A serverless approach for GitLab integration on AWS

27 May 2022 - 9 min. read

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

Cost optimization and operational efficiency are key value drivers for a successful Cloud adoption path; using managed serverless services significantly lowers maintenance costs while speeding up operations.

In this article, you'll find how to better integrate GitLab pipelines on AWS using ECS Fargate in a multi-environment scenario.

GitLab offers a lot of flexibility for computational resources: pipelines can run on Kubernetes clusters, Docker, on-premise, or custom platforms using GitLab custom executor drivers.

The tried and tested solution to run pipelines on the AWS Cloud uses EC2 instances as computational resources.

This approach leads to some inefficiency: starting instances on-demand will make pipeline executions slower and developers impatient (because of the initialization time). Keeping a spare runner available for builds, on the other hand, will increase costs.

We want to find a solution that can reduce execution time, ease maintenance and optimize costs.

Containers have a faster initialization time and help decrease costs: billing will be based only on used build time. Our goal is to use them for our pipeline executions, they will run on ECS clusters. Additionally, we will see how to use ECS Services for autoscaling.

Before describing our implementation, we need to know a few things: GitLab Runners are software agents that can execute pipeline scripts. We can configure a runner instance to manage the pipeline's computational resources autoscaling by adding or removing capacity as demand for build capacity changes.

In our scenario, we’ll also assume that we have three different environments: development, staging, and production: we'll define different IAM roles for our runners, so they will use the least privilege available to build and deploy our software.

GitLab Runners have associated tags that help choose the environment that will run the execution step when defined in a pipeline.

In this example, you can see a pipeline that builds and deploys in different environments:

```
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
```

Making a base Fargate runner

Let's assume that our codebase uses NodeJS: we can build a custom generic Docker image with all the dependencies (including GitLab runner).

Dockerfile

```
FROM ubuntu:20.04

# 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/releases/download/${TINI_VERSION}/tini-amd64 \
   && chmod +x /usr/local/bin/tini \
   && mkdir -p /run/sshd \
   && 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 \
   && 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
```

As you can see, there's no environment-dependent configuration.

Building a Runner for autoscaling (formerly Runner Manager)

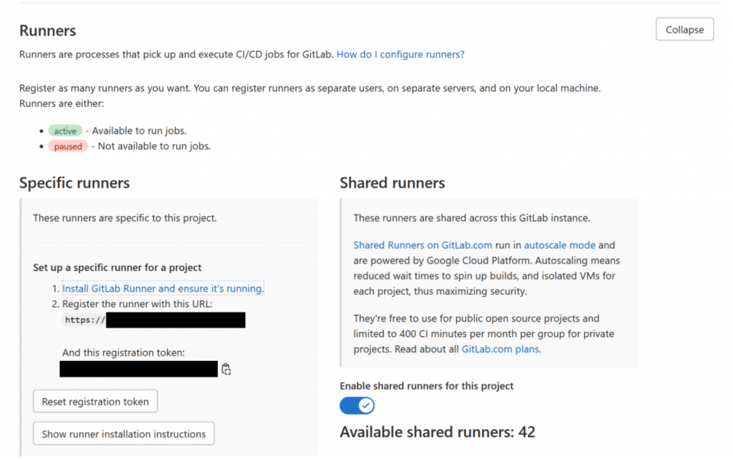
This runner instance needs to be specialized to handle the environment configuration; we'll use the Fargate Custom Executor provided by GitLab to interact and use different ECS Fargate Clusters for different environments.

We'll automatically handle our runner registration with the GitLab server during the Docker build phase by specifying its token using variables.

Our Fargate custom executor will need a configuration file ("config.toml") to specify a cluster, subnets, security groups, and task definition for our pipeline execution. We'll also handle this customization at build time.

First, we need to get a registration token from our GitLab server:

Go to your project CI/CD settings and expand the "Runners" section.



Runners

Runners are processes that pick up and execute CI/CD jobs for GitLab. How do I configure runners?

Register as many runners as you want. You can register runners as separate users, on separate servers, and on your local machine. Runners are either:

- active - Available to run jobs.
- paused - Not available to run jobs.

Specific runners

These runners are specific to this project.

