Utilizzo ottimale di cluster e GPU con Run ai

NetApp Solutions

NetApp
April 26, 2024

Sommario

Utilizzo ottimale di cluster e GPU con Run:ai

Le sezioni seguenti forniscono dettagli sull'installazione Run:ai, sugli scenari di test e sui risultati ottenuti in questa convalida.

Abbiamo validato il funzionamento e le performance di questo sistema utilizzando tool di benchmark standard di settore, inclusi i benchmark TensorFlow. Il set di dati ImageNet è stato utilizzato per formare ResNet-50, un famoso modello DL della rete neurale convoluzionale (CNN) per la classificazione delle immagini. ResNet-50 offre un risultato di training accurato con un tempo di elaborazione più rapido, che ci ha consentito di gestire una domanda sufficiente sullo storage.

Esegui:Installazione ai

Per installare Run:ai, attenersi alla seguente procedura:

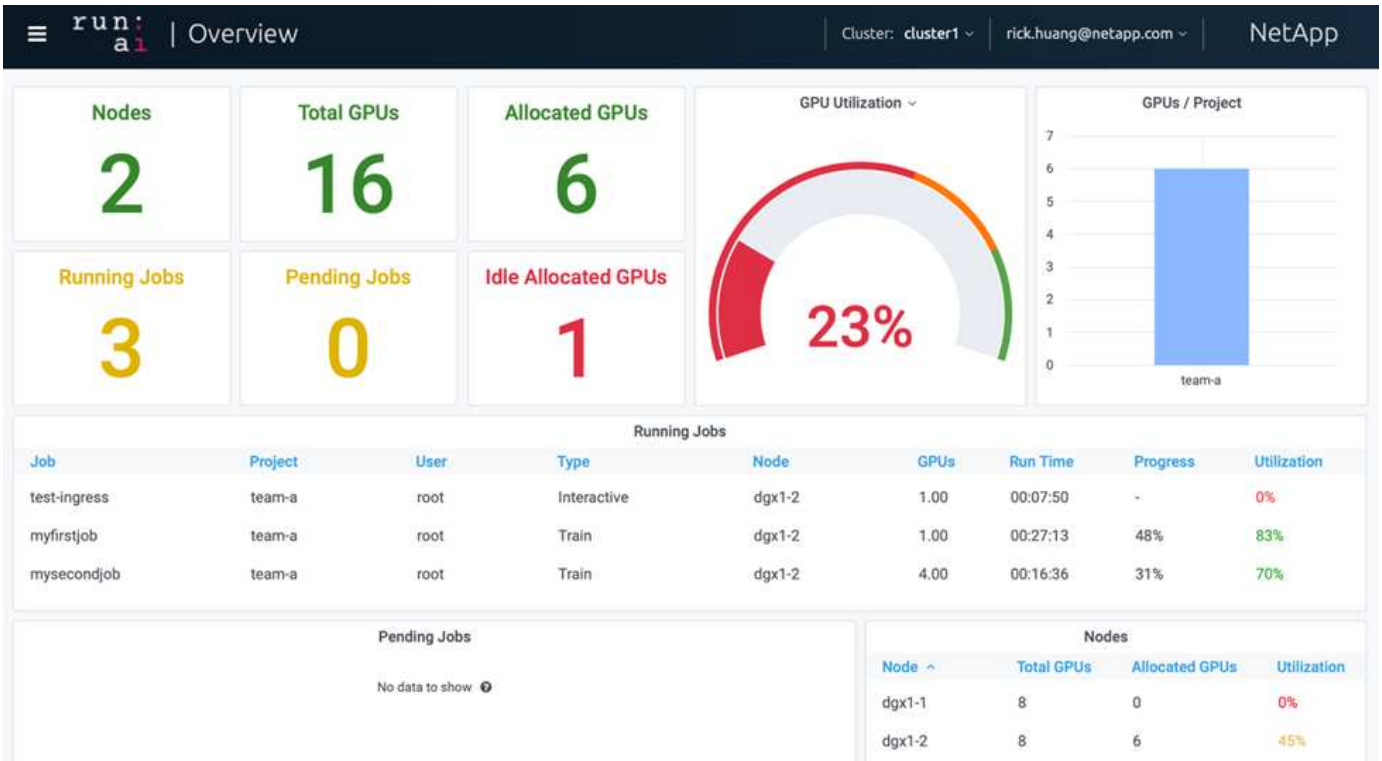
1. Installare il cluster Kubernetes utilizzando DeepOps e configurare la classe di storage predefinita di NetApp.
2. Preparare i nodi GPU:
   a. Verificare che i driver NVIDIA siano installati sui nodi GPU.
   b. Verificare che `nvidia-docker` è installato e configurato come runtime predefinito di docker.
3. Esecuzione dell'installazione:ai:
   a. Accedere a. "Esegui: UI Admin ai" per creare il cluster.
   b. Scarica il creato `runai-operator-<clustername>.yaml` file.
   c. Applicare la configurazione dell'operatore al cluster Kubernetes.

   ```
   kubectl apply -f runai-operator-<clustername>.yaml
   ```

4. Verificare l'installazione:
   a. Passare a. "https://app.run.ai/".
   b. Accedere alla dashboard Panoramica.
   c. Verificare che il numero di GPU in alto a destra rifletta il numero previsto di GPU e che i nodi GPU siano tutti nell'elenco dei server.per ulteriori informazioni sull'implementazione di Run:ai, vedere "Installazione di Run:ai su un cluster Kubernetes on-premise" e. "Installazione della CLI Run:ai".

Run:ai Dashboard e viste

Dopo aver installato Run:ai sul cluster Kubernetes e aver configurato correttamente i container, vengono visualizzate le seguenti dashboard e viste "https://app.run.ai" nel browser, come mostrato nella figura seguente.



Nel cluster sono presenti 16 GPU totali fornite da due nodi DGX-1. È possibile visualizzare il numero di nodi, il totale delle GPU disponibili, le GPU allocate assegnate con i carichi di lavoro, il numero totale di processi in esecuzione, i processi in sospeso e le GPU allocate inattive. Sul lato destro, il diagramma a barre mostra le GPU per progetto, che riepiloga il modo in cui i diversi team utilizzano la risorsa del cluster. Al centro è riportato l'elenco dei job attualmente in esecuzione con i relativi dettagli, inclusi nome del job, progetto, utente, tipo di job, Il nodo su cui ciascun processo è in esecuzione, il numero di GPU allocate per tale processo, il tempo di esecuzione corrente del processo, l'avanzamento del processo in percentuale e l'utilizzo della GPU per tale processo. Si noti che il cluster è sottoutilizzato (utilizzo della GPU al 23%) perché ci sono solo tre job in esecuzione inviati da un singolo team (`team-a`).

