載。

若要進一步了解 Rydberg 物理、類比 Hamiltonian 模擬和Aquila裝置，請參閱我們的範例筆記本。

使用 QuEra Aquila 提交類比程式

此頁面提供 Aquila機器功能的完整文件QuEra。此處涵蓋的詳細資訊如下：

1. 模擬的參數化 Hamiltonian Aquila
2. AHS 程式參數
3. AHS 結果內容

4. Aquila 功能參數

在本節中：

- 漢密爾頓文
- Braket AHS 程式結構描述
- Braket AHS 任務結果結構描述
- QuEra 裝置屬性結構描述

漢密爾頓文

來自 的Aquila機器原生QuEra模擬下列（時間相依）漢密爾頓文：

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

> **Note**
>
> 存取本機調校是一種實驗性功能，可透過 Braket Direct 請求取得。

where

- $H_{drive,k}(t)=({}^{1}/_{2}$ JPEG$(t)^{e i\varphi(t)} S_{-,k} + {}^{1}/_{2}$ JPEG$(t) e^{-i\varphi(t)} S_{+,k}) + (-{}_{global}\Delta(t) n_k)$，
  - DLR(t) 是時間相依的全域駕駛振幅（也稱為 Rabi 頻率），單位為 (rad / s)
  - φ(t) 是時間相依的全域階段，以弧度為單位
  - $S_{-,k}$ 和 ${}_{+,k}$Sare 是 atom k 的旋轉降低和提升運算子（根據 |↓#=|g#、|↑#=|r#，它們是 S_=|g# #r|、$S_+=(S_-)^{\dagger}$=|r##g|）
  - ${}_{global}\Delta(t)$ 是時間相依的全域調整
  - n_k 是 atom k 之 Rydberg 狀態的投影運算子（即 n=|r##r|）
- $H_{local\ detuning,k}(t)=-{}_{local}\Delta(t) h_k n_k$
  - ${}_{local}\Delta(t)$ 是本機頻率轉移的時間相依因素，單位為 (rad/s)
  - h_k 是網站相依因素，介於 0.0 和 1.0 之間的無維度數字
- $V_{vdw,k,l}=C_6/(d_{k,l})^6 n_k n_l$，
  - ${}_6$Cis van der Waals 係數，單位為 (rad / s) * (m)^6

- $d_{k,l}$ 是原子 k 和 l 之間的歐幾里得距離，以公尺為單位。

使用者可以透過 Braket AHS 程式結構描述控制下列參數。

- 2-d 原子排列 （每個 k 的 x_k和$_k$座標，以 um 為單位），以 k、l=1、2、...N 控制成對原子距離 $d_{k,l}$
- 以 (rad / s) 為單位的時間相依性、全域 Rabi 頻率
- φ(t)，時間相依的全域階段，單位為 (rad)
- $_{global}$Δ(t)，時間相依性、全域調整，單位為 (rad/s)
- $_{local}$Δ(t)，局部偏轉幅度的時間相依性 （全域） 因素，單位為 (rad / s)
- h_k，局部偏轉幅度的 （靜態） 網站相依因素，介於 0.0 和 1.0 之間的無維度數字

> **Note**
>
> 使用者無法控制涉及的關卡 （即 S_-、S_+、n 運算子是固定的） 或 Rydberg-Rydberg 互動係數 (C) 的強度$_6$。

Braket AHS 程式結構描述

braket.ir.ahs.program_v1.Program 物件 （範例）

> **Note**
>
> 如果您的帳戶未啟用本機調整功能，`localDetuning=[]`請在下列範例中使用 。

```
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（範例）

> **Note**
>
> 如果您的帳戶未啟用本機調整功能，`"localDetuning": []`請在下列範例中使用。

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

主要欄位

程式欄位	type	description
setup.ahs_register.sites	List【List【Decimal】】	鑷子捕捉原子的 2-d 座標清單
setup.ahs_register.filling	List【int】	使用 1 標記佔用陷阱站點的原子，使用 0 標記空站點
hamiltonian.drivingFields【】.amplitude.time_series.times	List【Decimal】	驅動振幅的時間點，Omega(t)
hamiltonian.drivingFields【】.amplitude.time_series.values	List【Decimal】	驅動振幅的值，Omega(t)

程式欄位	type	description
hamiltonian.drivingFields【】.amplitude.pattern	str	驅動振幅的空間模式，Omega(t)；必須是 'uniform'
hamiltonian.drivingFields【】.phase.time_series.times	List【Decimal】	駕駛階段的時間點，phi(t)
hamiltonian.drivingFields【】.phase.time_series.values	List【Decimal】	駕駛階段的值，phi(t)
hamiltonian.drivingFields【】.phase.pattern	str	駕駛階段的空間模式，phi(t)；必須是 'uniform'
hamiltonian.drivingFields【】.detuning.time_series.times	List【Decimal】	驅動調校的時間點，Delta_global(t)
hamiltonian.drivingFields【】.detuning.time_series.values	List【Decimal】	驅動調校的值，Delta_global(t)
hamiltonian.drivingFields【】.detuning.pattern	str	驅動調校的空間模式，Delta_global(t)；必須是 'uniform'
hamiltonian.localDetuning【】.magnitude.time_series.times	List【Decimal】	Delta_local(t) 本機調整幅度的時間相依因素時間點
hamiltonian.localDetuning【】.magnitude.time_series.values	List【Decimal】	Delta_local(t) 本機調整幅度的時間相依因素值

程式欄位	type	description
hamiltonian.localDetuning【】.magnitude.pattern	List【Decimal】	本機調整大小的站點相依因素，h_k（值對應於 setup.ahs_register.sites 中的站點）

中繼資料欄位

程式欄位	type	description
braketSchemaHeader.name	str	結構描述的名稱；必須是 'braket.ir.ahs.program'
braketSchemaHeader.version	str	結構描述的版本

Braket AHS 任務結果結構描述

braket.tasks.analog_hamiltonian_simulation_quantum_task_result.AnalogHamiltonianSimulationQuantumTas
（範例）

```
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（範例）

```
{
    "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
        }
    }
}
```

主要欄位

任務結果欄位	type	description
measurements【】.shotResult.preSequence	List【int】	每個鏡頭的序列前測量位元（每個原子站點一個）：如果站點為空，則 0；如果站點已填滿，則 1；在執行量子演進的脈衝序列之前測量
measurements【】.shotResult.postSequence	List【int】	每個鏡頭的序列後測量位元：如果原子處於 Rydberg 狀態或站點為空，則為 0；如果原子處於接地狀態，則為 1，在執行量子演進的脈衝序列結束時測量

中繼資料欄位

任務結果欄位	type	description
braketSchemaHeader.name	str	結構描述的名稱；必須是 'braket.task_result.analog_hamiltonian_simulation_task_result'
braketSchemaHeader.version	str	結構描述的版本
taskMetadata.braketSchemaHeader.name	str	結構描述的名稱；必須是 ‘braket.task_result.task_metadata'
taskMetadata.braketSchemaHeader.version	str	結構描述的版本
taskMetadata.id	str	量子任務的 ID。對於 AWS 量子任務，這是量子任務 ARN。
taskMetadata.shots	int	量子任務的鏡頭數量
taskMetadata.shots.deviceId	str	量子任務執行所在的裝置 ID。對於 AWS

任務結果欄位	type	description
		裝置，這是裝置 ARN。
taskMetadata.shots.createdAt	str	建立的時間戳記；格式必須是 ISO-8601/RFC3339 字串格式 YYYY-MM-DDTHH：mm：ss.sssZ。預設為無。
taskMetadata.shots.endedAt	str	量子任務結束時的時間戳記；格式必須是 ISO-8601/RFC3339 字串格式 YYYY-MM-DDTHH：mm：ss.sssZ。預設為無。
taskMetadata.shots.status	str	量子任務的狀態 (CREATED、QUEUED、RUNNING、COMPLETED、FAILED)。預設為無。
taskMetadata.shots.failureReason	str	量子任務的失敗原因。預設為無。

任務結果欄位	type	description
additionalMetadata.action	braket.ir.ahs.program_v1.Program	（請參閱 Braket AHS 程式結構描述一節）
additionalMetadata.action.braketSchemaHeader.queraMetadata.name	str	結構描述的名稱；必須是 'braket.task_result.quera_metadata'
additionalMetadata.action.braketSchemaHeader.queraMetadata.version	str	結構描述的版本
additionalMetadata.action.numSuccessfulShots	int	完全成功的鏡頭數量；必須等於請求的鏡頭數量
measurements【】.shotMetadata.shotStatus	int	鏡頭狀態（成功、部分成功、失敗）；必須為「成功」

QuEra 裝置屬性結構描述

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

```
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（範例）

```
{
    "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
        ...
    }
}
```

服務屬性欄位

服務屬性欄位	type	description
service.executionWindows【】.executionDay	ExecutionDay	執行時段的天數；必須是「每天」、「工作日」、「週末」、「星期一」、「星期二」、「星期三」、星期四」、「星期五」、「星期六」或「星期日」
service.executionWindows【】.windowStartHour	datetime.time	執行時段開始時的 UTC 24 小時格式
service.executionWindows【】.windowEndHour	datetime.time	執行時段結束時的 UTC 24 小時格式
service.qpu_capabilities.service.shotsRange	Tuple【int，int】	裝置的鏡頭數量下限和上限
service.qpu_capabilities.service.deviceCost.price	float	以美元為單位的裝置價格
service.qpu_capabilities.service.deviceCost.unit	str	收取價格的單位，例如：'minute'、'hour'、'shot'、'task'

中繼資料欄位

中繼資料欄位	type	description
action【】.version	str	AHS 程式結構描述的版本
action【】.actionType	ActionType	AHS 程式結構描述名稱；必須是 'braket.ir.ahs.program'
service.braketSchemaHeader.name	str	結構描述的名稱；必須是 'braket.device_schema.device_service_properties'
service.braketSchemaHeader.version	str	結構描述的版本
service.deviceDocumentation.imageUrl	str	裝置映像的 URL
service.deviceDocumentation.summary	str	裝置上的簡短描述
service.deviceDocumentation.externalDocumentationUrl	str	外部文件 URL
service.deviceLocation	str	裝置所在的地理位置
service.updatedAt	datetime	上次更新裝置屬性的時間

使用 AWS Boto3

Boto3 是適用於 Python 的 AWS SDK。透過 Boto3，Python 開發人員可以建立、設定和管理 AWS 服務，例如 Amazon Braket。Boto3 提供物件導向的 API，以及對 Amazon Braket 的低階存取。

遵循 Boto3 Quickstart 指南中的指示，了解如何安裝和設定 Boto3。

Boto3 提供與 Amazon Braket Python SDK 搭配使用的核心功能，協助您設定和執行量子任務。Python 客戶一律需要安裝 Boto3，因為這是核心實作。如果您想要使用其他協助程式方法，您也需要安裝 Amazon Braket SDK。

例如，當您呼叫 時`CreateQuantumTask`，Amazon Braket SDK 會將請求提交至 Boto3，然後呼叫 AWS API。

在本節中：

- 開啟 Amazon Braket Boto3 用戶端
- 設定 Boto3 和 Braket SDK 的 AWS CLI 設定檔

開啟 Amazon Braket Boto3 用戶端

若要搭配 Amazon Braket 使用 Boto3，您必須匯入 Boto3，然後定義用來連線至 Amazon Braket 的用戶端API。在下列範例中，Boto3 用戶端名為 `braket`。

```
import boto3
import botocore

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

> **Note**
>
> Braket 支援 IPv6。如果您使用IPv6-only的網路，或希望確保工作負載使用 IPv6 流量，請使用雙堆疊端點，如雙堆疊和 FIPS 端點指南中所述。

現在您已建立`braket`用戶端，您可以從 Amazon Braket 服務提出請求和處理回應。您可以在 API 參考中取得請求和回應資料的詳細資訊。

下列範例示範如何使用 裝置和量子任務。

- 搜尋裝置
- 擷取裝置
- 建立量子任務
- 擷取量子任務
- 搜尋量子任務
- 取消量子任務

搜尋裝置

- search_devices(**kwargs)

使用指定的篩選條件搜尋裝置。

```
# 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'])
```

擷取裝置

- get_device(deviceArn)

擷取 Amazon Braket 中可用的裝置。

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

建立量子任務

- create_quantum_task(**kwargs)

建立量子任務。

```
# 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")
```

擷取量子任務

- `get_quantum_task(quantumTaskArn)`

擷取指定的量子任務。

```
# 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'])
```

搜尋量子任務

- `search_quantum_tasks(**kwargs)`

搜尋符合指定篩選條件值的量子任務。

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

取消量子任務

- `cancel_quantum_task(quantumTaskArn)`

取消指定的量子任務。

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

設定 Boto3 和 Braket SDK 的 AWS CLI 設定檔

除非您另有明確指定，否則 Amazon Braket SDK 依賴預設 AWS CLI 登入資料。建議您在受管 Amazon Braket 筆記本上執行 時保持預設值，因為您必須提供具有啟動筆記本執行個體許可的 IAM 角色。

或者，如果您在本機（例如 Amazon EC2 執行個體）執行程式碼，您可以建立具名 AWS CLI 設定檔。您可以為每個設定檔提供不同的許可集，而不是定期覆寫預設設定檔。

本節提供如何設定此類 CLI `profile`以及如何將該設定檔併入 Amazon Braket 的簡短說明，以便使用來自該設定檔的許可進行API呼叫。

在本節中：

- 步驟 1：設定本機 AWS CLI profile
- 步驟 2：建立 Boto3 工作階段物件
- 步驟 3：將 Boto3 工作階段併入 Braket AwsSession

步驟 1：設定本機 AWS CLI **`profile`**

超出本文件的範圍，說明如何建立使用者，以及如何設定非預設設定檔。如需這些主題的資訊，請參閱：

- 入門
- 設定 AWS CLI 以使用 AWS IAM Identity Center

若要使用 Amazon Braket，您必須向此使用者和相關聯的 CLI `profile`提供必要的 Braket 許可。例如，您可以連接 AmazonBraketFullAccess 政策。

步驟 2：建立 Boto3 工作階段物件

若要建立 Boto3 工作階段物件，請利用下列程式碼範例。

```
from boto3 import Session

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

> **Note**
>
> 如果預期的API呼叫具有與`profile`預設區域不相符的區域型限制，您可以指定 Boto3 工作階段的區域，如下列範例所示。

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

對於指定為 的引數`region`，請取代對應至可使用 AWS 區域 Amazon Braket 的其中一個 的值`us-west-1`，例如 `us-east-1`、 等。

步驟 3：將 Boto3 工作階段併入 Braket AwsSession

下列範例示範如何初始化 Boto3 Braket 工作階段，並執行個體化該工作階段中的裝置。

```
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)
```

此設定完成後，您可以將量子任務提交到該執行個體化AwsDevice物件（例如呼叫 device.run(…) 命令）。該裝置進行的所有API呼叫都可以使用與您先前指定為 的 CLI 設定檔相關聯的 IAM 登入資料profile。

使用 Amazon Braket 測試您的量子任務

Amazon Braket 提供各種高效能量子電路模擬器，協助您在實際量子硬體上執行量子演算法之前進行測試和驗證。這些模擬器處理複雜的基礎軟體和基礎設施，以及 Amazon Elastic Compute Cloud (Amazon EC2) 叢集，以減輕在傳統高效能運算 (HPC) 基礎設施上模擬量子電路的負擔。這些資源可讓您專注於開發和最佳化您的量子應用程式。

使用 Braket 的模擬器，您可以徹底測試量子電路和演算法，而不受實體量子裝置的限制和約束。這可讓您探索各種量子運算概念，從基本量子閘道和電路到更進階的量子演算法和錯誤緩解技術。

Braket SDK 可簡化將量子任務提交到模擬器的過程，讓您控制模擬參數，例如鏡頭數量和雜訊模型，以更了解量子演算法的行為。您也可以使用 Amazon Braket 混合任務功能來結合傳統和量子運算元素，進一步擴展測試和驗證的範圍。

透過在 Braket 模擬器上徹底測試您的量子任務，您可以獲得寶貴的洞見、精簡您的演算法，並確保其正確性，然後再部署在實際量子硬體上。這有助於縮短開發時間、將錯誤降至最低，並最終加速量子運算領域的進度。

在本節中：

將量子任務提交至模擬器

Amazon Braket 可讓您存取多個可測試量子任務的模擬器。您可以個別提交量子任務，也可以設定量子任務批次。

模擬器

- 密度矩陣模擬器，DM1： `arn:aws:braket:::device/quantum-simulator/amazon/dm1`
- 狀態向量模擬器，SV1： `arn:aws:braket:::device/quantum-simulator/amazon/sv1`
- Tensor 網路模擬器，TN1： `arn:aws:braket:::device/quantum-simulator/amazon/tn1`
- 本機模擬器 ： `LocalSimulator()`

> **Note**
>
> 您可以取消 QPUs 和隨需模擬器CREATED處於 狀態的量子任務。您可以針對隨需模擬器和 QPUs，盡力取消QUEUED處於 狀態的量子任務。請注意，QPU QUEUED 量子任務不太可能在 QPU 可用性時段內成功取消。

在本節中：

本機狀態向量模擬器 (`braket_sv`)

本機狀態向量模擬器 (`braket_sv`) 是 Amazon Braket SDK 的一部分，可在您的環境中於本機執行。它非常適合在小型電路 （ 最多 25 個qubits) 上進行快速原型設計，具體取決於您的 Braket 筆記本執行個體或本機環境的硬體規格。

本機模擬器支援 Amazon Braket SDK 中的所有閘道，但 QPU 裝置支援較小的子集。您可以在裝置屬性中找到裝置支援的閘道。

> **Note**
>
> 本機模擬器支援進階 OpenQASM 功能，這些功能可能不支援 QPU 裝置或其他模擬器。如需支援功能的詳細資訊，請參閱 OpenQASM Local Simulator 筆記本中提供的範例。

如需如何使用模擬器的詳細資訊，請參閱 Amazon Braket 範例。

本機密度矩陣模擬器 (**braket_dm**)

本機密度矩陣模擬器 (braket_dm) 是 Amazon Braket SDK 的一部分，可在您環境中於本機執行。它非常適合在具有雜訊（最多 12 個qubits) 的小電路上快速原型設計，具體取決於您的 Braket 筆記本執行個體或本機環境的硬體規格。

您可以使用位元翻轉和去極化錯誤等閘道雜訊操作，從頭開始建置常見的雜訊電路。您也可以將雜訊操作套用至現有電路的特定 qubits和 閘道，這些電路旨在同時執行，無論是否有雜訊。

根據指定的 數量，braket_dm本機模擬器可以提供下列結果shots：

- 密度降低矩陣：Shots= 0

> **Note**
>
> 本機模擬器支援進階 OpenQASM 功能，這些功能可能不支援 QPU 裝置或其他模擬器。如需支援功能的詳細資訊，請參閱 OpenQASM Local Simulator 筆記本中提供的範例。

若要進一步了解本機密度矩陣模擬器，請參閱 Braket 雜訊模擬器簡介範例。

本機 AHS 模擬器 (**braket_ahs**)

本機 AHS (Analog Hamiltonian Simulation) 模擬器 (braket_ahs) 是 Amazon Braket SDK 的一部分，可在您的環境中於本機執行。它可用於模擬 AHS 程式的結果。它非常適合在小型註冊（最多 10-12 個原子）上進行原型設計，具體取決於您的 Braket 筆記本執行個體或本機環境的硬體規格。

本機模擬器支援具有一個統一駕駛欄位、一個（非統一）轉移欄位和任意原子排列的 AHS 程式。如需詳細資訊，請參閱 Braket AHS 類別和 Braket AHS 程式結構描述。

若要進一步了解本機 AHS 模擬器，請參閱 Hello AHS：執行您的第一個類比 Hamiltonian 模擬頁面和類比 Hamiltonian 模擬範例筆記本。

狀態向量模擬器 (SV1)

SV1 是一種隨需、高效能、通用狀態向量模擬器。它可以模擬最多 34 個 的電路qubits。視使用的閘道類型和其他因素而定，您可以預期 34-qubit、密集和方形電路（電路深度 = 34) 大約需要 1-2 小時才能完成。具有all-to-all閘道的電路非常適合 SV1。它以完整狀態向量或振幅陣列等形式傳回結果。

SV1 的執行時間上限為 6 小時。其預設有 35 個並行量子任務，以及最多 100 個（在 us-west-1 和 eu-west-2 中為 50 個）並行量子任務。

SV1 結果

SV1 根據指定的 數量， 可以提供下列結果shots：

- 範例：Shots> 0
- 預期：Shots>= 0
- 變異數：Shots>= 0
- 機率：Shots> 0
- 振幅：Shots= 0
- 聯合漸層：Shots= 0

如需結果的詳細資訊，請參閱結果類型。

SV1 永遠可用，它會隨需執行您的電路，而且可以平行執行多個電路。執行時間會隨操作數目線性擴展，並以 數目呈指數擴展qubits。的 數目對執行時間的影響shots很小。若要進一步了解，請造訪比較模擬器。

模擬器支援 Braket SDK 中的所有閘道，但 QPU 裝置支援較小的子集。您可以在裝置屬性中找到裝置支援的閘道。

密度矩陣模擬器 (DM1)

DM1 是一種隨需、高效能、密度矩陣模擬器。它可以模擬最多 17 個 的電路qubits。

DM1 的執行時間上限為 6 小時，預設為 35 個並行量子任務，以及最多 50 個並行量子任務。

DM1 結果

DM1 根據指定的 數量， 可以提供下列結果shots：

- 範例：Shots> 0
- 預期：Shots>= 0
- 變異數：Shots>= 0
- 機率：Shots> 0
- 降低密度矩陣：Shots= 0，最多 8 個 qubits

如需結果的詳細資訊，請參閱結果類型。

DM1 永遠可用，它會隨需執行您的電路，而且可以平行執行多個電路。執行時間會隨操作數目線性擴展，並以 數目呈指數擴展qubits。的 數目對執行時間的影響shots很小。若要進一步了解，請參閱比較模擬器。

雜訊閘道和限制

```
AmplitudeDamping
    Probability has to be within [0,1]
BitFlip
    Probability has to be within [0,0.5]
Depolarizing
    Probability has to be within [0,0.75]
GeneralizedAmplitudeDamping
    Probability has to be within [0,1]
PauliChannel
    The sum of the probabilities has to be within [0,1]
Kraus
    At most 2 qubits
    At most 4 (16) Kraus matrices for 1 (2) qubit
PhaseDamping
    Probability has to be within [0,1]
PhaseFlip
    Probability has to be within [0,0.5]
TwoQubitDephasing
    Probability has to be within [0,0.75]
TwoQubitDepolarizing
    Probability has to be within [0,0.9375]
```

Tensor 網路模擬器 (TN1)

TN1 是一種隨需、高效能的張量網路模擬器。 TN1可以模擬高達 50 個qubits且電路深度為 1，000 或更小的特定電路類型。 TN1特別適用於稀疏電路、具有本機閘道的電路，以及具有特殊結構的其他電路，例如量子傅立葉轉換 (QFT) 電路。 會以兩個階段TN1運作。首先，演練階段會嘗試識別電路的有效運算路徑，因此 TN1可以預估下一個階段的執行時間，稱為收縮階段。如果預估收縮時間超過TN1模擬執行時間限制， TN1 不會嘗試收縮。

TN1 的執行時間限制為 6 小時。其限制為最多 10 個 (eu-west-2 中的 5 個） 並行量子任務。

TN1 結果

收縮階段由一系列矩陣乘法組成。乘法序列會持續到達到結果，或直到判定無法達到結果為止。

注意： Shots 必須 > 0。

結果類型包括：

- 樣本
- 期望
- 變異數

如需結果的詳細資訊，請參閱結果類型。

TN1 永遠可用，它會隨需執行您的電路，而且可以平行執行多個電路。若要進一步了解，請參閱比較模擬器。

模擬器支援 Braket SDK 中的所有閘道，但 QPU 裝置支援較小的子集。您可以在裝置屬性中找到裝置支援的閘道。

請造訪 Amazon Braket GitHub 儲存庫以取得 TN1 範例筆記本，以協助您開始使用 TN1。

使用 的最佳實務 TN1

- 避免all-to-all電路。
- 使用少量 測試新電路或線路類別shots，以了解 的圓周「硬度」TN1。
- 在多個量子任務上分割大型shot模擬。

關於內嵌模擬器

內嵌模擬器的運作方式是讓模擬直接內嵌在演算法程式碼中。此外，它包含在相同的容器中，並直接在混合任務執行個體上執行模擬。此方法有助於消除通常與模擬和遠端裝置之間通訊相關的瓶頸。透過將所有運算保持在單一的凝聚性環境中，嵌入式模擬器可以大幅減少記憶體需求，並減少實現目標結果所需的電路執行次數。相較於依賴遠端模擬的傳統設定，這可能會導致顯著的效能改善，通常為十倍或更多。如需有關內嵌模擬器如何增強效能並啟用簡化混合任務的詳細資訊，請參閱使用 Amazon Braket 混合任務執行混合任務文件頁面。

PennyLane 的閃電模擬器

您可以使用 PennyLane 的閃電模擬器作為 Braket 上的內嵌模擬器。使用 PennyLane 的閃電模擬器，您可以使用進階梯度運算方法，例如並行差異化，以更快的速度評估梯度。Lightning.qubit 模擬器可透過 Braket NBIs做為裝置使用，並做為內嵌模擬器使用，而 Lightning.gpu 模擬器則需要做為具有 GPU

執行個體的內嵌模擬器執行。如需使用 Lightning.gpu 的範例，請參閱 Braket Hybrid Jobs 筆記本中的內嵌模擬器。

比較 Amazon Braket 模擬器

本節說明一些概念、限制和使用案例，協助您選取最適合量子任務的 Amazon Braket 模擬器。

在本機模擬器和隨需模擬器之間進行選擇 (SV1、TN1、DM1)

本機模擬器的效能取決於託管本機環境的硬體，例如用於執行模擬器的 Braket 筆記本執行個體。隨需模擬器在 AWS 雲端中執行，旨在擴展到超越典型的本機環境。隨需模擬器已針對較大的電路進行最佳化，但每個量子任務或量子任務批次會增加一些延遲額外負荷。如果涉及許多量子任務，這可能表示權衡。鑑於這些一般效能特性，下列指引可協助您選擇如何執行模擬，包括有雜訊的模擬。

對於模擬：

- 使用少於 18 個 時qubits，請使用本機模擬器。
- 使用 18–24 時qubits，請根據工作負載選擇模擬器。
- 使用超過 24 個 時qubits，請使用隨需模擬器。

對於雜訊模擬：

- 使用少於 9 個 時qubits，請使用本機模擬器。
- 使用 9–12 時qubits，請根據工作負載選擇模擬器。
- 使用超過 12 個 時qubits，請使用 DM1。

什麼是狀態向量模擬器？

SV1 是通用狀態向量模擬器。它會儲存量子狀態的完整波動函數，並依序將閘道操作套用至狀態。它儲存了所有可能性，即使是非常不可能的可能性。量子任務的SV1模擬器執行時間會隨著電路中的閘道數量線性增加。

什麼是密度矩陣模擬器？

DM1 模擬具有雜訊的量子電路。它會存放系統的完整密度矩陣，並依序套用電路的閘道和雜訊操作。最終密度矩陣包含迴路執行後量子狀態的完整資訊。執行時間通常會隨操作數目線性擴展，並以 數目呈指數擴展qubits。

什麼是張量網路模擬器？

TN1 將量子電路編碼為結構化圖形。

- 圖形的節點由量子閘道或 組成qubits。
- 圖形的邊緣代表閘道之間的連線。

由於此結構， TN1可以找到相對大型和複雜量子電路的模擬解決方案。

TN1 需要兩個階段

一般而言， 會以兩階段方法來TN1模擬量子運算。

- 演練階段：在此階段中， TN1 會想出一種以有效率的方式周遊圖形的方法，其中包括造訪每個節點，以便您取得所需的測量。身為客戶，您看不到此階段，因為 會為您同時TN1執行兩個階段。它會完成第一個階段，並根據實際限制，決定是否自行執行第二個階段。模擬開始後，您就沒有對該決策的輸入。
- 收縮階段：此階段類似於傳統電腦中運算的執行階段。階段由一系列矩陣乘法組成。這些乘法的順序對運算的難度有很大的影響。因此，演練階段會先完成，以找出圖形中最有效的運算路徑。在演練階段找到收縮路徑後， 會與您電路的閘道一起TN1收縮，以產生模擬的結果。

TN1 圖形類似於映射

簡言之，您可以將基礎TN1圖形與城市的街道進行比較。在具有計劃網格的城市中，使用地圖輕鬆找到目的地的路由。在具有意外街道、重複街道名稱等的城市中，查看地圖可能很難找到通往目的地的路線。

如果 TN1 未執行演練階段，就好像在城市的街道四處走動來尋找目的地，而不是先看地圖。在行走時間上，花更多時間看地圖，可以真正獲得回報。同樣地，演練階段提供寶貴的資訊。

您可能會說 對其周遊的基礎電路結構TN1有一定的「感知」。它會在演練階段期間獲得此意識。

最適合每種模擬器的問題類型

SV1 非常適合任何主要依賴具有特定數量 qubits和 閘道的問題類別。一般而言，所需的時間會隨著閘道數量線性增長，而它不取決於 的數量shots。 SV1 通常比 TN1 28 以下的電路更快qubits。

SV1 對於較高的qubit數字，速度可能會變慢，因為它實際上模擬了所有可能性，即使非常不可能也一樣。它無法判斷哪些結果可能。因此，對於30-qubit評估， SV1 必須計算 2^30 個組態。由於記憶體和儲存限制qubits，AmazonRaket SV1 模擬器的限制為 34。您可以像這樣想：每次將 新增至 qubit 時 SV1，問題會變得兩倍硬。

對於許多類別的問題， TN1可以在實際時間評估比 更大的電路SV1，因為 TN1 會利用圖形的結構。它基本上會從一開始就追蹤解決方案的演變，而且只會保留有助於有效周遊的組態。換句話說，它會儲存組態，以建立矩陣乘法的排序，進而產生更簡單的評估程序。

對於 TN1， qubits和 閘道的數量很重要，但圖形的結構更重要。例如， TN1 非常善於評估閘道為短距離的電路 （ 圖形 ） （ 亦即，每個閘道僅qubit連接到其最近的鄰近 qubits)，以及連線 （ 或閘道 ） 具有類似範圍的電路 （ 圖形 ）。的典型範圍TN1是只與 5 qubits 個qubits以外的其他 進行每個qubit通話。如果大多數結構可以分解為更簡單的關係，例如這些關係，可以用更多、更小或更統一的矩陣表示，則會有效率地TN1執行評估。

的限制 TN1

TN1 速度可能比SV1圖形的結構複雜度慢。對於某些圖形， 會在演練階段後TN1終止模擬，並顯示狀態為 FAILED，因為下列兩種原因之一：

- 找不到路徑 - 如果圖形太複雜，則很難找到良好的周遊路徑，且模擬器放棄運算。 TN1無法執行收縮。您可能會看到類似以下的錯誤訊息： `No viable contraction path found.`
- 收縮階段太困難 – 在某些圖形中， TN1可以找到周遊路徑，但評估非常長且非常耗時。在這種情況下，收縮非常昂貴，成本會過高，而是在演練階段之後TN1退出。您可能會看到類似以下的錯誤訊息： `Predicted runtime based on best contraction path found exceeds TN1 limit.`

> **Note**
>
> 即使未執行收縮且您看到 FAILED 狀態，TN1仍會向您收取 的演練階段費用。

預測的執行時間也取決於shot計數。在最壞的情況下，TN1收縮時間會線性取決於shot計數。電路可能可以與較少的 簽訂合約shots。例如，您可以提交具有 100 個 的量子任務shots，該任務TN1判定為不可收縮，但如果您只以 10 個重新提交，則收縮會繼續進行。在這種情況下，若要取得 100 個範例，您可以shots為相同電路提交 10 個量子任務 10 個量子任務，並在結尾合併結果。

最佳實務是，建議您一律使用幾個 shots (例如 10) 測試您的電路或電路類別，以了解您的電路對 的硬度TN1，然後再繼續執行更多數量的 shots。

> **Note**
>
> 形成收縮階段的乘法系列以小型 NxN 矩陣開頭。例如，2-qubit閘道需要 4x4 矩陣。在經判定為太困難的收縮期間所需的中繼矩陣是巨大的。這種運算需要數天才能完成。因此 Amazon Braket 不會嘗試非常複雜的收縮。

並行數量

所有 Braket 模擬器都可讓您同時執行多個電路。並行限制因模擬器和區域而異。如需並行限制的詳細資訊，請參閱配額頁面。

Amazon Braket 上的量子任務範例

本節逐步解說執行範例量子任務的階段，從選取裝置到檢視結果。作為 Amazon Braket 的最佳實務，建議您先在模擬器上執行電路，例如 SV1。

在本節中：

- 指定裝置
- 提交範例量子任務
- 提交參數化任務
- 指定 shots
- 輪詢結果
- 檢視範例結果

指定裝置

首先，選取並指定量子任務的裝置。此範例說明如何選擇模擬器 SV1。

