Un approccio serverless per l'integrazione con GitLab su AWS

27 Maggio 2022 - 11 min. read

Amazon ECS | AWS Fargate | CI/CD | Containers | Docker

L'ottimizzazione dei costi e l'eccellenza operativa sono fattori determinanti per una strategia vincente, che porti all'adozione del paradigma Cloud. I servizi gestiti serverless consentono di ridurre drasticamente i costi e di velocizzare le operazioni di mantenimento dell'infrastruttura.

In questo articolo descriveremo come integrare le pipeline GitLab su AWS usando ECS Fargate in uno scenario multi-ambiente.

GitLab, per quanto riguarda l'utilizzo di risorse computazionali, offre una grande flessibilità: le pipeline possono essere eseguite da cluster Kuberenetes, Docker, su virtual machine on prem o su piattaforme personalizzate usando driver custom.

La soluzione più diffusa per l'esecuzione di pipeline sul Cloud AWS prevede l'utilizzo di istanze EC2. Questo approccio porta ad alcune inefficienze: avviare le istanze on-demand aumenta il tempo di esecuzione, rendendo impazienti gli sviluppatori (a causa del tempo di inizializzazione). Tenere attive istanze "di scorta" durante la giornata disponibili per le build, di contro, aumenta i costi.

Il nostro obiettivo è trovare una soluzione che possa ridurre il tempo di esecuzione, facilitare la manutenzione e ottimizzare i costi.

I container, ad esempio, hanno un tempo di avvio breve e aiutano a contenere i costi: in questo caso d'uso il billing sarebbe proporzionale solamente al tempo di esecuzione effettivamente utilizzato.

Per raggiungere il nostro obiettivo faremo in modo di eseguire su task ECS Fargate le nostre build. In aggiunta vedremo anche come utilizzare i servizi ECS per implementarle e gestirne l'autoscaling.

Prima di passare all'implementazione pratica una piccola premessa: GitLab, per eseguire gli script definiti nelle pipeline, utilizza un agent software chiamato GitLab Runner, possiamo configurare una istanza del runner dedicata alla gestione dello scaling che si occupi di aggiungere e rimuovere risorse computazionali in base all'andamento delle richieste.

Nei nostri esempi assumeremo che siano utilizzati tre ambienti: sviluppo (dev), test (staging) e produzione (prod), utilizzeremo ruoli IAM differenti per i nostri runner per limitare i permessi assegnati ed utilizzare il principio di least privilege.

I runner GitLab possono essere associati a tag che rendono possibile la scelta dell'ambiente in cui verranno eseguite le build.

In questo esempio è definita una pipeline che compila e fa il deploy in 3 ambienti differenti:

```
stages:
 - build dev
 - deploy dev
 - build staging
 - deploy staging
 - build production
 - deploy production

build-dev:
 stage: build dev
 tags:
   - dev
 script:
   - ./scripts/build.sh
 artifacts:
   paths:
     - ./artifacts
```

```
    expire_in: 7d

deploy-dev:
 stage: deploy dev
 tags:
    - dev
 script:
    - ./scripts/deploy.sh

build-staging:
 stage: build staging
 tags:
    - staging
 script:
    - ./scripts/build.sh
 artifacts:
    paths:
      - ./artifacts
    expire_in: 7d

 deploy-staging:
 stage: deploy staging
 tags:
    - staging
 script:
    - ./scripts/deploy.sh

build-production:
 stage: build production
 tags:
    - production
 script:
    - ./scripts/build.sh
 artifacts:
    paths:
      - ./artifacts
```

```
    expire_in: 7d

 deploy-production:
 stage: deploy production
 tags:
    - production
 script:
    - ./scripts/deploy.sh
```

Implementare un runner Fargate base

Assumendo che il nostro software sia scritto utilizzando NodeJS possiamo sviluppare un Dockerfile che permetta di fare la build di una immagine Docker contenente tutte le dipendenze (GitLab runner incluso).