Set up a specific runner for a project

1. Install GitLab Runner and ensure it's running.
2. Register the runner with this URL:
   `https://`[redacted]

   And this registration token:
   [redacted]

Shared runners

These runners are shared across this GitLab instance.

Shared Runners on GitLab.com run in autoscale mode and are powered by Google Cloud Platform. Autoscaling means reduced wait times to spin up builds, and isolated VMs for each project, thus maximizing security.

They're free to use for public open source projects and limited to 400 CI minutes per month per group for private projects. Read about all GitLab.com plans.

Enable shared runners for this project: ✓

Available shared runners: 42

Copy the registration token and GitLab server address

You can embed the GitLab server address in your DockerFile; we'll treat the registration token as a secret.

As you'll see below, these lines will customize our configuration file:

```
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
```

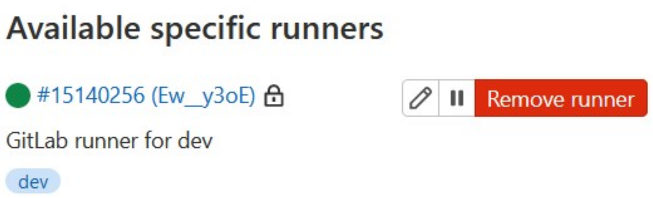
DockerFile

```
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"]
```

We can build our runner manager using:

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

When Docker build finishes, you can see runner registration.


Available specific runners

● #15140256 (Ew_y3oE)

GitLab runner for dev

dev

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", "custom", "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", "custom", "cleanup"]
```

ecs.toml

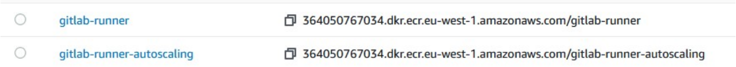
```
LogLevel = "info"
LogFormat = "text"

