Index

1. Section 1 - Introduction

This is just a tutorial of how we built a webform in .NET that talks through IAM/S3 to AWS DynamoDB and is using a Docker Swarm setup on a scalable AWS EC2-instance setup. I wrote this guide for my own good (learning by describing), but sharing it in case someone else might benefit from it too.

Link to repository: https://github.com/mymh13/swarm-dotnet-test

1.1 Pre-requisites

AWS account, AWS CLI installed, SSH key, AWS key pair, Docker (Docker Desktop is handy) and basic cloud and .NET knowledge.

2. Infrastructure basics

In this phase we will do the rudimentary basics: Create a security group for the swarm and launch EC2 instances. Further infrastructure (database) will be added later.

2.1 Create a Security Group

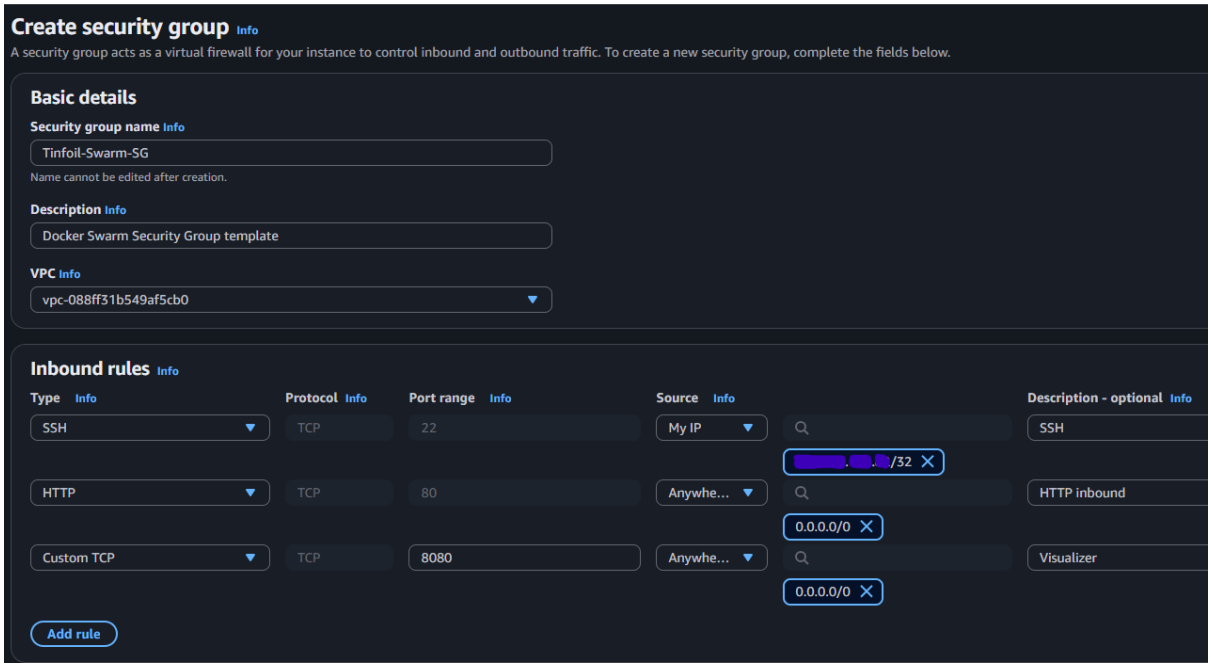
The SG sets the rules and boundaries for the system that uses that particular SG. In this first step, we want to create a SG that primarily handles the inbound rules for our Swarm network.

2.1.1 Create the initial inbound rules

In the AWS console, navigate to EC2 > Security Groups > Create Security Group. Fill in the following:

Name: Chose something descriptive so you know what role the SG plays. You will end up with many SGs eventually and then it will be hard to browse through them.
Description: Something descriptive, see above
VPC: use default VPC for this template, if you can do custom then you do not need this guide 😊
Inbound rules:

Type	Protocol	Port	Source	IP	Description
SSH	TCP	22	My IP	[yourIPhere]	SSH
HTTP	TCP	80	IPv4	0.0.0.0/0	HTTP inbound
Custom TCP	TCP	8080	IPv4	0.0.0.0/0	Visualizer