Dockerfile

```
# Ubuntu based GitLab runner with nodeJS, npm, and aws CLI
# ------------------------------------------------------------
--
# Install https://github.com/krallin/tini - a very small 'init' proce
ss
# that helps process signals sent to the container properly.
# ------------------------------------------------------------
--
ARG TINI_VERSION=v0.19.0

COPY docker-entrypoint.sh /usr/local/bin/docker-entrypoint.sh

RUN ln -snf /usr/share/zoneinfo/Europe/Rome /etc/localtime && echo Eu
rope/Rome > /etc/timezone \
    && echo "Installing base packaes" \
    && apt update && apt install -y curl gnupg unzip jq software-prope
rties-common \
    && echo "Installing awscli" \
    && curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip"
-o "awscliv2.zip" \
```

```
   && unzip awscliv2.zip \
   && ./aws/install \
   && rm -f awscliv2.zip \
   && apt update \
   && echo "Installing packages" \
   && apt install -y unzip openssh-server ca-certificates git git-lfs
nodejs npm \
   && echo "Installing tini and ssh" \
   && curl -Lo /usr/local/bin/tini https://github.com/krallin/tini/re
leases/download/${TINI_VERSION}/tini-amd64 \
   && chmod +x /usr/local/bin/tini \
   && mkdir -p /run/sshd \
   && curl -L https://packages.gitlab.com/install/repositories/runne
r/gitlab-runner/script.deb.sh | bash \
       && apt install -y gitlab-runner \
       && rm -rf /var/lib/apt/lists/* \
       && rm -f /home/gitlab-runner/.bash_logout \
   && git lfs install --skip-repo \
   && chmod +x /usr/local/bin/docker-entrypoint.sh \
   && echo "Done"

EXPOSE 22

ENTRYPOINT ["tini", "--", "/usr/local/bin/docker-entrypoint.sh"]
```

docker-entrypoint.sh

```
#!/bin/sh

# Create a folder to store the user's SSH keys if it does not exist.
USER_SSH_KEYS_FOLDER=~/.ssh
[ ! -d ${USER_SSH_KEYS_FOLDER} ] && mkdir -p ${USER_SSH_KEYS_FOLDER}

# Copy contents from the `SSH_PUBLIC_KEY` environment variable
# to the `$USER_SSH_KEYS_FOLDER/authorized_keys` file.
# The environment variable must be set when the container starts.
```

```
echo "${SSH_PUBLIC_KEY}" > ${USER_SSH_KEYS_FOLDER}/authorized_keys

# Clear the `SSH_PUBLIC_KEY` environment variable.
unset SSH_PUBLIC_KEY

# Start the SSH daemon
/usr/sbin/sshd -D
```

In questo caso non c'è nessuna dipendenza dall'ambiente in cui l'immagine dovrà essere eseguita.

Implementare un runner per l'autoscaling (Runner Manager)

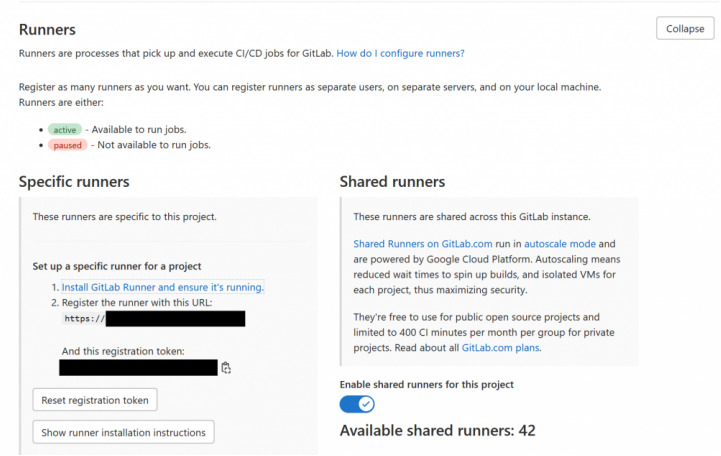
Questo tipo di runner deve essere specializzato per poter gestire l'ambiente in cui dovrà essere eseguito: adotteremo il Fargate custom executor sviluppato da GitLab per utilizzare cluster ECS Fargate differenti per i nostri ambienti.