```
# choose the on-demand simulator to run the circuit
from braket.aws import AwsDevice
device = AwsDevice("arn:aws:braket:::device/quantum-simulator/amazon/sv1")
```

您可以檢視此裝置的一些屬性，如下所示：

```
print (device.name)
for iter in device.properties.action['braket.ir.jaqcd.program']:
    print(iter)
```

```
SV1
('version', ['1.0', '1.1'])
('actionType', <DeviceActionType.JAQCD: 'braket.ir.jaqcd.program'>)
('supportedOperations', ['ccnot', 'cnot', 'cphaseshift', 'cphaseshift00',
 'cphaseshift01', 'cphaseshift10', 'cswap', 'cy', 'cz', 'h', 'i', 'iswap', 'pswap',
 'phaseshift', 'rx', 'ry', 'rz', 's', 'si', 'swap', 't', 'ti', 'unitary', 'v', 'vi',
 'x', 'xx', 'xy', 'y', 'yy', 'z', 'zz'])
('supportedResultTypes', [ResultType(name='Sample', observables=['x', 'y', 'z', 'h',
 'i', 'hermitian'], minShots=1, maxShots=100000), ResultType(name='Expectation',
 observables=['x', 'y', 'z', 'h', 'i', 'hermitian'], minShots=0, maxShots=100000),
 ResultType(name='Variance', observables=['x', 'y', 'z', 'h', 'i', 'hermitian'],
 minShots=0, maxShots=100000), ResultType(name='Probability', observables=None,
 minShots=1, maxShots=100000), ResultType(name='Amplitude', observables=None,
 minShots=0, maxShots=0)])
```

提交範例量子任務

提交範例量子任務以在隨需模擬器上執行。

```
# create a circuit with a result type
circ = Circuit().rx(0, 1).ry(1, 0.2).cnot(0,2).variance(observable=Observable.Z(),
 target=0)
# add another result type
circ.probability(target=[0, 2])

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

# submit the quantum task to run
my_task = device.run(circ, s3_location, shots=1000, poll_timeout_seconds = 100,
 poll_interval_seconds = 10)
# the positional argument for the S3 bucket is optional if you want to specify a bucket
 other than the default

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

`device.run()` 命令會透過 `CreateQuantumTask` API 建立量子任務。在短暫初始化時間之後，量子任務會排入佇列，直到容量存在以在裝置上執行量子任務為止。在此情況下，裝置為 SV1。裝置完

成運算後，AmazonRaket 會將結果寫入通話中指定的 Amazon S3 位置。除了本機模擬器之外，所有裝置`s3_location`都需要位置引數。

> **Note**
>
> Braket 量子任務動作的大小限制為 3MB。

提交參數化任務

Amazon Braket 隨需和本機模擬器和 QPUs 也支援在提交任務時指定免費參數的值。您可以使用`inputs`引數對 執行此操作`device.run()`，如下列範例所示。`inputs` 必須是字串浮點數對的字典，其中索引鍵是參數名稱。

參數編譯可以改善在特定 QPUs 上執行參數電路的效能。將參數電路做為量子任務提交至支援的 QPU 時，Raket 會編譯電路一次，並快取結果。後續參數更新不會重新編譯至相同電路，導致使用相同電路的任務執行時間更快。編譯電路時，Raket 會自動使用來自硬體提供者的更新校正資料，以確保最高品質的結果。

> **Note**
>
> 除了脈衝層級程式Rigetti Computing之外， 的所有超執行、以閘道為基礎的 QPUs都支援參數編譯。

```
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})
```

指定 shots

shots 引數是指所需的測量數量 shots。等模擬器SV1支援兩種模擬模式。

- 對於 shots = 0，模擬器會執行確切的模擬，並傳回所有結果類型的 true 值。（不適用於 TN1。）
- 對於 的非零值shots，來自輸出分佈的模擬器範例會模擬真實 QPUs的shot雜訊。QPU 裝置僅允許 shots > 0。

如需每個量子任務的最大射擊次數資訊，請參閱 Braket Quotas。

輪詢結果

執行 時my_task.result()，軟體開發套件會使用您在量子任務建立時定義的參數開始輪詢結果：

- poll_timeout_seconds 是在隨需模擬器和 或 QPU 裝置上執行量子任務時，輪詢量子任務逾時之前的秒數。預設值為 432，000 秒，即 5 天。
- 注意：對於 Rigetti和 等 QPU 裝置IonQ，建議您允許幾天的時間。如果您的輪詢逾時太短，則可能無法在輪詢時間內傳回結果。例如，當 QPU 無法使用時，會傳回本機逾時錯誤。
- poll_interval_seconds 是輪詢量子任務的頻率。它會指定您在隨需模擬器和 QPU 裝置上執行量子任務時呼叫 Braket API以取得狀態的頻率。預設值為 1 秒。

此非同步執行有助於與非一律可用的 QPU 裝置互動。例如，裝置在一般維護時段期間可能無法使用。

傳回的結果包含與量子任務相關聯的一系列中繼資料。您可以使用下列命令來檢查測量結果：

```
print('Measurement results:\n',result.measurements)
print('Counts for collapsed states:\n',result.measurement_counts)
print('Probabilities for collapsed states:\n',result.measurement_probabilities)
```

```
Measurement results:
 [[1 0 1]
 [0 0 0]
 [1 0 1]
 ...
 [0 0 0]
 [0 0 0]
 [0 0 0]]
Counts for collapsed states:
 Counter({'000': 761, '101': 226, '010': 10, '111': 3})
```

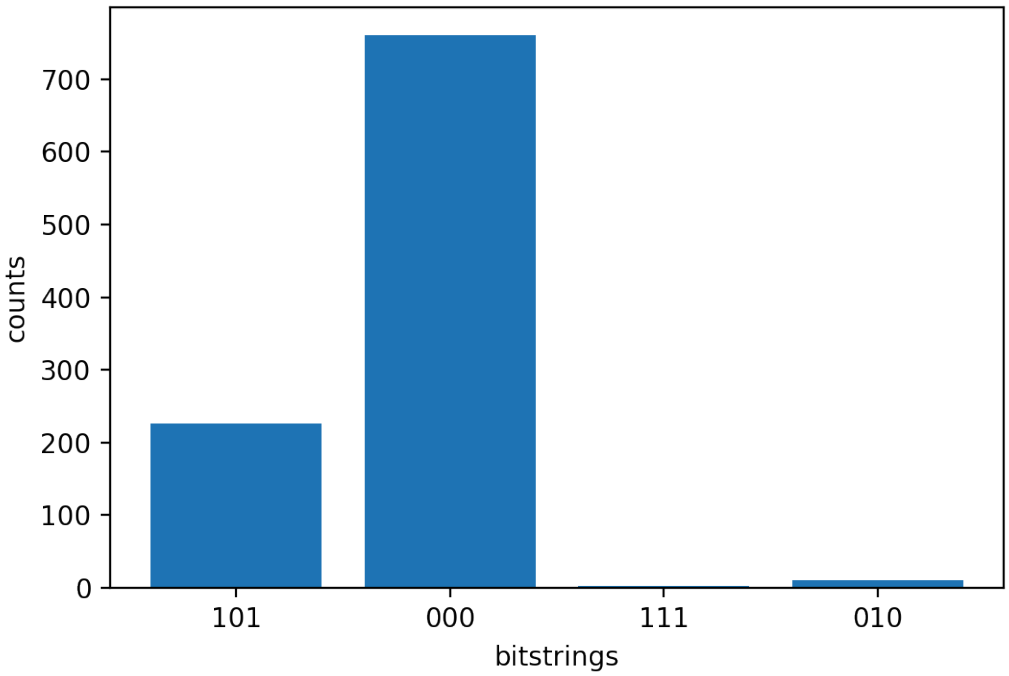
```
Probabilities for collapsed states:
 {'101': 0.226, '000': 0.761, '111': 0.003, '010': 0.01}
```

檢視範例結果

由於您已指定 ResultType，因此您可以檢視傳回的結果。結果類型會以新增至電路的順序顯示。

```
print('Result types include:\n', result.result_types)
print('Variance=',result.values[0])
print('Probability=',result.values[1])

# you can plot the result and do some analysis
import matplotlib.pyplot as plt
plt.bar(result.measurement_counts.keys(), result.measurement_counts.values());
plt.xlabel('bitstrings');
plt.ylabel('counts');
```

```
Result types include:
 [ResultTypeValue(type={'observable': ['z'], 'targets': [0], 'type': 'variance'},
 value=0.7062359999999999), ResultTypeValue(type={'targets': [0, 2], 'type':
 'probability'}, value=array([0.771, 0.   , 0.   , 0.229]))]
Variance= 0.7062359999999999
Probability= [0.771 0.    0.    0.229]
```



使用本機模擬器測試量子任務

您可以直接將量子任務傳送到本機模擬器，以進行快速原型設計和測試。此模擬器會在您的本機環境中執行，因此您不需要指定 Amazon S3 位置。結果會直接在您的工作階段中計算。若要在本機模擬器上執行量子任務，您只能指定 shots 參數。

> **Note**
>
> qubits 本機模擬器可以處理的執行速度和最大數量取決於 Amazon Braket 筆記本執行個體類型或本機硬體規格。

下列命令全部相同，並執行個體化狀態向量 （無雜訊） 本機模擬器。

```
# 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")
```

然後使用下列項目執行量子任務。

```
my_task = device.run(circ, shots=1000)
```

若要執行個體化本機密度矩陣（雜訊）模擬器，客戶會變更後端，如下所示。

```
# import the LocalSimulator module
from braket.devices import LocalSimulator
device = LocalSimulator(backend="braket_dm")
```

在本機模擬器上測量特定 qubit

本機狀態向量模擬器和本機密度矩陣模擬器支援執行中電路，其中可以測量電路的 qubit 子集，通常稱為部分測量。

例如，在以下程式碼中，您可以建立雙 qubit 電路，並只透過將具有目標 qubit 的`measure`指令新增至電路的結尾來測量第一個 qubit。

```
# Import the LocalSimulator module
from braket.devices import LocalSimulator

# Use the local simulator device
device = LocalSimulator()

# Define a bell circuit and only measure
circuit = Circuit().h(0).cnot(0, 1).measure(0)

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

# Get the results
result = task.result()

# Print the measurement counts for qubit 0
print(result.measurement_counts)
```

Quantum 任務批次處理

每個 Amazon Braket 裝置都可以使用 Quantum 任務批次處理，本機模擬器除外。批次處理對於您在隨需模擬器 (TN1 或 SV1) 上執行的量子任務特別有用，因為它們可以平行處理多個量子任務。為了協助您設定各種量子任務，Amazon Braket 提供範例筆記本。

批次處理可讓您平行啟動量子任務。例如，如果您想要進行需要 10 個量子任務的計算，且這些量子任務中的電路彼此獨立，最好使用批次處理。如此一來，您就不必等到某個量子任務完成，再開始另一個任務。

下列範例示範如何執行一批量子任務：

```
circuits = [bell for _ in range(5)]
batch = device.run_batch(circuits, s3_folder, shots=100)
print(batch.results()[0].measurement_counts)  # The result of the first quantum task in
 the batch
```

如需詳細資訊，請參閱 GitHub 或 Quantum 任務批次處理上的 Amazon Braket 範例，其中包含有關批次處理更具體的資訊。 https://github.com/aws/amazon-braket-sdk-python/blob/main/src/braket/aws/aws_quantum_task_batch.py

在本節中：

- 關於量子任務批次處理和成本
- Quantum 任務批次處理和 PennyLane
- 任務批次處理和參數化電路

關於量子任務批次處理和成本

有關量子任務批次處理和帳單成本的一些注意事項：

- 根據預設，量子任務批次會重試所有逾時或失敗量子任務 3 次。
- 一批長時間執行的量子任務，例如 qubits 的 34SV1，可能會產生大量成本。開始一批量子任務之前，請務必仔細仔細檢查`run_batch`指派值。我們不建議使用 TN1搭配 `run_batch`。
- TN1 可能產生失敗演練階段任務的成本（如需詳細資訊，請參閱 TN1 說明)。自動重試可以新增到成本中，因此我們建議在使用 時將批次處理的 'max_retries' 數目設定為 0 TN1（請參閱 Quantum 任務批次處理，第 186 行)。

Quantum 任務批次處理和 PennyLane

當您在 Amazon Braket 上使用 PennyLane 時，請透過在執行個體化 Amazon Braket 裝置`parallel = True`時設定 來利用批次處理，如下列範例所示。

```
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,)
```

如需使用 PennyLane 批次處理的詳細資訊，請參閱量子電路的平行最佳化。

任務批次處理和參數化電路

提交包含參數化電路的量子任務批次時，您可以提供inputs字典，用於批次中的所有量子任務，或輸入字典list的 ，在這種情況下，第 i- 個字典會與第 i- 個任務配對，如下列範例所示。

```
from braket.circuits import Circuit, FreeParameter, Observable
from braket.aws import AwsQuantumTaskBatch

# create the free parameters
alpha = FreeParameter('alpha')
beta = FreeParameter('beta')

# create two circuits
circ_a = Circuit().rx(0, alpha).ry(1, alpha).cnot(0,2).xx(0, 2, beta)
circ_a.variance(observable=Observable.Z(), target=0)

circ_b = Circuit().rx(0, alpha).rz(1, alpha).cnot(0,2).zz(0, 2, beta)
circ_b.expectation(observable=Observable.Z(), target=2)

# use the same inputs for both circuits in one batch

tasks = device.run_batch([circ_a, circ_b], inputs={'alpha': 0.1, 'beta':0.2})

# or provide each task its own set of inputs

inputs_list = [{'alpha': 0.3,'beta':0.1}, {'alpha': 0.1,'beta':0.4}]

tasks = device.run_batch([circ_a, circ_b], inputs=inputs_list)
```

您也可以為單一參數電路準備輸入字典清單，並以量子任務批次的形式提交。如果清單中有 N 個輸入字典，則批次包含 N 個量子任務。第 i- 個量子任務對應至使用第 i- 個輸入字典執行的電路。

```
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)
```

使用 Amazon Braket 混合任務

本節提供有關在 Amazon Braket 中建立和執行混合式任務的基本知識的說明。

您可以使用下列方式存取 Braket 中的混合式任務：

- Amazon Braket Python SDK。
- Amazon Braket 主控台。
- Amazon Braket API。

在本節中：

- 將本機程式碼作為混合任務執行
- 使用 Amazon Braket 混合任務執行混合任務
- 建立您的第一個混合任務
- 儲存您的任務結果
- 使用檢查點儲存和重新啟動混合任務
- 使用本機模式建置和偵錯混合式任務

將本機程式碼作為混合任務執行

Amazon Braket Hybrid Jobs 提供混合量子傳統演算法的全受管協調，結合 Amazon EC2 運算資源與 Amazon Braket Quantum Processing Unit (QPU) 存取。在混合任務中建立的量子任務具有個別量子任務的優先順序佇列，因此您的演算法不會因量子任務佇列中的波動而中斷。每個 QPU 都會維護個別的混合式任務佇列，確保任何指定時間只能執行一個混合式任務。

在本節中：

- 從本機 Python 程式碼建立混合任務
- 安裝其他 Python 套件和原始程式碼
- 將資料儲存並載入混合任務執行個體

- 混合式任務裝飾項目的最佳實務

從本機 Python 程式碼建立混合任務

您可以執行本機 Python 程式碼做為 Amazon Braket 混合任務。您可以透過使用@hybrid_job裝飾項目註釋程式碼來執行此操作，如下列程式碼範例所示。對於自訂環境，您可以選擇使用來自 Amazon Elastic Container Registry (ECR) 的自訂容器。

> **Note**
>
> 預設僅支援 Python 3.10。

您可以使用@hybrid_job裝飾項目來註釋函數。Braket 會將裝飾項目內的程式碼轉換為 Braket 混合任務演算法指令碼。然後，混合任務會在 Amazon EC2 執行個體的裝飾項目內叫用 函數。您可以使用 `job.state()`或 Braket 主控台來監控任務的進度。下列程式碼範例示範如何在 上執行五個狀態的序列State Vector Simulator (SV1) device。

```
from braket.aws import AwsDevice
from braket.circuits import Circuit, FreeParameter, Observable
from braket.devices import Devices
from braket.jobs.hybrid_job import hybrid_job
from braket.jobs.metrics import log_metric

device_arn = Devices.Amazon.SV1


@hybrid_job(device=device_arn)  # choose priority device
def run_hybrid_job(num_tasks=1):
    device = AwsDevice(device_arn)  # declare AwsDevice within the hybrid job

    # create a parametric circuit
    circ = Circuit()
    circ.rx(0, FreeParameter("theta"))
    circ.cnot(0, 1)
    circ.expectation(observable=Observable.X(), target=0)

    theta = 0.0  # initial parameter

    for i in range(num_tasks):
        task = device.run(circ, shots=100, inputs={"theta": theta})  # input parameters
```

```
        exp_val = task.result().values[0]

        theta += exp_val  # modify the parameter (possibly gradient descent)

        log_metric(metric_name="exp_val", value=exp_val, iteration_number=i)

    return {"final_theta": theta, "final_exp_val": exp_val}
```

您可以像一般 Python 函數一樣叫用 函數來建立混合式任務。不過，裝飾項目函數會傳回混合式任務控制代碼，而不是函數的結果。若要在結果完成後擷取結果，請使用 `job.result()`。

```
job = run_hybrid_job(num_tasks=1)
result = job.result()
```

@hybrid_job 裝飾項目中的裝置引數會指定混合任務可優先存取的裝置 - 在此情況下為SV1模擬器。若要取得 QPU 優先順序，您必須確保函數中使用的裝置 ARN 符合裝飾項目中指定的 ARN。為了方便起見，您可以使用 協助程式函數`get_job_device_arn()`來擷取 中宣告的裝置 ARN@hybrid_job。

> **Note**
>
> 每個混合任務至少都有一分鐘的啟動時間，因為它會在 Amazon EC2 上建立容器化環境。因此，對於非常短的工作負載，例如單一電路或一批電路，它可能就足以讓您使用量子任務。

超參數

`run_hybrid_job()` 函數會採用 引數`num_tasks`來控制建立的量子任務數量。混合任務會自動將此擷取為超參數。

> **Note**
>
> 超參數在 Braket 主控台中顯示為字串，限制為 2500 個字元。

指標和記錄

在 `run_hybrid_job()`函數中，來自反覆演算法的指標會以 記錄`log_metrics`。指標會自動繪製在混合任務索引標籤下的 Braket 主控台頁面中。您可以使用 指標，在混合式任務執行期間，使用 Braket

成本追蹤器近乎即時地追蹤量子任務成本。上述範例使用指標名稱「機率」，記錄結果類型的第一個機率。

擷取結果

混合任務完成後，您可以使用 `job.result()` 擷取混合任務結果。傳回陳述式中的任何物件都會由 Braket 自動擷取。請注意，函數傳回的物件必須是元組，且每個元素皆可序列化。例如，以下程式碼顯示運作中和失敗的範例。

```
@hybrid_job(device=Devices.Amazon.SV1)
def passing():
    np_array = np.random.rand(5)
    return np_array  # serializable

@hybrid_job(device=Devices.Amazon.SV1)
def failing():
    return MyObject() # not serializable
```

任務名稱

根據預設，此混合任務的名稱會從函數名稱推斷。您也可以指定長度上限為 50 個字元的自訂名稱。例如，在下列程式碼中，任務名稱為 "my-job-name"。

```
@hybrid_job(device=Devices.Amazon.SV1, job_name="my-job-name")
def function():
    pass
```

本機模式

本機任務是透過將 引數新增至裝飾項目`local=True`來建立。這會在本機運算環境的容器化環境中執行混合作業，例如您的筆記型電腦。本機任務沒有量子任務的優先順序佇列。對於多節點或 MPI 等進階案例，本機任務可以存取必要的 Braket 環境變數。下列程式碼會使用裝置做為 SV1 模擬器來建立本機混合任務。

```
@hybrid_job(device=Devices.Amazon.SV1, local=True)
def run_hybrid_job(num_tasks = 1):
    return ...
```

支援所有其他混合任務選項。如需選項清單，請參閱 braket.jobs.quantum_job_creation 模組。

安裝其他 Python 套件和原始程式碼

您可以自訂執行時間環境，以使用您偏好的 Python 套件。您可以使用requirements.txt檔案、套件名稱清單，或自備容器 (BYOC)。若要使用 requirements.txt 檔案自訂執行期環境，請參閱下列程式碼範例。

```
@hybrid_job(device=Devices.Amazon.SV1, dependencies="requirements.txt")
def run_hybrid_job(num_tasks = 1):
    return ...
```

例如， requirements.txt 檔案可能包含要安裝的其他套件。

```
qiskit
pennylane >= 0.31
mitiq == 0.29
```

或者，您可以提供套件名稱做為 Python 清單，如下所示。

```
@hybrid_job(device=Devices.Amazon.SV1, dependencies=["qiskit", "pennylane>=0.31",
 "mitiq==0.29"])
def run_hybrid_job(num_tasks = 1):
    return ...
```

其他原始程式碼可以指定為模組清單，也可以指定單一模組，如下列程式碼範例所示。

```
@hybrid_job(device=Devices.Amazon.SV1, include_modules=["my_module1", "my_module2"])
def run_hybrid_job(num_tasks = 1):
    return ...
```

將資料儲存並載入混合任務執行個體

指定輸入訓練資料

建立混合任務時，您可以透過指定 Amazon Simple Storage Service (Amazon S3) 儲存貯體來提供輸入訓練資料集。您也可以指定本機路徑，然後 Braket 會自動將資料上傳至位於 的 s3://<default_bucket_name>/jobs/<job_name>/<timestamp>/data/<channel_name> Amazon S3。如果您指定本機路徑，則頻道名稱預設為「輸入」。下列程式碼顯示來自本機路徑 的 numpy 檔案data/file.npy。

```
@hybrid_job(device=Devices.Amazon.SV1, input_data="data/file.npy")
```

```
def run_hybrid_job(num_tasks = 1):
    data = np.load("data/file.npy")
    return ...
```

對於 S3，您必須使用`get_input_data_dir()`協助程式 funciton。

```
s3_path = "s3://amazon-braket-us-west-1-961591465522/job-data/file.npy"

@hybrid_job(device=None, input_data=s3_path)
def job_s3_input():
    np.load(get_input_data_dir() + "/file.npy")

@hybrid_job(device=None, input_data={"channel": s3_path})
def job_s3_input_channel():
    np.load(get_input_data_dir("channel") + "/file.npy")
```

您可以透過提供頻道值和 S3 URIs 或本機路徑的字典來指定多個輸入資料來源。

```
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**
>
> 當輸入資料很大 (>1GB) 時，在建立任務之前會有很長的等待時間。這是因為本機輸入資料第一次上傳至 S3 儲存貯體時，S3 路徑會新增至任務請求。最後，任務請求會提交至 Braket 服務。

將結果儲存至 S3

若要儲存未包含在裝飾函數的傳回陳述式中的結果，您必須將正確的目錄附加至所有檔案寫入操作。下列範例顯示儲存 numpy 陣列和 matplotlib 圖。

```
@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 ...
```

所有結果都會壓縮為名為 的檔案model.tar.gz。您可以使用 Python job.result() 函數 下載結果，或從 Braket 管理主控台的混合任務頁面導覽至結果資料夾。

從檢查點儲存和繼續

對於長時間執行的混合任務，建議定期儲存演算法的中繼狀態。您可以使用內建save_job_checkpoint()協助程式函數，或將檔案儲存至AMZN_BRAKET_JOB_RESULTS_DIR路徑。稍後的 可與協助程式函數 搭配使用get_job_results_dir()。

以下是使用混合式任務裝飾工具儲存和載入檢查點的最小工作範例：