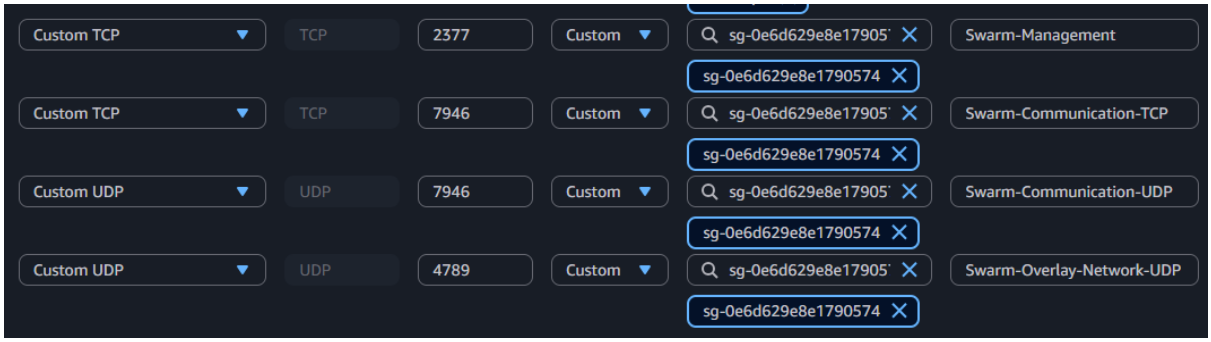


At this point you want to create the security group, there will be four more inbound rules but they will use self-reference so you want to create the SG before it can refer to itself!

2.1.2 Update the Security Group

We need to add the final inbound rules. Edit the Security Group. In the IP/search window, click the window and choice your SG you just created to self-reference.

Do note UDP on the two last ones! It should look something like this:



2.1.3 What the hell did we just do

Before we move on, let us briefly explain what this does:

- SSH (TCP 22) – Lets you connect to the server via SSH. We lock it down to your own IP only, so no one else can try to log in.
- HTTP (TCP 80) – This allows anyone on the internet to reach our application in the browser.
- Visualizer (TCP 8080) – Docker Swarms visualizer tool runs on port 8080. This rule makes us able to reach and view the visualizer in our browser.

- Swarm Management (TCP 2377) – Used for communication between the swarm manager and the worker nodes. Self-referenced so only other servers in the same SG are allowed in.
- Swarm Communication (TCP 7946 + UDP 7946) – These ports handle internal gossip (yes, that is the actual term in distributed systems! It means each node spread information to its neighbours) and node discovery within the system. Self-reference = self-contained, only swarm members allowed.
- Overlay Network (UDP 4789) – This port carries actual container traffic between swarm nodes over the same overlay network. Self-referenced, internal traffic inside the swarm.

2.1.4 Important note regarding SSH inbound:

Personally, I am always on VPN and my IP changes frequently. This means I have to consider how to handle my SSH IP inbound. In a brief project like this I chose to edit the SSH rule and adjust to reflect my new IP now and then, but for a longer term – and especially if you are more than one person handling this node – you need a more robust solution.

2.1.5 Standard vs custom ports

- Standard: TCP 22 is standard SSH-port. TCP 80 is standard HTTP- (web) traffic.
- Widely used: TCP 8080 is not an official standard, but a widely used convention for dashboards and dev tools.
- Custom: TCP 2377, TCP 7946 + UDP 7946 and UDP 4789 are specific to Docker Swarm.

Custom ports are chosen by the system- or application designer, it is important to know the standards and commonly used ones, so we don't accidentally re-use them for custom roles.

2.1.6 TCP vs UDP protocols

Most internet traffic uses one of the two protocols: TCP or UDP.

- TCP (Transmission Control Protocol) – It sets up a connection, checks that messages arrive, re-sends everything that gets lost. It is reliable but has a bit more overhead. We use this for SSH and HTTP because there accuracy is more important than speed.
- UDP (User Datagram Protocol) – It is sent without checking if it arrives, thus making it much faster and uses less overhead than TCP, but delivery is not guaranteed! This is useful for Docker Swarm's internal networking, especially if we are working idempotent as we can accept lost packets.

In short: TCP for reliable traffic, UDP for fast cluster chatter and data transport.

2.1.7 SG-Summary

We have create a Security Group that will handle inbound rules, boundaries and internal traffic.

2.2 Launch EC2 Instances

EC2 Instances is basically just Virtual Machines (VM) but let us stick to AWS-terminology. They will work as "servers" (nodes/hosts might be more appropriate description) for our Swarm. At this point we could also have considered other VMs for roles like a Bastion Host, or a Database or something else, but let us keep this tutorial simple for now and just focus on the Docker Swarm.

