**ECS STEPS**

Note that although ECS can use **Fargate** in place of EC2, FARGATE still requires a **VPC**, **Subnets** and **Security Groups.**

IF an ECS Task only needs to be called by other services within the same VPC, then it should live in a **private** subnet (EC2, Lambda, ECS). It would be fronted by an Internal Load Balancer.

ECS Tasks can lookup other ECS tasks using AWS Service Discovery to get the IP of the Task. Or you can hardcode the private Tasks IP address.

Something to consider with Private Subnets is that pulling images from either DockerHub or even ECR, requires public internet access. This means using a NAT GATEWAY on the public subnet so the private subnet can pull the data. NAT gateways charge per GB downloads so frequent updates can be expensive.

An alternative is to consider using VPC Endpoints. This charges an hourly fee but no GB charge and avoids going over the internet so more secure. It does mean an hourly charge though even when nothing is happening.

**VPC ENDPOINTS**

VPC EndPoints are PRIVATE, Secure Connections between your VPC and AWS Services without needing INTERNET or NAT Gateway. There are two types:

**GATEWAY** FREE. Only for S3 and DynamoDB. Uses VPC routetable.

**INETERFACE** CHARGED HOURLY – most services – SQS, SNS, CLOUDWATCH, KMS, SECRETSMANAGER, RDS, EC2, LAMBDA

So literally, most other AWS services will either need VPC ENDPOINT or NATGATEWAY if accessed from your ECS on Private Subnet (which is recommended).

**Step1**

We create a small Python Flask or SPringboot app – it receives XML and simulates processing it (a sleep).

**Step2.**

We Dockerise it.

docker build -t stp .

Now we can test the Container locally by doing:

docker run -p 5000:5000 stp

and simulate calling the service with: ocker run -p 5000:5000 my-flask-app

curl -X POST http://localhost:5000/process -d '<request><id>123</id></request>'

**Step3**

# Create a new REPO to store this in

**aws ecr create-repository --repository-name stp**

# Get Creds so docker can PUSH

* **aws ecr get-login-password --region eu-west-2 | docker login --username AWS --password-stdin <AWS\_ACCOUNT\_ID>.dkr.ecr.eu-west-2.amazonaws.com**
* **docker tag stp:latest <AWS\_ACCOUNT\_ID>.dkr.ecr. eu-west-2.amazonaws.com/my-flask-app:latest**
* **docker push <AWS\_ACCOUNT\_ID>.dkr.ecr. eu-west-2.amazonaws.com/stp:latest**

**NOW WE HAVE A SIMPLE FLASK CONTAINER IN ECR. NEXT STEP. SETUP ECS ON FARGATE (or EC2).**

**SETUP ECS.**

**Step 1**

Create the Initial Cluster if not already created. A cluster is just a logical grouping and has no other significance.

**aws ecs create-cluster --cluster-name my-stp-cluster**

**Step 2**

First we have to create a **ROLE** to allow ECS to:

* Pull images from ECR
* Write logs to CLoudwatch
* Use AWS Services once permission are added

aws iam create-role \

--role-name stpEcsTaskExecutionRole \

--assume-role-policy-document <file://ecs-trust-policy.json>

Where the trust policy file just **says ECS can assume this Role.**

Now we attach a Policy to that Role that says the Role can Pull Images from ECR and write to CLoudwatch.

aws iam attach-role-policy \

--role-name stpEcsTaskExecutionRole \

--policy-arn arn:aws:iam::aws:policy/service-role/AmazonECSTaskExecutionRolePolicy

aws iam attach-role-policy \

--role-name ecsTaskExecutionRole \

--policy-arn arn:aws:iam::aws:policy/CloudWatchLogsFullAccess

**Step 3**

Create and Register a **Task Definition.** Task Definition is just a blue print for how a Container Should run.

It will define things like:

Image name -> Docker or ECR Image to pull. How much CPU and Memory it needs. Env Variables, Ports it needs.

