Delivering Continuously — Automation Of Software Deployments and Rolling Deployments

Craig Godden-Payne
Feb 7 · 11 min read ★

What are blue green deployments?

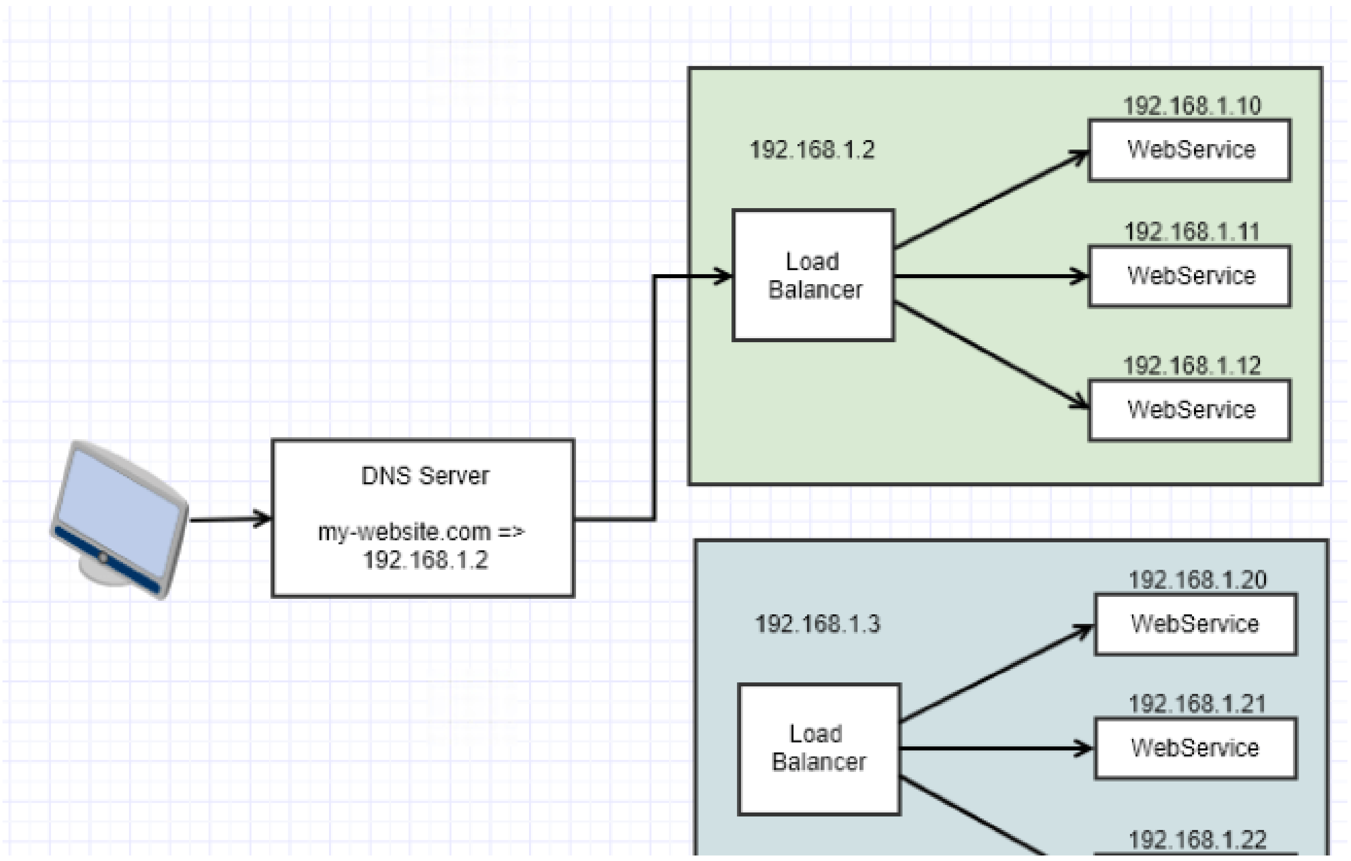
Traditionally, a blue-green deployment was described as a technique that reduces downtime and risk by running two identical production environments called Blue and Green. At any time, only one of the environments is live, with the live environment serving all production traffic. Because there is already another environment "ready to go" it's possible to switch environments if something happens to the green environment or a new version needs to be released, and can be really handy in a disaster recovery scenario.

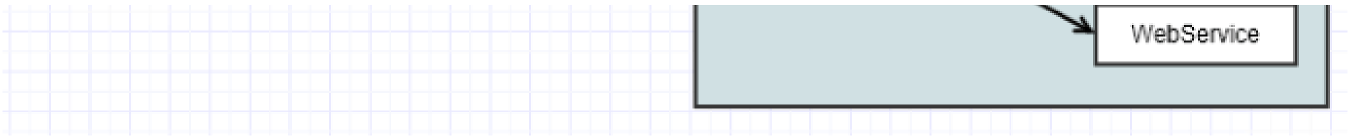
As times have changed, containerised technologies are now available, and it is possible to extend the description of blue green deployments to stacks or clusters of a containerised service, rather than just a traditional replica two environment blue green, as long as the tool you use supports some kind of connection draining, e.g. a load balancer.

How blue green works

To enable routing to applications using blue green, there is usually a central point of which the blue green switch can be made. This typically is done using a load balancer or changes to DNS.