2.2.1 First let us create the instances

AWS Console > EC2 > Instances > Launch Instance

Name: Just like with the SG group and classes/methods, always use descriptive names!
AMI: Amazon Linux 2023
Instance type: t3.small
Key pair: This was a pre-req 😊 chose your already pre-set-AWS-key-pair for this
Network settings:

- VPC – default VPC
- Subnet – default Subnet
- Auto-assign public IP – enable
- Firewall / Security Group – Select existing (the one you just created in step 1)

Advanced details > User Data: add a start script for the instance:

```
#!/bin/bash
dnf update -y
dnf install -y docker
systemctl enable --now docker
usermod -aG docker ec2-user
```

2.2.1.1 ..and this script does what?

`dnf update -y` - Updates all existing software on the instance. The -y flag means "say yes to all prompts" so the process doesn't stop to ask

`dnf install -y installs docker` - Installs Docker from the Amazon Linux package repository. Again, -y auto-confirms

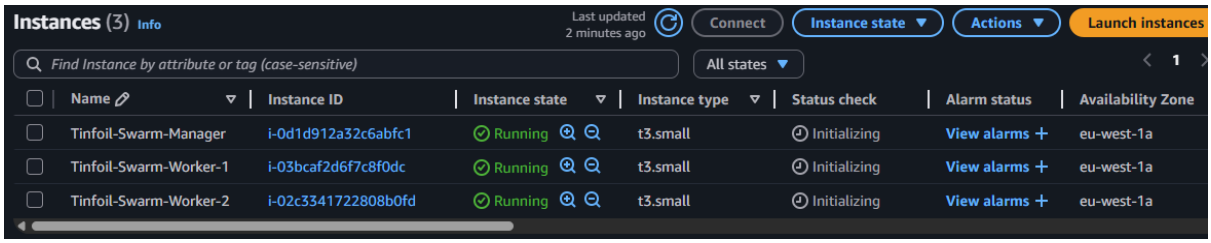
`systemctl enable –now docker` - Tells the system to start Docker right away (--now) and to also start it automatically on every reboot (enable)

`usermod -aG docker ec2-user` - Adds the default AWS login user (ec2-user) to the docker group, so you can run Docker commands without needing sudo every time

2.2.2 Summary, launch and multiples

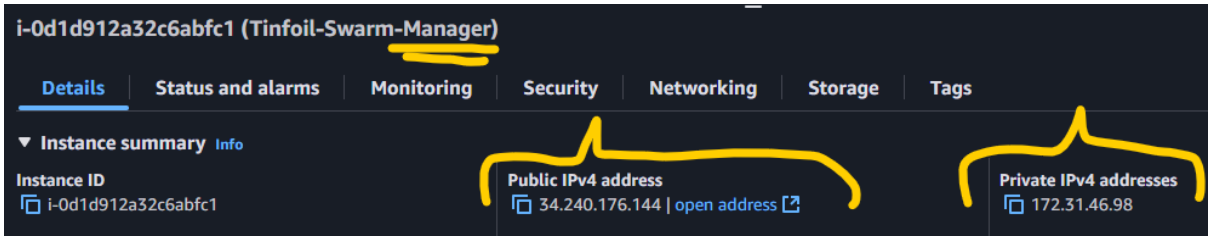
On your right hand, you should have a "Summary" window, with a box named "number of instances" – chose 3 and then click Launch instance.

Then go your EC2 > Instances and rename the second and third Manager-instances to Worker-1 + 2:



2.2.3 Note Instance IPs

View the instances (in the version above, September 2025, you can just click the instance tick box to the left and you get a quick view of the instance in a window below the instances). You want to make sure you have the Public IP and the Private IP for the three instances. I copy-pasted them down in a markdown document for easy access.



2.2.4 EC2-Summary

We have set up three EC2 Instances, one working as a Master and two Workers nodes.

2.3 Initialize Docker Swarm via SSH

I hope you have got the IPs handy and SSH + Key Pair is set up. In this phase we will initialize Docker Swarm on our EC2 Instances and verify the cluster. Let's go.