{

"family": "STPTask",

"networkMode": "awsvpc",

"**requiresCompatibilities**": ["FARGATE"],

"executionRoleArn": "arn:aws:iam::717279690473. role/ecsTaskExecutionRole",

"cpu": "256",

"memory": "512",

"**containerDefinitions**": [

{

"name": "flask-container",

"image": "717279690473.dkr.ecr.us-east-1.amazonaws.com/stp:latest",

"portMappings": [

{

"containerPort": 5000,

"hostPort": 5000

}

],

"**environment**": [

{

"name": "DYNAMODB\_TABLE",

"value": "ECS\_Requests"

}

]

}

]

}

**APPLY TASK DEFINITION**

aws ecs register-task-definition --cli-input-json <file://task-def.json>

**RUNNING THE ECS TASK**

Now we have created a Cluster and we have created a Task Definition Blue Print saying how to run a Task but we haven’t yet deployed the Task.

We have two ways we can run the task.

* Run the task directly (Manual, One Time Execution). Runs until Completion or Failure then STOPS.
* Run Task Via a Service. Long running and automatic restarts on failure.

To run directly as a one-off, we can do:

**aws ecs run-task** --cluster my-cluster --task-definition my-task-def --launch-type FARGATE --network-configuration "awsvpcConfiguration={subnets=[subnet-12345],securityGroups=[sg-12345],assignPublicIp=ENABLED}"

Or to run as a Service, we can do:

aws ecs create-service --cluster my-stp-cluster \

--service-name my-stp-service \

--task-definition my-stp-task \

--desired-count 1 \

--launch-type FARGATE \

--network-configuration "awsvpcConfiguration={subnets=[subnet-xxxxxxxx],securityGroups=[sg-xxxxxxxx],assignPublicIp=ENABLED}"

**But we need subnets and Security Groups ?**

SGs are required to enable traffic. In our case, we are running on port 5000 so inbound rules need setting up to permit traffic.

Subnets are required to say WHERE the Task is allowed to run. If we specify 3 subnets (One in each AZ), ECS will spread tasks across the subnets (It will only run as many tasks as the Service requests).

LOGGING

To enable logging, we need to include this in our task definition:

"logConfiguration": {

"logDriver": "awslogs",

"options": {

"awslogs-group": "/ecs/my-stp-task",

"awslogs-region": "eu-west-2",

"awslogs-stream-prefix": "ecs"

}

Note, the log group doesn’t get created automatically so need to create it first.

aws logs create-log-group --log-group-name "/ecs/my-flask-task"

**CI/CD**

There are a few ways to handle the creation here. Suggested way is to have 2 TF branches and use **terraform\_remote\_state** to refer to the infra created in the first TF setup.

Example structure with infra and service directories.

my-ecs-project/

**── infra/**

│ ├── iam.tf # Roles & policies

│ ├── security.tf # Security groups

│ ├── logs.tf # CloudWatch log group

│ ├── vpc.tf # VPC, subnets, route tables

│ ├── variables.tf # Shared variables

│ ├── outputs.tf # Expose values for cross-stack use

│ └── backend.tf # Remote state backend (e.g., S3)

│

**── service/** # App/service layer

│ ├── ecs.tf # ECS service config

│ ├── task-definition.tf # Task definition (container setup)

│ ├── variables.tf # App-specific vars (e.g., image URI)

│ └── backend.tf # Reads from infra's remote state

The reason for this is that we create the Infrastructure but we cant create an ECS Service unless the image is present in ECR so in this case, our CI/CD would execute the initial INFRA creation, then run the docker commands and upload to ECR, then run a second TF apply for the service directory.

LOAD BALANCER

Rather than point to public IP, we should create an ALB. The ALB keeps a static DNS name (Can also be mapped to a friendly Route53 name). The IP may change but DNS name will stay the same.

A LB can be made Internal Only or External with a Boolean flag.

If the LB is used only for this app, it can sit in the service directory.