DNS switching of load balancer



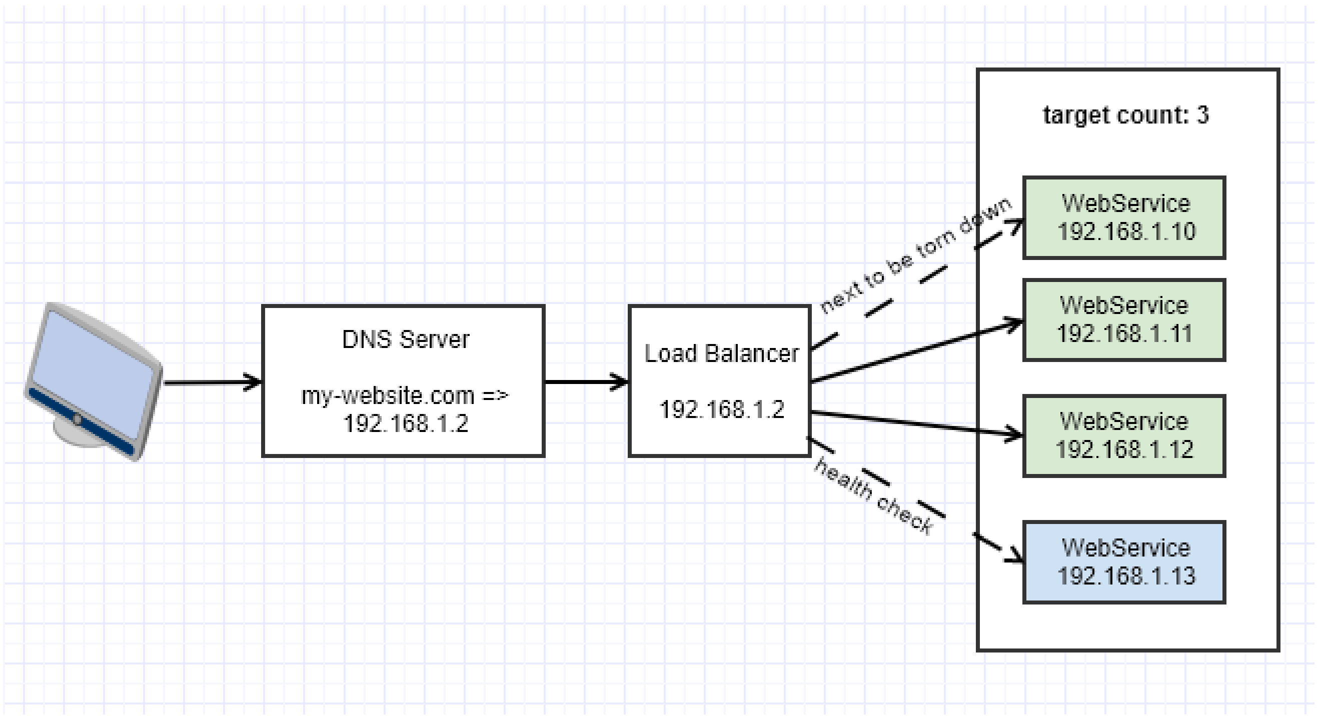


ProConChange can be made in your dns serverUnpredictable for when the switch will occur — TTL (although can be forced)Load balancer configuration doesn't need to be changedDNS does not check for outages, and will just return the IP of the machine you need to point toNo risk of breaking changes due to the whole environment being switched rather than a small partRequired more than one load balancer Can be more complex to setup, due to manually having to run a health check

Load balancer switching

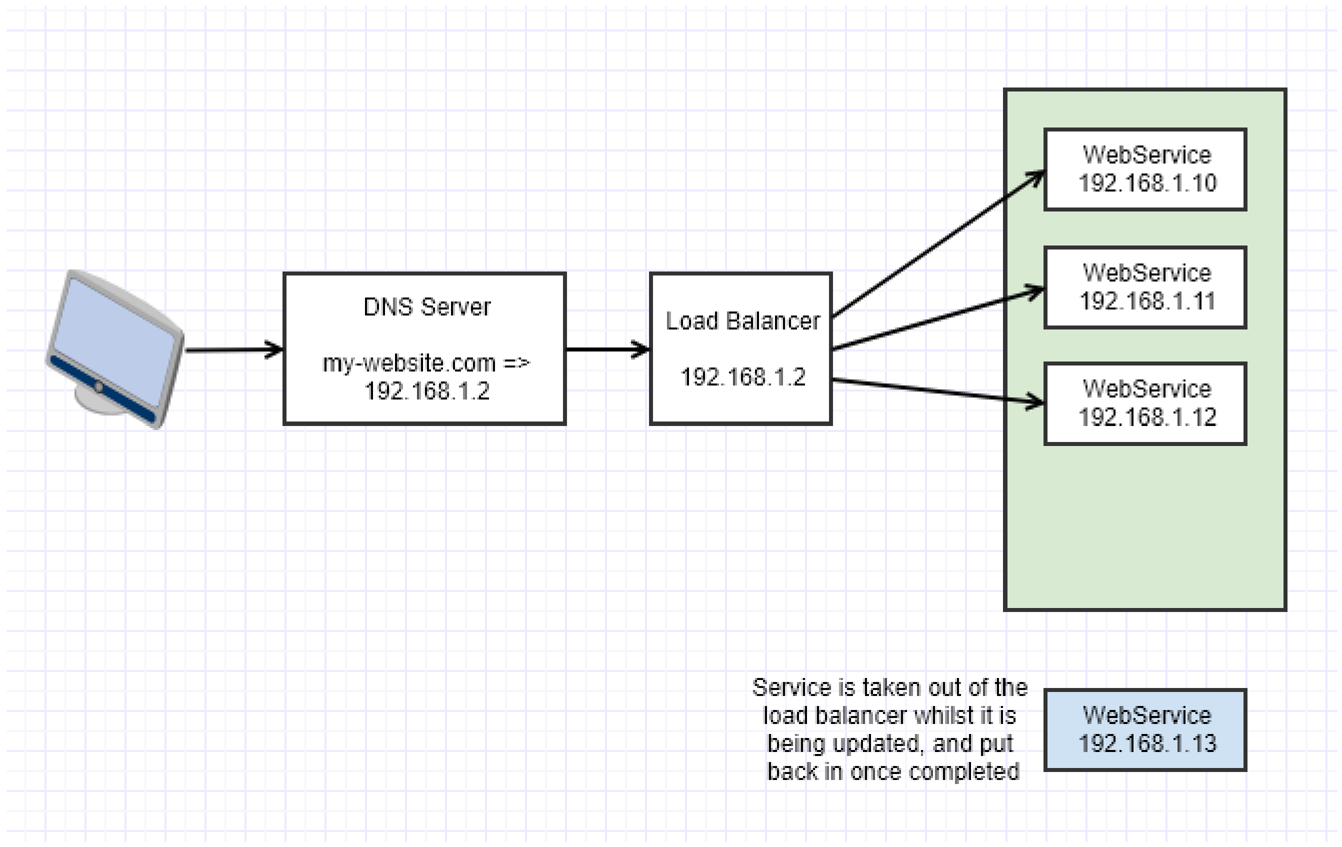
There are different approaches to this. When using containerised technologies, its common to only provision the infrastructure you need, so in this case, blue would not exist until its needed, and and soon as it is switched, green is torn down.