[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 "$@"
```

We can now push our Docker images to ECR repositories (we'll use gitlab-runner and gitlab-runner-autoscaling as repository names); please refer to ECR documentation for push commands.



○	gitlab-runner	364050767034.dkr.ecr.eu-west-1.amazonaws.com/gitlab-runner
○	gitlab-runner-autoscaling	364050767034.dkr.ecr.eu-west-1.amazonaws.com/gitlab-runner-autoscaling

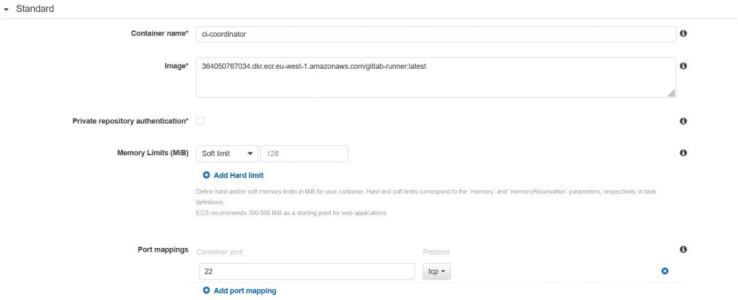
Once we finish pushing, we can proceed to define task definitions.

We'll describe our configuration for the development environment only; configuration steps will be the same for every environment.

You can find a complete guide on creating ECR repositories, task definitions, and services here:

We will configure task definitions for runners in our environments (gitlab-runner-dev-task, gitlab-runner-stage-task, gitlab-runner-prod-task).

Please note that the runner task definition has to define a container using "**ci-coordinator**" as the container name. You also need to define a port mapping for runner task definition for port 22 and a security group that accepts inbound connections on port 22: GitLab will use an ssh connection to execute the pipeline.



Standard

Container name* ci-coordinator

Image* 364050767034.dkr.ecr.eu-west-1.amazonaws.com/gitlab-runner:latest

Private repository authentication* □

Memory Limits (MiB) Soft limit 128

Add Hard limit

Define hard and/or soft memory limits in MiB for your container. Hard and soft limits correspond to the `memory` and `memoryReservation` parameters, respectively, in task definitions.

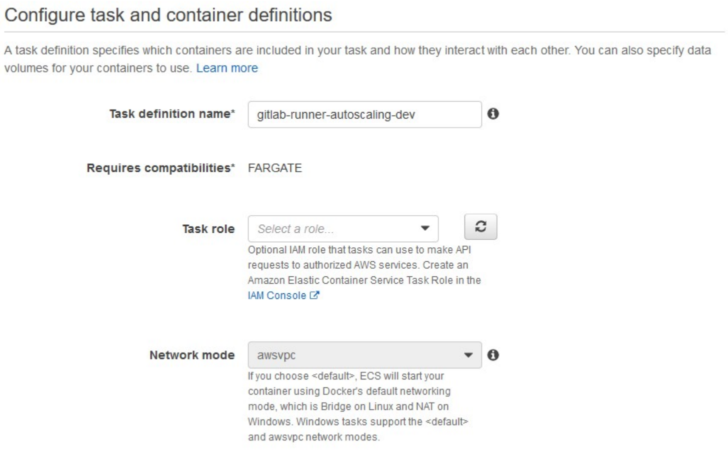
ECS recommends 300-500 MiB as a starting point for web applications.

Port mappings

Container port	Protocol
22	tcp

Add port mapping

Once we have defined our runner task definition, we can proceed to configure the task definition for autoscaling.



Configure task and container definitions

A task definition specifies which containers are included in your task and how they interact with each other. You can also specify data volumes for your containers to use. Learn more

Task definition name* gitlab-runner-autoscaling-dev

Requires compatibilities* FARGATE

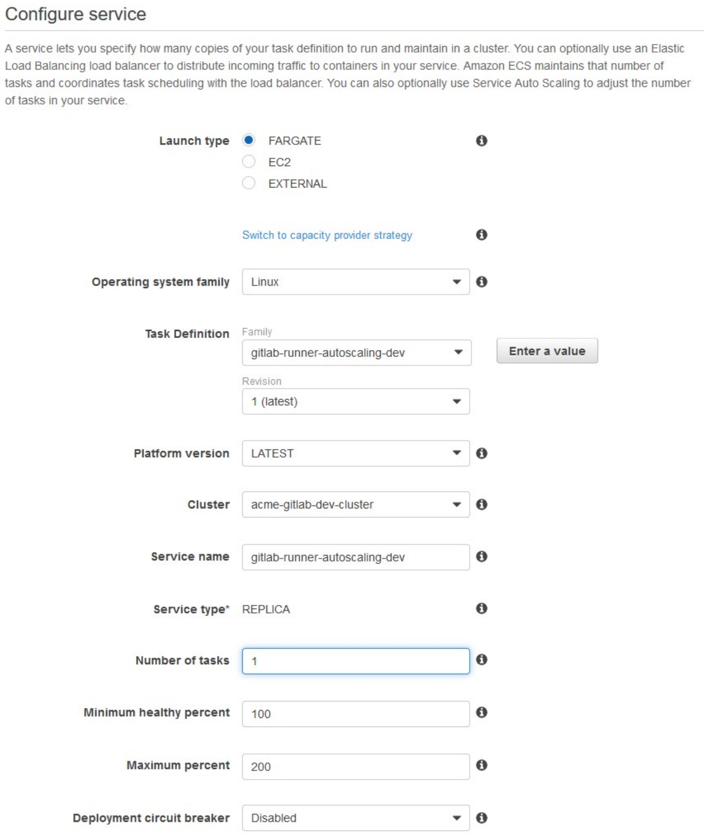
Task role Select a role...

Optional IAM role that tasks can use to make API requests to authorized AWS services. Create an Amazon Elastic Container Service Task Role in the IAM Console

Network mode awsvpc

If you choose <default>, ECS will start your container using Docker's default networking mode, which is Bridge on Linux and NAT on Windows. Windows tasks support the <default> and awsvpc network modes.

We then need to configure an ECS Service that keeps our runner alive.



Configure service

A service lets you specify how many copies of your task definition to run and maintain in a cluster. You can optionally use an Elastic Load Balancing load balancer to distribute incoming traffic to containers in your service. Amazon ECS maintains that number of tasks and coordinates task scheduling with the load balancer. You can also optionally use Service Auto Scaling to adjust the number of tasks in your service.

Launch type ● FARGATE ○ EC2 ○ EXTERNAL

Switch to capacity provider strategy

Operating system family Linux

Task Definition Family gitlab-runner-autoscaling-dev Enter a value

Revision 1 (latest)

Platform version LATEST

Cluster acme-gitlab-dev-cluster

Service name gitlab-runner-autoscaling-dev

Service type* REPLICA

Number of tasks 1

Minimum healthy percent 100

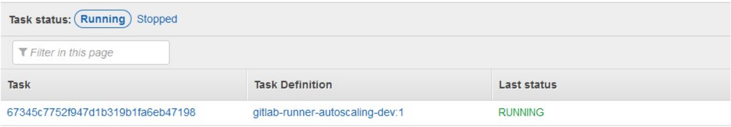
Maximum percent 200

Deployment circuit breaker Disabled

And then define a role with an associated policy to start and terminate tasks on our ECS cluster for the task role.