```
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()
```

在第一個混合任務中， save_job_checkpoint() 會使用包含要儲存之資料的字典來呼叫 。根據預設，每個值都必須可序列化為文字。若要檢查點更複雜的 Python 物件，例如 numpy 陣列，您可以設

定 data_format = PersistedJobDataFormat.PICKLED_V4。此程式碼會在稱為「檢查點」的子資料夾下，以混合任務成品<jobname>.json中的預設名稱建立和覆寫檢查點檔案。

若要建立新的混合任務以從檢查點繼續，我們需要傳遞 ，copy_checkpoints_from_job=job_arn其中 job_arn是先前任務的混合任務 ARN。然後，我們使用 從檢查點load_job_checkpoint(job_name)載入 。

混合式任務裝飾項目的最佳實務

接受非同步性

使用裝飾項目註釋建立的混合任務是非同步的 - 它們會在傳統和量子資源可用時執行。您可以使用 Braket Management Console或 Amazon CloudWatch 監控演算法的進度。當您提交演算法以執行時，Raket 會在可擴展的容器化環境中執行演算法，並在演算法完成時擷取結果。

執行反覆變化演算法

混合任務為您提供執行反覆式量子傳統演算法的工具。對於純量子問題，請使用量子任務或一批量子任務。對特定 QPUs優先順序存取，對於需要對 QPUs 進行多次反覆呼叫並對其進行傳統處理的長時間執行變化演算法最有利。

使用本機模式進行偵錯

在 QPU 上執行混合任務之前，建議您先在模擬器 SV1 上執行，以確認其如預期般執行。對於小規模測試，您可以使用本機模式執行 ，以進行快速反覆運算和偵錯。

使用使用自有容器 (BYOC) 改善重現性

在容器化環境中封裝您的軟體及其相依性，以建立可重現的實驗。透過將所有程式碼、相依性和設定封裝在容器中，您可以防止潛在的衝突和版本控制問題。

多執行個體分散式模擬器

若要執行大量電路，請考慮使用內建的 MPI 支援，在單一混合任務中的多個執行個體上執行本機模擬器。如需詳細資訊，請參閱內嵌模擬器。

使用參數電路

您從混合任務提交的參數電路會在特定 QPUs 上自動編譯，使用參數編譯來改善演算法的執行時間。

定期檢查點

對於長時間執行的混合任務，建議定期儲存演算法的中繼狀態。

如需進一步的範例、使用案例和最佳實務，請參閱 Amazon Braket 範例 GitHub。

使用 Amazon Braket 混合任務執行混合任務

若要使用 Amazon Braket 混合任務執行混合任務，您必須先定義演算法。您可以編寫演算法指令碼，也可以選擇使用 Amazon Braket Python SDK 或 PennyLane 來定義它。如果您想要使用其他（開放原始碼或專屬）程式庫，您可以使用 Docker 定義自己的自訂容器映像，其中包含這些程式庫。如需詳細資訊，請參閱使用您自己的容器 (BYOC)。

在任一情況下，接下來您使用 Amazon Braket 建立混合任務API，並在其中提供演算法指令碼或容器，選取混合任務要使用的目標量子裝置，然後從各種選用設定中選擇。這些選用設定的預設值適用於大多數使用案例。若要讓目標裝置執行混合任務，您可以選擇 QPU、隨需模擬器（例如 SV1DM1或 TN1）或傳統混合任務執行個體本身。透過隨需模擬器或 QPU，您的混合任務容器會對遠端裝置進行 API 呼叫。使用內嵌模擬器時，模擬器會內嵌在與演算法指令碼相同的容器中。PennyLane 的閃電模擬器內嵌預設預先建置的混合作業容器，供您使用。如果您使用內嵌 PennyLane 模擬器或自訂模擬器執行程式碼，您可以指定執行個體類型，以及您想要使用的執行個體數量。如需每個選擇的相關成本，請參閱 Amazon Braket 定價頁面。

如果您的目標裝置是隨需或內嵌模擬器，Amazon Braket 會立即開始執行混合任務。它會啟動混合任務執行個體（您可以在API呼叫中自訂執行個體類型）、執行演算法、將結果寫入 Amazon S3，以及釋出資源。此資源版本可確保您只需支付使用量的費用。

每個量子處理單元 (QPU) 的並行混合任務總數受到限制。今天，任何指定時間只能有一個混合任務在 QPU 上執行。佇列用於控制允許執行的混合任務數量，以免超過允許的限制。如果您的目標裝置是 QPU，您的混合任務會先進入所選 QPU 的任務佇列。Amazon Braket 會啟動所需的混合任務執行個

體，並在裝置上執行您的混合任務。在演算法期間，您的混合任務具有優先順序存取，這表示混合任務中的量子任務在裝置上排入佇列的其他 Braket 量子任務之前執行，前提是任務量子任務每隔幾分鐘提交一次至 QPU。一旦混合任務完成，就會釋出資源，這表示您只需支付使用量的費用。

> **Note**
>
> 裝置是區域性的，您的混合任務在 AWS 區域 與主要裝置相同的 中執行。

在模擬器和 QPU 目標案例中，您可以選擇定義自訂演算法指標，例如 Hamiltonian 的能量，作為演算法的一部分。這些指標會自動回報給 Amazon CloudWatch，並從那裡，在 Amazon Braket 主控台中以近乎即時的方式顯示。

> **Note**
>
> 如果您想要使用 GPU 型執行個體，請務必在 Braket 上使用其中一個 GPU 型模擬器搭配內嵌模擬器（例如 `lightning.gpu`)。如果您選擇其中一個 CPU 型內嵌模擬器（例如 `lightning.qubit`或 `braket:default-simulator`)，則不會使用 GPU，而且可能會產生不必要的成本。

建立您的第一個混合任務

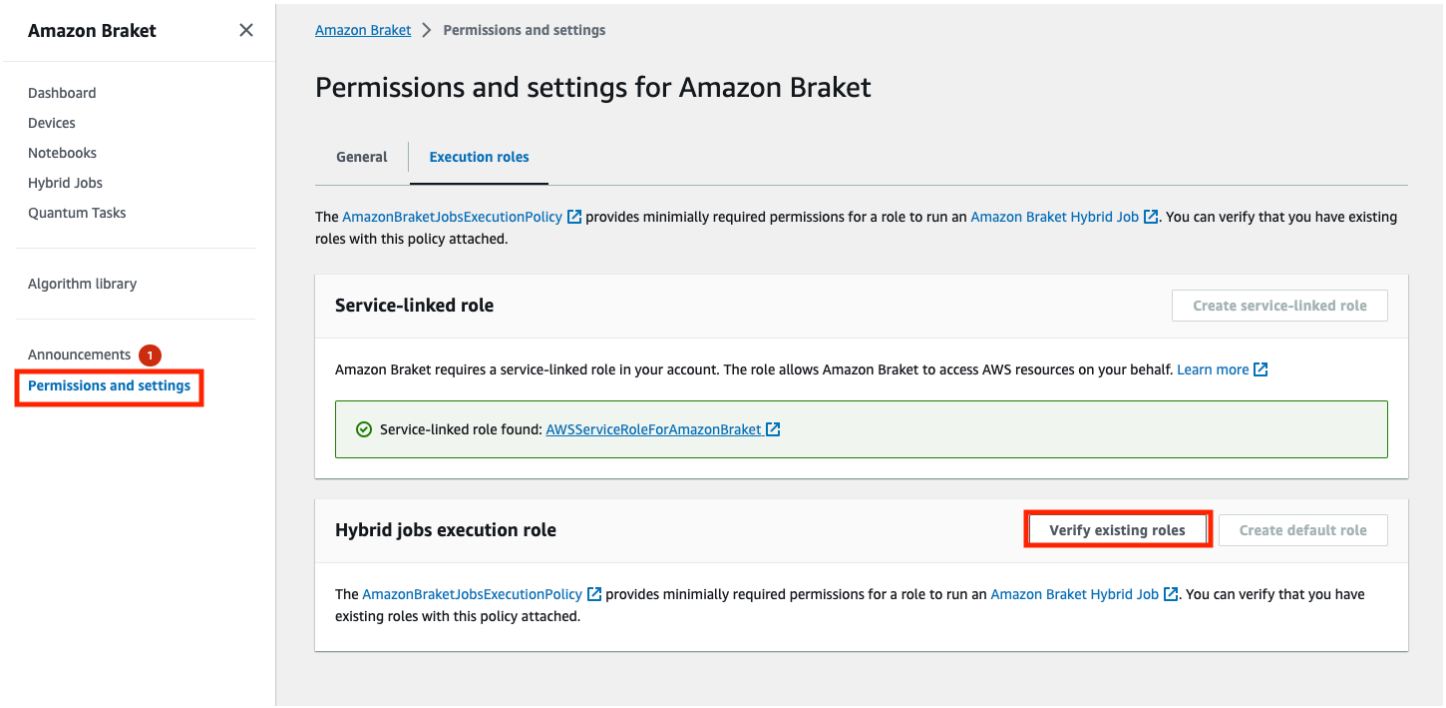
本節說明如何使用 Python 指令碼建立混合任務。或者，若要從本機 Python 程式碼建立混合任務，例如您偏好的整合開發環境 (IDE) 或 Braket 筆記本，請參閱 將本機程式碼作為混合任務執行。

在本節中：

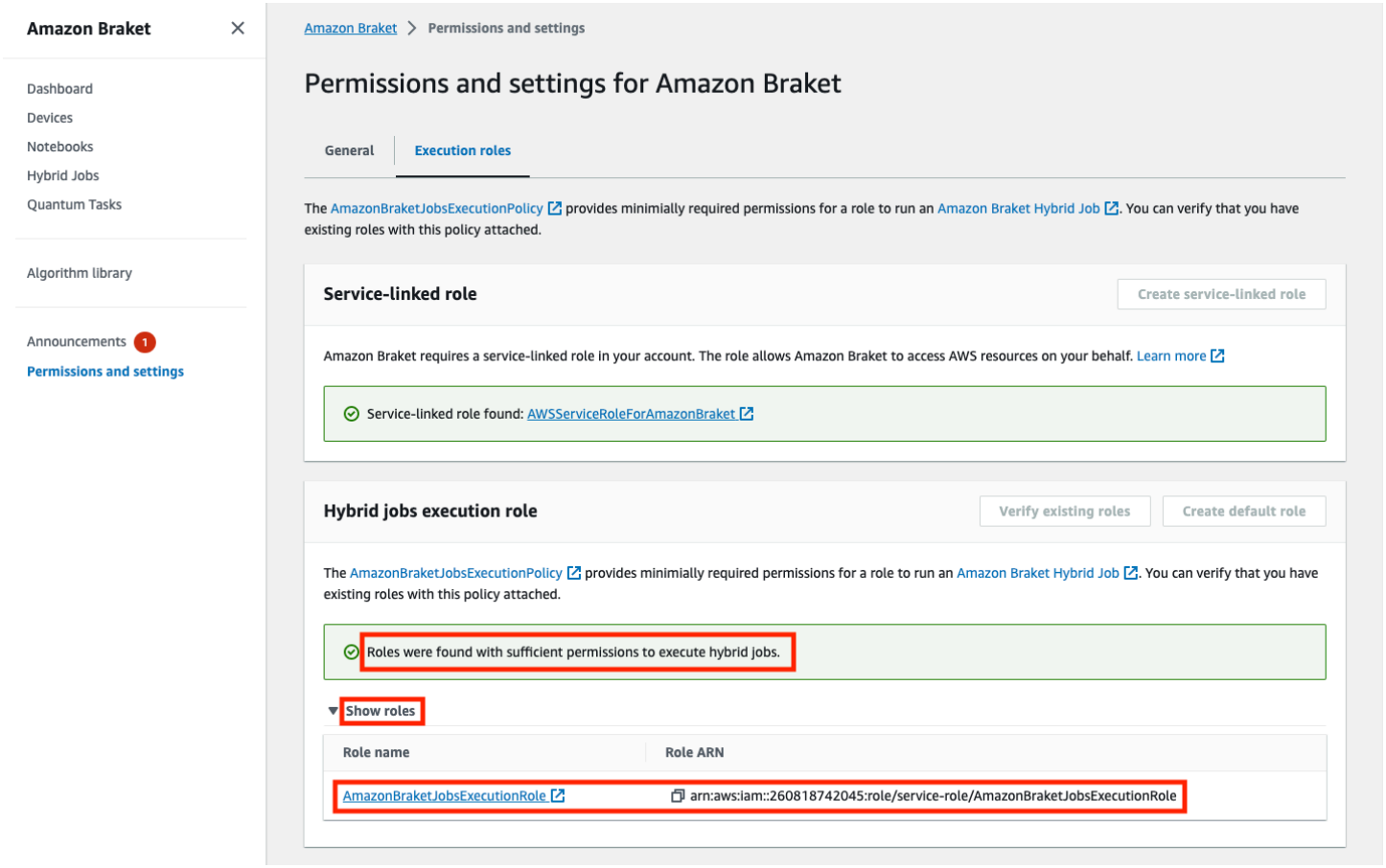
- 設定許可
- 建立並執行
- 監控結果

設定許可

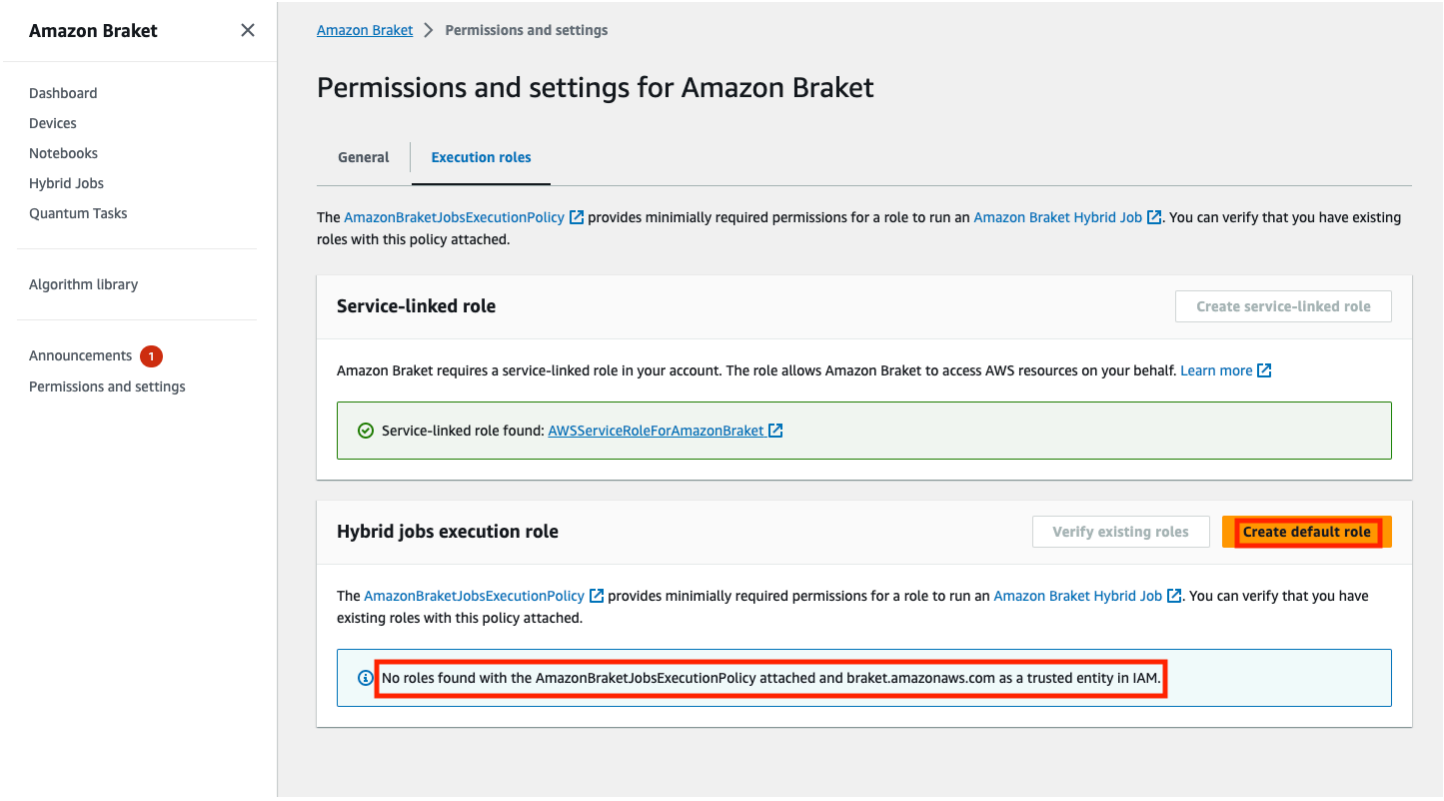
執行第一個混合任務之前，您必須確定您有足夠的許可以繼續此任務。若要判斷您擁有正確的許可，請從 Braket 主控台左側的選單中選取許可。Amazon Braket 的許可管理頁面可協助您驗證其中一個現有角色是否具有足以執行混合任務的許可，或引導您建立預設角色，以便在您尚未擁有此類角色時用來執行混合任務。



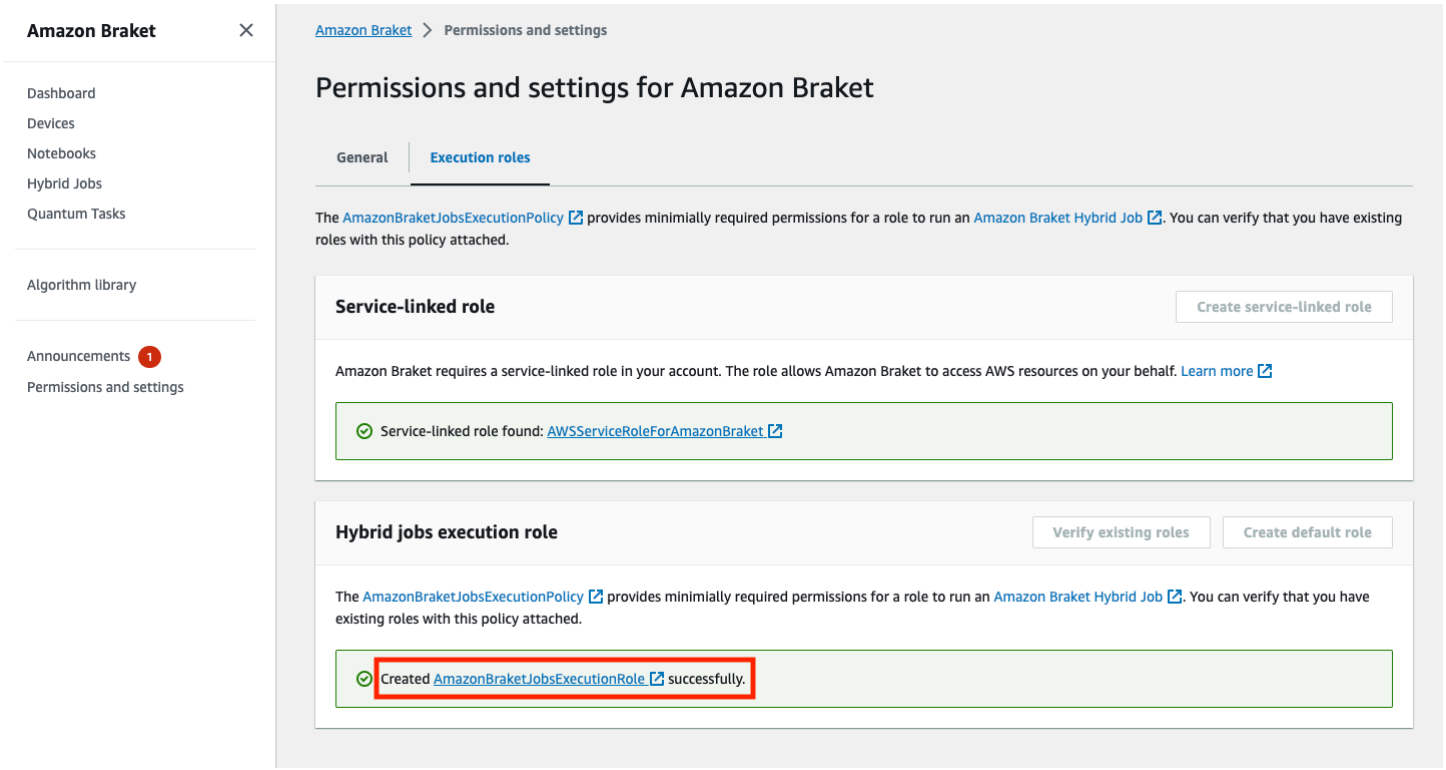
若要驗證您的角色是否具有執行混合任務的足夠許可，請選取驗證現有角色按鈕。如果您這麼做，您會收到找到角色的訊息。若要查看角色的名稱及其角色 ARNs，請選取顯示角色按鈕。



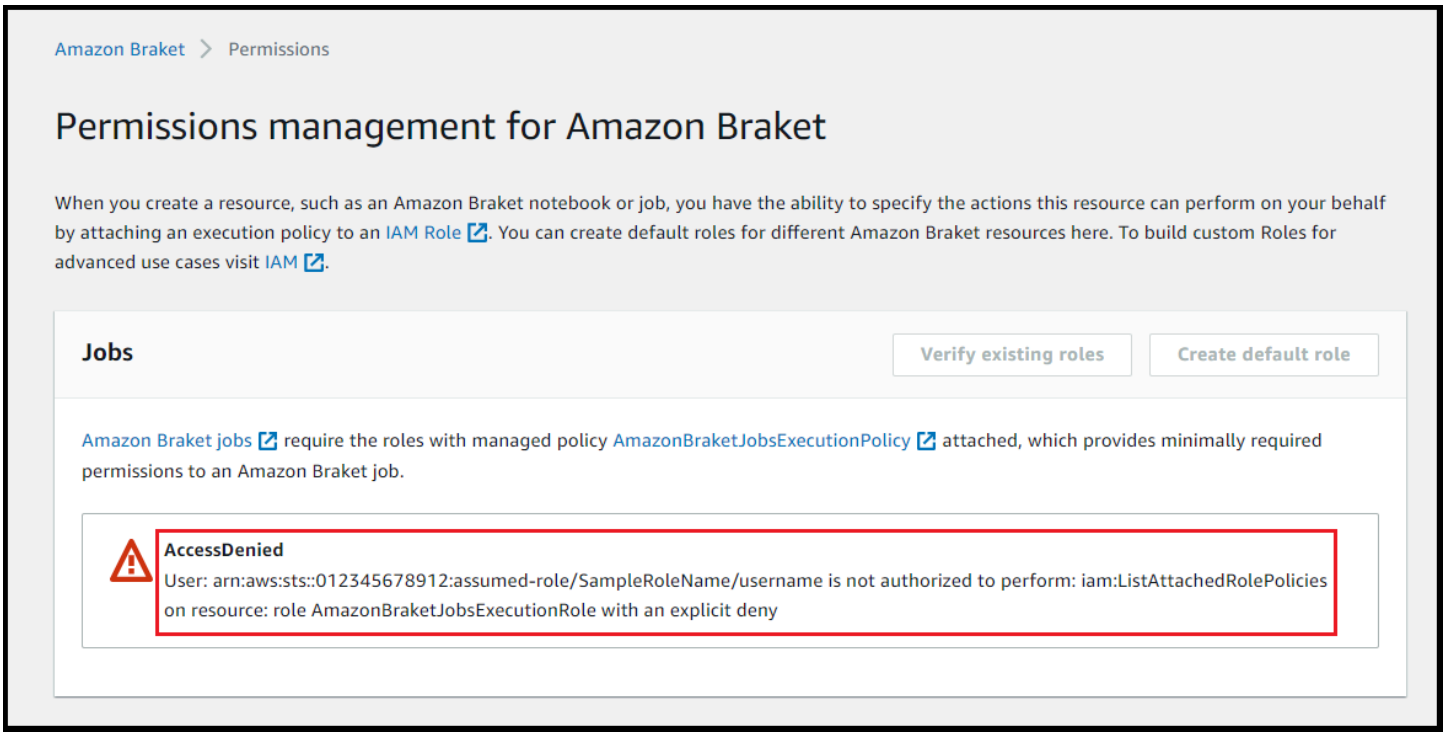
如果您沒有具有足夠執行混合任務許可的角色，您會收到一則訊息，指出找不到該角色。選取建立預設角色按鈕，以取得具有足夠許可的角色。



如果角色已成功建立，您會收到確認此訊息。

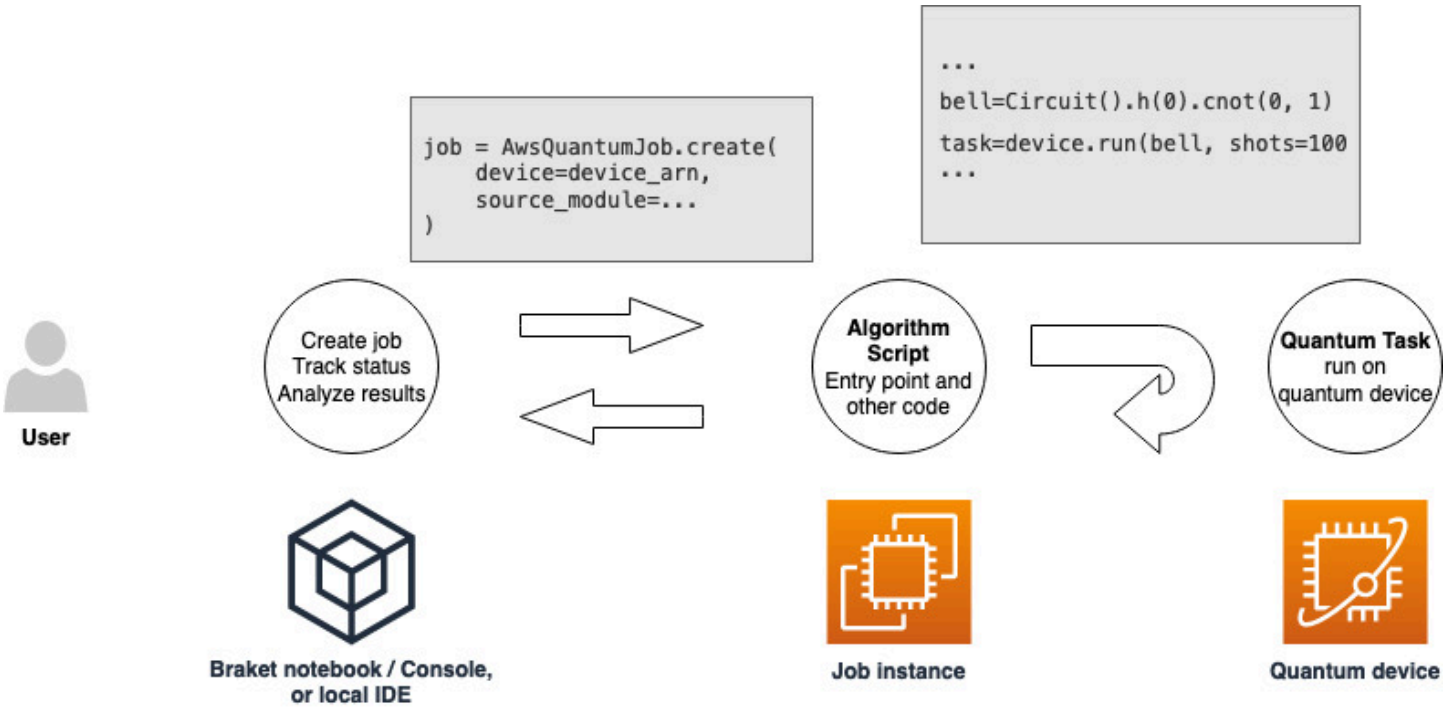


如果您沒有進行此查詢的許可，則會拒絕您存取。在此情況下，請聯絡您的內部 AWS 管理員。



建立並執行

一旦您有具有執行混合任務許可的角色，您就可以繼續進行。第一個 Braket 混合任務的關鍵部分是演算法指令碼。它定義了您想要執行的演算法，並包含屬於演算法一部分的傳統邏輯和量子任務。除了演算法指令碼之外，您還可以提供其他相依性檔案。演算法指令碼及其相依性稱為來源模組。進入點會定義混合任務啟動時，要在來源模組中執行的第一個檔案或函數。



首先，請考慮以下演算法指令碼的基本範例，該指令碼會建立五個鐘狀態並列印對應的測量結果。

```
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!")
```

在 Braket 筆記本或本機環境的目前工作目錄中，使用名稱 algorithm_script.py 儲存此檔案。algorithm_script.py 檔案具有 `start_here()`作為計劃的進入點。

接著，在與 algorithm_script.py 檔案相同的目錄中建立 Python 檔案或 Python 筆記本。此指令碼會啟動混合任務，並處理任何非同步處理，例如列印我們感興趣的狀態或關鍵結果。此指令碼至少需要指定混合任務指令碼和主要裝置。

> **Note**
>
> 如需如何在與筆記本相同的目錄中建立 Braket 筆記本或上傳檔案的詳細資訊，例如 algorithm_script.py 檔案，請參閱使用 Amazon Braket Python SDK 執行您的第一個電路

在這個基本的第一個案例中，您會以模擬器為目標。無論您以哪種類型的量子裝置、模擬器或實際量子處理單元 (QPU) 為目標，您在下列指令碼`device`中指定的裝置都會用來排程混合式任務，並可供演算法指令碼做為環境變數 使用`AMZN_BRAKET_DEVICE_ARN`。

> **Note**
>
> 您只能使用混合任務 AWS 區域 的 中可用的裝置。Amazon Braket SDK 會自動選取此選項 AWS 區域。例如，us-east-1 中的混合任務可以使用 IonQ、DM1、 SV1和 TN1 裝置，但不能使用 Rigetti 裝置。

如果您選擇量子電腦而非模擬器，則 Braket 會排程您的混合任務，以執行其具有優先順序存取的所有量子任務。

```
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
)
```

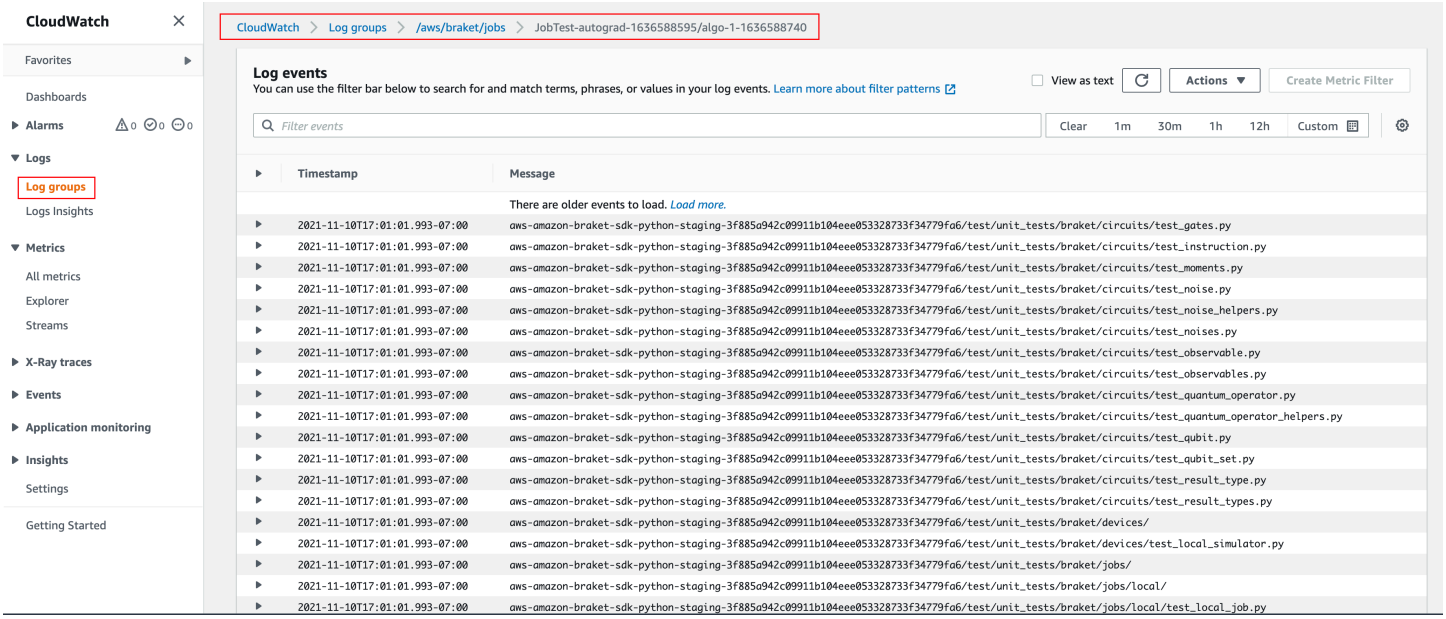
參數會wait_until_complete=True設定詳細模式，讓您的任務在執行時列印來自實際任務的輸出。您應該會看到類似下列範例的輸出。

```
job = AwsQuantumJob.create(
    Devices.Amazon.SV1,
    source_module="algorithm_script.py",
    entry_point="algorithm_script:start_here",
    wait_until_complete=True,
)
Initializing Braket Job: arn:aws:braket:us-west-2:<accountid>:job/<UUID>
.....................................
.
.
.