Approach using containers:



In a more traditional approach, for example with virtual machines, it is commonplace to have say 10 servers and remove 2 at a time from the load balancer, wait for the connections to drain, update the software and place back into the load balancer group. The traditional method is slower as you gradually deploy your new version, but it works well.

Approach using physical / virtualised machines


DNS Server
my-website.com => 192.168.1.2
Load Balancer
192.168.1.2
WebService 192.168.1.10
WebService 192.168.1.11
WebService 192.168.1.12
Service is taken out of the load balancer whilst it is being updated, and put back in once completed
WebService 192.168.1.13

ProConChange can be made in your load balancerCan be slow depending the time for your app to become healthy, but on the other hand will not switch until it is healthyNo changes to DNSConnection draining means that you could be serving both versions of a service, meaning a breaking change to a contract could cause errorsOnly one load balancer needed Load balancer will also perform health checks before making changes Predictable when the switch will occur Easy to automate

Why should we care so much about automated and blue green deployments?

- It reduces risk, it is very common for companies to have a person or rotation of people who will deploy, and it means they keep a lot of the knowledge to themselves. In blue green, since it is automated you can view the source of how the code / infrastructure is setup and deployed, and they deployment can be part of fully automated.
- It allows easy deployment and management of micro service architecture / breaking of monolithic architecture as we can quickly swap out smaller services during deployment / scale specific parts of our stack in response to demand, which

in turn can lead to lower spending on infrastructure, and less time managing deployment processes.

- These techniques are usually safer — it is generally safer to have immutable infrastructure where you can just spin up and replace existing infrastructure in almost all cases.
- Disaster recovery becomes very straightforward, as it usually would just require a deployment of a new set of infrastructure or parts of your infrastructure to priotise the most business critical functions after a catastrophic situation.
- Seemless deployments of newer versions, with no downtime, or warm up times. It's not so much a problem with dotnetcore, but IIS and .NetFramework warm up times were horrendous, and not something a user of your website would like to be involved in.
- Ability to horizontally scale your infrastructure and services as the business grows becomes straightforward
- No more deployments that break prod. It is almost impossible to push a broken build to production, as the blue green switch would never happen, so it's an extra safety net.

Photo by Thomas Jensen on Unsplash

Canary releases

It’s also worth mentioning at this point canary releases. It is possible to release new functionality to a proportion of users using blue green, where you allocate a proportion of your resources to the new functionality, although there are alternatives to this, such as using an experiments service, which will decide which bucket of users can see new functionality and managing this via code rather than resource. I won’t go into too much detail, but it’s worth noting that it’s a thing.

Automating blue green in AWS

There are three main automated ways to do blue green within AWS.

- Elastic Beanstalk
- ECS
- Kubenetes (EKS)

Photo by Markus Spiske on Unsplash

Elastic Beanstalk

Elastic beanstalk was AWS's first stab at blue green deployment technology. It can be used with containerised technology or without (EC2). If you use it with ECS, it is an abstraction layer above, and has some limitations.

Elastic beanstalk will interact with ECS/EC2 and ELB on your behalf, and it will make available health and metrics without much setup. It is very easy to use once initialised, and collects logs for all your apps in one place.

On the downside, it forces SSL termination at the load balancer level, and because of its simplistic nature, it isn't very customisable in terms of scheduling and scaling up/down. It is also known for being very slow in comparison to other technologies, especially with the EC2 offering, as it will start a server in EC2, wait for ready, then run a startup script before deploying the software to the server, before running. In EC2 this is all baked into the image. There are also licensing considerations (when running windows).

In my experience at a place I used to work, it also reported false positives regularly, and sometimes would kill instances regularly when they were seemingly not under much strain. This could have been down to bad implementation and early adoption though.

ECS — Elastic Container Service / Fargate

ECS is amazons version of container orchestration. It will run containers on EC2 instance(s) that are running on amazon linux with ECS agent installed (AMI image). You can use your own compatible image, but its recommended to run their optimised version. Because the containers run on your EC2 instance, they will be run within your VPC. As an alternative, you can run your containers on fargate, which is AWS's infrastructure as a service offering, meaning you run the cluster within your VPC, but you care less about the host that it's running on.

All the limitations of elastic beanstalk are removed in ECS, which gives access to much more sophisticated scheduling and scale up/ scale down logic, you can apply a rule based on any metric such as increase instance count by 1, if cpu > 80% or reduce instance count if network throughput is low.

ECS and fargate is defined through "tasks" and "services", where task describes your container, and service describes your instances within the cluster or lack of.

With ECS, problems can arise when you can't spin up a new container because of resource limitations, meaning that a deployment can get stuck as it needs to create a healthy instance before it can tear down a failing or obsolete container instance.

ECS is cheaper than Fargate, but Fargate is much simpler to setup and maintain.

ECS/Fargate has come really far in terms of startup time, and it's typical for having your container up and ready under 15 seconds.

I have only ever run nix based containers on ECS, although it is possible for windows. Amazon ECS is deeply integrated with IAM, enabling you to assign granular access permissions for each container and using IAM to restrict access to each service and delegate the resources that a container can access.

Kubernetes

Kubernetes is an open source container orchestration tool, sponsored by google and under the covers, EKS uses "kops" which is comparable to docker compose.

Kubernetes uses pods as the unit of containerisation apposed to a test in ECS, and both technologies have a concept of a service.

Deployments in EKS can be described easily within a deployment file, which means it is very close in simplicity to how Elastic Beanstalk works.

EKS does not integrate with IAM which can make securing more complex, and it costs more, as you have to pay for each EKS cluster, on top of any of the EC2 usage costs.

What about databases?

There are different techniques for this, but standard database practices apply where you want to only really make forward changes to a database, and usually when a database migration happens, you would tend to make the change before the blue green, to make sure that anything required by the latest version exists in the database.

Since blue green can be used for disaster recover scenarios, theres a topic in itself on this, so I won't go into detail.