2.3.1 Create a nifty helper-document (optional but recommended)

I used Obsidian and wrote in markdown, but anything will do. What I did was having a number of sections where I listed the public/private IPs of the Instances and a copy-paste command ready to SSH into the Instances. It should look something like this:

```
Swarm Manger
- public IP - 85.482.592.492
- private IP - 172.31.46.98
ssh -i ~/.ssh/<your-key-name>.pem ec2-user@85.482.592.492
```

Do note:

- This assumes your SSH key is in the default user/.ssh/ directory. If you are a Windows user I can highly recommend to always SSH keys in the C:\Users\<username>\.ssh directory
- The public IP in the example above has randomized numbers, this is information you should protect so make sure you do not save this information open in – for example – your project folder (if you use that for a GitHub repo or similar)

2.3.2 Connect to the Manger and initialize Swarm

Use the SSH command found in 2.3.1 to SSH into your Manager node:

```
ssh -i ~/.ssh/your-key.pem ec2-user@<manager-public-ip>
```

Then we will initialize the Swarm on the Manager:

```
sudo docker swarm init --advertise-addr <manager-private-ip>
```

We should get a verification that looks similar to this:

```
Swarm initialized: current node (xyz123) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-xxx... <manager-private-ip>:2377
```

Copy the join-command and save it to your helper-document.

```
sudo docker swarm join --token SWMTKN-1-<long-string> <manager-private-ip>:2377
```

2.3.2.1 I got most of that. I think

No worries, let me elaborate what you just did.

- By SSHing into the Manager node and running docker swarm init, we turned that server into the "brain" of the swarm.
- The `--advertise-addr <manager-private-ip>` flag makes sure other servers connect over the private AWS network (faster and more secure than public IP).
- Docker then gave us a join token and command. This is like a one-time password that worker nodes will use to join the swarm. Copy it somewhere safe — you'll need it in the next step.

2.3.3 Let us add the Worker nodes

Basically, we will join the existing Swarm as worker nodes, there is not much else to it. Open a new terminal window and run these commands:

```
ssh -i ~/.ssh/your-key.pem ec2-user@<worker1-public-ip>
# Run the join command from step 2.3.2
sudo docker swarm join --token SWMTKN-1-xxx... <manager-private-ip>:2377
```

Repeat the above step but replace worker1-public-ip with worker2's public IP. I do not think we need to explain the command in-depth, as you see we use "swarm join" which should be self-explanatory.

You should have three terminal windows that look something like this:

```
[ec2-user@ip-172-31-46-98 ~]$ sudo docker swarm init --advertise-addr 172.31.46.98
Swarm initialized: current node (sfzku1dl4tps762bvohufz89t) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-32wel7ks1h0hkee5q3t3virts7d74cjieti9kfivtksfq1nenn-7
```

```
[ec2-user@ip-172-31-39-19 ~]$ docker swarm join --token SWMTKN-1-32wel7ks1h0hkee5q3t3virts7
72.31.46.98:2377
This node joined a swarm as a worker.
[ec2-user@ip-172-31-39-19 ~]$
```

```
[ec2-user@ip-172-31-42-153 ~]$ docker swarm join --token SWMTKN-1-32wel7ks1h0hkee5q3t3virts
172.31.46.98:2377
This node joined a swarm as a worker.
[ec2-user@ip-172-31-42-153 ~]$
```

2.3.3.1 Let us quickly verify the Swarm cluster

In the Manager nodes' terminal, type this: `sudo docker node ls`

You should see this:

```
[ec2-user@ip-172-31-46-98 ~]$ sudo docker node ls
ID                            HOSTNAME                                    STATUS    AVAILABILITY   MANAGER STATUS   ENGINE VERSION
7dyzmrcn3ausuqq0mjf8we65f     ip-172-31-39-19.eu-west-1.compute.internal  Ready     Active                          25.0.8
1eahizoejo58fet83gfmkg06e     ip-172-31-42-153.eu-west-1.compute.internal Ready     Active                          25.0.8
sfzku1dl4tps762bvohufz89t *   ip-172-31-46-98.eu-west-1.compute.internal  Ready     Active         Leader           25.0.8
```

2.3.4 Swarm-Summary

We have initialized a Docker Swarm on our Manager, joined with Workers and verified it.

2.4 Deploying the services manually

Ideally this is something that we have an automatic solution for, some IaC solution would be preferable. But this is just a basic template to grasp the concept so we will deploy manually.