Completed 36.1 KiB/36.1 KiB (692.1 KiB/s) with 1 file(s) remaining#015download:
 s3://braket-external-assets-preview-us-west-2/HybridJobsAccess/models/
braket-2019-09-01.normal.json to ../../braket/additional_lib/original/
braket-2019-09-01.normal.json
Running Code As Process
Test job started!!!!!
Counter({'00': 55, '11': 45})
Counter({'11': 59, '00': 41})
Counter({'00': 55, '11': 45})
Counter({'00': 58, '11': 42})
Counter({'00': 55, '11': 45})
Test job completed!!!!!
Code Run Finished
2021-09-17 21:48:05,544 sagemaker-training-toolkit INFO     Reporting training SUCCESS
```

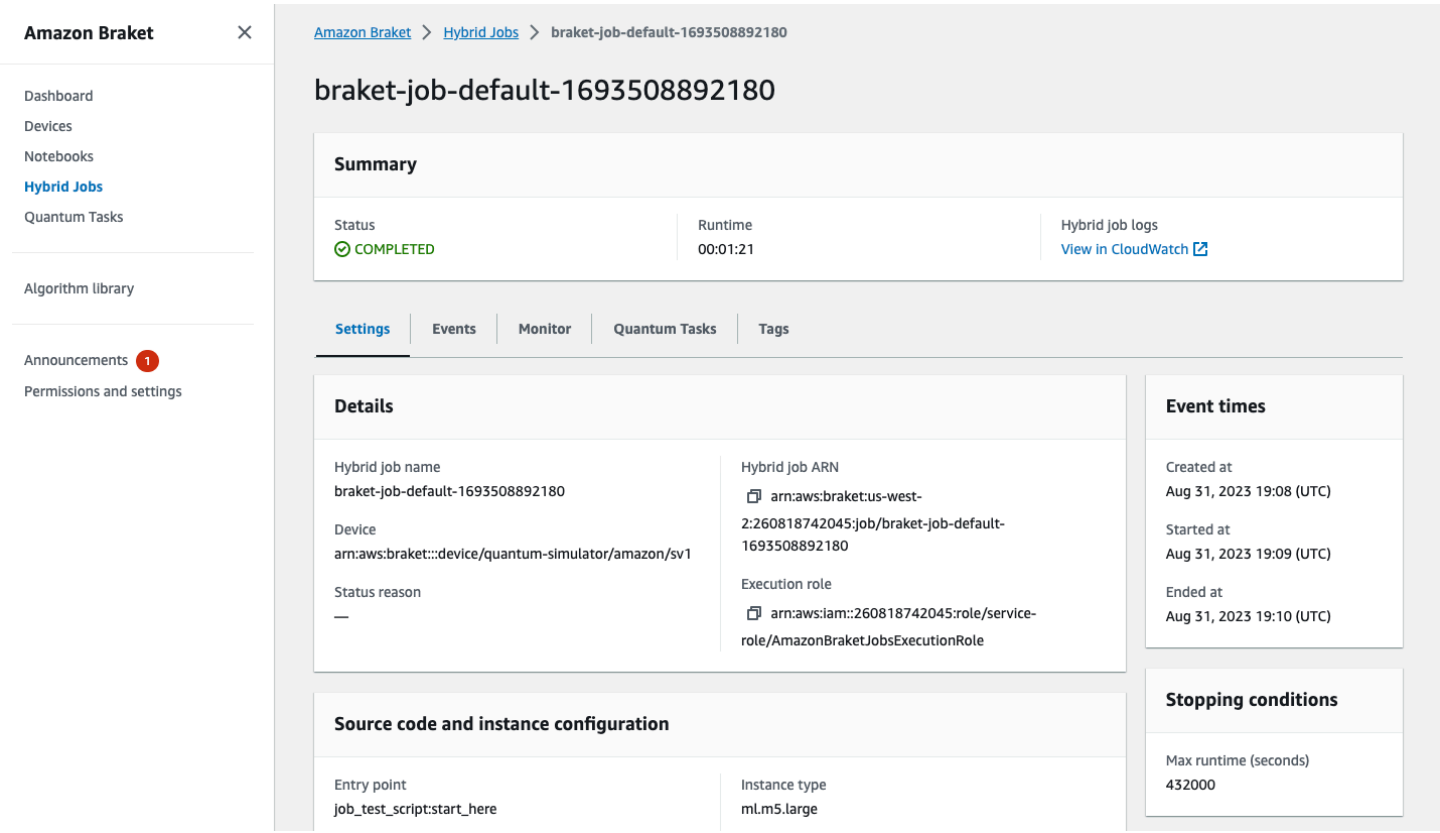
> **Note**
>
> 您也可以將自訂模組與 AwsQuantumJob.create 方法搭配使用，方法是傳遞其位置（本機目錄或檔案的路徑，或 tar.gz 檔案的 S3 URI)。如需工作範例，請參閱 Amazon Braket 範例 Github 儲存庫中混合任務資料夾中的 Parallelize_training_for_QML.ipynb 檔案。

監控結果

或者，您可以從 Amazon CloudWatch 存取日誌輸出。若要這樣做，請前往任務詳細資訊頁面左側選單上的日誌群組索引標籤，選取日誌群組 `aws/braket/jobs`，然後選擇包含任務名稱的日誌串流。在上述範例中，即為 `braket-job-default-1631915042705/algo-1-1631915190`。



您也可以在主控台中選取混合任務頁面，然後選擇設定，以檢視混合任務的狀態。



您的混合任務在執行時會在 Amazon S3 中產生一些成品。預設 S3 儲存貯體名稱為 ，amazon-braket-<region>-<accountid>且內容位於 jobs/<jobname>/<timestamp>目錄中。您可以使用 Braket Python SDK 建立混合任務code_location時，指定不同的 ，以設定存放這些成品的 S3 位置。

> **Note**
>
> 此 S3 儲存貯體必須與 AWS 區域 任務指令碼位於相同的 中。

jobs/<jobname>/<timestamp> 目錄包含一個子資料夾，其中包含model.tar.gz檔案中進入點指令碼的輸出。還有一個名為 的目錄script，其中包含source.tar.gz檔案中的演算法指令碼成品。您實際量子任務的結果位於名為 的目錄中jobs/<jobname>/tasks。

儲存您的任務結果

您可以儲存演算法指令碼產生的結果，以便從混合任務指令碼中的混合任務物件以及 Amazon S3 中的輸出資料夾 （ 在名為 model.tar.gz 的 tar 壓縮檔案中 ） 中使用這些結果。

輸出必須使用 JavaScript 物件標記法 (JSON) 格式儲存在檔案中。如果資料無法輕易序列化為文字，如同使用凹凸陣列的情況，您可以傳入 選項，以使用挑選的資料格式序列化。如需詳細資訊，請參閱 braket.jobs.data_persistence 模組。

若要儲存混合任務的結果，請將以下以 #ADD 註解的行新增至演算法指令碼。

```
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!!!!!")
```

然後，您可以透過附加加上 #ADD **print(job.result())** 註解的行來顯示任務指令碼中的任務結果。

```
import time
from braket.aws import AwsQuantumJob

job = AwsQuantumJob.create(
    source_module="algorithm_script.py",
    entry_point="algorithm_script:start_here",
    device_arn="arn:aws:braket:::device/quantum-simulator/amazon/sv1",
)

print(job.arn)
while job.state() not in AwsQuantumJob.TERMINAL_STATES:
    print(job.state())
```

```
    time.sleep(10)

print(job.state())
print(job.result())   #ADD
```

在此範例中，我們移除 wait_until_complete=True來隱藏詳細輸出。您可以在 中新增它以進行偵錯。當您執行此混合任務時，它會輸出識別符 和 job-arn，接著每 10 秒輸出一次混合任務的狀態，直到混合任務為 為止COMPLETED，之後會顯示鐘形電路的結果。請參閱以下範例。

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

使用檢查點儲存和重新啟動混合任務

您可以使用檢查點儲存混合任務的中繼反覆運算。在上一節的演算法指令碼範例中，您可以新增以下加上 #ADD 註解的行來建立檢查點檔案。

```
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!!!!!")
```

當您執行混合任務時，它會在檢查點目錄中的混合任務成品中以預設/opt/jobs/checkpoints路徑建立檔案 <jobname>-checkpoint-1.json。除非您想要變更此預設路徑，否則混合任務指令碼保持不變。

如果您想要從先前混合任務產生的檢查點載入混合任務，演算法指令碼會使用 `from braket.jobs import load_job_checkpoint`。要載入演算法指令碼的邏輯如下。

```
checkpoint_1 = load_job_checkpoint(
    "previous_job_name",
    checkpoint_file_suffix="checkpoint-1",
)
```

載入此檢查點之後，您可以根據載入至 的內容繼續邏輯checkpoint-1。

> **Note**
>
> checkpoint_file_suffix 必須符合先前在建立檢查點時指定的尾碼。

您的協同運作指令碼需要指定上一個混合任務job-arn的 ，並使用 #ADD 加上註解的行。

```
job = AwsQuantumJob.create(
    source_module="source_dir",
    entry_point="source_dir.algorithm_script:start_here",
    device_arn="arn:aws:braket:::device/quantum-simulator/amazon/sv1",
    copy_checkpoints_from_job="<previous-job-ARN>", #ADD
```

```
)
```

使用本機模式建置和偵錯混合式任務

當您建立新的混合演算法時，本機模式可協助您偵錯和測試演算法指令碼。本機模式是一項功能，可讓您執行您計劃在 Amazon Braket 混合任務中使用的程式碼，但不需要使用 Braket 來管理執行混合任務的基礎設施。相反地，請在 Amazon Braket 筆記本執行個體或偏好的用戶端本機上執行混合式任務，例如筆記型電腦或桌上型電腦。

在本機模式中，您仍可將量子任務傳送至實際裝置，但在本機模式中針對實際的 Quantum 處理單元 (QPU) 執行 時，不會獲得效能優勢。

若要使用本機模式，請將 修改`AwsQuantumJob`為程式內發生`LocalQuantumJob`的任何位置。例如，若要從建立您的第一個混合任務執行範例，請在程式碼中編輯混合任務指令碼，如下所示。

```
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**
>
> 已預先安裝在 Amazon Braket 筆記本中的 Docker 需要安裝在您的本機環境中，才能使用此功能。您可以在 Get Docker 頁面上找到安裝 Docker 的說明。此外，本機模式中不支援所有參數。

使用 Amazon Braket 執行您的量子任務

Braket 提供不同類型的量子電腦的安全隨需存取。您可以從 IonQ、 IQM和 存取閘道型量子電腦 Rigetti，以及從 QuEra 存取類比 Hamiltonian Simulator。您也不需要預先承諾，也不需要透過個別供應商取得存取權。

- Amazon Braket 主控台提供裝置資訊和狀態，協助您建立、管理和監控資源和量子任務。
- 透過 Amazon Braket Python SDK 以及主控台提交和執行量子任務。開發套件可透過預先設定的 Amazon Braket 筆記本存取。
- Amazon Braket API 可透過 Amazon Braket Python SDK 和筆記本存取。API 如果您要建置以程式設計方式使用量子運算的應用程式，您可以直接呼叫 。

本節中的範例示範如何API直接使用 Amazon Braket Python SDK 以及 Python SDK for Braket (Boto3) 來使用 Amazon Braket。 AWS

有關 Amazon Braket Python SDK 的詳細資訊

若要使用 Amazon Braket Python SDK，請先安裝 AWS Python SDK for Braket (Boto3)，以便與 通訊 AWS API。您可以將 Amazon Braket Python SDK 視為用於量子客戶之 Boto3 的便利包裝函式。

- Boto3 包含點選 所需的界面 AWS API。（請注意，Boto3 是與 通訊的大型 Python SDK AWS API。大多數 AWS 服務 支援 Boto3 介面。）
- Amazon Braket Python SDK 包含用於量子任務的電路、閘道、裝置、結果類型和其他部分的軟體模組。每次建立程式時，您都會匯入該量子任務所需的模組。
- Amazon Braket Python SDK 可透過筆記本存取，筆記本會預先載入執行量子任務所需的所有模組和相依性。
- 如果您不想使用筆記本，您可以將模組從 Amazon Braket Python SDK 匯入任何 Python 指令碼。

安裝 Boto3 之後，透過 Amazon Braket Python SDK 建立量子任務的步驟概觀如下所示：

1. （選用）開啟您的筆記本。
2. 匯入電路所需的 SDK 模組。
3. 指定 QPU 或模擬器。
4. 執行個體化電路。

5. 執行電路。
6. 收集結果。

本節中的範例顯示每個步驟的詳細資訊。

如需更多範例，請參閱 GitHub 上的 Amazon Braket 範例儲存庫。

在本節中：

將量子任務提交至 QPUs

Amazon Braket 可讓您存取數個可執行量子任務的裝置。您可以個別提交量子任務，也可以設定量子任務批次。

QPUs

您可以隨時向 QPUs 提交量子任務，但任務會在 Amazon Braket 主控台的裝置頁面上顯示的特定可用時段內執行。您可以使用量子任務 ID 擷取量子任務的結果，請參閱下一節。

- IonQ Aria-1 ：`arn:aws:braket:us-east-1::device/qpu/ionq/Aria-1`
- IonQ Aria-2 ：`arn:aws:braket:us-east-1::device/qpu/ionq/Aria-2`
- IonQ Forte-1 ：`arn:aws:braket:us-east-1::device/qpu/ionq/Forte-1`
- IonQ Forte-Enterprise-1 ：`arn:aws:braket:us-east-1::device/qpu/ionq/Forte-Enterprise-1`
- IQM Garnet ：`arn:aws:braket:eu-north-1::device/qpu/iqm/Garnet`
- IQM Emerald ：`arn:aws:braket:eu-north-1::device/qpu/iqm/Emerald`
- QuEra Aquila ：`arn:aws:braket:us-east-1::device/qpu/quera/Aquila`
- Rigetti Ankaa-3 ：`arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3`

> **Note**
>
> 您可以取消 QPUs 和隨需模擬器處於 CREATED 狀態的量子任務。您可以針對隨需模擬器和 QPUs，盡力取消QUEUED處於 狀態的量子任務。請注意，QPU QUEUED 量子任務不太可能在 QPU 可用性時段內成功取消。

在本節中：

IonQ

IonQ 提供以閘道為基礎的 QPUs以離子捕捉技術為基礎。IonQ's捕捉QPUs 是建置在捕捉的 171Yb+ 離子鏈上，該鏈透過真空室內的微製表面電波捕捉空間受限。

IonQ 裝置支援下列量子閘道。

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

透過逐字編譯，IonQQPUs 支援下列原生閘道。

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

如果您在使用原生 MS 閘道時只指定兩個階段參數，則會執行完全串連的 MS 閘道。完全連接的 MS 閘道一律會執行 π/2 輪換。若要指定不同的角度並執行部分連接的 MS 閘道，您可以新增第三個參數來指定所需的角度。如需詳細資訊，請參閱 braket. circuits.gate 模組。

這些原生閘道只能與逐字編譯搭配使用。若要進一步了解逐字編譯，請參閱逐字編譯。

IQM

IQM quantum 處理器是基於超導轉子 qubit 的通用閘道模型裝置。IQM Garnet 是 20 qubit 裝置，而 IQM Emerald 是 54 qubit 裝置。這兩個裝置都使用方形格狀拓撲，也稱為 Crystal 格狀拓撲。

IQM 裝置支援下列量子閘道。

```
"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"
```

透過逐字編譯，IQM裝置支援下列原生閘道。

```
'cz', 'prx'
```

Rigetti

Rigetti 量子處理器是通用的閘道模型機器，以全可調校的超導 為基礎qubits。

- Ankaa-3 系統是一種 84 qubit 裝置，利用可擴展的多晶片技術。

Rigetti 裝置支援下列量子閘道。

```
'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'
```

透過逐字編譯， Ankaa-3支援下列原生閘道。

```
'rx', 'rz', 'iswap'
```

Rigetti 超導量子處理器只能執行 'rx' 閘道，角度為 ±π/2 或 ±π。

Rigetti 裝置提供脈波層級控制，可支援Ankaa-3系統一組下列類型的預先定義影格。

```
`flux_tx`, `charge_tx`, `readout_rx`, `readout_tx`
```

如需這些影格的詳細資訊，請參閱影格和連接埠的角色。

QuEra

QuEra 提供中性原子型裝置，可執行類比漢密爾頓模擬 (AHS) 量子任務。這些特殊用途裝置會忠實地重現數百個同時互動 qubit 的時間相依量子動態。

一個人可以在類比 Hamiltonian 模擬的範式中編寫這些裝置的程式，方法是編列 qubit 暫存器的配置，以及控制欄位的時間和空間相依性。Amazon Braket 提供公用程式，可透過 python SDK 的 AHS 模組建構這類程式`braket.ahs`。

如需詳細資訊，請參閱類比 Hamiltonian 模擬範例筆記本或使用 QuEra 的 Aquila 頁面提交類比程式。

範例：將量子任務提交至 QPU

Amazon Braket 可讓您在 QPU 裝置上執行量子電路。下列範例示範如何將量子任務提交至 Rigetti或 IonQ 裝置。

選擇Rigetti Ankaa-3裝置，然後查看相關聯的連線圖表

```
# 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']}}
```

上述字典會connectivityGraph列出Rigetti裝置中每個 qubit 的鄰近 qubit。

選擇IonQ Aria-1裝置

對於IonQ Aria-1裝置， connectivityGraph是空的，如下列範例所示，因為裝置提供all-to-all連線。因此，connectivityGraph不需要詳細的 。

```
# 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': {}}
```

如下列範例所示，您可以選擇調整 shots（預設值 = 1000）、 `poll_timeout_seconds`（預設值 = 432000 = 5 天）、 `poll_interval_seconds`（預設值 = 1），以及如果您選擇指定預設S3儲存貯體以外的位置，則儲存結果的位置 (`s3_location`)。

```
my_task = device.run(circ, s3_location = 'amazon-braket-my-folder', shots=100,
 poll_timeout_seconds = 100, poll_interval_seconds = 10)
```

IonQ 和 Rigetti 裝置會自動將提供的電路編譯至其各自的原生閘道集，並將抽象qubit索引映射至個別 QPU qubits上的實體。

> **Note**
>
> QPU 裝置容量有限。達到容量時，您可以預期較長的等待時間。

Amazon Braket 可以在特定可用時段內執行 QPU 量子任務，但您仍可隨時提交量子任務（全年無休），因為所有對應的資料和中繼資料都可靠地存放在適當的 S3 儲存貯體中。如下一節所示，您可以使用 `AwsQuantumTask`和唯一的量子任務 ID 來復原量子任務。

檢查編譯的電路

當量子電路需要在硬體裝置上執行時，例如量子處理單元 (QPU)，必須先將電路編譯為裝置可以理解和處理的可接受格式。例如，將高階量子電路向下轉換為目標 QPU 硬體支援的特定原生閘道。檢查量子電路的實際編譯輸出對於偵錯和最佳化用途非常有用。這些知識有助於識別潛在問題、瓶頸或改善量子應用程式效能和效率的機會。您可以使用以下提供的程式碼，檢視和分析 Rigetti和 IQM量子運算裝置的量子電路編譯輸出。

```
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**
>
> 目前不支援檢視IonQ裝置的編譯電路輸出。

我的量子任務何時執行？

當您提交電路時，Amazon Braket 會將其傳送至您指定的裝置。Quantum Processing Unit (QPU) 和隨需模擬器量子任務會依收到順序排入佇列和處理。提交後處理規定人數任務所需的時間，取決於其他 Amazon Braket 客戶提交的任務數量和複雜性，以及所選 QPU 的可用性。

在本節中：

QPU 可用性時段和狀態

QPU 可用性因裝置而異。

在 Amazon Braket 主控台的裝置頁面中，您可以查看目前和近期的可用時段和裝置狀態。此外，每個裝置頁面都會顯示量子任務和混合任務的個別佇列深度。

如果客戶無法使用 裝置，無論可用性時段為何，裝置都會被視為離線。例如，它可能因排定的維護、升級或操作問題而離線。

佇列可見性

在提交量子任務或混合任務之前，您可以透過檢查裝置佇列深度來檢視您面前有多少量子任務或混合任務。

佇列深度

Queue depth 是指針對特定裝置排入佇列的量子任務和混合任務數量。裝置的量子任務和混合任務佇列計數可透過 Braket Software Development Kit (SDK)或 存取Amazon Braket Management Console。

1. 任務佇列深度是指目前正在等待以正常優先順序執行的量子任務總數。
2. 優先順序任務佇列深度是指等待透過 執行的已提交量子任務總數Amazon Braket Hybrid Jobs。這些任務會在獨立任務之前執行。
3. 混合任務佇列深度是指目前在裝置上排入佇列的混合任務總數。作為混合任務的一部分Quantum tasks提交的 具有優先順序，並在 中彙總Priority Task Queue。

想要透過 檢視佇列深度的客戶Braket SDK可以修改下列程式碼片段，以取得其量子任務或混合任務的佇列位置：

```
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'
```

將量子任務或混合任務提交至 QPU 可能會導致您的工作負載處於 QUEUED 狀態。Amazon Braket 可讓客戶查看其量子任務和混合任務佇列位置。

佇列位置

Queue position 是指各自裝置佇列中量子任務或混合任務的目前位置。可以透過 或 取得量子任務Braket Software Development Kit (SDK)或混合任務Amazon Braket Management Console。

想要透過 檢視佇列位置的客戶Braket SDK可以修改下列程式碼片段，以取得其量子任務或混合任務的佇列位置：

```
# 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
```

設定電子郵件或簡訊通知

當 QPU 的可用性變更或您量子任務的狀態變更時，Amazon Braket 會將事件傳送至 Amazon EventBridge。請依照下列步驟，透過電子郵件或簡訊接收裝置和量子任務狀態變更通知：

1. 建立 Amazon SNS 主題和電子郵件或簡訊的訂閱。電子郵件或簡訊的可用性取決於您的 區域。如需詳細資訊，請參閱 Amazon SNS 入門和傳送簡訊。
2. 在 EventBridge 中建立觸發 SNS 主題通知的規則。如需詳細資訊，請參閱使用 Amazon EventBridge 監控 Amazon Braket EventBridge。

（選用）設定 SNS 通知

您可以透過 Amazon Simple Notification Service (SNS) 設定通知，以便在 Amazon Braket 量子任務完成時收到提醒。如果您預期等待時間較長，作用中的通知會很有用；例如，當您提交大型量子任務，或在裝置可用時段之外提交量子任務時。如果您不想等待量子任務完成，您可以設定 SNS 通知。

Amazon Braket 筆記本會逐步引導您完成設定步驟。如需詳細資訊，請參閱 GitHub 上的 Amazon Braket 範例，特別是設定通知的範例筆記本。

管理您的 Amazon Braket 混合任務

本節說明如何在 Amazon Braket 中管理混合式任務。

您可以使用下列方式存取 Braket 中的混合式任務：

- Amazon Braket Python SDK。
- Amazon Braket 主控台。
- Amazon Braket API。

在本節中：

設定混合任務執行個體以執行指令碼

根據您的演算法，您可能有不同的需求。根據預設，Amazon Braket 會在`ml.m5.large`執行個體上執行您的演算法指令碼。不過，您可以在使用下列匯入和組態引數建立混合任務時自訂此執行個體類型。

```
from braket.jobs.config import InstanceConfig

job = AwsQuantumJob.create(
    ...
    instance_config=InstanceConfig(instanceType="ml.p3.8xlarge"), # Use NVIDIA Tesla
 V100 instance with 4 GPUs.
    ...
    ),
```

如果您正在執行內嵌模擬，並在裝置組態中指定本機裝置，您也可以在 中指定 `instanceCount`並將它設定為大於一個`InstanceConfig`，以請求多個執行個體。上限為 5。例如，您可以選擇 3 個執行個體，如下所示。

```
from braket.jobs.config import InstanceConfig
job = AwsQuantumJob.create(
    ...
    instance_config=InstanceConfig(instanceType="ml.p3.8xlarge", instanceCount=3), #
 Use 3 NVIDIA Tesla V100
    ...
    ),
```

當您使用多個執行個體時，請考慮使用資料平行功能分發混合任務。如需如何查看此 QML 平行化訓練範例的詳細資訊，請參閱下列範例筆記本。

下列三個資料表列出標準、高效能和 GPU 加速執行個體的可用執行個體類型和規格。

> **Note**
>
> 若要檢視混合任務的預設傳統運算執行個體配額，請參閱 Amazon Braket Quotas 頁面。

標準執行個體	vCPU	記憶體 (GiB)
ml.m5.large （預設）	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

高效能執行個體	vCPU	記憶體 (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

高效能執行個體	vCPU	記憶體 (GiB)
ml.c5n.18xlarge	72	192

GPU 加速執行個體	GPU	vCPU	記憶體 (GiB)	GPU 記憶體 (GiB)
ml.p3.2xlarge	1	8	61	16
ml.p3.8xlarge	4	32	244	64
ml.p3.16xlarge	8	64	488	128
ml.p4d.24xlarge	8	96	1152	320
ml.g4dn.xlarge	1	4	16	16
ml.g4dn.2xlarge	1	8	32	16
ml.g4dn.4xlarge	1	16	64	16
ml.g4dn.8xlarge	1	32	128	16
ml.g4dn.12xlarge	4	48	192	64
ml.g4dn.16xlarge	1	64	256	16

> **Note**
>
> p3 執行個體不適用於 us-west-1。如果您的混合任務無法佈建請求的 ML 運算容量，請使用另一個區域。

每個執行個體使用 30 GB 的資料儲存 (SSD) 預設組態。但是，您可以使用與設定 相同的方式來調整儲存體`instanceType`。下列範例示範如何將總儲存體增加到 50 GB。

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

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

在 中設定預設儲存貯體 **AwsSession**

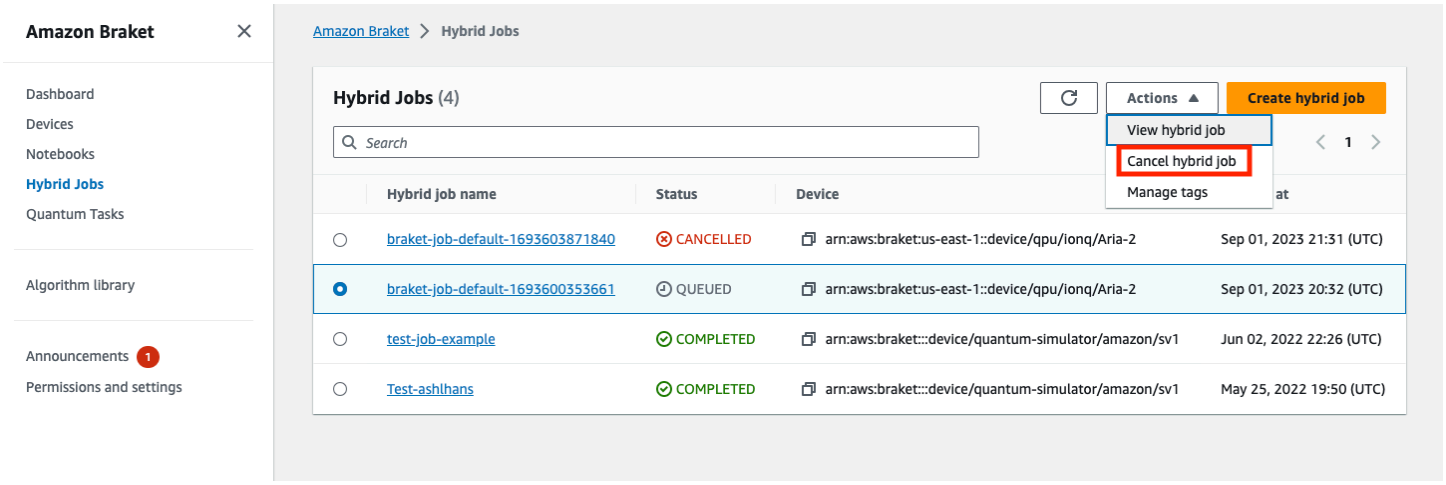
利用您自己的AwsSession執行個體可為您提供增強的彈性，例如能夠為您的預設 Amazon S3 儲存貯體指定自訂位置。根據預設， AwsSession具有預先設定的 Amazon S3 儲存貯體位置"amazon-braket-{id}-{region}"。不過，您可以選擇在建立 時覆寫預設的 Amazon S3 儲存貯體位置AwsSession。使用者可以選擇性地將AwsSession物件傳入 AwsQuantumJob.create()方法，方法是提供 aws_session 參數，如下列程式碼範例所示。

```
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
)
```

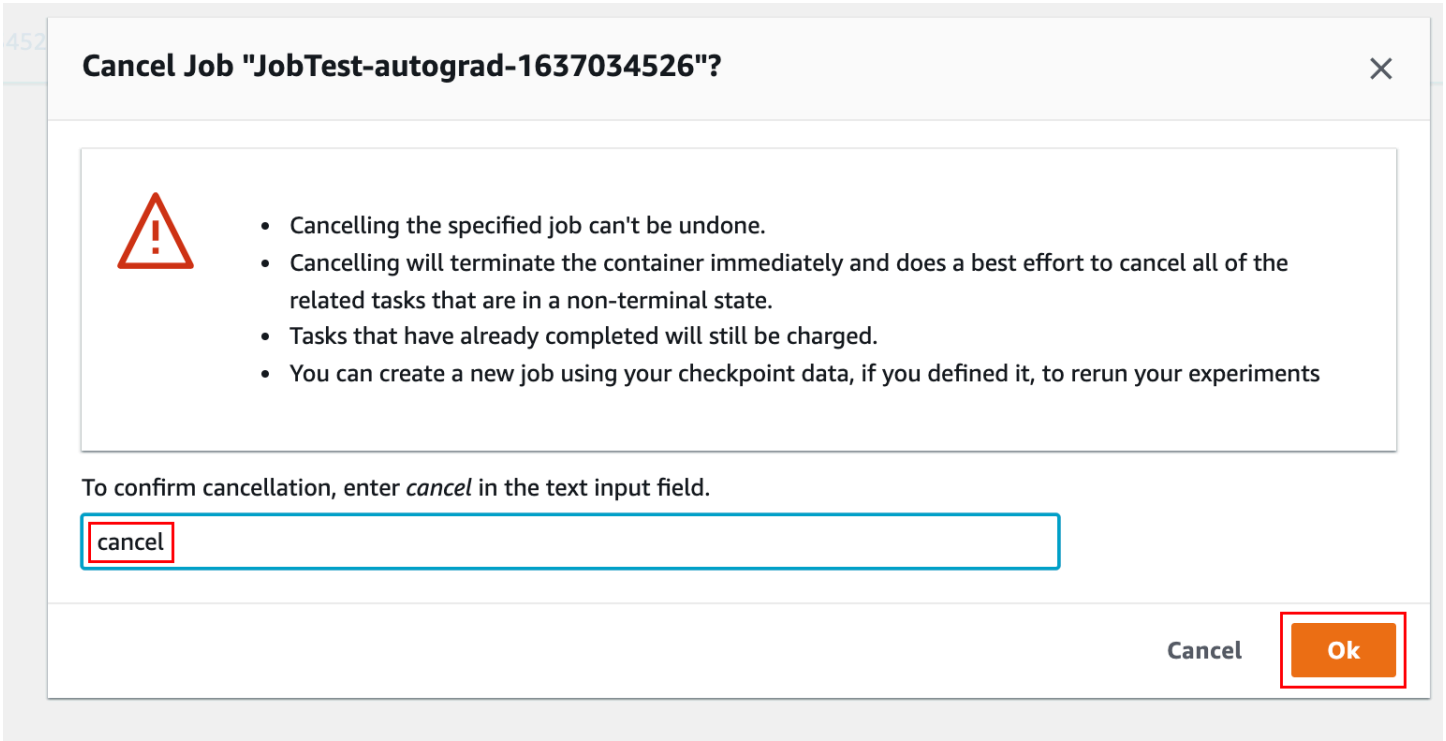
如何取消混合任務

您可能需要取消處於非終端狀態的混合式任務。這可以在 主控台或使用程式碼來完成。

若要在主控台中取消混合任務，請從混合任務頁面選取要取消的混合任務，然後從動作下拉式功能表中選取取消混合任務。



若要確認取消，請在出現提示時，在輸入欄位中輸入取消，然後選取確定。



若要使用 Braket Python SDK 的程式碼取消混合任務，請使用 `job_arn` 來識別混合任務，然後呼叫其上的 `cancel`命令，如下列程式碼所示。