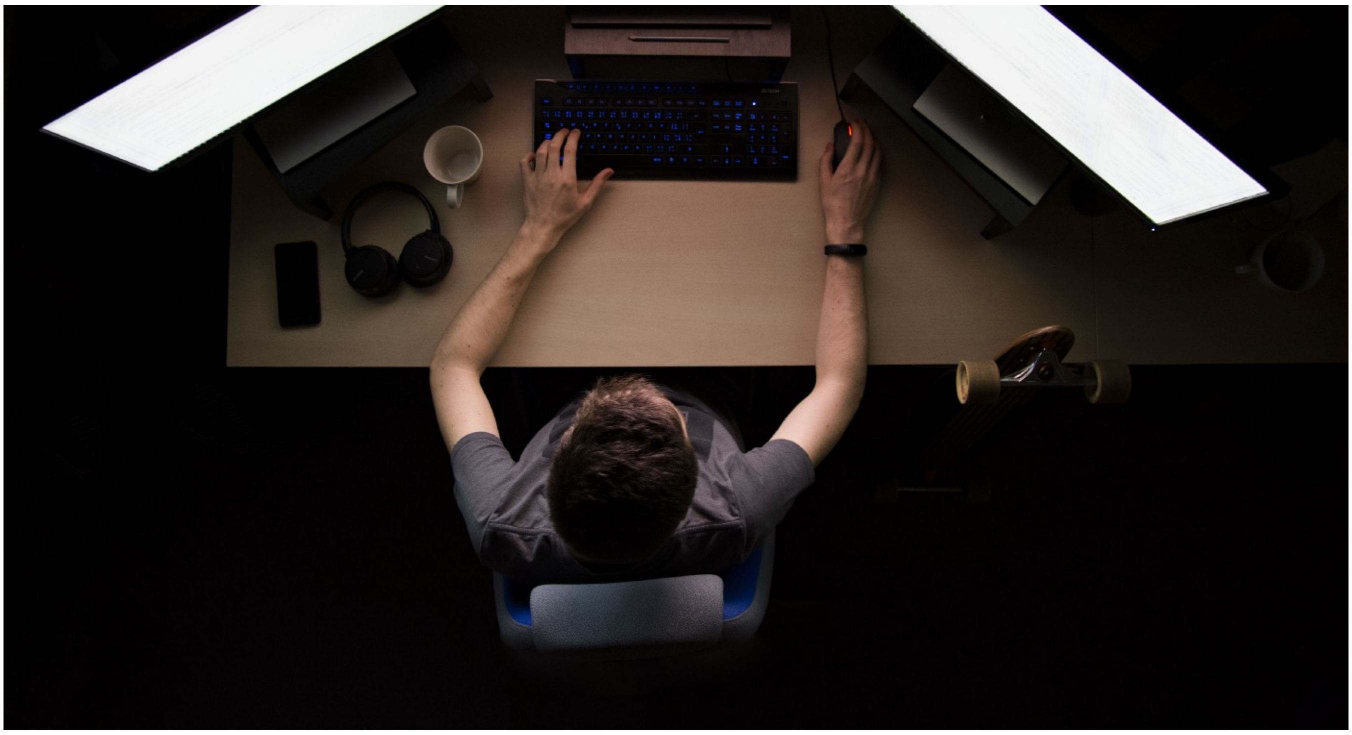


Photo by Max Duzij on Unsplash

Infrastructure as code

Infrastructure as code is the process of managing and provisioning resources (typically cloud resources) through definition files, rather than physical hardware configuration or interactive configuration tools or manual provisioning. IaC approaches are promoted for cloud computing, which is sometimes marketed as infrastructure as a service (IaaS). IaC supports IaaS, but should not be confused with it.

There are different tools for this, but the most reliable tend to be software which describes the target state of the infrastructure, and the software works out the delta to be applied.

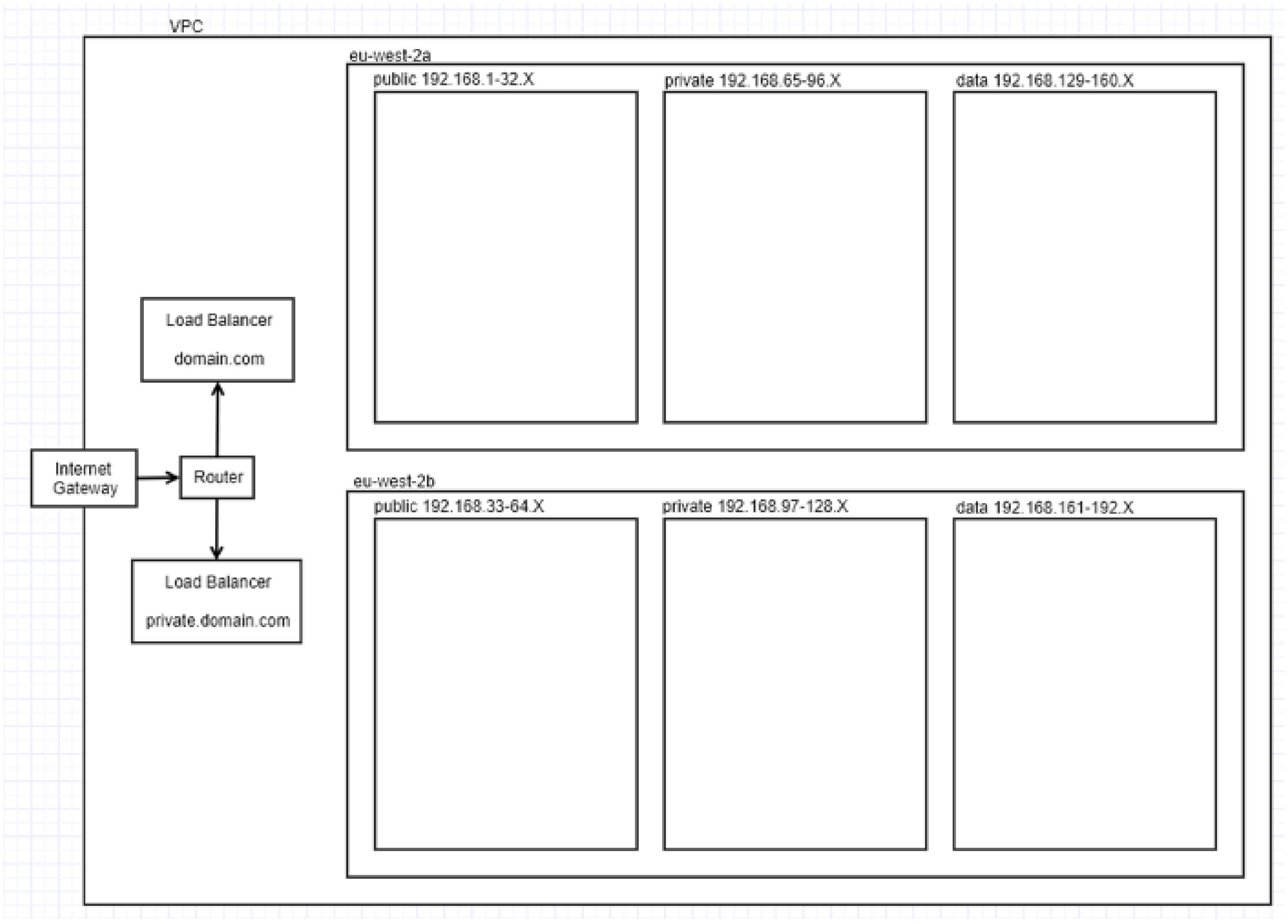
In the past, this is typically what operations teams were employed to do in a manual way, such as build me a new VM so that I can install my app etc. and as such plays in part to the Ops in DevOps.