2.4.1 Create a Docker Compose file

On the Manager node, we will create the stack file by typing (pasting) this:

```
cat > docker-stack.yml << 'EOF'
version: "3.8"
services:
  web:
    image: nginx:stable-alpine
    deploy:
      replicas: 3
      placement:
        preferences:
          - spread: node.id   # prefer even distribution across nodes
```

```
      restart_policy:
        condition: on-failure
      update_config:
        parallelism: 1
        delay: 5s
    ports:
      - "80:80"
    networks: [webnet]

  viz:
    image: dockersamples/visualizer:stable
    deploy:
      placement:
        constraints: [node.role == manager]
    ports:
      - "8080:8080"
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
    networks: [webnet]

networks:
  webnet:
    driver: overlay
EOF
```

2.4.1.1 Let us not go into all details, but roughly what this does is this

The cat > docker-stack.yml << 'EOF' ... EOF command is a way to create a new file (docker-stack.yml) and paste content directly into it. Everything between the two EOF markers becomes the file content.

This file is written in YAML, which is very sensitive to spacing and indentation — so it's best to copy-paste rather than type it manually.

The stack defines two services:

- Web: An nginx server with 3 replicas, exposed on port 80
- Viz: A Docker Swarm visualizer, only running on the manager node, exposed on port 8080 (we set that port in the SG but the Manager runs the Viz itself)

At the bottom, it also defines a custom overlay network (webnet) that lets the services talk to each other across swarm nodes.

2.4.2 Deploy the freshly created Stack

Still on the Manager node, run the deploy command:

```
sudo docker stack deploy -c docker-stack.yml myappname
```

```
[ec2-user@ip-172-31-46-98 ~]$ sudo docker stack deploy -c docker-stack.yml myappname
Creating network myappname_webnet
Creating service myappname_viz
Creating service myappname_web
```

This tells Docker Swarm to deploy the stack file (-c docker-stack.yml) and give the stack the name myappname.

2.4.3 Verify the deployment

```
# Show all stacks currently running in the Swarm
sudo docker stack ls

# List the services inside the stack, along with how many replicas are running. If we followed the guide you should see web with 3/3 replicas and viz with 1/1 – since we have 3x instances but only the Manager runs the viz.
sudo docker service ls

# Detailed info about each service: which node it is running on, current state, specific container ID. This is useful when troubleshooting.
sudo docker service ps myappname_web
sudo docker service ps myappname_viz
```

The comment above each command explains them further in-depth.

```
[ec2-user@ip-172-31-46-98 ~]$ sudo docker stack ls
NAME         SERVICES
myappname    2
[ec2-user@ip-172-31-46-98 ~]$ sudo docker service ls
ID             NAME             MODE         REPLICAS   IMAGE                             PORTS
orr1o0zokn16   myappname_viz    replicated   1/1        dockersamples/visualizer:stable   *:8080->8080/tcp
oq6uit43aetn   myappname_web    replicated   3/3        nginx:stable-alpine               *:80->80/tcp
```

```
[ec2-user@ip-172-31-46-98 ~]$ sudo docker service ps myappname_web
ID             NAME              IMAGE                NODE                                          DESIRED STATE   CURRENT STATE            ERROR     PORTS
pvilkvk75478   myappname_web.1   nginx:stable-alpine   ip-172-31-46-98.eu-west-1.compute.internal    Running         Running 3 minutes ago
z78t0hc6xyfj   myappname_web.2   nginx:stable-alpine   ip-172-31-42-153.eu-west-1.compute.internal   Running         Running 3 minutes ago
eg510zadh0b3   myappname_web.3   nginx:stable-alpine   ip-172-31-39-19.eu-west-1.compute.internal    Running         Running 3 minutes ago
[ec2-user@ip-172-31-46-98 ~]$ sudo docker service ps myappname_viz
ID             NAME              IMAGE                             NODE                                          DESIRED STATE   CURRENT STATE            ERROR
    PORTS
nzat3lxmepib   myappname_viz.1   dockersamples/visualizer:stable   ip-172-31-46-98.eu-west-1.compute.internal    Running         Running 3 minutes ago
```

2.4.4 Deploy-Summary

We have created and deployed the Docker Compose-file on the Manager and verified it.

2.5 Test web service and see if the service can scale

Now let's test that the web service is accessible, and then try scaling it up and down.

2.5.1 Let us first verify web access

Open a browser and go to the public IP of any node:

`http://<public-ip-of-any-node>`

You should see the default Nginx welcome page. Then check the visualizer service:

`http://<manager-public-ip>:8080/`

If everything is configured correctly, you'll see a live visualization of your swarm.

- Port 80 needs to be open for web access.
- Port 8080 needs to be open for the visualizer.