La registrazione del nostro runner con il server GitLab sarà automatizzata ed eseguita durante la fase di build, usando variabili per specificare il token e gli altri parametri di configurazione.

Il custom executor Fargate necessita di un file di configurazione ("config.toml") per poter specificare il cluster, le subnet, i security groups e la task definition da utilizzare per l'esecuzione della pipeline. Faremo in modo di automatizzare anche questa configurazione nella fase di build del container.

Per prima cosa dobbiamo ottenere un token per la registrazione del runner con il server GitLab.

Alla sezione "CI/CD" nelle impostazioni del progetto si può espandere la sezione "Runners".



Gli unici dati necessari sono il registration Token e l'indirizzo del server GitLab.

L'indirizzo del server GitLab può essere anche inserito direttamente nel Dockerfile, tratteremo invece il token di registrazione come un segreto.

Queste righe, contenute nel Dockerfile, si occupano della modifica del file di configurazione

```
RUNNER_TASK_TAGS=$(echo ${RUNNER_TAGS} | tr "," "-")
sed -i s/RUNNER_TAGS/${RUNNER_TASK_TAGS}/g /tmp/ecs.toml
sed -i s/SUBNET/${SUBNET}/g /tmp/ecs.toml
sed -i s/SECURITY_GROUP_ID/${SECURITY_GROUP_ID}/g /tmp/ecs.toml
```

DockerFile

```
FROM ubuntu:20.04

ARG GITLAB_TOKEN
ARG RUNNER_TAGS

ARG GITLAB_URL="https://gitlab.myawesomecompany.com"
ARG SUBNET
ARG SECURITY_GROUP_ID



COPY config.toml /tmp/
COPY ecs.toml /tmp/
COPY entrypoint /
COPY fargate-driver /tmp
```

```
RUN apt update && apt install -y curl unzip \
       && curl -L https://packages.gitlab.com/install/repositories/runner/gitlab-runner/script.deb.sh | bash \
       && apt install -y gitlab-runner \
       && rm -rf /var/lib/apt/lists/* \
       && rm -f "/home/gitlab-runner/.bash_logout" \
       && chmod +x /entrypoint \
       && mkdir -p /opt/gitlab-runner/metadata /opt/gitlab-runner/builds /opt/gitlab-runner/cache \
       && curl -Lo /opt/gitlab-runner/fargate https://gitlab-runner-custom-fargate-downloads.s3.amazonaws.com/latest/fargate-linux-amd64 \
       && chmod +x /opt/gitlab-runner/fargate \
       && RUNNER_TASK_TAGS=$(echo ${RUNNER_TAGS} | tr "," "-") \
       && sed -i s/RUNNER_TAGS/${RUNNER_TASK_TAGS}/g /tmp/ecs.toml \
       && sed -i s/SUBNET/${SUBNET}/g /tmp/ecs.toml \
       && sed -i s/SECURITY_GROUP_ID/${SECURITY_GROUP_ID}/g /tmp/ecs.toml \
       && cp /tmp/ecs.toml /etc/gitlab-runner/ \
       && echo "Token: ${GITLAB_TOKEN} url: ${GITLAB_URL} Tags: ${RUNNER_TAGS}" \
       && gitlab-runner register \
              --non-interactive \
              --url ${GITLAB_URL} \
              --registration-token ${GITLAB_TOKEN} \
              --template-config /tmp/config.toml \
              --description "GitLab runner for ${RUNNER_TAGS}" \
              --executor "custom" \
              --tag-list ${RUNNER_TAGS}

ENTRYPOINT ["/entrypoint"]
CMD ["run", "--user=gitlab-runner", "--working-directory=/home/gitlab-runner"]
```

A questo punto possiamo fare la build del container:

```
docker build . -t gitlab-runner --build-arg GITLAB_TOKEN="generatedgi
tlabtoken" --build-arg RUNNER_TAGS="dev" --build-arg SUBNET="subnet-1
2345" --build-arg SECURITY_GROUP_ID="sg-12345"
```