Nella sezione seguente, mostreremo come creare più team nella scheda progetti e allocare GPU per ciascun team per massimizzare l'utilizzo del cluster e gestire le risorse quando sono presenti molti utenti per cluster. Gli scenari di test imitano gli ambienti aziendali in cui le risorse di memoria e GPU sono condivise tra carichi di lavoro di training, deduzione e interattivi.

Creazione di progetti per i team Data Science e allocazione delle GPU

I ricercatori possono inviare i carichi di lavoro attraverso la CLI Run:ai, Kubeflow o processi simili. Per ottimizzare l'allocazione delle risorse e creare priorità, Run:ai introduce il concetto di progetti. I progetti sono entità di quota che associano un nome di progetto all'allocazione e alle preferenze della GPU. Si tratta di un metodo semplice e conveniente per gestire più team di data science.

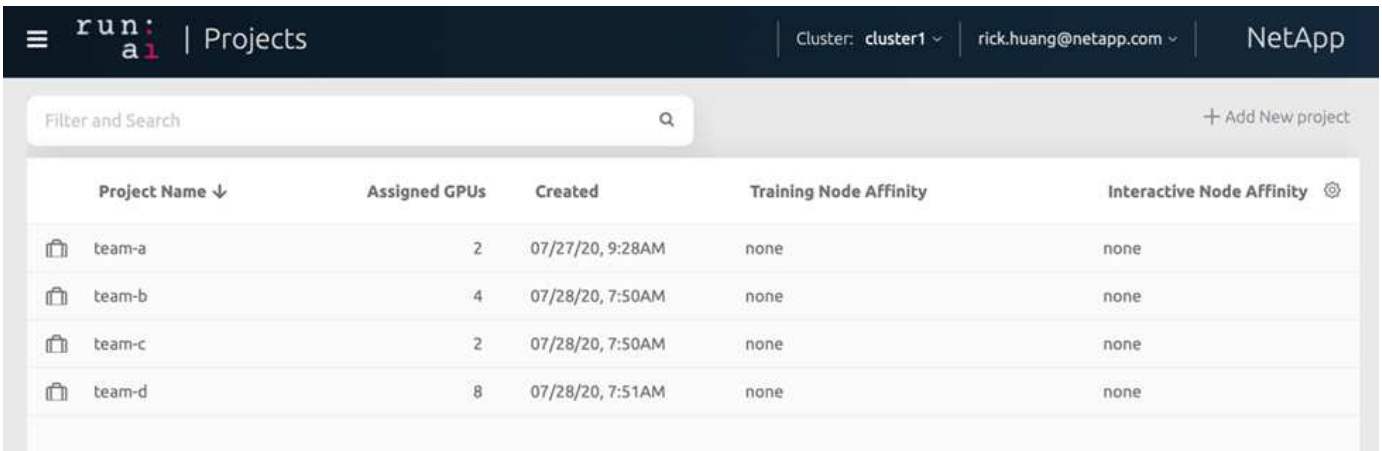
Un ricercatore che invia un workload deve associare un progetto a una richiesta di workload. Lo scheduler Run:ai confronta la richiesta con le allocazioni correnti e il progetto e determina se il carico di lavoro può essere allocato o se deve rimanere in uno stato in sospeso.

In qualità di amministratore di sistema, è possibile impostare i seguenti parametri nella scheda Run:ai Projects

(Esegui: Progetti ai):

- **Model projects.** Imposta un progetto per utente, imposta un progetto per team di utenti e imposta un progetto per un progetto organizzativo reale.
- **Quote di progetto.** ogni progetto è associato a una quota di GPU che può essere allocata per questo progetto contemporaneamente. Si tratta di una quota garantita nel senso che i ricercatori che utilizzano questo progetto possono ottenere questo numero di GPU indipendentemente dallo stato del cluster. Di norma, la somma dell'allocazione del progetto deve essere uguale al numero di GPU nel cluster. Oltre a questo, un utente di questo progetto può ricevere una quota eccessiva. Finché le GPU non vengono utilizzate, un ricercatore che utilizza questo progetto può ottenere più GPU. In vengono illustrati scenari di test con quote superiori e considerazioni di equità "Elevato utilizzo del cluster con allocazione della GPU con quota eccessiva", "Equità nell'allocazione delle risorse di base", e. "Equità nell'overquota".
- Creare un nuovo progetto, aggiornare un progetto esistente ed eliminare un progetto esistente.
- **Limita l'esecuzione dei job su gruppi di nodi specifici**. È possibile assegnare progetti specifici da eseguire solo su nodi specifici. Ciò è utile quando il team di progetto ha bisogno di hardware specializzato, ad esempio con memoria sufficiente. In alternativa, un team di progetto potrebbe essere il proprietario di hardware specifico acquistato con un budget specializzato, oppure quando potrebbe essere necessario indirizzare i carichi di lavoro di build o interattivi per lavorare su hardware più debole e indirizzare i carichi di lavoro di formazione più lunghi o non presidiati su nodi più veloci. Per i comandi per raggruppare i nodi e impostare l'affinità per un progetto specifico, vedere "Run:documentazione ai".
- **Limitare la durata dei lavori interattivi**. I ricercatori spesso si dimenticano di chiudere lavori interattivi. Ciò potrebbe comportare uno spreco di risorse. Alcune organizzazioni preferiscono limitare la durata dei lavori interattivi e chiuderli automaticamente.

La figura seguente mostra la vista progetti con quattro team creati. A ciascun team viene assegnato un numero diverso di GPU per i diversi carichi di lavoro, con il numero totale di GPU pari a quello delle GPU totali disponibili in un cluster costituito da due DGX-1.



Invio di job in Run:ai CLI

Questa sezione fornisce i dettagli sui comandi Run:ai di base che è possibile utilizzare per eseguire qualsiasi lavoro Kubernetes. È suddiviso in tre parti in base al tipo di carico di lavoro. I carichi di lavoro ai/ML/DL possono essere suddivisi in due tipi generici:

- **Sessioni di training non presidiate**. Con questi tipi di carichi di lavoro, il data scientist prepara un carico di lavoro a esecuzione automatica e lo invia per l'esecuzione. Durante l'esecuzione, il cliente può esaminare i risultati. Questo tipo di carico di lavoro viene spesso utilizzato in produzione o quando lo

sviluppo del modello si trova in una fase in cui non è richiesto alcun intervento umano.