Real life example

My previous experience / automated infrastructure as code deployments to ECS / fargate using terraform. At my previous job, I setup blue green deployments to utilise ECS (fargate was not available in the eu-west-2 London region at that time). But the stages we used are still applicable, albeit the networking configuration is a little bit more straightforward.

This is similar example to the VPC and subnet layout for the network I had at my last job. The VPC housed 3 subnets within 2 availability zones, public private and data. It was secured with NCL rules to allow public access to anything in public, private was only available from the public subnet or via the VPN, and data was only available from the data subnet.

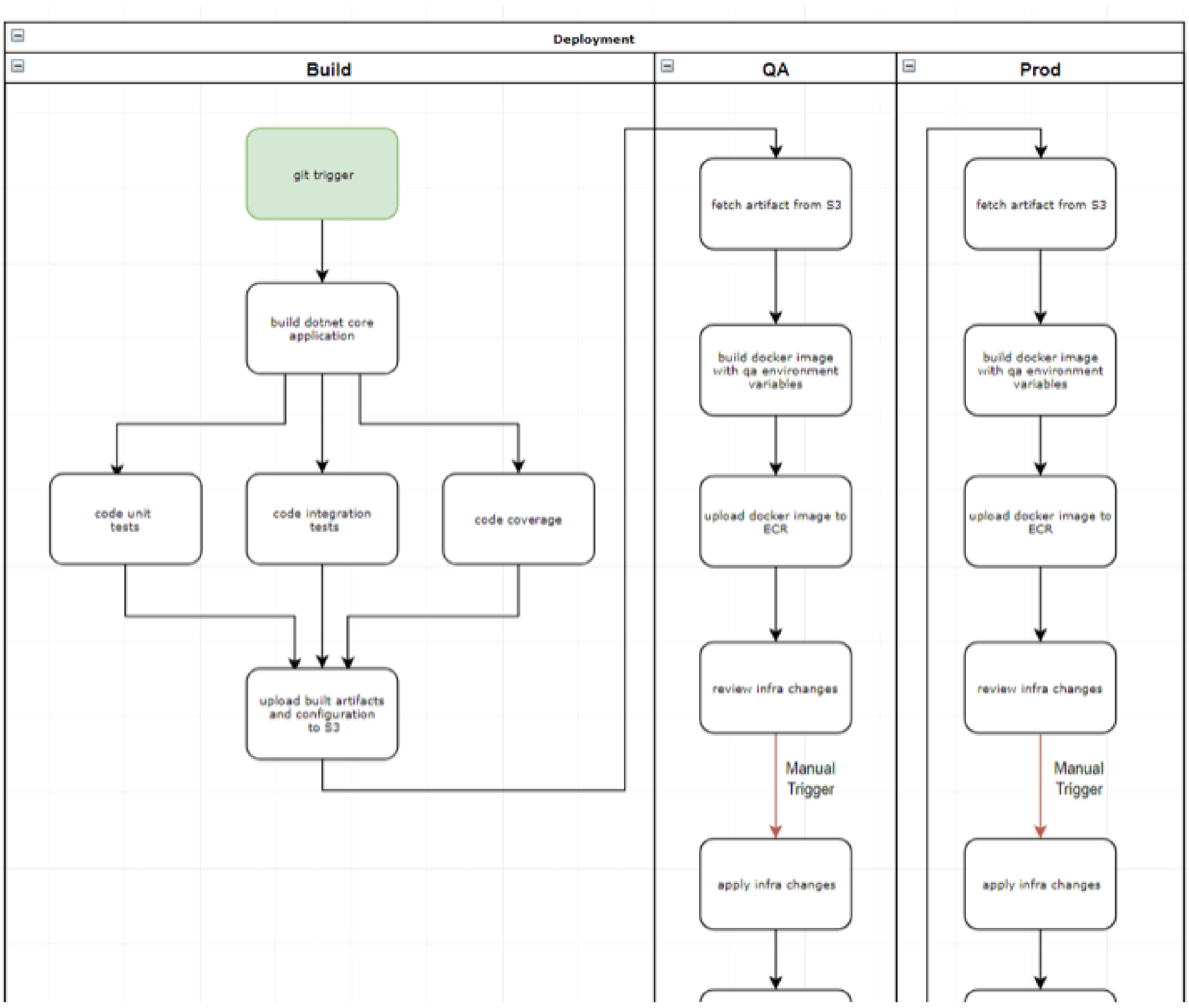
Two load balancers were created, public and private, and route53 DNS was set that the name servers for public facing domain.com would we accessible from the world, but DNS for internal.domain.com would only be available from within the VPC. This meant that each of the containers would require url catching to be setup within the load balancer, for a particular path or set of rules.

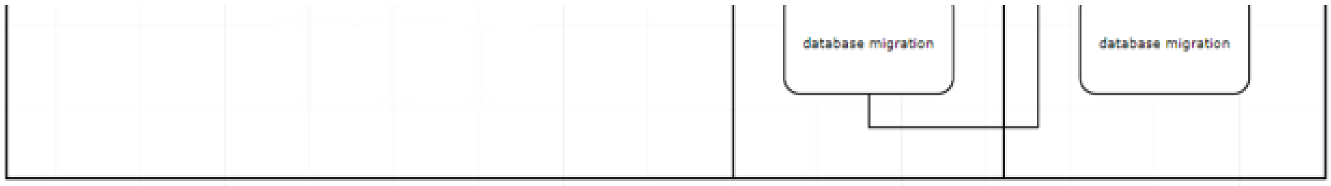


Since we wanted repeatable templated applications that could be daily deployed, the first thing I created was a pattern for deploying a “WebApp”.