Queste linee nel Dockerfile si occupano della personalizzazione del file di configurazione:

```
RUNNER_TASK_TAGS=$(echo ${RUNNER_TAGS} | tr "," "-")
sed -i s/RUNNER_TAGS/${RUNNER_TASK_TAGS}/g /tmp/ecs.toml
sed -i s/SUBNET/${SUBNET}/g /tmp/ecs.toml
sed -i s/SECURITY_GROUP_ID/${SECURITY_GROUP_ID}/g /tmp/ecs.toml
```

DockerFile

```
FROM ubuntu:20.04

ARG GITLAB_TOKEN
ARG RUNNER_TAGS
ARG GITLAB_URL="https://gitlab.myawesomecompany.com"
ARG SUBNET
ARG SECURITY_GROUP_ID


COPY config.toml /tmp/
COPY ecs.toml /tmp/
COPY entrypoint /
COPY fargate-driver /tmp


RUN apt update && apt install -y curl unzip \
       && curl -L https://packages.gitlab.com/install/repositories/ru

nner/gitlab-runner/script.deb.sh | bash \
       && apt install -y gitlab-runner \
```

```
        && rm -rf /var/lib/apt/lists/* \
        && rm -f "/home/gitlab-runner/.bash_logout" \
        && chmod +x /entrypoint \
        && mkdir -p /opt/gitlab-runner/metadata /opt/gitlab-runner/builds /opt/gitlab-runner/cache \
        && curl -Lo /opt/gitlab-runner/fargate https://gitlab-runner-custom-fargate-downloads.s3.amazonaws.com/latest/fargate-linux-amd64 \
        && chmod +x /opt/gitlab-runner/fargate \
        && RUNNER_TASK_TAGS=$(echo ${RUNNER_TAGS} | tr "," "-") \
        && sed -i s/RUNNER_TAGS/${RUNNER_TASK_TAGS}/g /tmp/ecs.toml \
        && sed -i s/SUBNET/${SUBNET}/g /tmp/ecs.toml \
        && sed -i s/SECURITY_GROUP_ID/${SECURITY_GROUP_ID}/g /tmp/ecs.toml \
        && cp /tmp/ecs.toml /etc/gitlab-runner/ \
        && echo "Token: ${GITLAB_TOKEN} url: ${GITLAB_URL} Tags: ${RUNNER_TAGS}" \
        && gitlab-runner register \
                --non-interactive \
                --url ${GITLAB_URL} \
                --registration-token ${GITLAB_TOKEN} \
                --template-config /tmp/config.toml \
                --description "GitLab runner for ${RUNNER_TAGS}" \
                --executor "custom" \
                --tag-list ${RUNNER_TAGS}

ENTRYPOINT ["/entrypoint"]
CMD ["run", "--user=gitlab-runner", "--working-directory=/home/gitlab-runner"]
```

Per la build del runner manager è sufficiente eseguire:

```
docker build . -t gitlab-runner-manager --build-arg GITLAB_TOKEN="generatedgitlabtoken" --build-arg RUNNER_TAGS="dev" --build-arg SUBNET="subnet-12345" --build-arg SECURITY_GROUP_ID="sg-12345"
```

Al termine delle operazioni, il runner si registrerà:

config.toml

```
concurrent = 1
check_interval = 0

[session_server]
 session_timeout = 1800

[[runners]]
 name = "ec2-ecs"
 executor = "custom"
 builds_dir = "/opt/gitlab-runner/builds"
 cache_dir = "/opt/gitlab-runner/cache"
 [runners.cache]
   [runners.cache.s3]
   [runners.cache.gcs]
 [runners.custom]
   config_exec = "/opt/gitlab-runner/fargate"
   config_args = ["--config", "/etc/gitlab-runner/ecs.toml", "custom"
, "config"]
   prepare_exec = "/opt/gitlab-runner/fargate"
   prepare_args = ["--config", "/etc/gitlab-runner/ecs.toml", "custo
m", "prepare"]
   run_exec = "/opt/gitlab-runner/fargate"
   run_args = ["--config", "/etc/gitlab-runner/ecs.toml", "custom",
"run"]
   cleanup_exec = "/opt/gitlab-runner/fargate"
   cleanup_args = ["--config", "/etc/gitlab-runner/ecs.toml", "custo
m", "cleanup"]
```