- **Sessioni di build interattive**. Con questi tipi di carichi di lavoro, il data scientist apre una sessione interattiva con Bash, Jupyter notebook, PyCharm remoto o IDE simili e accede direttamente alle risorse GPU. Abbiamo incluso un terzo scenario per l'esecuzione di workload interattivi con porte connesse per rivelare una porta interna all'utente del container.

Carichi di lavoro di training non presidiati

Dopo aver impostato i progetti e allocato le GPU, è possibile eseguire qualsiasi carico di lavoro Kubernetes utilizzando il seguente comando nella riga di comando:

```
$ runai project set team-a runai submit hyper1 -i gcr.io/run-ai-
demo/quickstart -g 1
```

Questo comando avvia un processo di training non assistito per il team-a con un'allocazione di una singola GPU. Il lavoro si basa su un'immagine del docker di esempio, `gcr.io/run-ai-demo/quickstart`. Abbiamo nominato il lavoro `hyper1`. È quindi possibile monitorare l'avanzamento del lavoro eseguendo il seguente comando:

```
$ runai list
```

La figura seguente mostra il risultato di `runai list` comando. Gli stati tipici che potrebbero essere visualizzati includono:

- `ContainerCreating`. Il container del docker viene scaricato dal repository cloud.
- `Pending`. Il lavoro è in attesa di essere pianificato.
- `Running`. Il processo è in esecuzione.

```
~> runai list
Showing jobs for project team-a
NAME    STATUS   AGE  NODE                                    IMAGE                        TYPE        PROJECT  USER   GPUs
hyper1  Running  11s  gke-dev-yaron1-gpu-4-pool-154f511d-5nk5  gcr.io/run-ai-demo/quickstart  Train       team-a   yaron  1
```

Per ottenere uno stato aggiuntivo sul lavoro, eseguire il seguente comando:

```
$ runai get hyper1
```

Per visualizzare i log del lavoro, eseguire `runai logs <job-name>` comando:

```
$ runai logs hyper1
```

In questo esempio, dovresti visualizzare il registro di una sessione DL in esecuzione, inclusi l'epoca di training corrente, l'ETA, il valore della funzione di perdita, l'accuratezza e il tempo trascorso per ogni fase.

È possibile visualizzare lo stato del cluster nell'interfaccia utente Run:ai all'indirizzo "https://app.run.ai/". In Dashboard > Panoramica, è possibile monitorare l'utilizzo della GPU.

Per arrestare questo carico di lavoro, eseguire il seguente comando:

```
$ runai delte hyper1
```

Questo comando interrompe il carico di lavoro del training. È possibile verificare questa azione eseguendo `runai list` di nuovo. Per ulteriori informazioni, vedere "lancio di workload di training non presidiati".

Workload di build interattivi

Dopo aver impostato i progetti e allocato le GPU, è possibile eseguire un carico di lavoro di build interattivo utilizzando il seguente comando dalla riga di comando:

```
$ runai submit build1 -i python -g 1 --interactive --command sleep --args
infinity
```

Il lavoro si basa su un python immagine del docker di esempio. Abbiamo chiamato la creazione di job 1.

> Il `-- interactive` flag indica che il lavoro non ha inizio o fine È responsabilità del ricercatore chiudere il lavoro. L'amministratore può definire un limite di tempo per i lavori interattivi dopo il quale vengono terminati dal sistema.

Il `--g 1` Flag assegna una singola GPU a questo lavoro. Il comando e l'argomento forniti sono `--command sleep—args infinity`. È necessario fornire un comando, altrimenti il container viene avviato e quindi chiuso immediatamente.

I seguenti comandi funzionano in modo simile ai comandi descritti in Carichi di lavoro di training non presidiati:

- `runai list`: Mostra il nome, lo stato, l'età, il nodo, l'immagine, Progetto, utente e GPU per i lavori.
- `runai get build1`: Visualizza lo stato aggiuntivo nella creazione del job 1.
- `runai delete build1`: Interrompe la creazione interattiva del workload 1.per ottenere una shell bash nel container, utilizzare il seguente comando:

```
$ runai bash build1
```

Questo fornisce una shell diretta nel computer. I data scientist possono quindi sviluppare o perfezionare i propri modelli all'interno del container.

È possibile visualizzare lo stato del cluster nell'interfaccia utente Run:ai all'indirizzo "https://app.run.ai". Per ulteriori informazioni, vedere "avvio e utilizzo di workload di build interattivi".

Carichi di lavoro interattivi con porte connesse

Come estensione dei carichi di lavoro di build interattivi, è possibile rivelare le porte interne all'utente del container quando si avvia un container con la CLI Run:ai. Questo è utile per ambienti cloud, per lavorare con i notebook Jupyter o per connettersi ad altri microservizi. "Ingresso" Consente l'accesso ai servizi Kubernetes dall'esterno del cluster Kubernetes. È possibile configurare l'accesso creando un insieme di regole che definiscono quali connessioni in entrata raggiungono i servizi.

Per una migliore gestione dell'accesso esterno ai servizi in un cluster, si consiglia agli amministratori del cluster di eseguire l'installazione "Ingresso" E configurare LoadBalancer.

Per utilizzare Ingress come tipo di servizio, eseguire il seguente comando per impostare il tipo di metodo e le porte durante l'invio del carico di lavoro:

```
$ runai submit test-ingress -i jupyter/base-notebook -g 1 \
  --interactive --service-type=ingress --port 8888 \
  --args="--NotebookApp.base_url=test-ingress" --command=start-notebook.sh
```

Una volta avviato il container, eseguire `runai list` per visualizzare `SERVICE URL(S)` Con cui accedere al Jupyter notebook. L'URL è composto dall'endpoint di ingresso, dal nome del processo e dalla porta. Ad esempio, vedere https://10.255.174.13/test-ingress-8888.

Per ulteriori informazioni, vedere "lancio di un workload di build interattivo con porte connesse".