I had to make sure the code we deployed would be able to run within a container, and since we had php, dotnetcore and node apps, they were all able to be run on linux. The environment structure that the business wanted was Build → QA → Prod with a focus to getting code into production within minutes if possible.

```
Build Stage
- build.sh -> Build artifacts (dotnet build, npm run build etc.)
- test.sh -> Run unit tests (dotnet tests, node tests)
- integration.sh -> Run integration tests (cypress, selenium and
jest)- coverage.sh -> Coverage reports (dotnet using open cover)
- upload the built artifacts to s3 e.g. s3://artifacts/my-
application/1.230.1/deployable.zip
QA Stage
- pull artifacts from S3
- build docker image
- upload docker image to repository (ECR)
- review infrastructure changes
- apply infrastructure changes (This causes a blue green deployment
to the environment)
- database migration
Prod Stage
- pull artifacts from S3
- build docker image
- upload docker image to repository (ECR)
- review infrastructure changes
- apply infrastructure changes (This causes a blue green deployment
to the environment)
- database migration
```





Here is an example of what one of the build docker/deploy docker scripts looked like:

```
#build.sh
#!/bin/bash
set -e

for dir in `find . -type d`
do

echo "using directory "$dir
if [ $dir = "." ]; then
    echo ""
else

    BASE_REPO=XXXXXXXXXXXX.dkr.ecr.eu-west-2.amazonaws.com
    IMAGE_NAME=${dir:2}
    VERSION_LATEST=latest
    VERSION=2.0
    echo "ImageName:"$IMAGE_NAME
    eval $(aws ecr get-login --region eu-west-2 --no-include-email)
    sleep 1

    docker build $dir -t $IMAGE_NAME:$VERSION
    docker tag $IMAGE_NAME:$VERSION 
$BASE_REPO/$IMAGE_NAME:$VERSION_LATEST
    docker tag $IMAGE_NAME:$VERSION $BASE_REPO/$IMAGE_NAME:$VERSION
    docker push $BASE_REPO/$IMAGE_NAME:$VERSION
    docker push $BASE_REPO/$IMAGE_NAME:$VERSION_LATEST

fi

done
```

And an example of what the docker file looked like:

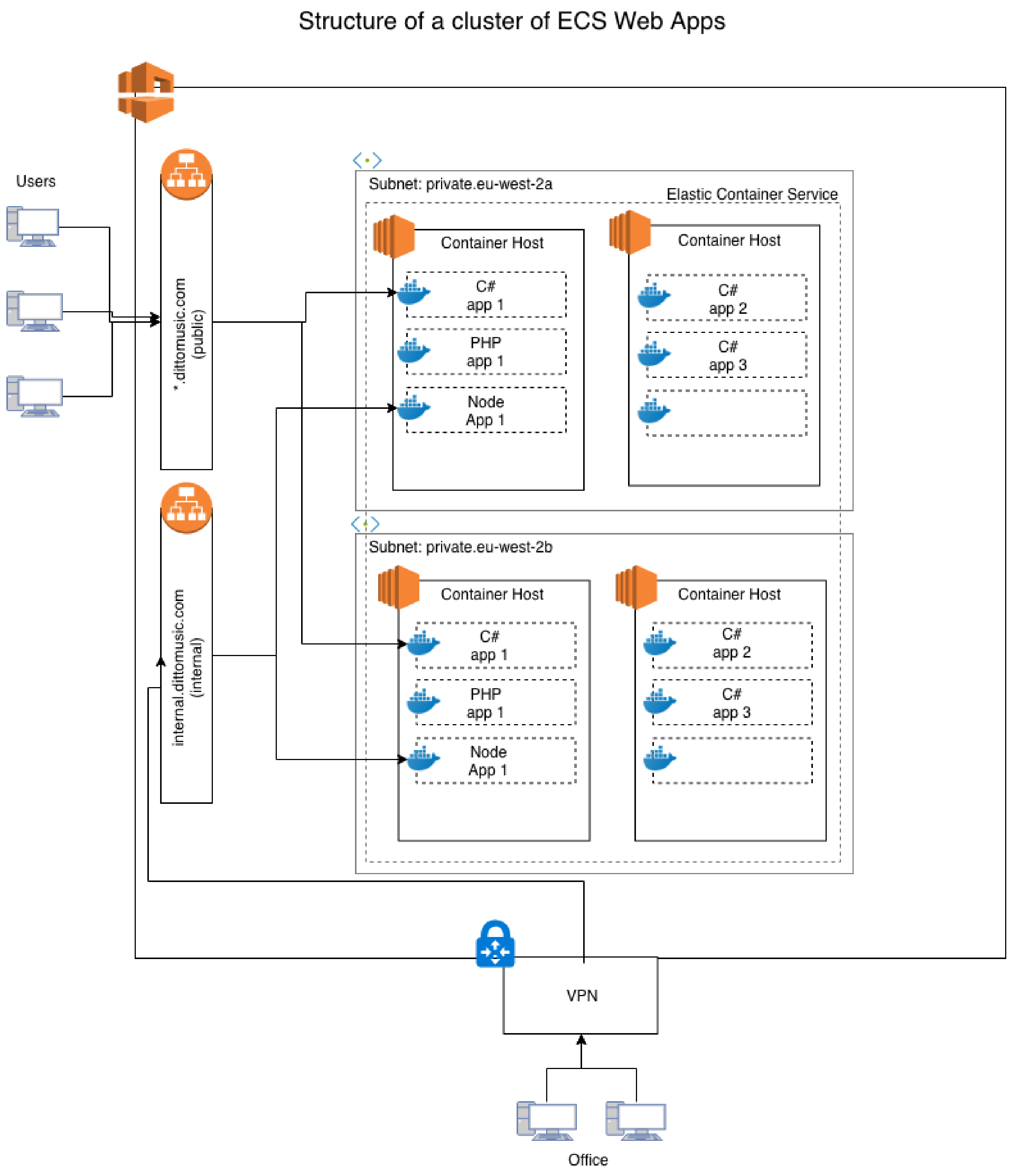
```
FROM microsoft/dotnet:2.1-sdk

WORKDIR /app
COPY . .

ENV Environment="qa"
ENV AwsRegion="eu-west-2"
ENV ApplicationVersion="{APPLICATION_VERSION}"
ENTRYPOINT ["dotnet", "my-application.dll"]
```

Since the WebApp could run any docker image, there were no restrictions on the technology we were able to host, although we used standardised images for dotnetcore applications, node applications and php applications.