ecs.toml

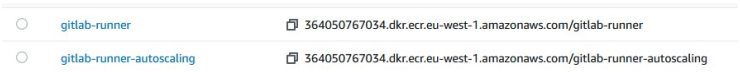
```
LogLevel = "info"
LogFormat = "text"

[Fargate]
 Cluster = "acme-gitlab-RUNNER-TAGS-cluster"
 Region = "eu-west-1"
 Subnet = "SUBNET"
 SecurityGroup = "SECURITY_GROUP_ID"
 TaskDefinition = "gitlab-runner-RUNNER_TAGS-task"
 EnablePublicIP = false

[TaskMetadata]
 Directory = "/opt/gitlab-runner/metadata"

[SSH]
 Username = "root"
 Port = 22
```

entrypoint

```
!/bin/bash

# gitlab-runner data directory
DATA_DIR="/etc/gitlab-runner"
CONFIG_FILE=${CONFIG_FILE:-$DATA_DIR/config.toml}
# custom certificate authority path
CA_CERTIFICATES_PATH=${CA_CERTIFICATES_PATH:-$DATA_DIR/certs/ca.crt}
LOCAL_CA_PATH="/usr/local/share/ca-certificates/ca.crt"

update_ca() {
  echo "Updating CA certificates..."
  cp "${CA_CERTIFICATES_PATH}" "${LOCAL_CA_PATH}"
  update-ca-certificates --fresh >/dev/null
}
```

```
if [ -f "${CA_CERTIFICATES_PATH}" ]; then
  # update the ca if the custom ca is different than the current
  cmp --silent "${CA_CERTIFICATES_PATH}" "${LOCAL_CA_PATH}" || update
_ca
fi

# launch gitlab-runner passing all arguments
exec gitlab-runner "$@"
```

Terminata la fase di build dobbiamo trasferire l'immagine nei repositoyi ECR (utilizzeremo gitlab-runner e gitlab-runner-autoscaling come nome), le istruzioni per la fase di push delle immagini docker sono disponibili nella documentazione del servizio ECR.

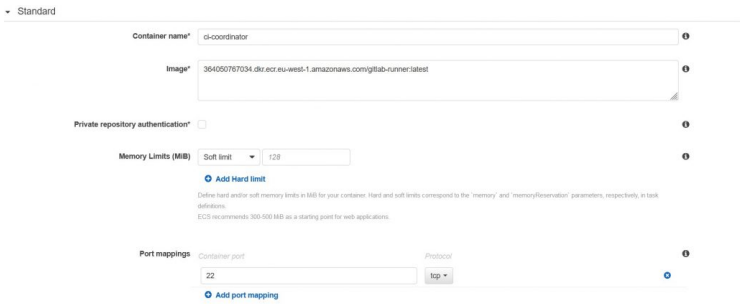


Una volta completata questa fase, possiamo procedere alla configurazione delle Task Definition

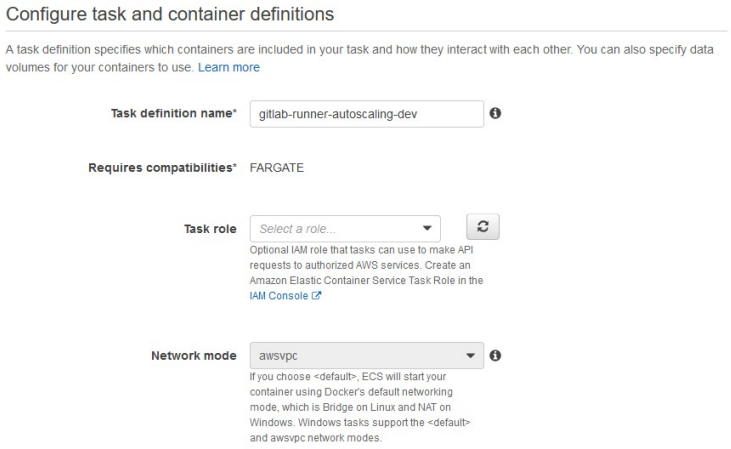
Ci occuperemo solamente la configurazione dell'ambiente di sviluppo, le operazioni per test e produzione sono simili.