```
job = AwsQuantumJob(arn=job_arn)
job.cancel()
```

`cancel` 命令會立即終止傳統混合任務容器，並盡最大努力取消所有仍處於非終端狀態的相關量子任務。

使用參數編譯來加速混合式任務

Amazon Braket 支援特定 QPUs的參數編譯。這可讓您只編譯電路一次，而不是混合演算法中的每次反覆運算，以減少與運算昂貴編譯步驟相關的額外負荷。這可以大幅改善混合任務的執行時間，因為您不必在每個步驟重新編譯電路。只需將參數化電路提交到我們支援的其中一個 QPUs 做為 Braket 混合任務。對於長時間執行的混合任務，Raket 會在編譯電路時自動使用來自硬體提供者的更新校正資料，以確保最高品質的結果。

若要建立參數電路，您必須先在演算法指令碼中提供參數做為輸入。在此範例中，我們使用小型參數電路，並忽略每次反覆運算之間的任何傳統處理。對於典型工作負載，您可以批次提交許多電路，並執行傳統處理，例如更新每個反覆運算中的參數。

```
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.")
```

您可以使用下列任務指令碼，提交演算法指令碼以混合任務的形式執行。在支援參數編譯的 QPU 上執行混合任務時，只會在第一次執行時編譯電路。在下列執行中，會重複使用編譯的電路，增加混合任務的執行時間效能，而不需要任何額外的程式碼行。

```
from braket.aws import AwsQuantumJob

job = AwsQuantumJob.create(
    device=device_arn,
    source_module="algorithm_script.py",
```

```
)
```

> **Note**
>
> 除了脈衝層級程式Rigetti Computing之外， 的所有超執行、以閘道為基礎的 QPUs都支援參數編譯。

使用您自己的容器 (BYOC)

Amazon Braket Hybrid Jobs 提供三個預先建置的容器，用於在不同環境中執行程式碼。如果其中一個容器支援您的使用案例，您只需在建立混合任務時提供演算法指令碼。您可以從演算法指令碼或使用從 `requirements.txt` 檔案新增次要缺少的相依性pip。

如果這些容器都不支援您的使用案例，或者如果您想要擴展它們，則 Braket Hybrid Jobs 支援使用您自己的自訂Docker容器映像執行混合式任務，或攜帶您自己的容器 (BYOC)。請確定這是適合您使用案例的正確功能。

在本節中：

- 何時將我自己的容器帶入正確的決策？
- 自攜容器的配方
- 在您自己的容器中執行 Braket 混合任務

何時將我自己的容器帶入正確的決策？

將您自己的容器 (BYOC) 帶到 Braket Hybrid Jobs，可讓您在封裝環境中安裝自己的軟體，藉此靈活地使用自己的軟體。根據您的特定需求，可能有一些方式可以實現相同的彈性，而無需完成完整BYOC Docker建置 - Amazon ECR 上傳 - 自訂映像 URI 週期。

> **Note**
>
> 如果您想要新增少量可公開取得的其他 Python 套件（通常少於 10 個），BYOC 可能不是正確的選擇。例如，如果您使用的是 PyPi。

在這種情況下，您可以使用其中一個預先建置的 Braket 映像，然後在任務提交時將`requirements.txt`檔案包含在來源目錄中。檔案會自動讀取，並正常`pip`安裝具有指定版本的套

件。如果您要安裝大量套件，則任務的執行時間可能會大幅增加。檢查 Python，如果適用，檢查您要用來測試軟體是否可運作的預先建置容器 CUDA 版本。

當您想要為任務指令碼使用非 Python 語言 （例如 C++ 或 Rust)，或想要使用透過 Braket 預先建置容器無法使用的 Python 版本時，BYOC 是必要的。如果出現下列情況，這也是不錯的選擇：

- 您使用的軟體具有授權金鑰，而且您需要向授權伺服器驗證該金鑰，才能執行軟體。使用 BYOC，您可以在Docker映像中嵌入授權金鑰，並包含要驗證授權金鑰的程式碼。
- 您正在使用不可公開使用的軟體。例如，軟體託管在您需要特定 SSH 金鑰才能存取的私有 GitLab 或 GitHub 儲存庫上。
- 您需要安裝未封裝在 Braket 提供的容器中的大型軟體套件。由於軟體安裝，BYOC 可讓您消除混合任務容器的長啟動時間。

BYOC 也可讓您使用軟體建置Docker容器並提供給使用者，藉此將自訂 SDK 或演算法提供給客戶。您可以在 Amazon ECR 中設定適當的許可來執行此操作。

> **Note**
>
> 您必須遵守所有適用的軟體授權。

自攜容器的配方

在本節中，我們提供step-by-step指南，說明您需要bring your own container (BYOC)執行的 Braket Hybrid Jobs – 指令碼、檔案，以及結合這些指令碼、檔案和步驟，以啟動和執行您的自訂Docker映像。兩個常見案例的配方：

1. 在Docker映像中安裝其他軟體，並在任務中僅使用 Python 演算法指令碼。
2. 使用非 Python 語言撰寫的演算法指令碼搭配混合任務，或 x86 以外的 CPU 架構。

對於案例 2，定義容器項目指令碼更為複雜。

當 Braket 執行您的混合任務時，它會啟動請求的 Amazon EC2 執行個體數量和類型，然後執行 Docker映像 URI 輸入指定的映像，以在其上建立任務。使用 BYOC 功能時，您可以指定私有 Amazon ECR 儲存庫中託管的映像 URI，而此儲存庫具有讀取存取權。Braket Hybrid Jobs 使用該自訂映像來執行任務。

建置可與混合任務搭配使用之Docker映像所需的特定元件。如果您不熟悉撰寫和建置 `Dockerfiles`，請參閱 Dockerfile 文件和 Amazon ECR CLI 文件。

使用要求：

- Dockerfile 的基礎映像
- （選用）修改過的容器進入點指令碼
- 使用 安裝所需的軟體和容器指令碼 Dockerfile

Dockerfile 的基礎映像

如果您使用的是 Python，並且想要在 Braket 提供的容器中提供的軟體上安裝軟體，則基礎映像的選項是託管於 GitHub 儲存庫和 Amazon ECR 上的其中一個 Braket 容器映像。您需要向 Amazon ECR 進行身分驗證，才能提取映像並建置映像。例如，BYOC Docker 檔案的第一行可以是： `FROM [IMAGE_URI_HERE]`

接著，填寫其餘的 Dockerfile，以安裝和設定您要新增至容器的軟體。預先建置的 Braket 映像將已包含適當的容器進入點指令碼，因此您不需要擔心是否包含該指令碼。

如果您想要使用非 Python 語言，例如 C++、Rust 或 Julia，或者如果您想要為非 x86 CPU 架構建置映像，例如 ARM，您可能需要在裸機公有映像之上建置 。您可以在 Amazon Elastic Container Registry Public Gallery 找到許多這類影像。請務必選擇適用於 CPU 架構的 GPU，並視需要選擇您要使用的 GPU。

（選用）修改過的容器進入點指令碼

> **Note**
>
> 如果您只將其他軟體新增至預先建置的 Braket 映像，則可以略過本節。

若要在混合任務中執行非 Python 程式碼，請修改定義容器進入點的 Python 指令碼。例如，`braket_container.py`Amazon Braket Github 上的 python 指令碼。這是 Braket 預先建置的映像用來啟動演算法指令碼並設定適當環境變數的指令碼。容器進入點指令碼本身必須使用 Python，但可以啟動非 Python 指令碼。在預先建置的範例中，您可以看到 Python 演算法指令碼是以 Python 子程序或全新的程序啟動。透過修改此邏輯，您可以啟用進入點指令碼來啟動非 Python 演算法指令碼。例如，您可以修改 `thekick_off_customer_script()` 函數，根據副檔名結尾啟動 Rust 程序。

您也可以選擇撰寫全新的 `braket_container.py`。它應該將輸入資料、來源封存和其他必要的檔案從 Amazon S3 複製到容器中，並定義適當的環境變數。

使用 安裝所需的軟體和容器指令碼 **Dockerfile**

> **Note**
>
> 如果您使用預先建置的 Braket 映像作為Docker基礎映像，則容器指令碼已存在。

如果您在上一個步驟中建立了修改過的容器指令碼，則需要將其複製到容器中，並將環境變數定義為 SAGEMAKER_PROGRAM `braket_container.py`，或您已命名為新容器進入點指令碼的內容。

以下是`Dockerfile`可讓您在 GPU 加速任務執行個體上使用 Julia 的 範例：

```
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
```

此範例會下載並執行 提供的指令碼 AWS ，以確保符合所有相關開放原始碼授權。例如，透過正確歸因由 管理的任何已安裝程式碼MIT license。

如果您需要包含非公有程式碼，例如託管在私有 GitHub 或 GitLab 儲存庫中的程式碼，請勿在Docker映像中嵌入 SSH 金鑰來存取它。相反地，請在建置Docker Compose時使用 ，以允許 Docker 存取其建置所在的主機機器上的 SSH。如需詳細資訊，請參閱 Docker 中的安全使用 SSH 金鑰存取私有 Github 儲存庫指南。

建置和上傳Docker映像

使用正確定義的 Dockerfile，您現在可以依照步驟建立私有 Amazon ECR 儲存庫，如果其中尚未存在的話。您也可以建置、標記容器映像並將其上傳至儲存庫。

您已準備好建置、標記和推送映像。如需 選項的完整說明docker build和一些範例，請參閱 Docker 建置文件。

對於上述定義的範例檔案，您可以執行：

```
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
```

指派適當的 Amazon ECR 許可

Braket Hybrid Jobs Docker 映像必須在私有 Amazon ECR 儲存庫中託管。根據預設，私有 Amazon ECR 儲存庫不會提供對 Braket Hybrid Jobs IAM role或任何其他要使用您映像的使用者的讀取存取權，例如協作者或學生。您必須設定儲存庫政策來授予適當的許可。一般而言，只將許可授予您想要存取映像的特定使用者和IAM角色，而不是允許具有 的任何人image URI提取它們。

在您自己的容器中執行 Braket 混合任務

若要使用您自己的容器建立混合任務，AwsQuantumJob.create()請使用image_uri指定的引數呼叫 。您可以使用 QPU、隨需模擬器，或在 Braket Hybrid Jobs 提供的傳統處理器上於本機執行程式碼。我們建議您先在 SV1, DM1，然後再在真實的 QPU 上執行。 TN1

若要在傳統處理器上執行程式碼，請更新 來指定instanceCount您使用的 instanceType和 InstanceConfig。請注意，如果您指定 instance_count > 1，則需要確保您的程式碼可以在多個主機上執行。您可以選擇的執行個體數量上限為 5。例如：

```
job = AwsQuantumJob.create(
    source_module="source_dir",
    entry_point="source_dir.algorithm_script:start_here",
    image_uri="111122223333.dkr.ecr.us-west-2.amazonaws.com/my-byoc-container:latest",
    instance_config=InstanceConfig(instanceType="ml.p3.8xlarge", instanceCount=3),
    device="local:braket/braket.local.qubit",
    # ...)
```

> **Note**
>
> 使用裝置 ARN 追蹤您用作混合任務中繼資料的模擬器。可接受的值必須遵循格式 `device = "local:<provider>/<simulator_name>"`。請記住，`<provider>`和 `<simulator_name>` 只能由字母、數字、_、 -和 . 組成。字串限制為 256 個字元。
> 如果您計劃使用 BYOC 且未使用 Braket 開發套件建立量子任務，您應該將環境變數的值傳遞`AMZN_BRAKET_JOB_TOKEN`給`CreateQuantumTask`請求中的 `jobToken` 參數。如果沒有，則量子任務不會獲得優先順序，並以一般獨立量子任務計費。

直接使用 與混合任務互動 API

您可以使用 直接存取 Amazon Braket 混合任務並與之互動API。不過，API直接使用 時，無法使用預設值和便利方法。

> **Note**
>
> 我們強烈建議您使用 Amazon Braket Python SDK 與 Amazon Braket 混合任務互動。它提供方便的預設值和保護，協助您的混合式任務成功執行。

本主題涵蓋使用 的基本概念API。如果您選擇使用 API，請記住，此方法可能更為複雜，並準備好進行多次反覆運算，讓您的混合式任務得以執行。

若要使用 API，您的帳戶應該具有具有 `AmazonBraketFullAccess`受管政策的角色。

> **Note**
>
> 如需如何使用 `AmazonBraketFullAccess`受管政策取得角色的詳細資訊，請參閱啟用 Amazon Braket 頁面。

此外，您需要 執行角色。此角色將傳遞給 服務。您可以使用 Amazon Braket 主控台建立角色。使用許可和設定頁面上的執行角色索引標籤，為混合任務建立預設角色。

`CreateJob` API 需要您指定混合任務的所有必要參數。若要使用 Python，請將演算法指令碼檔案壓縮為 tar 套件，例如 input.tar.gz 檔案，然後執行下列指令碼。更新角括號 (<>) 中的程式碼部分，以符合您的帳戶資訊和指定混合任務啟動路徑、檔案和方法的進入點。

```
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
    },
)
```

建立混合任務後，您可以透過 `GetJob`API或 主控台存取混合任務詳細資訊。若要從執行`createJob`程式碼的 Python 工作階段取得混合式任務詳細資訊，如先前範例所示，請使用下列 Python 命令。

```
getJob = client.get_job(jobArn=job["jobArn"])
```

若要取消混合任務，`CancelJob`API請使用任務Amazon Resource Name的 () 呼叫 'JobArn'。

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

您可以使用 checkpointConfig 參數createJobAPI將檢查點指定為 的一部分。

```
checkpointConfig = {
    "localPath" : "/opt/omega/checkpoints",
    "s3Uri": f"s3://{bucket}/{s3_prefix}/checkpoints"
},
```

> **Note**
>
> localPath checkpointConfig不能以下列任一預留路徑開頭：/opt/ml、/tmp、 /opt/braket或 /usr/local/nvidia。

使用保留

預留可讓您獨家存取您選擇的量子裝置。您可以方便地排程保留，以便確切了解工作負載開始和結束執行的時間。預訂以 1 小時遞增提供，最多可提前 48 小時取消，無需額外費用。您可以選擇為即將到來的保留預先將量子任務和混合任務排入佇列，或在保留期間提交工作負載。

無論您在 Quantum Processing Unit (QPU) 上執行多少量子任務和混合任務，專用裝置存取的成本取決於保留的持續時間。

下列規定人數電腦可供保留：

- IonQ 的 Aria and Forte
- Rigetti 的 Ankaa-3
- IQM 的 Garnet 和 Emerald
- QuEra 的 Aquila

> **Note**
>
> 搭配 IonQ 裝置使用直接保留時，沒有閘道擷取限制，且錯誤緩解任務至少需要 500 個擷取。

何時使用保留

利用具有保留的專用裝置存取，可讓您方便且可預測地知道量子工作負載何時開始和結束執行。相較於隨需提交任務和混合任務，您不需要等待其他客戶任務的佇列。由於您在保留期間擁有裝置的專屬存取權，因此只有工作負載會在裝置上執行整個保留。

我們建議您將隨需存取用於研究的設計和原型設計階段，以便快速且經濟實惠地迭代演算法。準備好產生最終實驗結果後，請考慮在方便的時候安排裝置保留，以確保您可以滿足專案或發佈截止日期。當您想要在特定時間執行任務時，例如在量子電腦上執行即時示範或研討會時，我們也建議使用保留。

在本節中：

如何建立保留

若要建立保留，請依照下列步驟聯絡 Braket 團隊：

1. 開啟 Amazon Braket 主控台。
2. 在左側窗格中選擇 Braket Direct，然後在預留區段中，選擇預留裝置 。
3. 選取您要保留的裝置。
4. 提供您的聯絡資訊，包括名稱和電子郵件。請務必提供您定期檢查的有效電子郵件地址。
5. 在告訴我們工作負載的相關資訊下，提供使用預留執行工作負載的任何詳細資訊。例如，所需的保留長度、相關限制條件或所需的排程。
6. 如果您有興趣在確認保留後與 Braket 專家連線進行保留準備工作階段，可選擇是否選取我對準備工作階段感興趣。

您也可以依照下列步驟聯絡我們來建立保留：

1. 開啟 Amazon Braket 主控台。
2. 在左側窗格中選擇裝置，然後選擇您要保留的裝置。
3. 在摘要區段中，選擇預留裝置。
4. 請遵循先前程序中的步驟 4-6。

提交表單後，您會收到來自 Braket 團隊的電子郵件，其中包含建立保留的後續步驟。確認保留後，您將透過電子郵件收到保留 ARN。

> **Note**
>
> 只有在您收到保留 ARN 後，才會確認您的保留。

保留以最少 1 小時的增量提供，某些裝置可能會有額外的保留長度限制（包括最短和最長保留持續時間）。Braket 團隊會在確認保留之前與您分享任何相關資訊。

如果您表示對保留準備工作階段感興趣，則 Braket 團隊將透過您的電子郵件與您聯絡，以安排與 Braket 專家的 30 分鐘工作階段。

在保留期間執行量子任務

從建立保留取得有效的保留 ARN 後，您可以建立在保留期間執行的量子任務。這些任務會保持 `QUEUED` 狀態，直到您的保留開始為止。

> **Note**
>
> 預留是 AWS 帳戶和裝置特定的。只有建立保留 AWS 的帳戶才能使用您的保留 ARN。

> **Note**
>
> 使用保留 ARN 提交的任務和任務沒有佇列可見性，因為只有您的任務會在保留期間執行。

您可以使用 Braket、、 QiskitPennyLane或直接搭配 boto3 (使用 Boto3) Python SDKs來建立量子任務。若要使用保留，您必須擁有 Amazon Braket Python SDK的 1.79.0 版或更新版本。您可以使用下列程式碼更新至最新的 Braket SDK、Qiskit供應商和PennyLane外掛程式。

```
pip install --upgrade amazon-braket-sdk amazon-braket-pennylane-plugin qiskit-braket-provider
```

使用`DirectReservation`內容管理員執行任務

在您的排程保留內執行任務的建議方法是使用DirectReservation內容管理員。透過指定您的目標裝置和保留 ARN，內容管理員可確保在 Python with陳述式中建立的所有任務都以裝置獨佔存取權執行。

首先，定義量子電路和裝置。然後使用保留內容並執行任務。

```
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)
```

您可以使用 PennyLane和 Qiskit 外掛程式在保留中建立量子任務，只要DirectReservation內容在建立量子任務時處於作用中狀態即可。例如，透過 Qiskit-Braket 提供者，您可以執行任務，如下所示。

```
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)
```

同樣地，下列程式碼會使用 Braket-PennyLane 外掛程式在保留期間執行電路。

```
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()
```

手動設定保留內容

或者，您可以使用下列程式碼手動設定保留內容。

```
# set reservation context
reservation = DirectReservation(device, reservation_arn="<my_reservation_arn>").start()

# run circuit during reservation
task = device.run(bell, shots=100)
```

這非常適合可在第一個儲存格中執行內容的 Jupyter 筆記本，且所有後續任務將在保留中執行。

> **Note**
>
> 包含`.start()`呼叫的儲存格應該只執行一次。

若要切換回隨需模式：重新啟動 Jupyter 筆記本，或呼叫以下內容將內容變更回隨需模式。

```
reservation.stop()  # unset reservation context
```

> **Note**
>
> 保留有排定的開始和結束時間 （請參閱建立保留）。`reservation.start()` 和 `reservation.stop()`方法不會開始或終止保留。這些是修改所有後續量子任務以在保留期間執行的方法。這些方法不會影響排定的保留時間。

建立任務時明確傳遞保留 ARN

在保留期間建立任務的另一個方法是在呼叫 時明確傳遞保留 ARN`device.run()`。

```
task = device.run(bell, shots=100, reservation_arn="<my_reservation_arn>")
```

此方法會直接將量子任務與保留 ARN 建立關聯，確保在保留期間執行。針對此選項，將保留 ARN 新增至您計劃在保留期間執行的每個任務。此外，請檢查在 Qiskit或 中建立的任務PennyLane是否使用正確的保留 ARN。由於這些額外的考量，建議使用前面兩種方式。

直接使用 boto3 時，請在建立任務時傳遞保留 ARN 做為關聯。

```
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)
```

在保留期間執行混合式任務

將 Python 函數作為混合任務執行後，您可以透過傳遞`reservation_arn`關鍵字引數在保留中執行混合任務。混合任務中的所有任務都會使用保留 ARN。重要的是， 的混合任務`reservation_arn`只會在保留開始時啟動傳統運算。

> **Note**
>
> 在保留期間執行的混合任務只會在預留裝置上成功執行規定人數任務。嘗試使用不同的隨需 Braket 裝置將導致錯誤。如果您需要在相同混合任務中的隨需模擬器和預留裝置上執行任務，請`DirectReservation`改用 。

下列程式碼示範如何在保留期間執行混合式任務。

```
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)
```

對於使用 Python 指令碼的混合式任務（請參閱開發人員指南中建立您的第一個混合式任務一節），您可以在建立任務時傳遞`reservation_arn`關鍵字引數，在保留中執行這些任務。

```
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>"
)
```

保留結束時會發生什麼情況

保留結束後，您就不再擁有裝置的專用存取權。使用此保留排入佇列的任何剩餘工作負載都會自動取消。

> **Note**
>
> 保留結束時處於 RUNNING 狀態的任何任務都會取消。我們建議您使用檢查點，在方便的時候儲存和重新啟動任務。

無法延長持續保留，例如保留開始之後和保留結束之前，因為每個保留都代表獨立的專用裝置存取。例如，兩個back-to-back保留會被視為個別保留，而第一個保留中的任何待定任務都會自動取消。它們不會在第二個保留中繼續。

> **Note**
>
> 預留代表 AWS 帳戶的專用裝置存取。即使裝置保持閒置狀態，其他客戶也無法使用它。因此，無論使用時間為何，都會向您收取預留時間長度的費用。

取消或重新排程現有的保留

您可以在排定的保留開始時間前至少 48 小時內取消保留。若要取消，請以取消請求回應您收到的保留確認電子郵件。

若要重新排程，您必須取消現有的保留，然後建立新的保留。

錯誤緩解技術

Quantum 錯誤緩解是一組旨在降低量子電腦中錯誤影響的技術。

Quantum 裝置會受到環境雜訊的影響，進而降低執行的運算品質。雖然容錯量子運算向此問題承諾解決方案，但目前的量子裝置受限於 qubit 數量和相對較高的錯誤率。為了在短期內解決這個問題，研究人員正在調查方法，以提高雜訊量子運算的準確性。這種方法稱為量子錯誤緩解，涉及使用各種技術從雜訊測量資料中擷取最佳訊號。

在本節中：

- IonQ 裝置上的錯誤緩解技術

IonQ 裝置上的錯誤緩解技術

錯誤緩解包括執行多個實體電路，並結合其測量結果來改善結果。

> **Note**
>
> 對於所有 IonQ的裝置：使用隨需模型時，有 100 萬個閘道截圖限制，以及至少 2，500 個錯誤緩解任務的鏡頭。對於直接保留，沒有閘道擷取限制，且錯誤緩解任務至少需要 500 個擷取。

緩和

IonQ 裝置具有稱為消弱的錯誤緩解方法。

偏差會將電路映射到多個變體中，這些變體作用於不同的 qubit 排列或具有不同的閘道分解。這可減少系統性錯誤的影響，例如閘道過度旋轉或單一故障的 qubit，方法是使用不同的電路實作，否則可能會使測量結果偏差。這會產生額外的額外負荷，以校正多個 qubit 和閘道。

如需消除偏差的詳細資訊，請參閱透過對稱增強量子電腦效能。

> **Note**
>
> 使用去偏差至少需要 2500 個鏡頭。

您可以使用下列程式碼在IonQ裝置上執行具有解除偏差的量子任務：

```
from braket.aws import AwsDevice
from braket.circuits import Circuit
from braket.error_mitigation import Debias

# choose an IonQ device
device = AwsDevice("arn:aws:braket:us-east-1::device/qpu/ionq/Aria-1")
circuit = Circuit().h(0).cnot(0, 1)

task = device.run(circuit, shots=2500, device_parameters={"errorMitigation": Debias()})

result = task.result()
print(result.measurement_counts)
>>> {"00": 1245, "01": 5, "10": 10 "11": 1240} # result from debiasing
```

當量子任務完成時，您可以查看量子任務的測量機率和任何結果類型。所有變體的測量機率和計數會彙總為單一分佈。電路中指定的任何結果類型，例如預期值，都是使用彙總測量計數來計算。

銳利化

您也可以存取使用稱為銳化的不同後製處理策略所計算的測量機率。銳利化會比較每個變體的結果，並捨棄不一致的鏡頭，有利於變體中最有可能的測量結果。如需詳細資訊，請參閱透過對稱增強量子電腦效能。

重要的是，銳化會假設輸出分佈的形式稀疏，具有少數高機率狀態和許多零機率狀態。如果此假設無效，可能會扭曲機率分佈。

您可以從 `GateModelTaskResult` Braket Python SDK 中 的 `additional_metadata` 欄位中的銳化分佈存取機率。請注意，銳化不會傳回測量計數，而是傳回重新標準化的機率分佈。下列程式碼片段顯示如何在銳利化後存取 分佈。

```
print(result.additional_metadata.ionqMetadata.sharpenedProbabilities)
>>> {"00": 0.51, "11": 0.549} # sharpened probabilities
```

Amazon Braket 故障診斷

使用本節中的疑難排解資訊和解決方案，以協助解決 Amazon Braket 的問題。

在本節中：

AccessDeniedException

如果您在啟用或停用 Braket 時收到 AccessDeniedException，您可能會嘗試在受限角色沒有存取權的區域中啟用或停用 Braket。

在這種情況下，請聯絡您的內部 AWS 管理員，以了解下列哪些條件適用：

- 如果有禁止存取區域的角色限制。
- 如果您嘗試使用的角色允許使用 Braket。

如果您的角色在使用 Braket 時無法存取指定區域，則您將無法在該特定區域中使用裝置。

呼叫 CreateQuantumTask 操作時發生錯誤 (ValidationException)

如果您收到類似 的錯誤：`An error occurred (ValidationException) when calling the CreateQuantumTask operation: Caller doesn't have access to amazon-braket-…` 請檢查您指的是現有的 s3_folder。Braket 不會為您自動建立新的 Amazon S3 儲存貯體和字首。

如果您API直接存取 並收到類似 的錯誤：`Failed to create quantum task: Caller doesn't have access to s3://MY_BUCKET`請檢查您是否未包含在 Amazon S3 儲存貯體路徑`s3://`中。

SDK 功能無法運作

您的 Python 版本必須為 3.9 或更新版本。對於 Amazon Braket 混合任務，我們建議使用 Python 3.10。

確認您的 SDK 和結構描述是up-to-date。若要從筆記本或您的 python 編輯器更新 SDK，請執行下列命令：

```
pip install amazon-braket-sdk --upgrade --upgrade-strategy eager
```

若要更新結構描述，請執行下列命令：

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

如果您是從自己的用戶端存取 Amazon Braket，請確認您的AWS 區域已設定為 Amazon Braket 支援的區域。

由於 ServiceQuotaExceededException，混合任務失敗

如果您超過目標模擬器裝置的並行量子任務限制，則針對 Amazon Braket 模擬器執行量子任務的混合任務可能無法建立。如需服務限制的詳細資訊，請參閱配額主題。

如果您在帳戶中的多個混合式任務中對模擬器裝置執行並行任務，您可能會遇到此錯誤。

若要查看針對特定模擬器裝置的並行量子任務數量，請使用 `search-quantum-tasks` API，如下列程式碼範例所示。

```
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
```

您也可以使用 Amazon CloudWatch 指標來檢視針對裝置建立的量子任務：Raket > By Device。

若要避免執行這些錯誤：

1. 請求增加模擬器裝置的並行量子任務數量的服務配額。這僅適用於SV1裝置。
2. 處理程式碼中的`ServiceQuotaExceeded`例外狀況，然後重試。

元件在筆記本執行個體中停止運作

如果筆記本的某些元件停止運作，請嘗試下列動作：

1. 將您建立或修改的任何筆記本下載至本機磁碟機。
2. 停止您的筆記本執行個體。
3. 刪除您的筆記本執行個體。
4. 使用不同的名稱建立新的筆記本執行個體。
5. 將筆記本上傳至新執行個體。

故障診斷 OpenQASM

本節提供使用 OpenQASM 3.0 遇到錯誤時可能有用的疑難排解指標。

在本節中：

- 包含陳述式錯誤
- 非連續qubits錯誤
- 混合實體qubits與虛擬qubits錯誤
- 在相同的程式錯誤qubits中請求結果類型和測量
- 超過傳統和qubit註冊限制的錯誤
- 未出現逐字 pragma 錯誤的方塊
- 逐字方塊缺少原生閘道錯誤
- 逐字方塊缺少實體qubits錯誤
- 逐字 pragma 缺少 "braket" 錯誤
- 單一 qubits無法編製索引錯誤
- 兩個qubit閘道qubits中的實體未連線錯誤
- 本機模擬器支援警告

包含陳述式錯誤

Braket 目前沒有要包含在 OpenQASM 程式中的標準閘道程式庫檔案。例如，下列範例會引發剖析器錯誤。