We wanted something like this:


Structure of a cluster of ECS Web Apps
Users
*.dittomusic.com (public)
internal.dittomusic.com (internal)
Subnet: private.eu-west-2a
Elastic Container Service
Container Host
C# app 1
PHP app 1
Node App 1
Container Host
C# app 2
C# app 3
Subnet: private.eu-west-2b
Container Host
C# app 1
PHP app 1
Node App 1
Container Host
C# app 2
C# app 3
VPN
Office

Configuration using terraform modules

The configuration for the container was standardised using terraform modules, so that I had full control over the security aspects and what resources each container could access. Each deployment was able to override these default settings if needed.

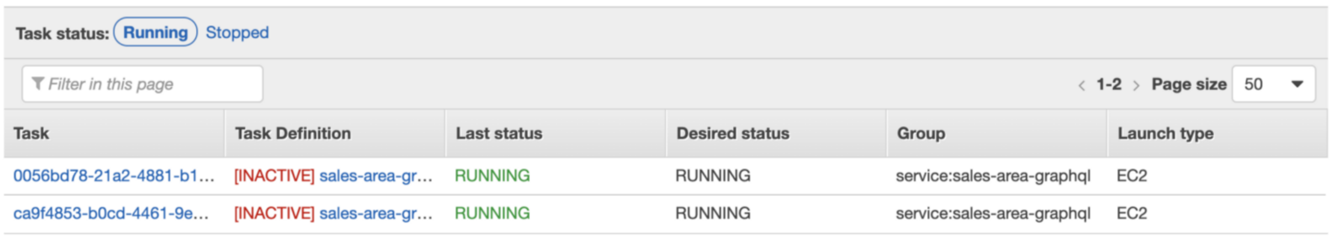
All the containers were hosted within a private subnet, and could only be accessed by being attached to a load balancer, and never externally.

This is an example of the module I used, which only provides the necessary configuration to the template to create the application on deployment, important things being the desired count and the lb_pattern. I also provided functionality to apply scaling rules based on metrics etc. as the module returned a reference to the service, so it was possible to chain extra functionality on.

```
module "ecs-web-app" {
  source                  =
"git::git@github.com:DittoMusic/SCRUBBED"
  app_name                = "${var.app_name}"
  region                  = "${var.region}"
  application_version     = "${var.application_version}"
  task_memory             = "${var.memory}"
  container_port          = "${var.container_port}"
  attach_to_load_balancer = "internal"
  lb_pattern              = "/reporting*"
  lb_rule_number          = 6
  desired_count           = 2
  health_check_period     = 30
}
```

To do the blue green deployment without manual intervention, I setup the blue green switch to utilise the health check support of ALB/ELB.

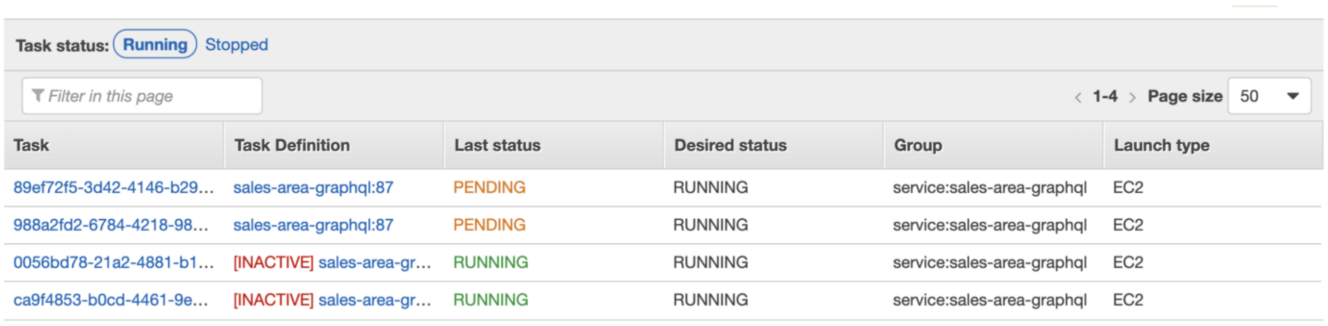
Deployment is triggered, tasks are marked as inactive in AWS and starts the draining process


Task status: Running Stopped

Filter in this page — 1-2 — Page size 50

Task	Task Definition	Last status	Desired status	Group	Launch type
0056bd78-21a2-4881-b1...	[INACTIVE] sales-area-gr...	RUNNING	RUNNING	service:sales-area-graphql	EC2
ca9f4853-b0cd-4461-9e...	[INACTIVE] sales-area-gr...	RUNNING	RUNNING	service:sales-area-graphql	EC2

The new task is started, and enters a pending state



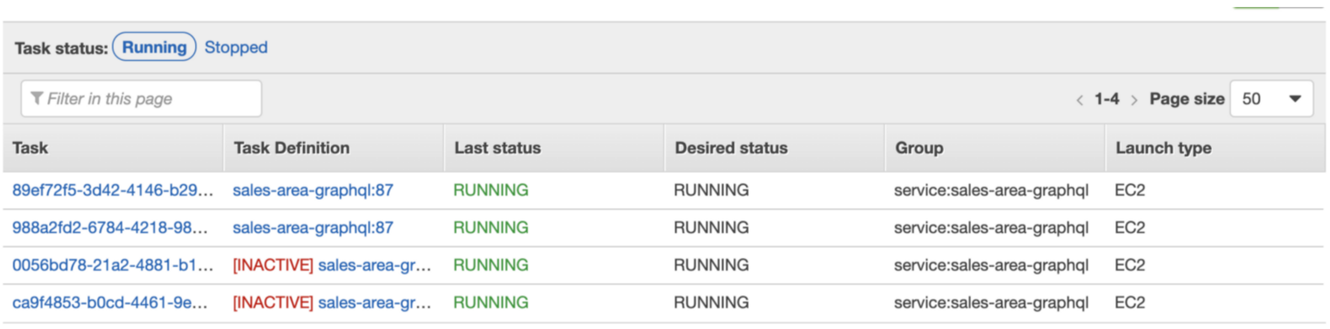
Task status: Running Stopped

Filter in this page

‹ 1-4 › Page size 50

Task	Task Definition	Last status	Desired status	Group	Launch type
89ef72f5-3d42-4146-b29…	sales-area-graphql:87	PENDING	RUNNING	service:sales-area-graphql	EC2
988a2fd2-6784-4218-98…	sales-area-graphql:87	PENDING	RUNNING	service:sales-area-graphql	EC2
0056bd78-21a2-4881-b1…	[INACTIVE] sales-area-gr…	RUNNING	RUNNING	service:sales-area-graphql	EC2
ca9f4853-b0cd-4461-9e…	[INACTIVE] sales-area-gr…	RUNNING	RUNNING	service:sales-area-graphql	EC2