```
{
    "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-dev-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": "*"
        }
    ]
}
```

After a minute, our runner service will be ready:



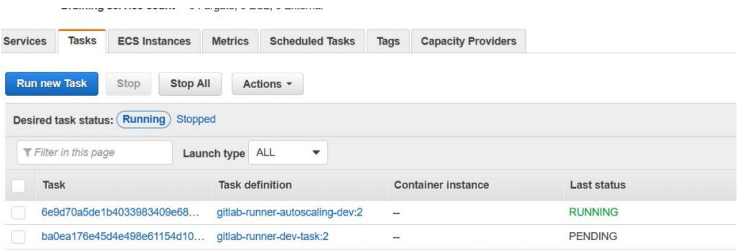
Task status: Running Stopped

Task	Task Definition	Last status
67345c7752f947d1b319b1fa6eb47198	gitlab-runner-autoscaling-dev:1	RUNNING

We can now define a test execution pipeline in *.gitlab-ci.yml*:

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

Our runner will run a new task when you execute the pipeline:

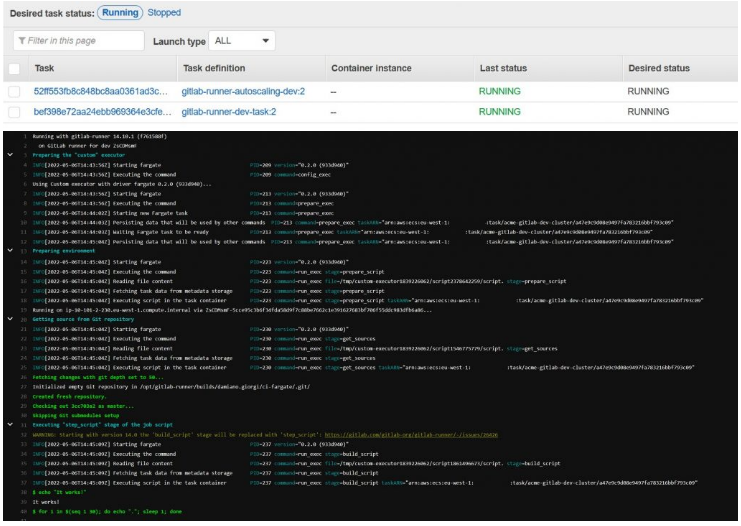


Desired task status: Running Stopped

Launch type ALL

	Task	Task definition	Container instance	Last status
□	6e9d70a5de1b4033983409e68...	gitlab-runner-autoscaling-dev:2	--	RUNNING
□	ba0ea176e45d4e498e61154d10...	gitlab-runner-dev-task:2	--	PENDING

The task will run, and pipeline execution will start:



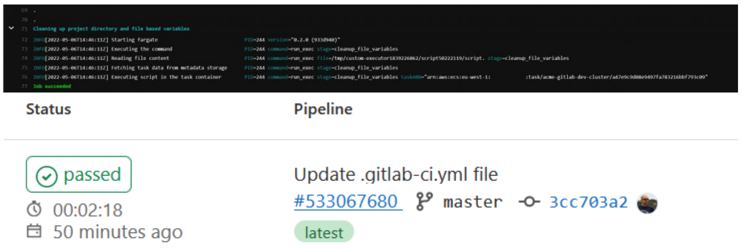
Desired task status: Running Stopped

Launch type ALL

	Task	Task definition	Container instance	Last status	Desired status
□	52ff553fb8c848bc8aa0361ad3c...	gitlab-runner-autoscaling-dev:2	--	RUNNING	RUNNING
□	bef398e72aa24ebb969364e3cfe...	gitlab-runner-dev-task:2	--	RUNNING	RUNNING