In questo articolo è possibile trovare una guida più dettagliata per la creazione di repository ECR e task definition.

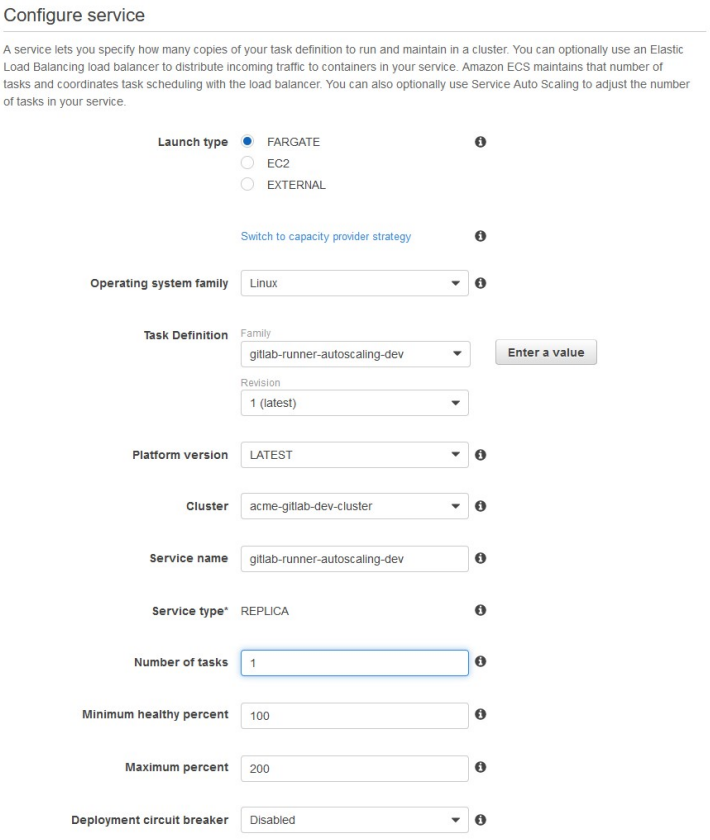
Configureremo le task definition per eseguire i runner nei nostri ambienti (gitlab-runner-dev-task, gitlab-runner-stage-task, gitlab-runner-prod-task).Attenzione: la task definition deve includere un container con nome "**ci-coordinator**". Occorre anche definire un port mapping per la porta 22 ed un security group che permetta le connessioni in entrata su quella porta: GitLab utilizza infatti una connessione ssh per l'esecuzione delle fasi della pipeline.



Una volta terminata la configurazione della task definition per il runner possiamo procedere con la parte dedicata all'autoscaling.



A questo punto basta aggiungere un ECS Service che si può occupare di gestire il runner.



Occorre anche definire un ruolo con una policy associata che permetta l'esecuzione di task nel cluster:

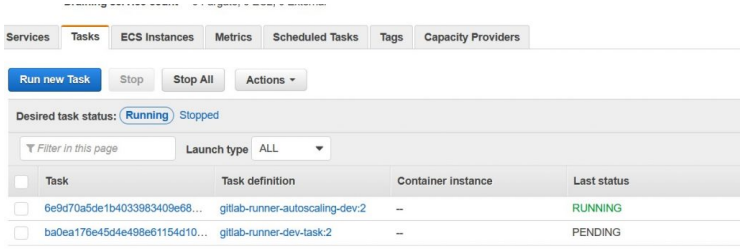
```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowRunTask",
            "Effect": "Allow",
            "Action": [
                "ecs:RunTask",
                "ecs:ListTasks",
                "ecs:StartTask",
                "ecs:StopTask",
                "ecs:ListContainerInstances",
                "ecs:DescribeTasks"
            ],
            "Resource": [
                "arn:aws:ecs:eu-west-1:account-id:task/acme-gitlab-de
v-cluster/*",
                "arn:aws:ecs:eu-west-1:account-id:cluster/acme-gitlab
-dev-cluster",
                "arn:aws:ecs:eu-west-1:account-id:task-definition/*:
*",
                "arn:aws:ecs:*:account-id:container-instance/*/*"
            ]
        },
        {
            "Sid": "AllowListTasks",
            "Effect": "Allow",
            "Action": [
                "ecs:ListTaskDefinitions",
                "ecs:DescribeTaskDefinition"
            ],
            "Resource": "*"
        }
    ]
}
```