```
OPENQASM 3;
include "standardlib.inc";
```

此程式碼會產生錯誤訊息：`No terminal matches '"' in the current parser context, at line 2 col 17.`

非連續qubits錯誤

在裝置功能`true`中`requiresContiguousQubitIndices`設定為 的裝置qubits上使用不連續的 會導致錯誤。

在模擬器和 上執行量子任務時IonQ，下列程式會觸發錯誤。

```
OPENQASM 3;

qubit[4] q;

h q[0];
cnot q[0], q[2];
cnot q[0], q[3];
```

此程式碼會產生錯誤訊息：`Device requires contiguous qubits. Qubit register q has unused qubits q[1], q[4].`

混合實體qubits與虛擬qubits錯誤

不允許在相同程式qubits中混合實體qubits與虛擬 ，並導致錯誤。下列程式碼會產生錯誤。

```
OPENQASM 3;

qubit[2] q;
cnot q[0], $1;
```

此程式碼會產生錯誤訊息：`[line 4] mixes physical qubits and qubits registers.`

在相同的程式錯誤qubits中請求結果類型和測量

請求在相同程式中qubits明確測量的結果類型和 會導致錯誤。下列程式碼會產生錯誤。

```
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])
```

此程式碼會產生錯誤訊息：`Qubits should not be explicitly measured when result types are requested.`

超過傳統和qubit註冊限制的錯誤

只允許一個傳統註冊和一個qubit註冊。下列程式碼會產生錯誤。

```
OPENQASM 3;

qubit[2] q0;
qubit[2] q1;
```

此程式碼會產生錯誤訊息：`[line 4] cannot declare a qubit register. Only 1 qubit register is supported.`

未出現逐字 pragma 錯誤的方塊

所有方塊前面都必須加上逐字法。下列程式碼會產生錯誤。

```
box{
rx(0.5) $0;
}
```

此程式碼會產生錯誤訊息：`In verbatim boxes, native gates are required. x is not a device native gate.`

逐字方塊缺少原生閘道錯誤

逐字方塊應具有原生閘道和實體 qubits。下列程式碼會產生原生閘道錯誤。

```
#pragma braket verbatim
box{
x $0;
}
```

此程式碼會產生錯誤訊息： `In verbatim boxes, native gates are required. x is not a device native gate.`

逐字方塊缺少實體qubits錯誤

逐字方塊必須具有實體 qubits。下列程式碼會產生缺少的實體qubits錯誤。

```
qubit[2] q;

#pragma braket verbatim
box{
rx(0.1) q[0];
}
```

此程式碼會產生錯誤訊息： `Physical qubits are required in verbatim box.`

逐字 pragma 缺少 "braket" 錯誤

您必須在逐字法中包含「括號」。下列程式碼會產生錯誤。

```
#pragma braket verbatim          // Correct
#pragma verbatim                 // wrong
```

此程式碼會產生錯誤訊息： `You must include “braket” in the verbatim pragma`

單一 qubits無法編製索引錯誤

單一 qubits無法編製索引。下列程式碼會產生錯誤。

```
OPENQASM 3;
```

```
qubit q;
h q[0];
```

此程式碼會產生錯誤：`[line 4] single qubit cannot be indexed.`

不過，單一qubit陣列的索引如下所示：

```
OPENQASM 3;

qubit[1] q;
h q[0];   // This is valid
```

兩個qubit閘道qubits中的實體未連線錯誤

若要使用實體 qubits，請先檢查 qubits 以確認裝置使用實體，`device.properties.action[DeviceActionType.OPENQASM].supportPhysicalQubits`然後檢查 `device.properties.paradigm.connectivity.connectivityGraph`或 來驗證連線圖表`device.properties.paradigm.connectivity.fullyConnected`。

```
OPENQASM 3;

cnot $0, $14;
```

此程式碼會產生錯誤訊息：`[line 3] has disconnected qubits 0 and 14`

本機模擬器支援警告

`LocalSimulator` 支援 OpenQASM 中的進階功能，這些功能可能無法在 QPUs 或隨需模擬器上使用。如果您的程式包含 特有的語言功能`LocalSimulator`，如下列範例所示，您將會收到警告。

```
qasm_string = """
qubit[2] q;

h q[0];
ctrl @ x q[0], q[1];
"""
qasm_program = Program(source=qasm_string)
```

此程式碼會產生警告：`此程式只會使用 LocalSimulator 中支援的 OpenQASM 語言功能。QPUs 或隨需模擬器可能不支援其中一些功能。

如需支援的 OpenQASM 功能的詳細資訊，請探索本機模擬器上 OpenQASM 的進階功能支援頁面。

Amazon Braket 的安全性

的雲端安全 AWS 是最高優先順序。身為 AWS 客戶，您可以受益於資料中心和網路架構，這些架構是為了滿足最安全敏感組織的需求而建置。

安全性是 AWS 與您之間的共同責任。共同責任模型將其描述為雲端的安全性和雲端中的安全性：

- 雲端的安全性 – AWS 負責保護在 中執行 AWS 服務的基礎設施 AWS 雲端。 AWS 也為您提供可安全使用的服務。作為AWS 合規計劃的一部分，第三方稽核人員會定期測試和驗證我們安全的有效性。若要了解適用於 Amazon Braket 的合規計劃，請參閱AWS 合規計劃的 服務範圍。
- 雲端的安全性 – 您的責任取決於您使用 AWS 的服務。您也必須對其他因素負責，包括資料的機密性、您公司的要求和適用法律和法規。

本文件可協助您了解如何在使用 Braket 時套用共同責任模型。下列主題說明如何設定 Braket 以符合您的安全與合規目標。您也會了解如何使用其他 AWS 服務來協助您監控和保護 Braket 資源。

在本節中：

- 共同的安全責任
- 資料保護
- 資料保留
- 管理對 Amazon Braket 的存取
- Amazon Braket 服務連結角色
- Amazon Braket 的合規驗證
- Amazon Braket 中的基礎設施安全性
- Amazon Braket 硬體供應商的安全性
- Amazon Braket 的 Amazon VPC 端點

共同的安全責任

安全性是 AWS 與您之間的共同責任。共同責任模型 將此描述為雲端的安全和雲端內的安全：

- 雲端的安全性 – AWS 負責保護在 AWS 服務 中執行的基礎設施 AWS 雲端。 AWS 也為您提供可安全使用的服務。在 AWS 合規計劃中，第三方稽核員會定期測試並驗證我們的安全功效。若要了解適用於 Amazon Braket 的合規計劃，請參閱AWS 合規計劃範圍內的服務。

- 雲端的安全性 – 您負責控制在此 AWS 基礎設施上託管的內容。此內容包含 AWS 服務 您使用之 的安全組態和管理任務。

資料保護

AWS 共同責任模型適用於 Amazon Braket 中的資料保護。如此模型所述， AWS 負責保護執行所有的全域基礎設施 AWS 雲端。您負責維護在此基礎設施上託管內容的控制權。您也同時負責所使用 AWS 服務 的安全組態和管理任務。如需資料隱私權的詳細資訊，請參閱資料隱私權常見問答集。如需有關歐洲資料保護的相關資訊，請參閱 AWS 安全性部落格上的 AWS 共同的責任模型和 GDPR 部落格文章。

基於資料保護目的，我們建議您保護 AWS 帳戶 登入資料，並使用 AWS IAM Identity Center 或 AWS Identity and Access Management (IAM) 設定個別使用者。如此一來，每個使用者都只會獲得授與完成其任務所必須的許可。我們也建議您採用下列方式保護資料：

- 每個帳戶均要使用多重要素驗證 (MFA)。
- 使用 SSL/TLS 與 AWS 資源通訊。我們需要 TLS 1.2 並建議使用 TLS 1.3。
- 使用 設定 API 和使用者活動記錄 AWS CloudTrail。如需有關使用 CloudTrail 追蹤擷取 AWS 活動的資訊，請參閱AWS CloudTrail 《 使用者指南》中的使用 CloudTrail 追蹤。
- 使用 AWS 加密解決方案，以及其中的所有預設安全控制 AWS 服務。
- 使用進階的受管安全服務 (例如 Amazon Macie)，協助探索和保護儲存在 Amazon S3 的敏感資料。
- 如果您在 AWS 透過命令列界面或 API 存取 時需要 FIPS 140-3 驗證的密碼編譯模組，請使用 FIPS 端點。如需有關 FIPS 和 FIPS 端點的更多相關資訊，請參閱聯邦資訊處理標準 (FIPS) 140-3。

我們強烈建議您絕對不要將客戶的電子郵件地址等機密或敏感資訊，放在標籤或自由格式的文字欄位中，例如名稱欄位。這包括當您使用 Amazon Braket 或使用主控台、API AWS CLI或 AWS SDKs的其他 AWS 服務 時。您在標籤或自由格式文字欄位中輸入的任何資料都可能用於計費或診斷日誌。如果您提供外部伺服器的 URL，我們強烈建議請勿在驗證您對該伺服器請求的 URL 中包含憑證資訊。

資料保留

90 天後，Amazon Braket 會自動移除與量子任務相關聯的所有量子任務 IDs 和其他中繼資料。由於此資料保留政策，這些任務和結果無法再透過從 Amazon Braket 主控台進行搜尋來擷取，但會保留在您的 S3 儲存貯體中。

如果您需要存取存放在 S3 儲存貯體中超過 90 天的歷史量子任務和結果，您必須單獨記錄任務 ID 和與該資料相關聯的其他中繼資料。請務必在 90 天前儲存資訊。您可以使用儲存的資訊來擷取歷史資料。

管理對 Amazon Braket 的存取

本章說明執行 Amazon Braket 或限制特定使用者和角色存取所需的許可。您可以授予（或拒絕）帳戶中任何使用者或角色所需的許可。若要這樣做，請將適當的 Amazon Braket 政策連接到您帳戶中的該使用者或角色，如以下各節所述。

作為先決條件，您必須啟用 Amazon Braket。若要啟用 Braket，請務必以具有 (1) 管理員許可或 (2) 獲指派 AmazonBraketFullAccess 政策的使用者或角色身分登入，並具有建立 Amazon Simple Storage Service (Amazon S3) 儲存貯體的許可。

在本節中：

- Amazon Braket 資源
- 筆記本和角色
- AWS Amazon Braket 的 受管政策
- 限制使用者存取特定裝置
- 限制使用者存取特定筆記本執行個體
- 限制使用者存取特定 S3 儲存貯體

Amazon Braket 資源

Braket 會建立一種類型的資源：quantum-task 資源。此 AWS 資源類型的資源名稱 (ARN) 如下所示：

- 資源名稱：AWS：Service：：Braket
- ARN Regex：arn：${Partition}：braket：${Region}：${Account}：quantum-task/${RandomId}

筆記本和角色

您可以在 Braket 中使用 noteboook 資源類型。筆記本是 Braket 可以共用的 Amazon SageMaker AI 資源。若要搭配 Braket 使用筆記本，您必須指定名稱開頭為 的 IAM 角色`AmazonBraketServiceSageMakerNotebook`。

若要建立筆記本，您必須使用具有管理員許可或具有下列內嵌政策的角色。

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*"
          ]
        }
      }
    }
  ]
```

```
}
```

若要建立角色，請遵循建立筆記本頁面中提供的步驟，或讓您的管理員為您建立。確定已連接 AmazonBraketFullAccess 政策。

建立角色之後，您可以為未來啟動的所有筆記本重複使用該角色。

AWS Amazon Braket 的 受管政策

AWS 受管政策是由 AWS AWS 受管政策建立和管理的獨立政策旨在為許多常用案例提供許可，以便您可以開始將許可指派給使用者、群組和角色。

請記住， AWS 受管政策可能不會授予特定使用案例的最低權限許可，因為這些許可可供所有 AWS 客戶使用。我們建議您定義使用案例專屬的客戶管理政策，以便進一步減少許可。

您無法變更 AWS 受管政策中定義的許可。如果 AWS 更新 AWS 受管政策中定義的許可，則更新會影響政策連接的所有主體身分（使用者、群組和角色）。 AWS 服務 當新的 啟動或新的 API 操作可供現有 服務使用時， AWS 最有可能更新 AWS 受管政策。

如需詳細資訊，請參閱《IAM 使用者指南》中的 AWS 受管政策。

主題

- AWS 受管政策：AmazonBraketFullAccess
- AWS 受管政策：AmazonBraketJobsExecutionPolicy
- AWS 受管政策：AmazonBraketServiceRolePolicy
- AWS 受管政策的 Amazon Braket 更新

AWS 受管政策：AmazonBraketFullAccess

AmazonBraketFullAccess 政策會授予 Amazon Braket 操作的許可，包括這些任務的許可：

- 從 Amazon Elastic Container Registry 下載容器 – 讀取和下載用於 Amazon Braket Hybrid Jobs 功能的容器映像。容器必須符合格式 "arn：aws：ecr：：repository/amazon-braket"。
- 保留 AWS CloudTrail 日誌 – 針對所有描述、取得和列出除了開始和停止查詢、測試指標篩選條件和篩選日誌事件以外的動作。 AWS CloudTrail 日誌檔案包含您帳戶中發生的所有 Amazon Braket API 活動記錄。

- 利用角色來控制資源 – 在帳戶中建立服務連結角色。服務連結角色可以代表您存取 AWS 資源。它只能由 Amazon Braket 服務使用。此外，將 IAM 角色傳遞至 Amazon Braket `CreateJob`API並建立角色，並將範圍為 AmazonBraketFullAccess 的政策連接至角色。
- 建立日誌群組、日誌事件和查詢日誌群組，以維護帳戶的用量日誌檔案 – 建立、存放和檢視帳戶中 Amazon Braket 用量的記錄資訊。查詢混合任務日誌群組上的指標。包含適當的 Braket 路徑，並允許放置日誌資料。在 CloudWatch 中放置指標資料。
- 在 Amazon S3 儲存貯體中建立和存放資料，並列出所有儲存貯體 – 若要建立 S3 儲存貯體，請列出您帳戶中的 S3 儲存貯體，並將物件放入您帳戶中名稱開頭為 amazon-braket- 的任何儲存貯體，並從中取得物件。Braket 需要這些許可，才能將包含已處理量子任務結果的檔案放入儲存貯體，並從儲存貯體擷取它們。
- 傳遞 IAM 角色 – 將 IAM 角色傳遞至 `CreateJob` API。
- Amazon SageMaker AI 筆記本 – 建立和管理範圍為資源的SageMaker筆記本執行個體，來源為 "arn：aws：sagemaker：：notebook-instance/amazon-braket-"。
- 驗證服務配額 – 若要建立 SageMaker AI 筆記本和 Amazon Braket Hybrid 任務，您的資源計數不得超過您帳戶的配額。
- 檢視產品定價 – 在提交工作負載之前，先檢閱並規劃量子硬體成本。

若要檢視此政策的許可，請參閱 AWS 受管政策參考中的 AmazonBraketFullAccess。

AWS 受管政策：AmazonBraketJobsExecutionPolicy

AmazonBraketJobsExecutionPolicy 政策會授予 Amazon Braket 混合任務中使用的執行角色許可，如下所示：

- 從 Amazon Elastic Container Registry 下載容器 - 讀取和下載用於 Amazon Braket Hybrid Jobs 功能的容器映像的許可。容器必須符合格式 "arn：aws：ecr：*：*：repository/amazon-braket*"。
- 建立日誌群組和日誌事件和查詢日誌群組，以維護帳戶的用量日誌檔案 – 建立、存放和檢視帳戶中 Amazon Braket 用量的記錄資訊。查詢混合任務日誌群組上的指標。包含適當的 Braket 路徑，並允許放置日誌資料。在 CloudWatch 中放置指標資料。
- 將資料儲存在 Amazon S3 儲存貯體中 – 列出您帳戶中的 S3 儲存貯體、將物件放入您的帳戶中以 amazon-braket- 開頭的任何儲存貯體，並從其名稱中取得物件。Braket 需要這些許可，才能將包含已處理量子任務結果的檔案放入儲存貯體，並從儲存貯體擷取它們。
- 傳遞 IAM 角色 – 將 IAM 角色傳遞至 CreateJobAPI。角色必須符合 arn：aws：iam：：*:role/service-role/AmazonBraketJobsExecutionRole* 格式。

若要檢視此政策的許可，請參閱 AWS 受管政策參考中的 AmazonBraketJobsExecutionPolicy。

AWS 受管政策：AmazonBraketServiceRolePolicy

AmazonBraketServiceRolePolicy 政策會授予 Amazon Braket 操作的許可，包括這些任務的許可：

- Amazon S3 – 列出您帳戶中儲存貯體，並將物件放入帳戶中名稱開頭為 的任何儲存貯體，並從中取得物件的許可`amazon-braket-`。
- Amazon CloudWatch Logs – 列出和建立日誌群組、建立相關日誌串流，並將事件放入為 Amazon Braket 建立的日誌群組的許可。

如需服務連結角色的詳細資訊，請參閱 Amazon Braket 服務連結角色。

若要檢視此政策的許可，請參閱 AWS 受管政策參考中的 AmazonBraketServiceRolePolicy。

AWS 受管政策的 Amazon Braket 更新

下表提供有關從此服務開始追蹤 Amazon Braket AWS 受管政策更新的詳細資訊。

變更	Description	日期
AmazonBraketServiceRolePolicy - 資源管理政策	已將「aws：ResourceAccount」：「${aws：PrincipalAccount}」條件範圍新增至 Amazon S3 和 CloudWatch 日誌動作。	2025 年 7 月 11 日
AmazonBraketFullAccess - Braket 的完整存取政策	新增「pricing：GetProducts」動作。	2025 年 4 月 14 日
AmazonBraketFullAccess - Braket 的完整存取政策	已將「aws：ResourceAccount」：「${aws：PrincipalAccount}」條件範圍新增至 S3 動作。	2025 年 3 月 7 日
AmazonBraketFullAccess - Braket 的完整存取政策	新增 servicequotas：GetServiceQuota 和 cloudwatch：GetMetricData 動作。	2023 年 3 月 24 日

變更	Description	日期
AmazonBraketFullAccess - Braket 的完整存取政策	新增 s3：ListAllMyBuckets 許可，以檢視和檢查使用的 Amazon S3 儲存貯體。	2022 年 3 月 31 日
AmazonBraketFullAccess - Braket 的完整存取政策	Braket 調整了 AmazonBraketFullAccess 的 iam：PassRole 許可，以包含 `service-role/` 路徑。	2021 年 11 月 29 日
AmazonBraketJobsExecutionPolicy - Amazon Braket 混合任務的混合任務執行政策	Braket 已更新混合任務執行角色 ARN，以包含 `service-role/` 路徑。	2021 年 11 月 29 日
Braket 已開始追蹤變更	Braket 開始追蹤其 AWS 受管政策的變更。	2021 年 11 月 29 日

限制使用者存取特定裝置

若要限制特定 Braket 裝置的使用者存取，您可以將拒絕許可政策新增至特定IAM角色。

下列動作可以受到限制：

- `CreateQuantumTask` - 拒絕在指定裝置上建立量子任務。
- `CreateJob` - 拒絕在特定裝置上建立混合任務。
- `GetDevice` - 拒絕取得指定裝置的詳細資訊。

下列範例會限制存取 的所有 QPUs AWS 帳戶 `123456789012`。

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
    "Effect": "Deny",
```

```
    "Action": [
      "braket:CreateQuantumTask",
      "braket:CreateJob",
      "braket:GetDevice"
    ],
    "Resource": [
    "arn:aws:braket:*:*:device/qpu/*"
    ],
    "Condition": {
            "StringEquals": {
                "aws:PrincipalAccount": "123456789012"
            }
      }
    }
  ]
}
```

> **Note**
>
> 從政策中排除`braket:GetDevice`動作，以透過 Braket 主控台啟用使用者對裝置屬性的讀取存取權，例如裝置可用性、校正資料和定價。

若要調整此程式碼，請將受限裝置Amazon的資源編號 (ARN) 取代為上一個範例中顯示的字串。此字串提供 資源值。在 Braket 中，裝置代表您可以呼叫以執行量子任務的 QPU 或模擬器。可用的裝置會列在 裝置頁面上。有兩個結構描述用於指定對這些裝置的存取：

- `arn:aws:braket:<region>:*:device/qpu/<provider>/<device_id>`
- `arn:aws:braket:<region>:*:device/quantum-simulator/<provider>/<device_id>`

以下是各種裝置存取類型的範例

- 若要選取所有區域的所有 QPUs： `arn:aws:braket:*:*:device/qpu/*`
- 僅選取 us-west-2 區域中的所有 QPUs： `arn:aws:braket:us-west-2:*:device/qpu/*`
- 相當於，若要選取 us-west-2 區域中的所有 QPUs（因為裝置是服務資源，而不是客戶資源）：`arn:aws:braket:us-west-2:*:device/qpu/*`
- 若要限制對所有隨需模擬器裝置的存取： `arn:aws:braket:*:*:device/quantum-simulator/*`

- 若要限制從特定提供者存取裝置（例如，存取RigettiQPU裝置）：`arn:aws:braket:*:*:device/qpu/rigetti/*`
- 若要限制對TN1裝置的存取： `arn:aws:braket:*:*:device/quantum-simulator/amazon/tn1`
- 若要限制對所有Create動作的存取： `braket:Create*`

限制使用者存取特定筆記本執行個體

若要限制特定使用者存取特定 Braket 筆記本執行個體，您可以將拒絕許可政策新增至特定角色、使用者或群組。

下列範例使用政策變數來有效限制 中啟動、停止和存取特定筆記本執行個體的許可 AWS 帳戶 123456789012，該執行個體是根據應具有存取權的使用者命名（例如，使用者Alice可存取名為 的筆記本執行個體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}*"
      ]
    }
  ]
}
```

限制使用者存取特定 S3 儲存貯體

若要限制特定使用者存取特定 Amazon S3 儲存貯體，您可以將拒絕政策新增至特定角色、使用者或群組。

下列範例會限制擷取物件並將物件放入特定S3儲存貯體 (`arn:aws:s3:::amazon-braket-us-east-1-123456789012-Alice`) 的許可，也會限制這些物件的清單。

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/*"
      ]
    }
  ]
}
```

若要限制存取特定筆記本執行個體的 儲存貯體，您可以將上述政策新增至筆記本執行角色。

Amazon Braket 服務連結角色

當您啟用 Amazon Braket 時，會在您的帳戶中建立服務連結角色。

服務連結角色是一種獨特的 IAM 角色類型，在此情況下會直接連結至 Amazon Braket。Amazon Braket 服務連結角色預先定義為包含 Braket AWS 服務 代表您呼叫其他 時所需的所有許可。

服務連結角色可讓您更輕鬆地設定 Amazon Braket，因為您不必手動新增必要的許可。Amazon Braket 定義其服務連結角色的許可。除非您變更這些定義，否則只有 Amazon Braket 可以擔任其角色。定義的許可包括信任政策和許可政策。許可原則無法附加到其他任何 IAM 實體。

Amazon Braket 設定的服務連結角色是 AWS Identity and Access Management (IAM) 服務連結角色功能的一部分。如需有關 AWS 服務 其他支援服務連結角色的資訊，請參閱AWS 使用 IAM 的服務，並在服務連結角色欄中尋找具有是的服務。選擇有連結的是，以檢視該服務的服務連結角色文件。

如需服務連結角色受 AWS 管政策的詳細資訊，請參閱 AmazonBraketServiceRolePolicy。

Amazon Braket 的合規驗證

> **Note**
>
> AWS 合規報告不涵蓋第三方硬體供應商的 QPUs，他們可以選擇進行自己的獨立稽核。

若要了解 是否 AWS 服務 在特定合規計劃的範圍內，請參閱AWS 服務 合規計劃範圍內然後選擇您感興趣的合規計劃。如需一般資訊，請參閱 AWS Compliance Programs。

您可以使用 下載第三方稽核報告 AWS Artifact。如需詳細資訊，請參閱在 中下載報告 AWS Artifact。

您使用 時的合規責任 AWS 服務 取決於資料的機密性、您公司的合規目標，以及適用的法律和法規。AWS 提供下列資源來協助合規：

- 安全合規與治理 - 這些解決方案實作指南內容討論了架構考量，並提供部署安全與合規功能的步驟。
- HIPAA 合格服務參考 – 列出 HIPAA 合格服務。並非所有 AWS 服務 都符合 HIPAA 資格。
- AWS 合規資源 – 此工作手冊和指南的集合可能適用於您的產業和位置。
- AWS 客戶合規指南 – 透過合規的角度了解共同責任模型。本指南摘要說明跨多個架構 （包括國家標準技術研究所 (NIST)、支付卡產業安全標準委員會 (PCI) 和國際標準化組織 (ISO)) 保護 AWS 服務和映射指南至安全控制的最佳實務。
- 《AWS Config 開發人員指南》中的使用規則評估資源 – AWS Config 服務會評估資源組態符合內部實務、產業準則和法規的程度。
- AWS Security Hub – 這 AWS 服務 可讓您全面檢視其中的安全狀態 AWS。Security Hub 使用安全控制，可評估您的 AWS 資源並檢查您的法規遵循是否符合安全業界標準和最佳實務。如需支援的服務和控制清單，請參閱「Security Hub 控制參考」。
- Amazon GuardDuty – 這會監控您的環境是否有可疑和惡意活動，以 AWS 服務 偵測對您 AWS 帳戶、工作負載、容器和資料的潛在威脅。GuardDuty 可滿足特定合規架構所規定的入侵偵測需求，以協助您因應 PCI DSS 等各種不同的合規需求。
- AWS Audit Manager – 這 AWS 服務 可協助您持續稽核 AWS 用量，以簡化您管理風險和符合法規和產業標準的方式。

Amazon Braket 中的基礎設施安全性

Amazon Braket 是受管服務，受到 AWS 全球網路安全的保護。如需 AWS 安全服務和 如何 AWS 保護基礎設施的相關資訊，請參閱AWS 雲端安全。若要使用基礎設施安全最佳實務來設計您的 AWS 環境，請參閱安全支柱 AWS Well-Architected Framework 中的基礎設施保護。

您可以使用 AWS 發佈的 API 呼叫，透過網路存取 Amazon Braket。使用者端必須支援下列專案：

- Transport Layer Security (TLS)。我們需要 TLS 1.2 並建議使用 TLS 1.3。
- 具備完美轉送私密(PFS)的密碼套件，例如 DHE (Ephemeral Diffie-Hellman)或 ECDHE (Elliptic Curve Ephemeral Diffie-Hellman)。現代系統(如 Java 7 和更新版本)大多會支援這些模式。

此外，請求必須使用存取金鑰 ID 和與 IAM 主體相關聯的私密存取金鑰來簽署。或者，您可以透過 AWS Security Token Service (AWS STS) 來產生暫時安全憑證來簽署請求。

您可以從任何網路位置呼叫這些 API 操作，但 Braket 確實支援以資源為基礎的存取政策，其中可能包括根據來源 IP 地址的限制。您也可以使用 Braket 政策來控制來自特定 Amazon Virtual Private Cloud (Amazon VPC) 端點或特定 VPCs存取。實際上，這只會隔離網路中特定 VPC 對指定 Braket 資源 AWS 的網路存取。

Amazon Braket 硬體供應商的安全性

Amazon Braket 上的 QPUs由第三方硬體供應商託管。當您在 QPU 上執行量子任務時，Amazon Braket 會在將電路傳送至指定的 QPU 進行處理時，使用 DeviceARN 做為識別符。

如果您使用 Amazon Braket 存取其中一個第三方硬體提供者操作的量子運算硬體，您的電路及其相關資料將由 操作設施之外的硬體提供者處理 AWS。您可以在 Amazon Braket 主控台的裝置詳細資訊區段中找到每個 QPU 可用之實體位置和 AWS 區域的相關資訊。

您的內容已匿名化。只有處理電路所需的內容才會傳送給第三方。 AWS 帳戶 資訊不會傳輸給第三方。

所有資料都會在靜態和傳輸中加密。資料會解密，僅用於處理。Amazon Braket 第三方供應商不允許基於處理電路以外的目的存放或使用您的內容。電路完成後，結果會傳回至 Amazon Braket，並存放在 S3 儲存貯體中。

Amazon Braket 第三方量子硬體供應商的安全性會定期稽核，以確保符合網路安全、存取控制、資料保護和實體安全的標準。

Amazon Braket 的 Amazon VPC 端點

您可以建立介面 VPC 端點，在 VPC 和 Amazon Braket 之間建立私有連線。介面端點採用 AWS PrivateLink技術，此技術可讓您在沒有網際網路閘道、NAT 裝置、VPN 連線或 AWS Direct Connect 連線的情況下存取 Braket APIs。VPC 中的執行個體不需要公有 IP 地址，即可與 Braket APIs通訊。

每個介面端點都是由您子網路中的一或多個彈性網路介面表示。

使用 時 AWS PrivateLink，VPC 和 Braket 之間的流量不會離開Amazon網路，這會提高您與雲端應用程式共用之資料的安全性，因為它可降低資料對公有網際網路的暴露。如需詳細資訊，請參閱《Amazon VPC 使用者指南》中的使用介面 VPC 端點存取 AWS 服務。

在本節中：

Amazon Braket VPC 端點的考量事項

設定 Braket 的介面 VPC 端點之前，請務必檢閱《Amazon VPC 使用者指南》中的介面端點先決條件。

Braket 支援從您的 VPC 呼叫其所有 API 動作。

根據預設，允許透過 VPC 端點完整存取 Braket。如果您指定 VPC 端點政策，您可以控制存取。如需詳細資訊，請參閱《Amazon VPC 使用者指南》中的使用端點政策控制 VPC 端點的存取。

設定 Braket 和 PrivateLink

若要 AWS PrivateLink 搭配 Amazon Braket 使用 ，您必須建立 Amazon Virtual Private Cloud (Amazon VPC) 端點做為界面，然後透過 Amazon Braket API服務連線至端點。

以下是此程序的一般步驟，稍後章節會詳細說明這些步驟。

- 設定並啟動 Amazon VPC 來託管您的 AWS 資源。如果您已有 VPC，則可以略過此步驟。
- 為 Braket 建立 Amazon VPC 端點
- 透過端點連接和執行 Braket 量子任務

步驟 1：視需要啟動 Amazon VPC

請記住，如果您的帳戶已在操作中具有 VPC，您可以略過此步驟。

VPC 控制您的網路設定，例如 IP 地址範圍、子網路、路由表和網路閘道。基本上，您會在自訂虛擬網路中啟動 AWS 資源。如需有關 Amazon VPC 的詳細資訊，請參閱《Amazon VPC 使用者指南》。

開啟 Amazon VPC 主控台，並使用子網路、安全群組和網路閘道建立新的 VPC。

步驟 2：建立 Braket 的介面 VPC 端點

您可以使用 Amazon VPC 主控台或 AWS Command Line Interface () 為 Braket 服務建立 VPC 端點 AWS CLI。如需詳細資訊，請參閱《Amazon VPC 使用者指南》中的建立 VPC 端點。

若要在主控台中建立 VPC 端點，請開啟 Amazon VPC 主控台、開啟端點頁面，然後繼續建立新的端點。請記下端點 ID 以供日後參考。當您對 Braket 進行特定呼叫時，這是 `–endpoint-url`旗標的一部分API。

使用下列服務名稱建立 Braket 的 VPC 端點：

- `com.amazonaws.substitute_your_region.braket`

如需詳細資訊，請參閱《Amazon VPC 使用者指南》中的使用介面 VPC 端點存取 AWS 服務。

步驟 3：透過端點連接並執行 Braket 量子任務

建立 VPC 端點之後，您可以執行包含 `endpoint-url` 參數的 CLI 命令，將界面端點指定給 API或 執行時間，例如下列範例：