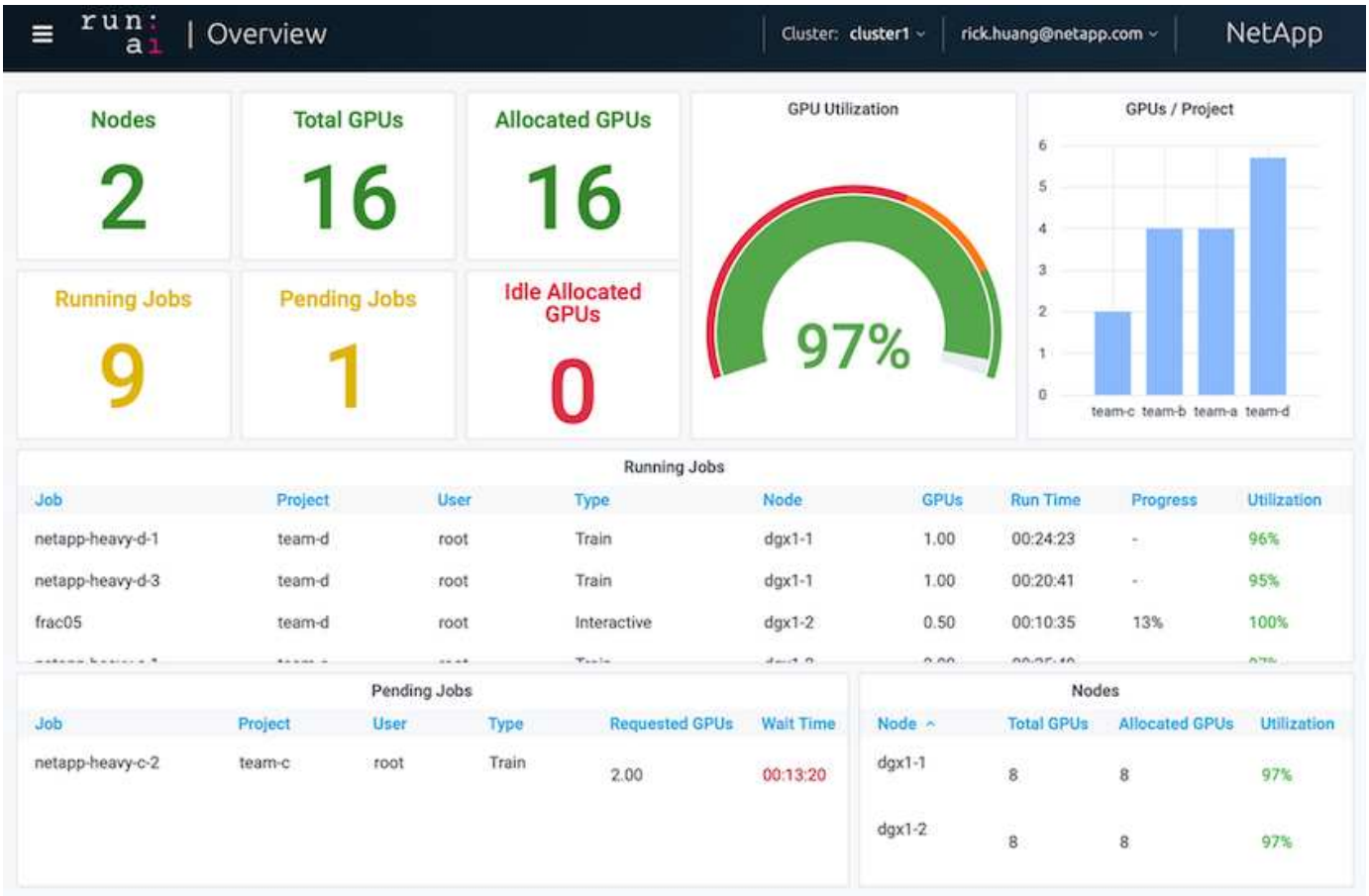
Elevato utilizzo del cluster

In questa sezione, emuliamo uno scenario realistico in cui quattro team di data science inviano ciascuno i propri carichi di lavoro per dimostrare la soluzione di orchestrazione Run:ai che raggiunge un elevato utilizzo del cluster mantenendo al contempo la prioritizzazione e il bilanciamento delle risorse GPU. Iniziamo utilizzando il benchmark ResNet-50 descritto nella sezione "ResNet-50 con ImageNet dataset Benchmark Summary":

```
$ runai submit netapp1 -i netapp/tensorflow-tf1-py3:20.01.0 --local-image
--large-shm  -v /mnt:/mnt -v /tmp:/tmp --command python --args
"/netapp/scripts/run.py" --args "--
dataset_dir=/mnt/mount_0/dataset/imagenet/imagenet_original/" --args "--
num_mounts=2"  --args "--dgx_version=dgx1" --args "--num_devices=1" -g 1
```

Abbiamo eseguito lo stesso benchmark ResNet-50 di in "NVA-1121". Abbiamo utilizzato la bandiera `--local-image` per i container che non risiedono nel repository del dock pubblico. Abbiamo montato le directory `/mnt` e. `/tmp` Sul nodo host DGX-1 a. `/mnt` e. `/tmp` al container, rispettivamente. Il set di dati è disponibile presso NetApp AFFA800 con `dataset_dir` argomento che punta alla directory. Entrambi `--num_devices=1` e. `-g 1` Significa che allociamo una GPU per questo lavoro. Il primo è un argomento per `run.py` script, mentre quest'ultimo è un flag per `runai submit` comando.

La figura seguente mostra una dashboard panoramica del sistema con il 97% di utilizzo della GPU e tutte le sedici GPU disponibili allocate. È possibile visualizzare facilmente il numero di GPU allocate per ciascun team nel grafico a barre GPU/progetto. Il riquadro dei job in esecuzione mostra i nomi dei job in esecuzione, il progetto, l'utente, il tipo, il nodo, GPU consumate, tempo di esecuzione, avanzamento e dettagli di utilizzo. Un elenco dei workload in coda con il relativo tempo di attesa viene visualizzato in lavori in sospeso. Infine, la casella Nodes offre i numeri GPU e l'utilizzo per i singoli nodi DGX-1 nel cluster.