Dopo un minuto il nostro servizio sarà disponibile:

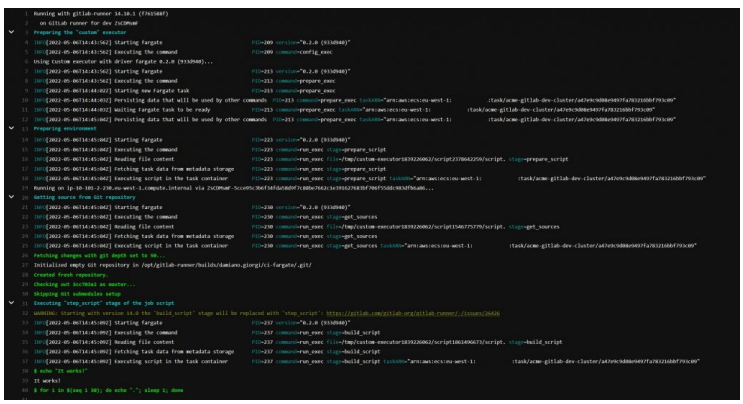
A questo punto possiamo definire una pipeline di test in .gitlab-ci.yml

```
test:
  tags:
    - dev
  script:
  - echo "It works!"
  - for i in $(seq 1 30); do echo "."; sleep 1; done
```

Il runner avvierà un nuovo task per l'esecuzione della pipeline



Il task passerà in stato running e l'esecuzione inizierà:

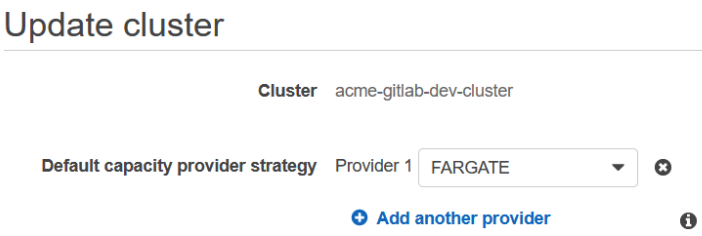


E l'esecuzione terminerà con successo!

Terminata la pipeline il container non sarà più in esecuzione.

Troubleshooting

Nel caso si verifichi un errore di timeout verificare i security group ed il routing dalle subnet ai repository ECR (se sono in uso subnet private). Nel caso di subnet isolate occorre invece aggiungere un VPC endpoint per accedere al servizio ECR.
Nel caso l'errore sia: "*starting new Fargate task: running new task on Fargate: error starting AWS Fargate Task: InvalidParameterException: No Container Instances were found in your cluster.*" occorre verificare che sia definito un capacity provider di default per il cluster ECS (fare click su "Update Cluster" e selezionare un capacity provider):



Oggi abbiamo visto come adottare un approccio serverless per l'esecuzione delle pipeline GitLab, si tratta solo dell'inizio: ci sono molte altre strade da esplorare: container spot, build e deploy cross-account, architetture differenti (come ad esempio ARM e Windows).

Avete già provato ad ottimizzare le vostre build e pipeline? Fatecelo sapere nei commenti!

Resources:

- GitHub Repository

Damiano Giorgi

Ex sistemista on-prem, pigro e incline all'automazione di task noiosi. Alla ricerca costante di novità tecnologiche e quindi passato al cloud per trovare nuovi stimoli.L'unico hardware a cui mi dedico ora è quello del mio basso; se non mi trovate in ufficio o in sala prove provate al pub o in qualche aeroporto!