```
aws braket search-quantum-tasks --endpoint-url
 VPC_Endpoint_ID.braket.substituteYourRegionHere.vpce.amazonaws.com
```

如果您為 VPC 端點啟用私有 DNS 主機名稱，則不需要在 CLI 命令中將端點指定為 URL。相反地，CLI 和 Braket SDK 預設使用的 Amazon Braket API DNS 主機名稱會解析為您的 VPC 端點。其格式如下列範例所示：

```
https://braket.substituteYourRegionHere.amazonaws.com
```

部落格文章名為使用端點從 Amazon VPC 直接存取 Amazon SageMaker AI 筆記本， AWS PrivateLink提供如何設定端點以安全連線至 SageMaker 筆記本的範例，這與 Amazon Braket 筆記本類似。

如果您遵循部落格文章中的步驟，請記得將名稱 Amazon Braket 取代為 Amazon SageMaker AI。針對服務名稱，如果您的區域不是 us-east-1，請在該字串中輸入`com.amazonaws.us-east-1.braket`或替換您的正確 AWS 區域 名稱。

有關建立端點的其他資訊

- 如需有關如何使用私有子網路建立 VPC 的資訊，請參閱使用私有子網路建立 VPC。
- 如需有關使用 Amazon VPC 主控台或 建立和設定端點的資訊 AWS CLI，請參閱《Amazon VPC 使用者指南》中的建立 VPC 端點。
- 如需有關使用 建立和設定端點的資訊 AWS CloudFormation，請參閱AWS CloudFormation 《 使用者指南》中的 AWS：：EC2：：VPCEndpoint 資源。

使用 Amazon VPC 端點政策控制存取

若要控制 Amazon Braket 的連線存取，您可以將 AWS Identity and Access Management (IAM) 端點政策連接至 Amazon VPC 端點。此政策會指定下列資訊：

- 可執行動作的委託人（使用者或角色）。
- 可執行的動作。
- 可供執行動作的資源。

如需詳細資訊，請參閱《Amazon VPC 使用者指南》中的使用端點政策控制 VPC 端點的存取。

範例：Raket 動作的 VPC 端點政策

下列範例顯示 Braket 的端點政策。當連接至 端點時，此政策會授予所有資源上所有委託人的列出的 Braket 動作存取權。

```
{
 "Statement":[
 {
   "Principal":"*",
   "Effect":"Allow",
   "Action":[
     “braket:action-1",
     “braket:action-2",
     “braket:action-3”
     ],
   "Resource":"*"
   }
  ]
}
```

您可以連接多個端點政策來建立複雜的 IAM 規則。如需詳細資訊和範例，請參閱：

- 適用於 Step Functions 的 Amazon Virtual Private Cloud 端點政策
- 為非管理員使用者建立精細 IAM 許可
- 使用端點政策控制對 VPC 端點的存取

日誌記錄和監控

透過 Amazon Braket 服務提交量子任務後，您可以透過 Amazon Braket SDK 和主控台密切監控該任務的狀態和進度。這可提供集中式界面，以追蹤工作負載的實作、找出任何潛在的瓶頸或問題，並採取適當的動作來最佳化量子應用程式的效能和可靠性。當量子任務完成時，Raket 會將結果儲存在您指定的 Amazon S3 位置。量子任務的完成時間可能有所不同，尤其是在量子處理單元 (QPU) 裝置上執行的任務。這主要是由於執行佇列的長度，因為量子硬體資源在多個使用者之間共用。

狀態類型清單：

- `CREATED` – Amazon Braket 已收到您的量子任務。
- `QUEUED` – Amazon Braket 已處理您的量子任務，現在正在等待在裝置上執行。
- `RUNNING` – 您的量子任務正在 QPU 或隨需模擬器上執行。
- `COMPLETED` – 您的量子任務已完成在 QPU 或隨需模擬器上執行。
- `FAILED` – 您的量子任務嘗試執行並失敗。根據您的量子任務失敗的原因，請嘗試再次提交量子任務。
- `CANCELLED` – 您已取消規定人數任務。量子任務未執行。

在本節中：

從 Amazon Braket SDK 追蹤量子任務

命令`device.run(…)` 會定義具有唯一量子任務 ID 的量子任務。您可以使用 查詢和追蹤狀態`task.state()`，如下列範例所示。

注意： `task = device.run()` 是一種非同步操作，這表示您可以在系統在背景處理量子任務時繼續工作。

擷取結果

當您呼叫 時`task.result()`，開發套件會開始輪詢 Amazon Braket，以查看量子任務是否完成。SDK 會使用您在 中定義的輪詢參數`.run()`。量子任務完成後，軟體開發套件會從 S3 儲存貯體擷取結果，並將其傳回為`QuantumTaskResult`物件。

```
# 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
```

取消量子任務

若要取消量子任務，請呼叫 `cancel()`方法，如下列範例所示。

```
# cancel quantum task
task.cancel()
status = task.state()
```

```
print('Status of task:', status)
```

```
Status of task: CANCELLING
```

檢查中繼資料

您可以檢查已完成量子任務的中繼資料，如下列範例所示。

```
# 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
```

擷取量子任務或結果

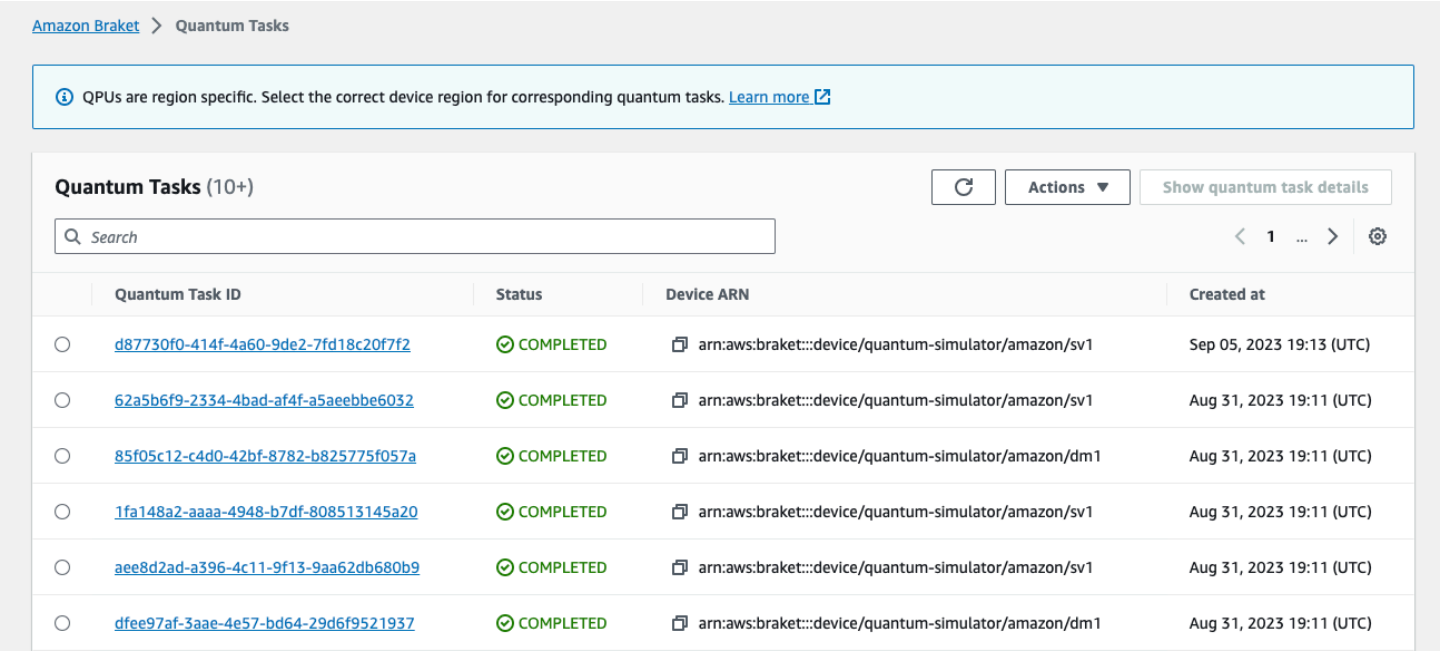
如果您的核心在您提交量子任務後死亡，或者您關閉筆記本或電腦，您可以使用其唯一的 ARN （量子任務 ID) 重建`task`物件。然後，您可以呼叫 從儲存該儲存貯體的 S3 儲存貯體`task.result()`取得結果。

```
from braket.aws import AwsSession, AwsQuantumTask
```

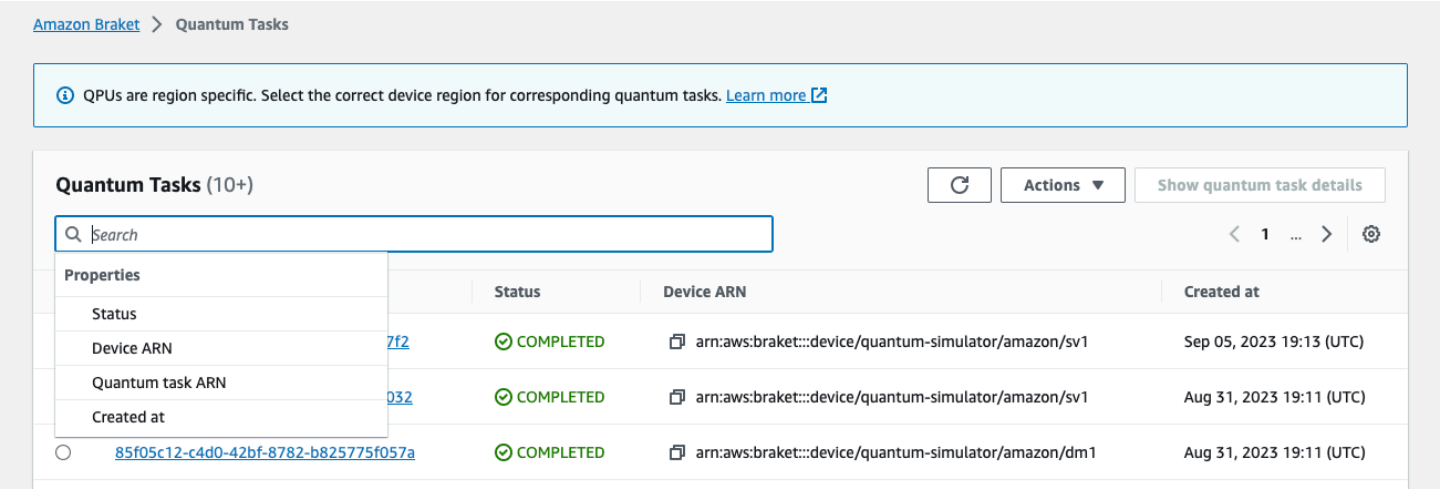
```
# restore task with unique arn
task_load = AwsQuantumTask(arn=task_id)
# retrieve the result of the task
result = task_load.result()
```

透過 Amazon Braket 主控台監控量子任務

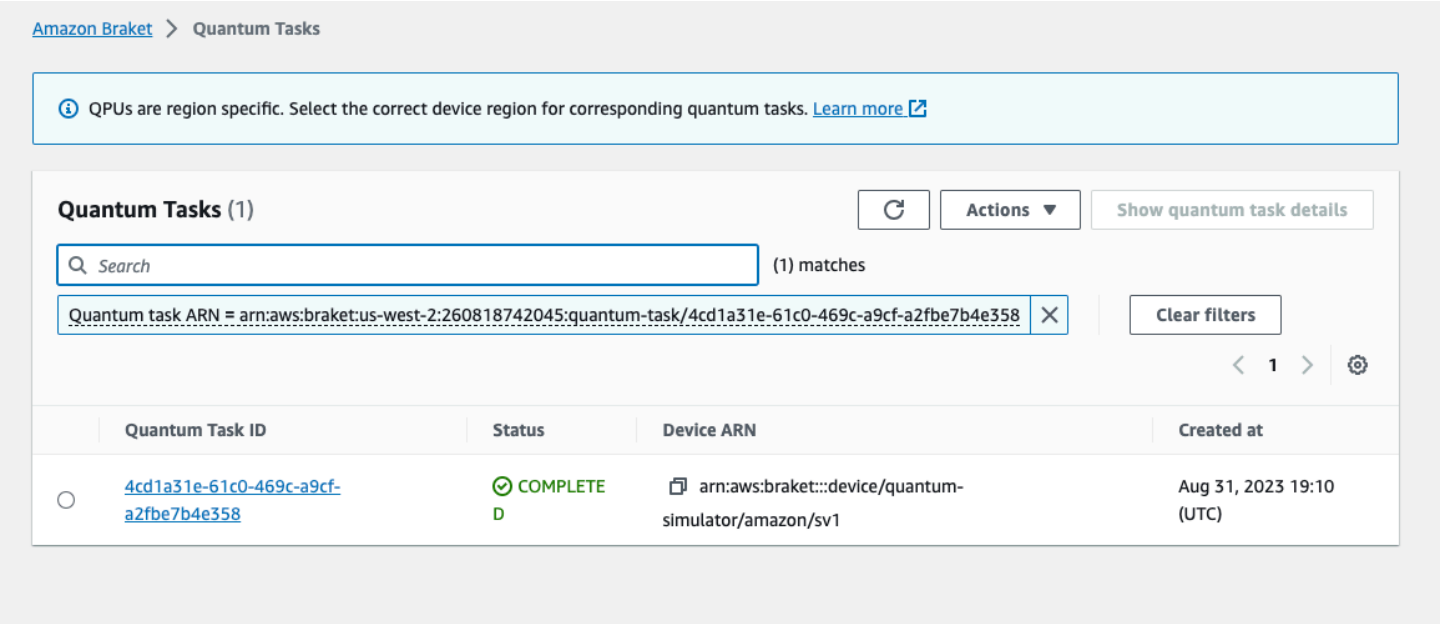
Amazon Braket 提供透過 Amazon Braket 主控台監控量子任務的便利方式。所有提交的量子任務都會列在 Quantum 任務欄位中，如下圖所示。此服務是區域特定的，這表示您只能檢視在特定 中建立的量子任務 AWS 區域。



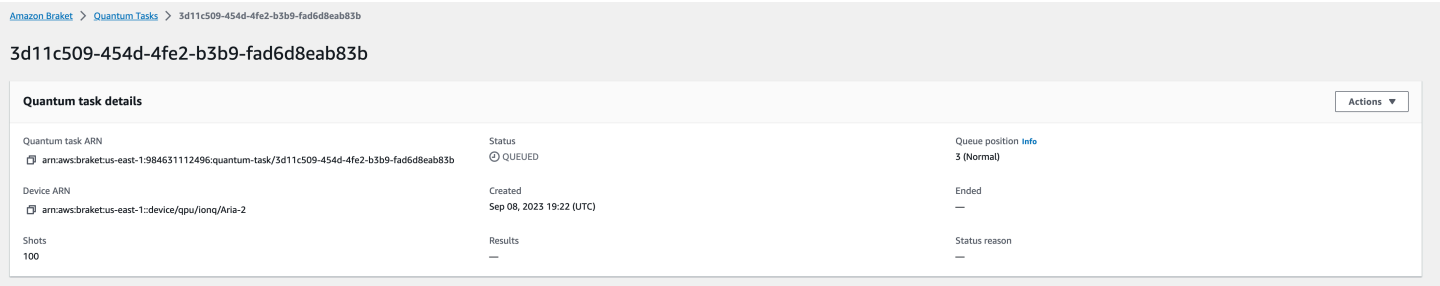
您可以透過導覽列搜尋特定量子任務。搜尋可以根據 Quantum 任務 ARN (ID)、狀態、裝置和建立時間。當您選取導覽列時，選項會自動顯示，如下列範例所示。



下圖顯示根據其唯一量子任務 ID 搜尋量子任務的範例，可透過呼叫 來取得`task.id`。



此外，如下圖所示，量子任務的狀態可在處於 QUEUED 狀態時受到監控。按一下量子任務 ID 會顯示詳細資訊頁面。此頁面會顯示相對於處理裝置量子任務的動態佇列位置。



在佇列中提交做為混合任務一部分的 Quantum 任務將具有優先順序。在混合任務之外提交的 Quantum 任務將具有正常的佇列優先順序。

想要查詢 Braket SDK 的客戶可以透過程式設計方式取得其量子任務和混合任務佇列位置。如需詳細資訊，請參閱我的任務何時執行頁面。

標記 Amazon Braket 資源

標籤是您指派或 AWS 指派給 AWS 資源的自訂屬性標籤。標籤是說明資源詳細資訊的中繼資料。每個標籤皆包含鍵與值。這些合稱為鍵值組。對於您指派的標籤，您可以定義鍵與值。

在 Amazon Braket 主控台中，您可以導覽至量子任務或筆記本，並檢視與其相關聯的標籤清單。您可以新增標籤、移除標籤或修改標籤。您可以在建立時標記量子任務或筆記本，然後透過主控台 AWS CLI或 管理相關聯的標籤API。

有關 AWS 和 標籤的詳細資訊

- 如需標記的一般資訊，包括命名和使用慣例，請參閱《標記 AWS 資源和標籤編輯器使用者指南》中的什麼是標籤編輯器？。
- 如需有關標記限制的資訊，請參閱《標記 AWS 資源和標籤編輯器使用者指南》中的標籤命名限制和要求。
- 如需最佳實務和標記策略，請參閱標記 AWS 資源的最佳實務。
- 關於支援標記的服務清單，請參閱Resource Groups 標記 API 參考。

以下各節提供有關 Amazon Braket 標籤的更具體資訊。

在本節中：

使用標籤

標籤可以將資源組織成對您有用的類別。例如，您可以指派「部門」標籤來指定擁有此資源的部門。

每個標籤有兩個部分：

- 標籤金鑰 （例如 CostCenter、環境或專案)。標籤鍵會區分大小寫。
- 稱為標籤值的選用欄位 （例如 111122223333 或 Production)。忽略標籤值基本上等同於使用空字串。與標籤鍵相同，標籤值會區分大小寫。

標籤可協助您執行下列動作：

- 識別和組織您的 AWS 資源。許多 AWS 服務 支援標記，因此您可以將相同的標籤指派給來自不同服務的資源，以指出資源相關。

- 追蹤您的 AWS 成本。您可以在 AWS 帳單與成本管理 儀表板上啟用這些標籤。 AWS 會使用標籤來分類您的成本，並傳送每月成本分配報告給您。如需詳細資訊，請參閱 AWS 帳單與成本管理 使用者指南中的使用成本配置標籤。
- 控制對 AWS 資源的存取。如需詳細資訊，請參閱使用標籤控制存取。

Amazon Braket 中用於標記的支援資源

Amazon Braket 中的下列資源類型支援標記：

- quantum-task 資源
- 資源名稱： `AWS::Service::Braket`
- ARN Regex： `arn:${Partition}:braket:${Region}:${Account}:quantum-task/${RandomId}`

注意：雖然Amazon筆記本實際上是 Amazon SageMaker AI 資源，但您可以使用主控台導覽至筆記本資源，在 Amazon Braket 主控台中套用和管理 Braket 筆記本的標籤。如需詳細資訊，請參閱 SageMaker 文件中的筆記本執行個體中繼資料。

使用 Amazon Braket API標記

- 如果您使用 Amazon Braket 在資源上API設定標籤，請呼叫 TagResourceAPI。

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

- 若要從資源中移除標籤，請呼叫 UntagResourceAPI。

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

- 若要列出連接至特定資源的所有標籤，請呼叫 ListTagsForResourceAPI。

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

標記限制

下列基本限制適用於 Amazon Braket 資源上的標籤：

- 您可以指派給資源的標籤數量上限：50
- 索引鍵長度上限：128 個 Unicode 字元
- 數值長度上限：256 個 Unicode 字元
- 索引鍵和值的有效字元：`a-z, A-Z, 0-9, space`和這些字元：`_ . : / = + -`和 `@`
- 金鑰和值會區分大小寫。
- 不要使用 `aws`做為金鑰的字首；它保留供 AWS 使用。

在 Amazon Braket 中管理標籤

您可以將標籤設定為資源上的屬性。您可以透過 Amazon Braket 主控台、Amazon Braket 或 來檢視、新增API、修改、列出和刪除標籤 AWS CLI。如需詳細資訊，請參閱 Amazon Braket API 參考。

在本節中：

新增標籤

您可以在下列時間將標籤新增至可標記的資源：

- 當您建立 資源時：使用 主控台，或在 AWS API 中包含 `Tags` 參數與 `Create`操作。
- 建立資源後：使用 主控台導覽至量子任務或筆記本資源，或在 AWS API 中呼叫 `TagResource`操作。

若要在建立資源時新增標籤，您也需要建立指定類型資源的許可。

檢視標籤

您可以使用主控台導覽至任務或筆記本資源，或呼叫 `ListTagsForResource`API操作，來檢視 Amazon Braket 中任何可標記資源的 AWS 標籤。

您可以使用下列 AWS API命令來檢視資源上的標籤：

- AWS API: `ListTagsForResource`

編輯標籤

您可以使用 主控台導覽至量子任務或筆記本資源來編輯標籤，或使用下列命令來修改連接至可標記資源之標籤的值。當您指定已存在的標籤金鑰時，會覆寫該金鑰的值：

- AWS API: `TagResource`

移除標籤

您可以透過指定要移除的金鑰、使用主控台導覽至量子任務或筆記本資源，或在呼叫 `UntagResource`操作時，從資源移除標籤。

- AWS API: `UntagResource`

Amazon Braket 中的 AWS CLI 標記範例

當您使用 AWS Command Line Interface (AWS CLI) 與 Amazon Braket 互動時，以下程式碼是示範如何建立套用至您建立之量子任務的標籤的範例命令。在此範例中，任務正在SV1量子模擬器上執行，並指定量子處理單元 (QPU) Rigetti 的參數設定。請務必在範例命令中，在所有其他必要參數之後，在最結尾指定標籤。在此情況下，標籤的索引鍵為 `state`，值為 `Washington`。這些標籤可用來協助分類或識別此特定量子任務。

```
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\"}
```

此範例示範如何在透過 執行時將標籤套用至量子任務 AWS CLI，這有助於組織和追蹤您的 Braket 資源。

使用 EventBridge 監控您的量子任務

Amazon EventBridge 會監控 Amazon Braket 量子任務中的狀態變更事件。來自 Amazon Braket 的事件幾乎會即時交付至 EventBridge。您可以撰寫規則來指出您感興趣的事件，包括當事件符合規則時要採取的自動化動作。可觸發的自動動作包括下列項目：

- 叫用 AWS Lambda 函數
- 啟用 AWS Step Functions 狀態機器
- 通知 Amazon SNS 主題

EventBridge 會監控這些 Amazon Braket 狀態變更事件：

- Qauntum 任務的狀態會變更

Amazon Braket 保證交付量子任務狀態變更事件。這些事件至少會交付一次，但可能會失序。

如需詳細資訊，請參閱 Amazon EventBridge 中的事件。

在本節中：

- 使用 EventBridge 監控量子任務狀態
- Amazon Braket EventBridge 事件範例

使用 EventBridge 監控量子任務狀態

使用 EventBridge，您可以建立規則，定義當 Amazon Braket 傳送有關 Braket 量子任務的狀態變更通知時要採取的動作。例如，您可以建立規則，在每次量子任務狀態變更時傳送電子郵件訊息給您。

1. AWS 使用具有使用 EventBridge 和 Amazon Braket 之許可的帳戶登入 。
2. 開啟 Amazon EventBridge 主控台。
3. 使用下列值，建立 EventBridge 規則：
   - 針對規則類型，選擇具有事件模式的規則。
   - 在 Event source (事件來源) 中，選擇 Other (其他)。
   - 在事件模式區段中，選擇自訂模式 (JSON 編輯器），然後將下列事件模式貼入文字區域：

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

     若要從 Amazon Braket 擷取所有事件，請排除 `detail-type`區段，如下列程式碼所示：

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

   - 對於目標類型，選擇 AWS 服務，對於選取目標，選擇目標，例如 Amazon SNS 主題或 AWS Lambda 函數。從 Amazon Braket 收到量子任務狀態變更事件時，會觸發目標。

     例如，使用 Amazon Simple Notification Service (SNS) 主題在事件發生時傳送電子郵件或文字訊息。若要這樣做，請先使用 Amazon SNS 主控台建立 Amazon SNS 主題。若要進一步了解，請參閱使用 Amazon SNS 傳送使用者通知。

如需建立規則的詳細資訊，請參閱建立對事件做出反應的 Amazon EventBridge 規則。

Amazon Braket EventBridge 事件範例

如需 Amazon Braket Quantum 任務狀態變更事件欄位的資訊，請參閱 Amazon EventBridge 中的事件。

下列屬性會出現在 JSON "detail" 欄位中。

- **quantumTaskArn** (str)：產生此事件的量子任務。
- **status** （選用 【str】）：量子任務轉換至的狀態。
- **deviceArn** (str)：建立此量子任務的使用者指定的裝置。
- shots (int)：使用者shots請求的 數量。
- **outputS3Bucket** (str)：使用者指定的輸出儲存貯體。
- **outputS3Directory** (str)：使用者指定的輸出金鑰字首。
- **createdAt** (str)：做為 ISO-8601 字串的量子任務建立時間。
- **endedAt** （選用 【str】）：量子任務達到結束狀態的時間。此欄位只有在量子任務已轉換為終端機狀態時才存在。

下列 JSON 程式碼顯示 Amazon Braket Quantum 任務狀態變更事件的範例。

```
{
    "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"
      }
}
```

使用 CloudWatch 監控指標

您可以使用 Amazon CloudWatch 監控 Amazon Braket，這會收集原始資料並將其處理為可讀且近乎即時的指標。 Amazon CloudWatch 您可以在 Amazon CloudWatch 主控台中檢視最多 15 個月前產生的歷史資訊，或過去 2 週內更新的搜尋指標，以更清楚了解 Amazon Braket 的效能。若要進一步了解，請參閱使用 CloudWatch 指標。

> **Note**
>
> 您可以導覽至 Amazon SageMaker AI 主控台上的筆記本詳細資訊頁面，以檢視 Amazon Braket 筆記本的 CloudWatch 日誌串流。 Amazon SageMaker 其他 Amazon Braket 筆記本設定可透過 SageMaker 主控台取得。

在本節中：

Amazon Braket 指標和維度

指標為 CloudWatch 中的基本概念。指標代表按時間順序發佈到 CloudWatch 的一組資料點。每個指標的特徵都是一組維度。若要進一步了解 CloudWatch 中的指標維度，請參閱 CloudWatch 維度。

Amazon Braket 會將下列 Amazon Braket 特有的指標資料傳送至 Amazon CloudWatch 指標：

Quantum 任務指標

如果量子任務存在，則可使用指標。它們會顯示在 CloudWatch 主控台中的 AWS/Braket/By Device 下。

指標	描述
計數	量子任務的數量。
Latency (延遲)	當量子任務完成時，會發出此指標。它代表從量子任務初始化到完成的總時間。

Quantum 任務指標的維度

量子任務指標會以以 deviceArn 參數為基礎的維度發佈，格式為 arn：aws：braket：：device/xxx。

使用 CloudTrail 記錄您的量子任務

Amazon Braket 已與 整合 AWS CloudTrail，此服務提供 Amazon Braket AWS 服務 中使用者、角色或所採取動作的記錄。CloudTrail 會將 Amazon Braket 的所有API呼叫擷取為事件。擷取的呼叫包括來自 Amazon Braket 主控台的呼叫，以及對 Amazon Braket 操作的程式碼呼叫。如果您建立線索，您可以將 CloudTrail 事件持續交付至 Amazon S3 儲存貯體，包括 Amazon Braket 的事件。如果您未設定追蹤，您仍然可以在 CloudTrail 主控台的事件歷史記錄中檢視最新的事件。您可以使用 CloudTrail 所收集的資訊，判斷對 Amazon Braket 提出的請求、提出請求的 IP 地址、提出請求的人員、提出請求的時間，以及其他詳細資訊。

若要進一步了解 CloudTrail，請參閱「AWS CloudTrail 使用者指南」。

在本節中：

- CloudTrail 中的 Amazon Braket 資訊
- 了解 Amazon Braket 日誌檔案項目

CloudTrail 中的 Amazon Braket 資訊

當您建立帳戶 AWS 帳戶 時，您的 上會啟用 CloudTrail。當活動在 Amazon Braket 中發生時，該活動會與事件歷史記錄中的其他 AWS 服務 事件一起記錄在 CloudTrail 事件中。您可以在 中檢視、搜尋和下載最近的事件 AWS 帳戶。如需詳細資訊，請參閱《使用 CloudTrail 事件歷史記錄檢視事件》https://docs.aws.amazon.com/awscloudtrail/latest/userguide/view-cloudtrail-events.html。

若要持續記錄 中的事件 AWS 帳戶，包括 Amazon Braket 的事件，請建立追蹤。線索能讓 CloudTrail 將日誌檔案交付至 Amazon S3 儲存貯體。依預設，當您在主控台中建立追蹤時，該追蹤會套用至所有的 AWS 區域。線索會記錄 AWS 分割區中所有區域的事件，並將日誌檔案交付至您指定的 Amazon S3 儲存貯體。此外，您可以設定其他 AWS 服務 以進一步分析和處理 CloudTrail 日誌中收集的事件資料。如需詳細資訊，請參閱下列內容：

- 建立追蹤的概觀
- CloudTrail 支援的服務和整合
- 設定 CloudTrail 的 Amazon SNS 通知
- 從多個區域接收 CloudTrail 日誌檔案，以及從多個帳戶接收 CloudTrail 日誌檔案

CloudTrail 會記錄所有 Amazon Braket 動作。例如，對 GetQuantumTask或 GetDevice動作的呼叫會在 CloudTrail 日誌檔案中產生項目。

每一筆事件或日誌專案都會包含產生請求者的資訊。身分資訊可協助您判斷下列事項：

- 提出該請求時，是否使用了特定角色或聯合身分使用者的暫時安全憑證。
- 該請求是否由另一項 AWS 服務服務提出。

如需詳細資訊，請參閱 CloudTrail userIdentity 元素。

了解 Amazon Braket 日誌檔案項目

追蹤是一種組態，能讓事件以日誌檔案的形式交付到您指定的 Amazon S3 儲存貯體。CloudTrail 日誌檔案包含一或多個日誌專案。一個事件為任何來源提出的單一請求，並包含請求動作、請求的日期和時間、請求參數等資訊。CloudTrail 日誌檔案不是公開API呼叫的排序堆疊追蹤，因此不會以任何特定順序顯示。

下列範例是 GetQuantumTask動作的日誌項目，可取得量子任務的詳細資訊。