Allocazione frazionale della GPU per carichi di lavoro meno impegnativi o interattivi

Quando ricercatori e sviluppatori stanno lavorando sui propri modelli, sia nelle fasi di sviluppo, tuning di iperparametri o debug, tali carichi di lavoro richiedono di solito meno risorse di calcolo. È quindi più efficiente eseguire il provisioning di GPU e memoria frazionarie in modo che la stessa GPU possa essere allocata contemporaneamente ad altri carichi di lavoro. La soluzione di orchestrazione di Run:ai offre un sistema di condivisione della GPU frazionale per carichi di lavoro containerizzati su Kubernetes. Il sistema supporta i carichi di lavoro che eseguono programmi CUDA ed è particolarmente adatto per attività ai leggere come inferenza e costruzione di modelli. Il sistema di GPU frazionale offre ai team di progettazione ai e data science la possibilità di eseguire più carichi di lavoro contemporaneamente su una singola GPU. Ciò consente alle aziende di eseguire più carichi di lavoro, ad esempio visione artificiale, riconoscimento vocale ed elaborazione del linguaggio naturale sullo stesso hardware, riducendo così i costi.

Esegui: Il sistema di GPU frazionale di ai crea efficacemente GPU logiche virtualizzate con la propria memoria e spazio di calcolo che i container possono utilizzare e accedere come se fossero processori autonomi. In questo modo, è possibile eseguire diversi carichi di lavoro in container, uno accanto all'altro, sulla stessa GPU senza interferire l'uno con l'altro. La soluzione è trasparente, semplice e portatile e non richiede modifiche ai container stessi.

Un'usecase tipica potrebbe visualizzare da due a otto lavori in esecuzione sulla stessa GPU, il che significa

che è possibile eseguire otto volte il lavoro con lo stesso hardware.

Per il lavoro `frac05` appartenente al progetto `team-d` Nella figura seguente, è possibile vedere che il numero di GPU allocate era 0.50. Questo è ulteriormente verificato da `nvidia-smi` Che indica che la memoria GPU disponibile per il container era di 16,255 MB: Metà dei 32 GB per GPU V100 nel nodo DGX-1.

```
root@run-deploy:~# runai bash frac05 -p team-d
root@frac05-0:/workload# nvidia-smi
Tue Jul 28 15:17:03 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 450.51.05    Driver Version: 450.51.05    CUDA Version: 11.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla V100-SXM2...  On   | 00000000:07:00.0 Off |                    0 |
| N/A   57C    P0   240W / 300W |  15525MiB / 16255MiB |    100%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A       156      C   python3                         15525MiB |
+-----------------------------------------------------------------------------+
```

Elevato utilizzo del cluster con allocazione della GPU con quota eccessiva

In questa sezione e nelle sezioni "Equità nell'allocazione delle risorse di base", e. "Equità nell'overquota", Abbiamo ideato scenari di test avanzati per dimostrare le funzionalità di orchestrazione Run:ai per la gestione di workload complessi, la pianificazione preventiva automatica e il provisioning di GPU con overquota. Lo abbiamo fatto per ottenere un elevato utilizzo delle risorse del cluster e ottimizzare la produttività del team di data science di livello Enterprise in un ambiente ai ONTAP.

Per queste tre sezioni, impostare i seguenti progetti e quote:

Progetto	Quota
squadra a.	4
team-b	2
team-c	2

Progetto	Quota
team-d	8

Inoltre, per queste tre sezioni vengono utilizzati i seguenti container:

- Notebook Jupyter: `jupyter/base-notebook`
- Run:Avvio rapido ai: `gcr.io/run-ai-demo/quickstart`

Per questo scenario di test sono stati stabiliti i seguenti obiettivi:

- Mostra la semplicità del provisioning delle risorse e il modo in cui le risorse vengono estratte dagli utenti
- Mostrare come gli utenti possono eseguire facilmente il provisioning di frazioni di GPU e numero intero di GPU
- Mostra come il sistema elimina i colli di bottiglia di calcolo consentendo a team o utenti di superare la quota di risorse se nel cluster sono presenti GPU gratuite
- Mostra come vengono eliminati i colli di bottiglia della pipeline di dati utilizzando la soluzione NetApp durante l'esecuzione di processi a elaborazione intensiva, come il container NetApp
- Mostrare come vengono eseguiti diversi tipi di container utilizzando il sistema
  - Notebook Jupyter
  - Container Run:ai
- Mostra un utilizzo elevato quando il cluster è pieno

Per informazioni dettagliate sulla sequenza di comandi effettiva eseguita durante il test, vedere "Dettagli sui test per la Sezione 4.8".

Una volta inviati tutti i 13 carichi di lavoro, è possibile visualizzare un elenco di nomi di container e GPU allocati, come mostrato nella figura seguente. Disponiamo di sette corsi di formazione e sei lavori interattivi, che simulano quattro team di data science, ciascuno con i propri modelli in esecuzione o in fase di sviluppo. Per i lavori interattivi, i singoli sviluppatori utilizzano Jupyter Notebooks per scrivere o eseguire il debug del codice. Pertanto, è adatto per eseguire il provisioning delle frazioni GPU senza utilizzare troppe risorse del cluster.

```
root@run-deploy:~# runai list -A
NAME           STATUS   AGE  NODE    IMAGE                          TYPE        PROJECT  USER  GPUs  CREATED BY CLI  SERVICE URL(S)
b-4-gg         Running  2m   dgx1-2  gcr.io/run-ai-demo/quickstart  Train       team-b   root  2     true
c-5-g          Running  2m   dgx1-2  gcr.io/run-ai-demo/quickstart  Train       team-c   root  1     true
c-4-gg         Running  2m   dgx1-1  gcr.io/run-ai-demo/quickstart  Train       team-c   root  2     true
b-3-g          Running  2m   dgx1-1  gcr.io/run-ai-demo/quickstart  Train       team-b   root  1     true
c-3-g02        Running  2m   dgx1-1  gcr.io/run-ai-demo/quickstart  Interactive team-c   root  0.2   true
d-1-gggg       Running  2m   dgx1-2  gcr.io/run-ai-demo/quickstart  Train       team-d   root  4     true
c-2-g03        Running  2m   dgx1-1  gcr.io/run-ai-demo/quickstart  Interactive team-c   root  0.3   true
c-1-g05        Running  2m   dgx1-1  gcr.io/run-ai-demo/quickstart  Interactive team-c   root  0.5   true
a-2-gg         Running  3m   dgx1-1  gcr.io/run-ai-demo/quickstart  Train       team-a   root  2     true
b-2-g04        Running  3m   dgx1-2  gcr.io/run-ai-demo/quickstart  Interactive team-b   root  0.4   true
a-1-g          Running  3m   dgx1-1  gcr.io/run-ai-demo/quickstart  Train       team-a   root  1     true
b-1-g06        Running  3m   dgx1-2  gcr.io/run-ai-demo/quickstart  Interactive team-b   root  0.6   true
a-1-1-jupyter  Running  3m   dgx1-1  jupyter/base-notebook          Interactive team-a   root  1     true            http://10.61.218.134/a-1-1-jupyter,
https://10.61.218.134/a-1-1-jupyter
```

I risultati di questo scenario di test mostrano quanto segue:

- Il cluster deve essere pieno: Vengono utilizzate 16/16 GPU.
- Elevato utilizzo del cluster.
- Più esperimenti rispetto alle GPU a causa dell'allocazione frazionale.