If either page doesn't load, double-check your Security Group and your stack file to make sure the correct ports are open to the right IPs.

2.5.2 Scale the services

Keep that Visualizer browser tab open, and run these commands on the Manager node:

```
# Scale up nginx to 5 replicas
sudo docker service scale myappname_web=5

# Check scaling
sudo docker service ps myappname_web

# Scale back down to 3
sudo docker service scale myappname_web=3
```

You should see containers appear and disappear in the visualizer as the service scales up and down.

Do note: Scaling here means adding or removing containers, not EC2 instances. Your swarm is still running on the same set of servers. You're just changing how many copies of the nginx container run across them.

2.5.3 Tools to monitor the services

You won't need these right now, as you saw the services running, but leaving this here as it is helpful tools in the future:

```
# Watch service status (Ctrl + C to exit)
watch sudo docker service ls

# View service logs
sudo docker service logs myappname_web
sudo docker service logs myappname_viz
```

"Watch" is live monitoring, so you will have to Ctrl + C to exit / cancel the operation.

Logs are useful tools, as a developer these are your best friends. Logging separate logfiles for separate services / functions is something you will want to do and learn.

2.5.4 Web-Summary

We visit the public IPs, check that the service is running and try the Visualizer.

2.6 Tools for cleaning up (optional)

At this point you might want to clean up a stack (if you are running one), or leave / reset the Swarm.

2.6.1 Remove stack

On the manager node: `sudo docker stack rm myappname`

rm means "remove", it is a good command to know if you are working with files and directories too.

2.6.2 Leave Swarm

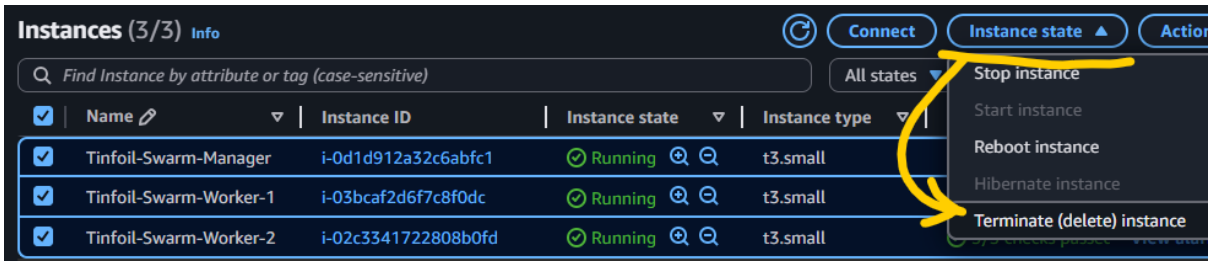
You might want to reset the swarm, alter instances or something else:

```
# On workers
sudo docker swarm leave
```

```
# On manager (force)
sudo docker swarm leave --force
```

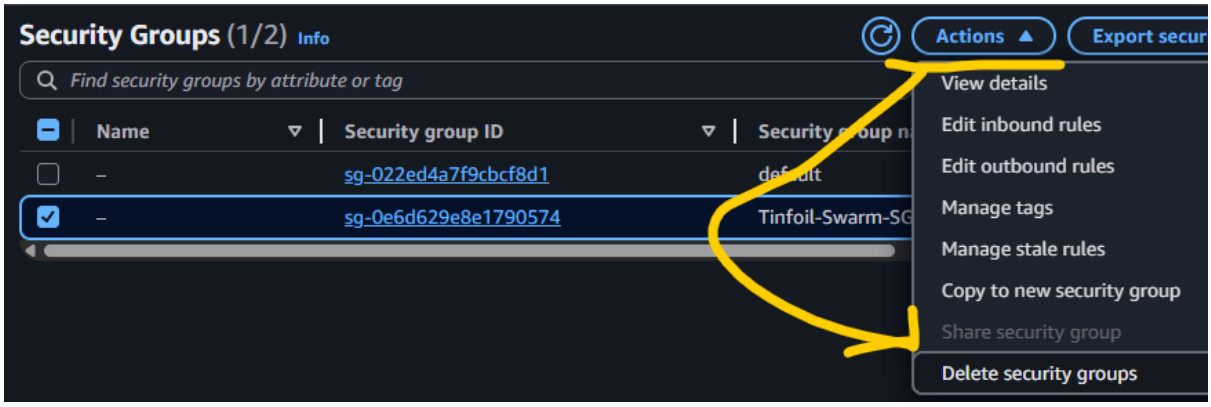
2.6.3 Terminate Instances

Visit EC2 > Instances. Select all three instances. Under “Instance state” chose Terminate.