```
Running with gitlab-runner 14.10.1 (f761588f)
  on GitLab runner for dev ZsCDMsmF
Preparing the "custom" executor
INFO[2022-05-06T14:43:56Z] Starting fargate                              PID=209 version="0.2.0 (933d940)"
INFO[2022-05-06T14:43:56Z] Executing the command                         PID=209 command=config_exec
Using Custom executor with driver fargate 0.2.0 (933d940)...
INFO[2022-05-06T14:43:56Z] Starting fargate                              PID=213 version="0.2.0 (933d940)"
INFO[2022-05-06T14:43:56Z] Executing the command                         PID=213 command=prepare_exec
INFO[2022-05-06T14:44:02Z] Starting new Fargate task                     PID=213 command=prepare_exec
INFO[2022-05-06T14:44:03Z] Persisting data that will be used by other commands  PID=213 command=prepare_exec taskARN="arn:aws:ecs:eu-west-1:            :task/acme-gitlab-dev-cluster/a47e9c9d08e9497fa783216bbf793c09"
INFO[2022-05-06T14:44:03Z] Waiting Fargate task to be ready              PID=213 command=prepare_exec taskARN="arn:aws:ecs:eu-west-1:            :task/acme-gitlab-dev-cluster/a47e9c9d08e9497fa783216bbf793c09"
INFO[2022-05-06T14:45:04Z] Persisting data that will be used by other commands  PID=213 command=prepare_exec taskARN="arn:aws:ecs:eu-west-1:            :task/acme-gitlab-dev-cluster/a47e9c9d08e9497fa783216bbf793c09"
Preparing environment
INFO[2022-05-06T14:45:04Z] Starting fargate                              PID=223 version="0.2.0 (933d940)"
INFO[2022-05-06T14:45:04Z] Executing the command                         PID=223 command=run_exec stage=prepare_script
INFO[2022-05-06T14:45:04Z] Reading file content                          PID=223 command=run_exec file=/tmp/custom-executor1839226062/script2378642259/script. stage=prepare_script
INFO[2022-05-06T14:45:04Z] Fetching task data from metadata storage      PID=223 command=run_exec stage=prepare_script
INFO[2022-05-06T14:45:04Z] Executing script in the task container        PID=223 command=run_exec stage=prepare_script taskARN="arn:aws:ecs:eu-west-1:            :task/acme-gitlab-dev-cluster/a47e9c9d08e9497fa783216bbf793c09"
Running on ip-10-101-2-230.eu-west-1.compute.internal via ZsCDMsmF-5cce95c3b6f34fda58d9f7c88be7662c1e391627683bf706f55ddc983dfb6a86...
Getting source from Git repository
INFO[2022-05-06T14:45:04Z] Starting fargate                              PID=230 version="0.2.0 (933d940)"
INFO[2022-05-06T14:45:04Z] Executing the command                         PID=230 command=run_exec stage=get_sources
INFO[2022-05-06T14:45:04Z] Reading file content                          PID=230 command=run_exec file=/tmp/custom-executor1839226062/script1546775779/script. stage=get_sources
INFO[2022-05-06T14:45:04Z] Fetching task data from metadata storage      PID=230 command=run_exec stage=get_sources
INFO[2022-05-06T14:45:04Z] Executing script in the task container        PID=230 command=run_exec stage=get_sources taskARN="arn:aws:ecs:eu-west-1:            :task/acme-gitlab-dev-cluster/a47e9c9d08e9497fa783216bbf793c09"
Fetching changes with git depth set to 50...
Initialized empty Git repository in /opt/gitlab-runner/builds/damiano.giorgi/ci-fargate/.git/
Created fresh repository.
Checking out 3cc703a2 as master...
Skipping Git submodules setup
Executing "step_script" stage of the job script
WARNING: Starting with version 14.0 the 'build_script' stage will be replaced with 'step_script': https://gitlab.com/gitlab-org/gitlab-runner/-/issues/26426