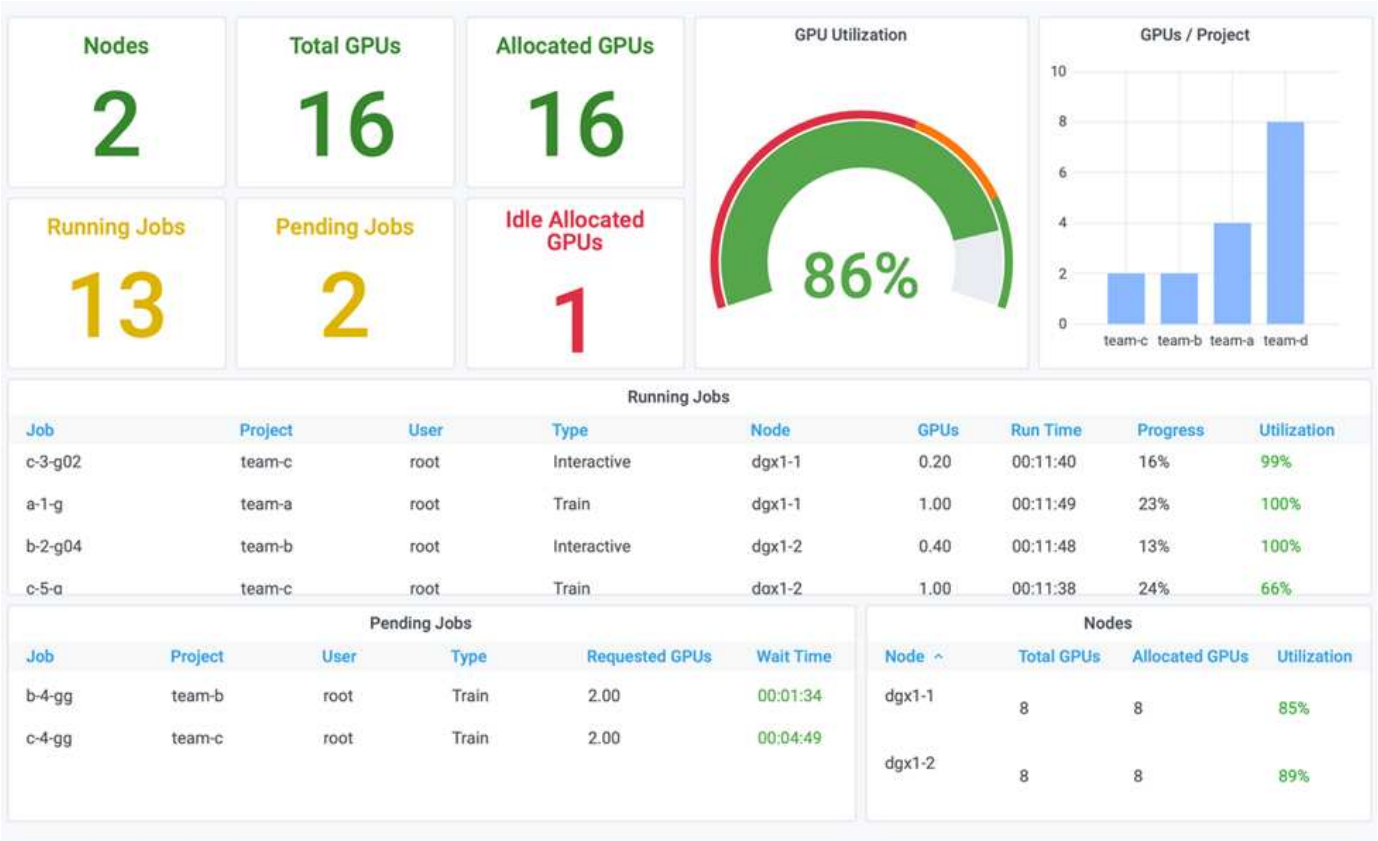
- `team-d` non utilizza tutta la quota; pertanto, `team-b` e. `team-c` Possibilità di utilizzare GPU aggiuntive per i propri esperimenti, con conseguente riduzione dei tempi di innovazione.

Equità nell'allocazione delle risorse di base

In questa sezione, mostreremo quando `team-d` Richiede più GPU (sono sotto la loro quota), il sistema mette in pausa i carichi di lavoro di `team-b` e. `team-c` e li sposta in uno stato in sospeso in modo equo e condiviso.

Per ulteriori informazioni, tra cui invio dei job, immagini container utilizzate e sequenze di comandi eseguite, vedere la sezione "Dettagli sui test per la Sezione 4.9".

La figura seguente mostra l'utilizzo del cluster risultante, le GPU allocate per team e i processi in sospeso a causa del bilanciamento automatico del carico e della pianificazione preventiva. Possiamo osservare che quando il numero totale di GPU richieste da tutti i carichi di lavoro del team supera il numero totale di GPU disponibili nel cluster, l'algoritmo di equità interna di Run:ai mette in pausa un job per ciascuno `team-b` e. `team-c` perché hanno soddisfatto la quota di progetto. In questo modo si ottiene un elevato utilizzo generale del cluster, mentre i team di data science continuano a lavorare sotto i limiti delle risorse stabiliti da un amministratore.



I risultati di questo scenario di test dimostrano quanto segue:

- **Bilanciamento automatico del carico.** il sistema bilancia automaticamente la quota delle GPU, in modo che ogni team utilizzi ora la propria quota. I carichi di lavoro che sono stati sospesi appartengono ai team che hanno superato la quota.
- **Fair share pause.** il sistema sceglie di arrestare il carico di lavoro di un team che ha superato la quota e quindi di arrestare il carico di lavoro dell'altro team. Run:ai dispone di algoritmi interni per la correttezza.