The new task starts successfully, and a health check is performed from the load balancer to container:port



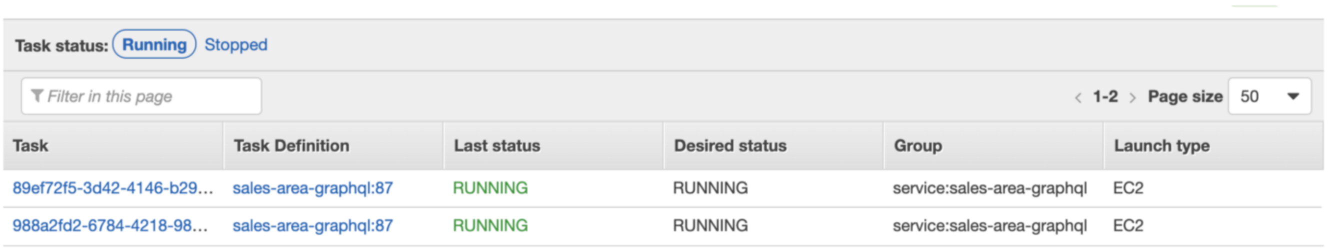
Task status: Running Stopped

Filter in this page

‹ 1-4 › Page size 50

Task	Task Definition	Last status	Desired status	Group	Launch type
89ef72f5-3d42-4146-b29…	sales-area-graphql:87	RUNNING	RUNNING	service:sales-area-graphql	EC2
988a2fd2-6784-4218-98…	sales-area-graphql:87	RUNNING	RUNNING	service:sales-area-graphql	EC2
0056bd78-21a2-4881-b1…	[INACTIVE] sales-area-gr…	RUNNING	RUNNING	service:sales-area-graphql	EC2
ca9f4853-b0cd-4461-9e…	[INACTIVE] sales-area-gr…	RUNNING	RUNNING	service:sales-area-graphql	EC2

After 30 seconds or so, the old version of the containers drained from the cluster



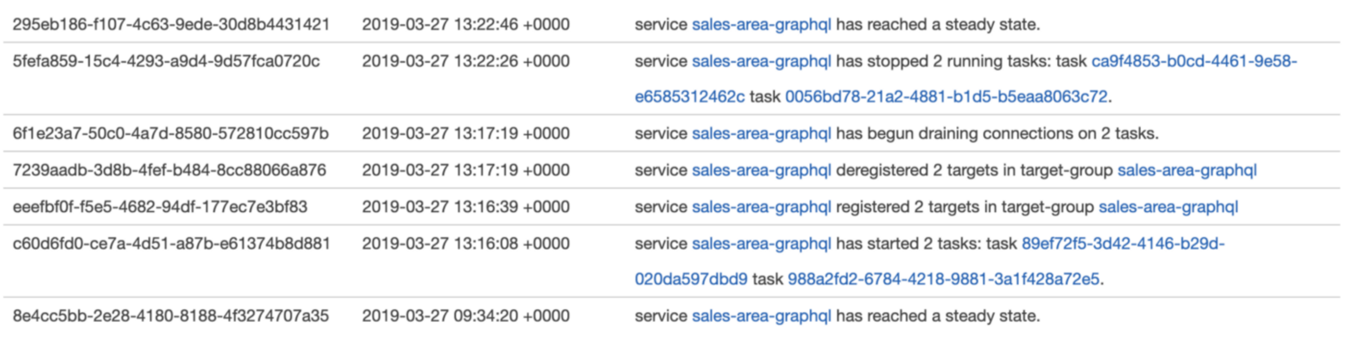
Task status: Running Stopped

Filter in this page

‹ 1-2 › Page size 50

Task	Task Definition	Last status	Desired status	Group	Launch type
89ef72f5-3d42-4146-b29…	sales-area-graphql:87	RUNNING	RUNNING	service:sales-area-graphql	EC2
988a2fd2-6784-4218-98…	sales-area-graphql:87	RUNNING	RUNNING	service:sales-area-graphql	EC2

Here is an example of the log, which shows the blue green deployment.



295eb186-f107-4c63-9ede-30d8b4431421	2019-03-27 13:22:46 +0000	service sales-area-graphql has reached a steady state.
5fefa859-15c4-4293-a9d4-9d57fca0720c	2019-03-27 13:22:26 +0000	service sales-area-graphql has stopped 2 running tasks: task ca9f4853-b0cd-4461-9e58-e6585312462c task 0056bd78-21a2-4881-b1d5-b5eaa8063c72.
6f1e23a7-50c0-4a7d-8580-572810cc597b	2019-03-27 13:17:19 +0000	service sales-area-graphql has begun draining connections on 2 tasks.
7239aadb-3d8b-4fef-b484-8cc88066a876	2019-03-27 13:17:19 +0000	service sales-area-graphql deregistered 2 targets in target-group sales-area-graphql
eeefbf0f-f5e5-4682-94df-177ec7e3bf83	2019-03-27 13:16:39 +0000	service sales-area-graphql registered 2 targets in target-group sales-area-graphql
c60d6fd0-ce7a-4d51-a87b-e61374b8d881	2019-03-27 13:16:08 +0000	service sales-area-graphql has started 2 tasks: task 89ef72f5-3d42-4146-b29d-020da597dbd9 task 988a2fd2-6784-4218-9881-3a1f428a72e5.
8e4cc5bb-2e28-4180-8188-4f3274707a35	2019-03-27 09:34:20 +0000	service sales-area-graphql has reached a steady state.

Blue Green DevOps Automation Load Balancing