INFO[2022-05-06T14:45:09Z] Starting fargate                              PID=237 version="0.2.0 (933d940)"
INFO[2022-05-06T14:45:09Z] Executing the command                         PID=237 command=run_exec stage=build_script
INFO[2022-05-06T14:45:09Z] Reading file content                          PID=237 command=run_exec file=/tmp/custom-executor1839226062/script1861496673/script. stage=build_script
INFO[2022-05-06T14:45:09Z] Fetching task data from metadata storage      PID=237 command=run_exec stage=build_script
INFO[2022-05-06T14:45:09Z] Executing script in the task container        PID=237 command=run_exec stage=build_script taskARN="arn:aws:ecs:eu-west-1:            :task/acme-gitlab-dev-cluster/a47e9c9d08e9497fa783216bbf793c09"
$ echo "It works!"
It works!
$ for i in $(seq 1 30); do echo "."; sleep 1; done
```

And, as you can see, execution is successful!



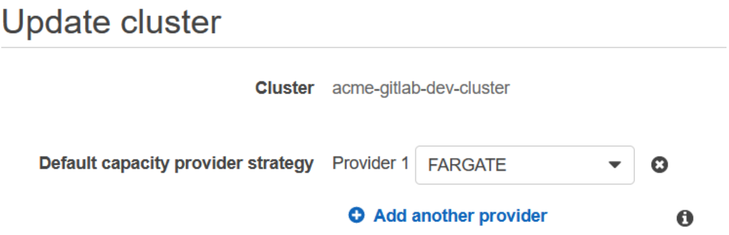
```
.
.
Cleaning up project directory and file based variables
INFO[2022-05-06T14:46:11Z] Starting fargate                              PID=244 version="0.2.0 (933d940)"
INFO[2022-05-06T14:46:11Z] Executing the command                         PID=244 command=run_exec stage=cleanup_file_variables
INFO[2022-05-06T14:46:11Z] Reading file content                          PID=244 command=run_exec file=/tmp/custom-executor1839226062/script50222119/script. stage=cleanup_file_variables
INFO[2022-05-06T14:46:11Z] Fetching task data from metadata storage      PID=244 command=run_exec stage=cleanup_file_variables
INFO[2022-05-06T14:46:11Z] Executing script in the task container        PID=244 command=run_exec stage=cleanup_file_variables taskARN="arn:aws:ecs:eu-west-1:            :task/acme-gitlab-dev-cluster/a47e9c9d08e9497fa783216bbf793c09"
Job succeeded
```

Status	Pipeline
passed 00:02:18 50 minutes ago	Update .gitlab-ci.yml file #533067680 master 3cc703a2 latest

Once the pipeline execution finishes, our container terminates, and our build container ends.

Troubleshooting

If you get a timeout error, verify your security groups definition and routing from the subnets to the ECR repositories (if you use private subnets). If you use isolated subnets, provide a VPC endpoint for ECR service
If you receive the error: "*starting new Fargate task: running new task on Fargate: error starting AWS Fargate Task: InvalidParameterException: No Container Instances were found in your cluster.*" verify that you have set a default capacity provider for your ECS Cluster (click on "Update Cluster" and select a capacity provider)


Update cluster

Cluster acme-gitlab-dev-cluster

Default capacity provider strategy Provider 1 FARGATE

Today we explored a serverless approach for running GitLab pipelines, scratching only the surface. There's a lot more to explore: Spot Container Instances, cross-account build and deploy, and different architectures (ARM and Windows, anyone?).

Do you already have a strategy for optimizing your builds? Have you already tinkered with custom executors for GitLab pipelines? Let us know in the comments!

Resources:

- GitHub Repository

Damiano Giorgi

Ex on-prem systems engineer, lazy and prone to automating boring tasks. In constant search of technological innovations and new exciting things to experience. And that's why I love Cloud Computing! At this moment, the only "hardware" I regularly dedicate myself to is that my bass; if you can't find me in the office or in the band room try at the pub or at some airport, then!