Equità nell'overquota

In questa sezione, espandiamo lo scenario in cui più team inviano carichi di lavoro e superano la loro quota. In questo modo, dimostreremo come l'algoritmo di equità di Run:ai alloca le risorse del cluster in base al rapporto delle quote preimpostate.

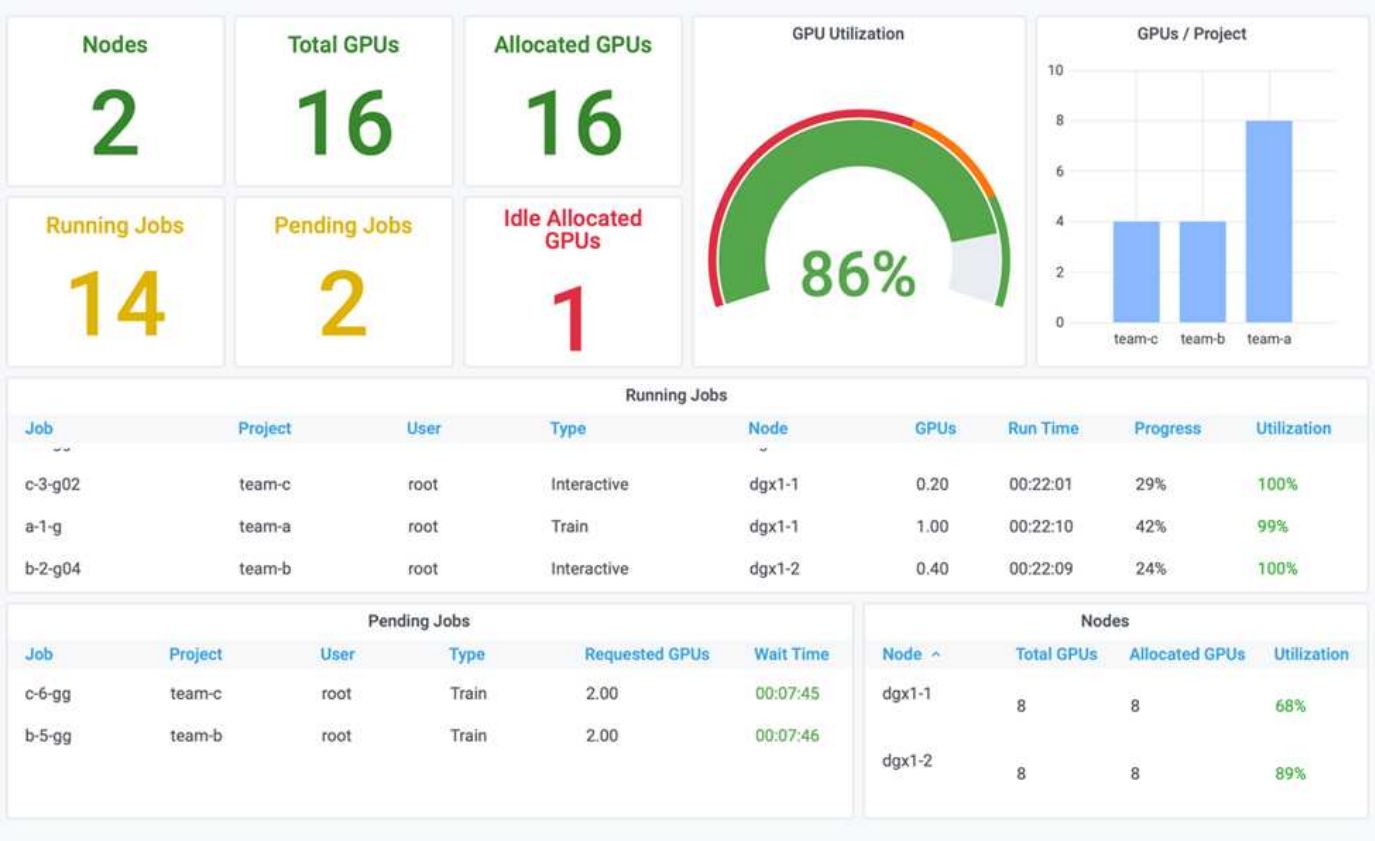
Obiettivi per questo scenario di test:

- Mostra il meccanismo di accodamento quando più team richiedono GPU sulla propria quota.
- Mostrare come il sistema distribuisce una quota equa del cluster tra più team che superano la quota in base al rapporto tra le quote, in modo che il team con la quota maggiore ottenga una quota maggiore della capacità di riserva.

Alla fine di "Equità nell'allocazione delle risorse di base", sono presenti due carichi di lavoro in coda: uno per `team-b` e uno per `team-c`. In questa sezione, vengono accodati carichi di lavoro aggiuntivi.

Per ulteriori informazioni, tra cui invio di lavori, immagini container utilizzate e sequenze di comandi eseguite, vedere "Dettagli sui test per la sezione 4.10".

Quando tutti i lavori vengono inoltrati in base alla sezione "Dettagli sui test per la sezione 4.10", il dashboard di sistema lo mostra `team-a`, `team-b`, e. `team-c` Tutti hanno più GPU rispetto alla quota preimpostata. `team-a` Occupa quattro GPU in più rispetto alla quota soft preimpostata (quattro), mentre `team-b` e. `team-c` Ciascuna di esse occupa due GPU in più rispetto alla propria quota soft (due). Il rapporto delle GPU con overquota allocate è uguale a quello della quota preimpostata. Questo perché il sistema ha utilizzato la quota preimpostata come riferimento di priorità e fornito di conseguenza quando più team richiedono più GPU, superando la quota. Tale bilanciamento automatico del carico offre equità e prioritizzazione quando i team di data science aziendali sono attivamente impegnati nello sviluppo e nella produzione di modelli di ai.