2.6.4 Delete Security Group

Visit EC2 > Security Groups. Under Actions, Delete the SG.



2.7 Summary of above sections

We have built a SG, launched three instances and deployed a Docker Swarm on them, making sure it is healthy, and all the roles are functional.

In the class when we were at this point, we were supposed to do a .NET application and use DynamoDB for handling files that we send (and possibly retrieve, but that was a bonus) from our .NET application. As I have taken down and up these swarms now multiple times, I feel it is appropriate to create section 3 here – create the above system but in CloudFormation (or Terraform, but let us begin with CF).

3 CloudFormation me up, Scotty

I will build this using the Nested Stacks system as I really, strongly dislike messy long files. It has other benefits too:

- Separations of Concern – Each section is independent from other sections. This is good both from a modularity concept but it also easier when trying to solve problems that arise
- Deploy – Deployment benefits greatly from being separated. You can work on one module and just deploy that one, especially if we are working idempotent
- Adaptability – We might want to expand, or reduce, the program. Adapt to various needs. Then it might be beneficial to just replace or remove or add sections that we need.

That said, let us get going!

3.1 S3 Bucket roles first

Before CloudFormation can deploy nested stacks, we need a storage location for the templates and a user that has permissions to interact with that location. This is done in two parts.

3.1.1 Create the bucket first

1. Go to AWS Console > S3 > Create bucket
2. Enter a unique bucket name (ex., cf-artifacts-<account-id>-eu-west-1). Bucket names must be globally unique
3. Select the region where you plan to deploy your stacks (ex.., eu-west-1)
4. Leave other settings at defaults for now
5. Click Create bucket

You now have a dedicated bucket for storing CloudFormation templates and artifacts.

3.1.2 Then attach the policy to the user so you can use the bucket

1. Go to AWS Console > IAM > Users > select your user
2. Click Add Permissions > Create inline policy
3. Switch to the JSON tab and paste in your policy (replace <your-bucket-name> with the one you just created in 3.1.1.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:ListBucket",
        "s3:GetBucketLocation"
      ],
      "Resource": "arn:aws:s3:::<your-bucket-name>"
    },
    {
      "Effect": "Allow",
      "Action": [
        "s3:GetObject",
        "s3:PutObject",
        "s3:DeleteObject",
        "s3:DeleteObjectVersion"
      ],
      "Resource": "arn:aws:s3:::<your-bucket-name>/*"
```

```
    }
  ]
}
```

Name the policy something descriptive (ex. S3_bucket_policy or cf-s3access-policy) and save it. Your IAM user now has the necessary permissions to interact with the S3 bucket – congrats!

3.2 Quick conversion – CloudFormation style

I recommend visiting the repo for the latest code: https://github.com/mymh13/swarm-dotnet-test

Goal: take the manual Swarm setup from sections 1–2 and express it as Infrastructure as Code using nested CloudFormation stacks. We're not changing what we build—just *how* we build it.

Meow

3.2.1 Nested stacks structure

Child templates live in S3 and a tiny **root** template orchestrates them in order.