```
{
  "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"
}
```

以下顯示 GetDevice動作的日誌項目，其會傳回裝置事件的詳細資訊。

```
{
  "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"
}
```

使用 Amazon Braket 的進階記錄

您可以使用記錄器記錄整個任務處理程序。這些進階記錄技術可讓您查看背景輪詢，並建立記錄以供日後偵錯。

若要使用記錄器，建議您變更 `poll_timeout_seconds`和 `poll_interval_seconds` 參數，以便量子任務可以長時間執行，且量子任務狀態會持續記錄，並將結果儲存至檔案。您可以將此程式碼轉移到 Python 指令碼，而不是 Jupyter 筆記本，以便指令碼可以作為背景中的程序執行。

設定記錄器

首先，設定記錄器，將所有日誌自動寫入文字檔案中，如下列範例行所示。

```
# 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
```

建立並執行電路

現在您可以建立電路、將其提交至裝置以執行，並查看發生的情況，如本範例所示。

```
# 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
)
```

檢查日誌檔案

您可以輸入下列命令來檢查寫入 檔案的內容。

```
# 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})
```

從日誌檔案取得 ARN

從傳回的日誌檔案輸出，如先前範例所示，您可以取得 ARN 資訊。使用 ARN ID，您可以擷取已完成量子任務的結果。

```
# 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 配額

下表列出 Amazon Braket 的服務配額。服務配額 (也稱為限制) 是 AWS 帳戶的服務資源或操作的最大數量。

某些配額可以增加。如需詳細資訊，請參閱 AWS 服務 配額。

- 爆量速率配額無法增加。
- 可調整配額 （爆量速率除外，無法調整） 的最大速率增加為指定預設速率限制的 2X。例如，預設配額為 60，最多可調整為 120。
- 並行 SV1(DM1) 量子任務的可調整配額最多允許每個 60 AWS 區域個。
- 混合任務的運算執行個體允許數目上限為 1，配額可調整。

資源	描述	限制	可調整
API 請求率	在目前區域中，您可以在此帳戶中傳送的每秒請求數上限。	140	是
API 請求爆量率	您可以在目前區域中此帳戶中傳送的每秒額外請求數 (RPS) 上限。	600	否
CreateQuantumTask 請求率	您可以在每個區域此帳戶中每秒傳送的CreateQuantumTask 請求數量上限。	每秒 20 個	是
CreateQuantumTask 請求爆量率	您可以在目前區域中此帳戶中傳送的每秒額外CreateQuantumTask 請求數 (RPS) 上限。	40	否

資源	描述	限制	可調整
SearchQuantumTasks 請求率	您可以在每個區域中此帳戶中每秒傳送的SearchQuantumTasks 請求數量上限。	每秒 5	是
SearchQuantumTasks 請求爆量率	在目前區域中，您可以在此帳戶中傳送的每秒額外SearchQuantumTasks 請求數 (RPS) 上限。	50	否
GetQuantumTask 請求率	您可以在每個區域中此帳戶中每秒傳送的GetQuantumTask 請求數量上限。	每秒 100 次	是
GetQuantumTask 請求爆量率	在目前區域中，您可以在此帳戶中傳送的每秒額外GetQuantumTask 請求數 (RPS) 上限。	500	否
CancelQuantumTask 請求率	您可以在每個區域中此帳戶中每秒傳送的CancelQuantumTask 請求數量上限。	每秒 2 個	是
CancelQuantumTask 請求爆量率	在目前區域中，您可以在此帳戶中傳送的每秒額外CancelQuantumTask 請求數 (RPS) 上限。	20	否

資源	描述	限制	可調整
GetDevice 請求率	您可以在每個區域中此帳戶中每秒傳送的 GetDevice 請求數量上限。	每秒 5	是
GetDevice 請求爆量率	在目前區域中，您可以在此帳戶中傳送的每秒額外GetDevice 請求數 (RPS) 上限。	50	否
SearchDevices 請求率	您可以在每個區域中此帳戶中每秒傳送的 SearchDevices 請求數量上限。	每秒 5	是
SearchDevices 請求爆量率	在目前區域中，您可以在此帳戶中傳送的每秒額外SearchDevices 請求數 (RPS) 上限。	50	否
CreateJob 請求率	您可以在每個區域中此帳戶中每秒傳送的 CreateJob 請求數量上限。	每秒 1 個	是
CreateJob 請求爆量率	您可以在目前區域中此帳戶中傳送的每秒額外CreateJob 請求數 (RPS) 上限。	5	否
SearchJobs 請求率	您可以在每個區域中此帳戶中每秒傳送的 SearchJob 請求數量上限。	每秒 5	是

資源	描述	限制	可調整
SearchJobs 請求爆量率	在目前區域中，您可以在此帳戶中傳送的每秒額外SearchJob 請求數 (RPS) 上限。	50	否
GetJob 請求率	您可以在每個區域中此帳戶中每秒傳送的GetJob請求數量上限。	每秒 5	是
GetJob 請求爆量率	在目前區域中，您可以在此帳戶中傳送的每秒額外GetJob請求數 (RPS) 上限。	25	否
CancelJob 請求率	您可以在每個區域中此帳戶中每秒傳送的CancelJob 請求數量上限。	每秒 2 個	是
CancelJob 請求爆量率	在目前區域中，您可以在此帳戶中傳送的每秒額外CancelJob 請求數 (RPS) 上限。	5	否
並行SV1量子任務數量	在目前區域中狀態向量模擬器 (SV1) 上執行的並行量子任務數量上限。	100 us-east-1， 50 us-west-1， 100 us-west-2， 50 eu-west-2	否

資源	描述	限制	可調整
並行DM1量子任務數量	在目前區域中密度矩陣模擬器 (DM1) 上執行的並行量子任務數量上限。	100 us-east-1， 50 us-west-1， 100 us-west-2， 50 eu-west-2	否
並行TN1量子任務數量	在目前區域中張量網路模擬器 (TN1) 上執行的並行量子任務數量上限。	10 us-east-1， 10 us-west-2， 5 eu-west-2、	是
並行混合任務的數量	目前區域中並行混合任務的數量上限。	3	是
混合任務執行時間限制	混合任務可以執行的天數上限。	5	否

以下是混合任務的預設傳統運算執行個體配額。若要提高這些配額，請聯絡 支援。此外，每個執行個體都會指定可用的區域。

資源	描述	限制	可調整	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
混合任務的 ml.c4.xlarge 執行個體數目上限	此帳戶和區域中所有 Amazon Braket 混合任務允許的 ml.c4.xlarge 類	5	是	是	是	是	是	否

資源	描述	限制	可調整	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
	型執行個體數目上限。							
混合任務的 ml.c4.2xlarge 執行個體數目上限	此帳戶和區域中所有 Amazon Braket 混合任務允許的 ml.c4.2xlarge 類型執行個體數目上限。	5	是	是	是	是	是	否
混合任務的 ml.c4.4xlarge 執行個體數目上限	此帳戶和區域中所有 Amazon Braket 混合任務允許的 ml.c4.4xlarge 類型執行個體數目上限。	5	是	是	是	是	是	否

資源	描述	限制	可調整	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
混合任務的 ml.c4.8xlarge 執行個體數目上限	此帳戶和區域中所有 Amazon Braket 混合任務允許的 ml.c4.8xlarge 類型執行個體數目上限。	5	是	是	是	是	否	否
混合任務的 ml.c5.xlarge 執行個體數目上限	此帳戶和區域中所有 Amazon Braket 混合任務允許的 ml.c5.xlarge 類型執行個體數目上限。	5	是	是	是	是	是	是

資源	描述	限制	可調整	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
混合任務的 ml.c5.2xlarge 執行個體數目上限	此帳戶和區域中所有 Amazon Braket 混合任務允許的 ml.c5.2xlarge 類型執行個體數目上限。	5	是	是	是	是	是	是
混合任務的 ml.c5.4xlarge 執行個體數目上限	此帳戶和區域中所有 Amazon Braket 混合任務允許的 ml.c5.4xlarge 類型執行個體數目上限。	1	是	是	是	是	是	是

資源	描述	限制	可調整	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
混合任務的 ml.c5.9xlarge 執行個體數目上限	此帳戶和區域中所有 Amazon Braket 混合任務允許的 ml.c5.9xlarge 類型執行個體數目上限。	1	是	是	是	是	是	是
混合任務的 ml.c5.18xlarge 執行個體數目上限	此帳戶和區域中所有 Amazon Braket 混合任務允許的 ml.c5.18xlarge 類型執行個體數目上限。	0	是	是	是	是	是	是

資源	描述	限制	可調整	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
混合任務的 ml.c5n.xlarge 執行個體數目上限	此帳戶和區域中所有 Amazon Braket 混合任務允許的 ml.c5n.xlarge 類型執行個體數目上限。	0	是	是	是	是	否	否
混合任務的 ml.c5n.2xlarge 執行個體數目上限	此帳戶和區域中所有 Amazon Braket 混合任務允許的 ml.c5n.2xlarge 類型執行個體數目上限。	0	是	是	是	是	否	否

資源	描述	限制	可調整	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
混合任務的 ml.c5n.4xlarge 執行個體數目上限	此帳戶和區域中所有 Amazon Braket 混合任務允許的 ml.c5n.4xlarge 類型執行個體數目上限。	0	是	是	是	是	否	否
混合任務的 ml.c5n.9xlarge 執行個體數目上限	此帳戶和區域中所有 Amazon Braket 混合任務允許的 ml.c5n.9xlarge 類型執行個體數目上限。	0	是	是	是	是	否	否

資源	描述	限制	可調整	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
混合任務的 ml.c5n.18xlarge 執行個體數目上限	此帳戶和區域中所有 Amazon Braket 混合任務允許的 ml.c5n.18xlarge 類型執行個體數目上限。	0	是	是	是	是	否	否
混合任務的 ml.g4dn.xlarge 執行個體數目上限	此帳戶和區域中所有 Amazon Braket 混合任務允許的 ml.g4dn.xlarge 類型執行個體數目上限。	0	是	是	是	是	是	是

資源	描述	限制	可調整	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
混合任務的 ml.g4dn.2xlarge 執行個體數目上限	此帳戶和區域中所有 Amazon Braket 混合任務允許的 ml.g4dn.2xlarge 類型執行個體數目上限。	0	是	是	是	是	是	是
混合任務的 ml.g4dn.4xlarge 執行個體數目上限	此帳戶和區域中所有 Amazon Braket 混合任務允許的 ml.g4dn.4xlarge 類型執行個體數目上限。	0	是	是	是	是	是	是

資源	描述	限制	可調整	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
混合任務的 ml.g4dn.8xlarge 執行個體數目上限	此帳戶和區域中所有 Amazon Braket 混合任務允許的 ml.g4dn.8xlarge 類型執行個體數目上限。	0	是	是	是	是	是	是
混合任務的 ml.g4dn.12xlarge 執行個體數目上限	此帳戶和區域中所有 Amazon Braket 混合任務允許的 ml.g4dn.12xlarge 類型執行個體數目上限。	0	是	是	是	是	是	是

資源	描述	限制	可調整	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
混合任務的 ml.g4dn.16xlarge 執行個體數目上限	此帳戶和區域中所有 Amazon Braket 混合任務允許的 ml.g4dn.16xlarge 類型執行個體數目上限。	0	是	是	是	是	是	是
混合任務的 ml.m4.xlarge 執行個體數目上限	此帳戶和區域中所有 Amazon Braket 混合任務允許的 ml.m4.xlarge 類型執行個體數目上限。	5	是	是	是	是	是	否

資源	描述	限制	可調整	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
混合任務的 ml.m4.2xlarge 執行個體數目上限	此帳戶和區域中所有 Amazon Braket 混合任務允許的 ml.m4.2xlarge 類型執行個體數目上限。	5	是	是	是	是	是	否
混合任務的 ml.m4.4xlarge 執行個體數目上限	此帳戶和區域中所有 Amazon Braket 混合任務允許的 ml.m4.4xlarge 類型執行個體數目上限。	2	是	是	是	是	是	否

資源	描述	限制	可調整	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
混合任務的 ml.m4.10› large 執行個體數目上限	此帳戶和區域中所有 Amazon Braket 混合任務允許的 ml.m4.10› large 類型執行個體數目上限。	0	是	是	是	是	是	否
混合任務的 ml.m4.16› large 執行個體數目上限	此帳戶和區域中所有 Amazon Braket 混合任務允許的 ml.m4.16› large 類型執行個體數目上限。	0	是	是	是	是	是	否

資源	描述	限制	可調整	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
混合任務的 ml.m5.large 執行個體數目上限	此帳戶和區域中所有 Amazon Braket 混合任務允許的 ml.m5.large 類型執行個體數目上限。	5	是	是	是	是	是	是
混合任務的 ml.m5.xlarge 執行個體數目上限	此帳戶和區域中所有 Amazon Braket 混合任務允許的 ml.m5.xlarge 類型執行個體數目上限。	5	是	是	是	是	是	是

資源	描述	限制	可調整	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
混合任務的 ml.m5.2xlarge 執行個體數目上限	此帳戶和區域中所有 Amazon Braket 混合任務允許的 ml.m5.2xlarge 類型執行個體數目上限。	5	是	是	是	是	是	是
混合任務的 ml.m5.4xlarge 執行個體數目上限	此帳戶和區域中所有 Amazon Braket 混合任務允許的 ml.m5.4xlarge 類型執行個體數目上限。	5	是	是	是	是	是	是

資源	描述	限制	可調整	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
混合任務的 ml.m5.12› large 執行個體數目上限	此帳戶和區域中所有 Amazon Braket 混合任務允許的 ml.m5.12› large 類型執行個體數目上限。	0	是	是	是	是	是	是
混合任務的 ml.m5.24› large 執行個體數目上限	此帳戶和區域中所有 Amazon Braket 混合任務允許的 ml.m5.24› large 類型執行個體數目上限。	0	是	是	是	是	是	是

資源	描述	限制	可調整	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
混合任務的 ml.p2.xlarge 執行個體數目上限	此帳戶和區域中所有 Amazon Braket 混合任務允許的 ml.p2.xlarge 類型執行個體數目上限。	0	是	是	否	是	否	否
混合任務的 ml.p2.8xlarge 執行個體數目上限	此帳戶和區域中所有 Amazon Braket 混合任務允許的 ml.p2.8xlarge 類型執行個體數目上限。	0	是	是	否	是	否	否

資源	描述	限制	可調整	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
混合任務的 ml.p2.16xlarge 執行個體數目上限	此帳戶和區域中所有 Amazon Braket 混合任務允許的 ml.p2.16xlarge 類型執行個體數目上限。	0	是	是	否	是	否	否
混合任務的 ml.p3.2xlarge 執行個體數目上限	此帳戶和區域中所有 Amazon Braket 混合任務允許的 ml.p3.2xlarge 類型執行個體數目上限。	0	是	是	否	是	否	否

資源	描述	限制	可調整	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
混合任務的 ml.p4d.24xlarge 執行個體數目上限	此帳戶和區域中所有 Amazon Braket 混合任務允許的 ml.p4d.24xlarge 類型執行個體數目上限。	0	是	是	否	是	否	否
混合任務的 ml.p3dn.24xlarge 執行個體數目上限	此帳戶和區域中所有 Amazon Braket 混合任務允許的 ml.p3dn.24xlarge 類型執行個體數目上限。	0	是	是	否	是	否	否

資源	描述	限制	可調整	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
混合任務的 ml.p3.8xlarge 執行個體數目上限	此帳戶和區域中所有 Amazon Braket 混合任務允許的 ml.p3.8xlarge 類型執行個體數目上限。	0	是	是	否	是	是	否
混合任務的 ml.p3.16xlarge 執行個體數目上限	此帳戶和區域中所有 Amazon Braket 混合任務允許的 ml.p3.16xlarge 類型執行個體數目上限。	0	是	是	否	是	是	否

請求限制更新

如果您收到 執行個體類型的 ServiceQuotaExceeded 例外狀況，且沒有足夠的執行個體可供使用，您可以從 AWS 主控台的 Service Quotas 頁面請求提高限制，並在 AWS 服務下搜尋 Amazon Braket。

> **Note**
>
> 如果您的混合任務無法佈建請求的 ML 運算容量，請使用另一個區域。此外，如果您在資料表中沒有看到執行個體，它不適用於混合任務。

其他配額和限制

- Amazon Braket 量子任務動作的大小限制為 3MB。
- 對於 SV1，最多 31 個 qubits 的電路最長執行持續時間為 3 小時，超過 31 個 qubits 的電路最長執行持續時間為 11 小時。
- 對於 SV1、 和 Rigetti 裝置DM1，每個任務允許的鏡頭數量上限為 50，000。
- 每個 任務允許的鏡頭數量上限為 TN1 1000。
- 對於所有 IonQ的裝置：使用隨需模型時，有 100 萬個門框限制，以及至少 2500 個錯誤緩解任務的鏡頭。對於直接保留，沒有閘道擷取限制，且錯誤緩解任務至少需要 500 個擷取。
- 對於 QuEra的 Aquila 裝置，每個任務的拍攝次數上限為 1，000 次。
- 對於 IQM的 Garnet 和翡翠裝置，每個任務的射擊次數上限為 20，000 次。
- 對於 TN1和 QPU 裝置，每個任務的鏡頭必須 > 0。

Amazon Braket 開發人員指南的文件歷史記錄

下表說明 Amazon Braket 的文件版本。

- 最新API參考更新時間：2025 年 8 月 14 日
- 文件最近更新時間：2025 年 8 月 15 日

變更	Description	日期
在建置區段下移動 Pennylane 和 CUDA-Q 頁面	將 Pennylane 和 CUDA-Q 頁面移至 目錄的建置區段下方。	2025 年 8 月 15 日
新`ProgramSet` 功能	新增對程式集的支援，這是在單一量子任務中執行多個量子電路的操作。	2025 年 8 月 14 日
新裝置 IQM Emerald	新增對IQM Emerald裝置的支援。具有方形 (Crystal) 格狀拓撲的 54 位元裝置。	2025 年 7 月 21 日
已更新 AmazonBraketServiceRolePolicy 政策	AmazonBraketServiceRolePolicy 現在僅向 aws：PrincipalAccount 提供 s3：* 和 log：* 動作。 PrincipalAccount 這只會限制對請求者的儲存貯體和日誌群組的存取。	2025 年 7 月 11 日
新的實驗功能功能：動態電路	中電路測量和前饋操作可作為實驗功能，請參閱存取 IQM 裝置上的動態電路。	2025 年 6 月 26 日
已更新 AmazonBraketFullAccess 政策	AmazonBraketFullAccess 現在包含 pricing：GetProducts，以在主控台上顯示硬體成本。	2025 年 4 月 14 日

新裝置 IonQ Forte-Enterprise-1	新增對 IonQ Forte-Enterprise-1 裝置的支援。利用截獲的離子技術的 36 個 quibit 裝置。	2025 年 3 月 17 日
改善 S3 條件許可	為了提高安全性，`AmazonBraketFullAccess` 現在只提供 `s3:*`動作給 `aws:PrincipalAccount` 。這只會限制對請求者自有儲存貯體的存取。	2025 年 3 月 7 日
新裝置 Rigetti Ankaa-3	新增對Rigetti Ankaa-3裝置的支援。利用可擴展多晶片技術的 84 個 quibit 裝置。	2025 年 1 月 14 日
Rigetti Ankaa-2 裝置淘汰	已移除Rigetti Ankaa-2裝置的支援。	2025 年 1 月 14 日
支援 IPv6 流量	Amazon Braket 現在支援使用雙堆疊端點 `braket.{region}.api.aws` 的 IPv6 流量。	2024 年 12 月 12 日
NVIDIA's CUDA-Q Amazon Braket 上的 支援	客戶現在可以在 Amazon Braket 上使用NVIDIA's CUDA-Q開發人員架構來執行量子程式。	2024 年 12 月 6 日
IonQ Forte-1 裝置隨時可用	IonQ Forte-1 裝置不再僅保留，現在隨時可供客戶使用。	2024 年 11 月 22 日
Rigetti Aspen-M-3 裝置淘汰	已移除對Rigetti Aspen-M-3裝置的支援。	2024 年 9 月 27 日
IonQ Harmony 裝置淘汰	已移除對IonQ Harmony裝置的支援。	2024 年 8 月 29 日

新裝置 Rigetti Ankaa-2	新增對Rigetti Ankaa-2裝置的支援。利用可擴展多晶片技術的 84 個 quibit 裝置。	2024 年 8 月 26 日
開發人員指南重組	新的開發人員指南採用現有的建置、測試、執行客戶旅程，並透過 Amazon Braket 引導使用者沿著此路徑前進。	2024 年 8 月 23 日
OQC Lucy 裝置淘汰	已移除OQC Lucy裝置的支援。	2024 年 6 月 28 日
新裝置IQM Garnet和區域 Europe North 1	新增對 IQM Garnet 裝置的支援。具有方形格狀拓撲的 20 qubit 裝置。將 Braket 支援的區域擴展至歐洲北部 1 （斯德哥爾摩）。	2024 年 5 月 22 日
本機調整已釋放	實驗功能現在包含 QuEra Aquila QPU 的本機調整功能。	2024 年 4 月 11 日
已釋出筆記本閒置管理員	建立筆記本執行個體時，請啟用閒置管理員並設定閒置持續時間，以自動重設 Braket 筆記本執行個體。	2024 年 3 月 27 日
重做內容表	重組 Amazon Braket 目錄，以遵守 AWS 風格指南要求，並改善內容流程以提供客戶體驗。	2023 年 12 月 12 日
Braket 直接釋放	新增對 Braket 直接功能的支援，包括： • 使用保留 • 取得專家建議 • 探索實驗功能	2023 年 11 月 27 日

已更新 建立 Amazon Braket 筆記本執行個體	更新文件以新增資訊，為新的和現有的 Amazon Braket 客戶建立筆記本執行個體。	2023 年 11 月 27 日
已更新 使用您自己的容器 (BYOC)	更新文件，以新增何時到 BYOC、配方到 BYOC，以及在容器上執行 Braket 混合任務的相關資訊。	2023 年 10 月 18 日
混合任務裝飾項目已發佈	新增將本機程式碼作為混合任務執行頁面。包含範例： • 從本機 Python 程式碼建立混合任務 • 安裝其他 Python 套件和原始程式碼 • 將資料儲存並載入混合任務執行個體 • 混合式任務裝飾項目的最佳實務	2023 年 10 月 16 日
新增佇列可見性	更新開發人員指南文件，以包含 queue depth和 queue position。 更新 API 括號，以反映佇列可見性的新 API 變更。	2023 年 9 月 25 日
標準化文件中的命名	更新文件，將任何 "job" 執行個體變更為 "hybrid job"，並將 "task" 變更為 "quantum task"	2023 年 9 月 11 日
新裝置 IonQ Aria 2	新增對IonQ Aria 2裝置的支援	2023 年 9 月 8 日
更新原生閘道	更新文件以新增從 Rigetti 對原生閘道進行程式設計存取的相關資訊。	2023 年 8 月 16 日

Xanadu 離開	更新文件以移除所有Xanadu裝置	2023 年 6 月 2 日
新裝置 IonQ Aria	新增對IonQ Aria裝置的支援	2023 年 5 月 16 日
淘汰Rigetti的裝置	停止對 的支援 Rigetti Aspen-M-2	2023 年 5 月 2 日
已更新 AmazonBraketFullAccess 政策資訊	已更新定義 AmazonBraketFullAccess 政策內容的指令碼，以包含 servicequotas：GetServiceQuota 和 cloudwatch：GetMetricData 動作，以及有關配額限制的資訊。	2023 年 4 月 19 日
引導式旅程啟動	變更文件以反映更最新且簡化的 Braket 加入方法。	2023 年 4 月 5 日
新裝置 Rigetti Aspen-M-3	新增對Rigetti Aspen-M-3裝置的支援	2023 年 1 月 17 日
新的並行漸層功能	新增 提供的輔助漸層功能的相關資訊 SV1	2022 年 12 月 7 日
新的演算法程式庫功能	新增有關 Braket 演算法程式庫的資訊，該程式庫提供預先建置量子演算法的目錄	2022 年 11 月 28 日
D-Wave 離開	更新文件以適應移除所有D-Wave裝置	2022 年 11 月 17 日
新裝置 QuEra Aquila	新增對QuEra Aquila裝置的支援	2022 年 10 月 31 日
支援 Braket Pulse	新增對 Braket Pulse 的支援，允許在 Rigetti和 OQC 裝置上使用脈衝控制	2022 年 10 月 20 日

支援 IonQ 原生閘道	新增支援 IonQ 裝置提供的原生閘道集	2022 年 9 月 13 日
新的執行個體配額	更新與混合任務相關聯的預設傳統運算執行個體配額	2022 年 8 月 22 日
新的服務儀表板	更新主控台螢幕擷取畫面以包含服務儀表板	2022 年 8 月 17 日
新裝置 Rigetti Aspen-M-2	新增對Rigetti Aspen-M-2裝置的支援	2022 年 8 月 12 日
新的 OpenQASM 功能	新增本機模擬器 (braket_sv 和 braket_dm) 的 OpenQASM 功能支援	2022 年 8 月 4 日
新的成本追蹤程序	新增如何取得模擬器和硬體工作負載的近乎即時的成本預估上限	2022 年 7 月 18 日
新Xanadu Borealis裝置	新增對Xanadu Borealis裝置的支援	2022 年 6 月 2 日
新的加入簡化程序	新增新加入程序和簡化加入程序如何運作的相關資訊	2022 年 5 月 16 日
新裝置 D-Wave Advantage _system6.1	新增對D-WaveAdvantage _system6.1裝置的支援	2022 年 5 月 12 日
支援內嵌模擬器	新增如何使用混合任務執行內嵌模擬，以及如何使用 PennyLane 閃電模擬器	2022 年 5 月 4 日
AmazonBraketFullAccess - Amazon Braket 的完整存取政策	新增 s3：ListAllMyBuckets 許可，允許使用者檢視和檢查為 Amazon Braket 建立和使用的儲存貯體	2022 年 3 月 31 日

支援 OpenQASM	新增對閘道式量子裝置和模擬器的 OpenQASM 3.0 支援	2022 年 3 月 7 日
新的 Quantum 硬體供應商 Oxford Quantum Circuits和新的區域，eu-west-2	新增對 OQC 和 eu-west-2 的支援	2022 年 2 月 28 日
新Rigetti裝置	新增對 Rigetti Aspen M-1 的支援	2022 年 2 月 15 日
新的資源限制	將並行DM1和SV1任務的數量上限從 55 個增加到 100 個	2022 年 1 月 5 日
新Rigetti裝置	新增對 Rigetti Aspen-11 的支援	2021 年 12 月 20 日
已淘汰Rigetti的裝置	停止對Rigetti Aspen-10裝置的支援	2021 年 12 月 20 日
新的結果類型	本機密度矩陣模擬器和DM1裝置支援的低密度矩陣結果類型	2021 年 12 月 20 日
更新政策描述	Amazon Braket 已更新角色 ARN 以包含`servicerole/` 路徑。如需政策更新的資訊，請參閱 Amazon Braket 受 AWS 管政策更新表格。	2021 年 11 月 29 日
Amazon Braket 任務	Amazon Braket Hybrid Jobs 和 API 新增的使用者指南	2021 年 11 月 29 日
新Rigetti裝置	新增對 Rigetti Aspen-10 的支援	2021 年 11 月 20 日
淘汰D-Wave的裝置	停止對 D-Wave QPU 的支援，Advantage_system1	2021 年 11 月 4 日
新D-Wave裝置	新增對其他 D-Wave QPU 的支援， Advantage_system4	2021 年 10 月 5 日

新的雜訊模擬器	新增對密度矩陣模擬器 (DM1) 的支援，可模擬高達 17 的電路qubits和本機雜訊模擬器 braket_dm	2021 年 5 月 25 日
PennyLane 支援	新增對 Amazon Braket 上的 PennyLane 的支援	2020 年 12 月 8 日
新的模擬器	新增對 Tensor 網路模擬器 (TN1) 的支援，允許更大的電路	2020 年 12 月 8 日
任務批次處理	Braket 支援客戶任務批次處理	2020 年 11 月 24 日
手動qubit配置	Braket 支援Rigetti裝置上的手動qubit配置	2020 年 11 月 24 日
可調整配額	Braket 支援任務資源的自助式可調整配額	2020 年 10 月 30 日
支援 PrivateLink	您可以為您的 Braket 任務設定私有 VPC 端點	2020 年 10 月 30 日
標籤的支援	Braket 支援以 API為基礎的 quantum-task 資源標籤	2020 年 10 月 30 日
新D-Wave裝置	新增對其他 D-Wave QPU 的支援， Advantage_system1	2020 年 9 月 29 日
初始版本	Amazon Braket 文件的初始版本	2020 年 8 月 12 日

本文為英文版的機器翻譯版本，如內容有任何歧義或不一致之處，概以英文版為準。