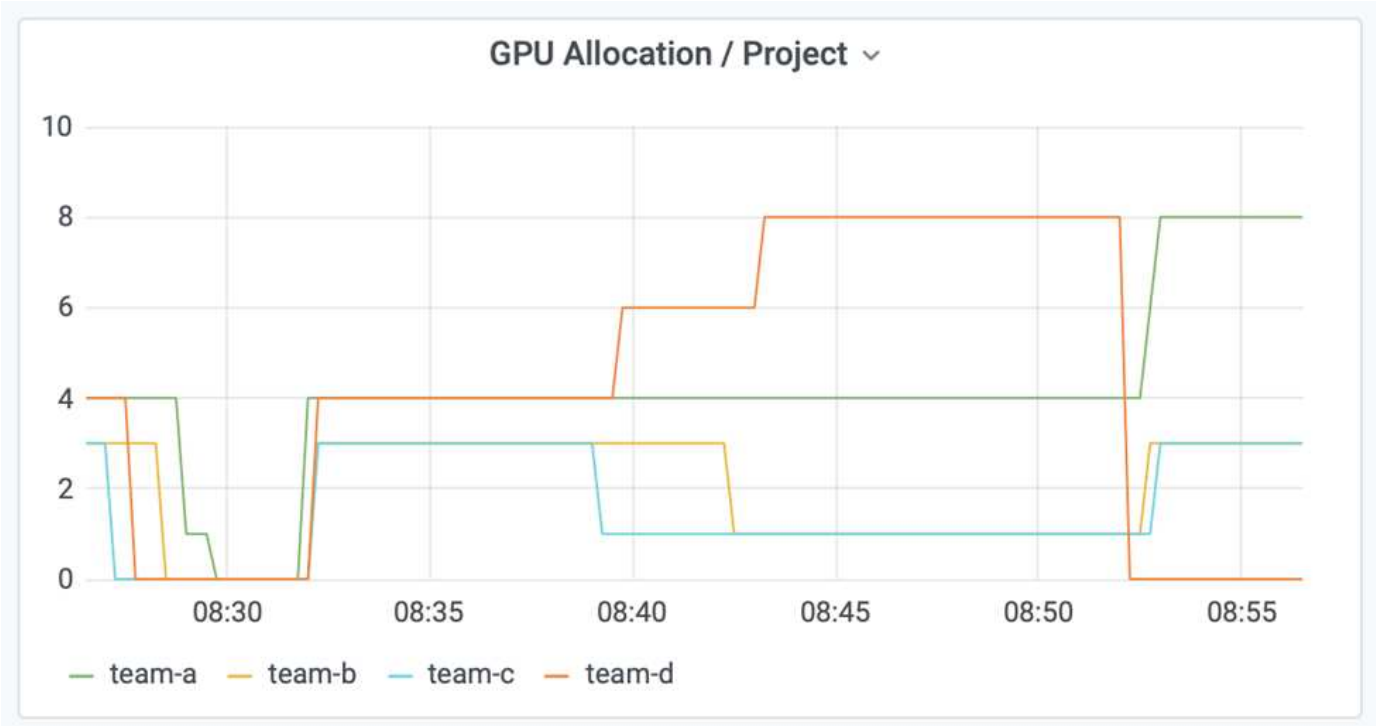
I risultati di questo scenario di test mostrano quanto segue:

- Il sistema inizia a demettere in coda i carichi di lavoro di altri team.
- L'ordine di dequeuing viene stabilito in base agli algoritmi di equità, in modo che `team-b` e. `team-c` Ottenere la stessa quantità di GPU con quote eccessive (poiché hanno una quota simile), e. `team-a` Ottiene una quantità doppia di GPU poiché la loro quota è due volte superiore alla quota di `team-b` e. `team-c`.
- Tutta l'allocazione viene eseguita automaticamente.

Pertanto, il sistema dovrebbe stabilizzarsi nei seguenti stati:

Progetto	GPU allocate	Commento
squadra a.	8/4	Quattro GPU oltre la quota. Coda vuota.
team-b	4/2	Due GPU oltre la quota. Un carico di lavoro in coda.
team-c	4/2	Due GPU oltre la quota. Un carico di lavoro in coda.
team-d	0/8	Non utilizzare le GPU, non i carichi di lavoro in coda.

La figura seguente mostra l'allocazione della GPU per progetto nel tempo nella dashboard Run:ai Analytics per le sezioni "Elevato utilizzo del cluster con allocazione della GPU con quota eccessiva", "Equità nell'allocazione delle risorse di base", e. "Equità nell'overquota". Ciascuna riga della figura indica il numero di GPU fornite per un determinato team di data science in qualsiasi momento. Possiamo vedere che il sistema alloca dinamicamente le GPU in base ai carichi di lavoro inviati. Ciò consente ai team di superare la quota quando nel cluster sono disponibili GPU, quindi di prevenire i lavori in base all'equità, prima di raggiungere infine uno stato stabile per tutti e quattro i team.



Salvataggio dei dati in un PersistentVolume con provisioning Trident

NetApp Trident è un progetto open source completamente supportato, progettato per aiutarti a soddisfare le sofisticate esigenze di persistenza delle tue applicazioni containerizzate. È possibile leggere e scrivere i dati su un volume di prestazioni (PV) Kubernetes con provisioning Trident, con il vantaggio aggiunto di tiering dei dati, crittografia, tecnologia Snapshot NetApp, conformità e performance elevate offerte dal software di gestione dei dati NetApp ONTAP.

Riutilizzo dei PVC in uno spazio dei nomi esistente

Per i progetti ai più grandi, potrebbe essere più efficiente per diversi container leggere e scrivere i dati sullo stesso PV Kubernetes. Per riutilizzare un PVC (Persistent Volume Claim) Kubernetes, l'utente deve aver già creato un PVC. Vedere "Documentazione di NetApp Trident" Per informazioni dettagliate sulla creazione di un PVC. Ecco un esempio di riutilizzo di un PVC esistente:

```
$ runai submit pvc-test -p team-a --pvc test:/tmp/pvc1mount -i gcr.io/run-
ai-demo/quickstart -g 1
```

Eseguire il seguente comando per visualizzare lo stato del lavoro `pvc-test` per il progetto `team-a`:

```
$ runai get pvc-test -p team-a
```

Dovrebbe essere montato PV /tmp/pvc1mount su `team-a` lavoro `pvc-test`. In questo modo, più container possono leggere dallo stesso volume, il che è utile quando ci sono più modelli concorrenti in fase di sviluppo o in produzione. Gli scienziati dei dati possono creare un insieme di modelli e quindi combinare i risultati delle previsioni con il voto a maggioranza o altre tecniche.

Per accedere alla shell container, utilizzare quanto segue:

```
$ runai bash pvc-test -p team-a
```

È quindi possibile controllare il volume montato e accedere ai dati all'interno del container.

Questa funzionalità di riutilizzo dei PVC funziona con i volumi NetApp FlexVol e NetApp ONTAP FlexGroup, consentendo ai data engineer di utilizzare opzioni di gestione dei dati più flessibili e solide per sfruttare il data fabric basato su NetApp.