```
/infra
  /templates
    root.yaml                  # orchestrates children
    00-sg-swarm.yaml           # Security Group for swarm
    10-ec2-swarm.yaml          # EC2 Manager + 2 Workers (Docker installed)
  /parameters
    dev.json                   # example parameter set for root.yaml
```

3.2.2 What why when how all those cookies, conceptually? IaC is cool

Mental (music) model:

- root.yaml = the conductor. Look at me waving this tiny stick!
- Child templates = sections of the orchestra (SG, EC2).
- S3 = the sheet music library (where templates live).
- update-stack = the downbeat that makes every-stack play the latest score.

What each piece does (conceptually):

- root.yaml (master): Orchestrates order and wiring. It calls the SG stack first, then the EC2 stack, and passes outputs (like the SG ID) to the next stack's inputs. It's the single command you use to create, update, and delete *everything*.
- 00-sg-swarm.yaml (security): Encodes the exact ports you opened manually (SSH 22 locked to your IP, HTTP 80, Visualizer 8080 to world; Swarm mgmt/overlay ports self-referenced). Same rules, now declarative.
- 10-ec2-swarm.yaml (compute): Launches three EC2s (1 manager, 2 workers) in your chosen subnet/SG and runs user-data to install Docker. This mirrors your earlier "install Docker on each node" step.

Why nested stacks for this:

- Separation of concerns: Each child template does one thing well. Easier to reason about, change, and reuse.

- Safer iteration: You can update just one child (ex., SG rules) and then update-stack the root—CloudFormation figures out the minimal change.
- Single tear-down: Deleting the root stack removes the whole setup in the right order.
- S3 as source of truth: You upload the child templates to S3; root.yaml references them via TemplateURL. When you push new child templates to S3, the next update-stack picks them up.

What we do not automate (yet):

We *stop* at "Docker installed." You still:

- SSH to the manager > docker swarm init --advertise-addr <manager-private-ip>
- SSH to each worker > run the docker swarm join ... the manager printed
- Deploy docker-stack.yml as before
  (We'll automate init/join later with roles/SSM or cloud-init once the foundation is stable.)

Parameters & environments (lightweight, IaC-friendly):

We keep a small infra/parameters/dev.json (local, ignored by Git) with things like:

- TemplateBucket, TemplatePrefix (where child templates live in S3)
- VpcId, PublicSubnetId (use default VPC/subnet for now)
- AllowedSshCidr, KeyPairName, InstanceType, AmiId

This makes create/update/teardown a single command each (no ad-hoc clicking, no scripts).

3.2.3 root.yaml (Master stack)

This will be a long "skeleton" code, and this all assumes you have basic understanding of AWS and Cloudformation. The SG/EC2-setup we run should be self-explanatory more or less though. Template:

```
AWSTemplateFormatVersion: '2010-09-09'
Description: Root stack for Docker Swarm (manager + 2 workers) using nested
stacks

Parameters:
  StackNamePrefix:
    Type: String
    Default: swarm-cf
    Description: Prefix used for naming
  TemplateBucket:
    Type: String
    Description: S3 bucket that stores child templates (e.g. cf-artifacts-
<acct>-<region>)
  TemplatePrefix:
    Type: String
    Default: swarm-iac/templates/
    Description: Key prefix inside the S3 bucket
  VpcId:
    Type: AWS::EC2::VPC::Id
```

```
    Description: VPC to deploy into (choose your default VPC)
  PublicSubnetId:
    Type: AWS::EC2::Subnet::Id
    Description: Public subnet for all three instances (default subnet is
fine)
  AllowedSshCidr:
    Type: String
    Description: Your IP in CIDR for SSH (e.g. 1.2.3.4/32)
  KeyPairName:
    Type: AWS::EC2::KeyPair::KeyName
    Description: Existing EC2 key pair name
  InstanceType:
    Type: String
    Default: t3.small
  AmiId:
    Type: AWS::EC2::Image::Id
    Description: Amazon Linux 2023 AMI in this region

Resources:
  SwarmSecurityGroupStack:
    Type: AWS::CloudFormation::Stack
    Properties:
      TemplateURL: !Sub
'https://s3.${AWS::Region}.amazonaws.com/${TemplateBucket}/${TemplatePrefix}00
-sg-swarm.yaml'
      Parameters:
        StackNamePrefix: !Ref StackNamePrefix
        VpcId: !Ref VpcId
        AllowedSshCidr: !Ref AllowedSshCidr

  SwarmEc2Stack:
    Type: AWS::CloudFormation::Stack
    DependsOn: SwarmSecurityGroupStack
    Properties:
      TemplateURL: !Sub
'https://s3.${AWS::Region}.amazonaws.com/${TemplateBucket}/${TemplatePrefix}10
-ec2-swarm.yaml'
      Parameters:
        StackNamePrefix: !Ref StackNamePrefix
        PublicSubnetId: !Ref PublicSubnetId
        SecurityGroupId: !GetAtt
SwarmSecurityGroupStack.Outputs.SwarmSecurityGroupId
        KeyPairName: !Ref KeyPairName
        InstanceType: !Ref InstanceType
        AmiId: !Ref AmiId

Outputs:
  ManagerPublicIp:
```

```
    Value: !GetAtt SwarmEc2Stack.Outputs.ManagerPublicIp
  Worker1PublicIp:
    Value: !GetAtt SwarmEc2Stack.Outputs.Worker1PublicIp
  Worker2PublicIp:
    Value: !GetAtt SwarmEc2Stack.Outputs.Worker2PublicIp
  SecurityGroupId:
    Value: !GetAtt SwarmSecurityGroupStack.Outputs.SwarmSecurityGroupId
```

3.2.4 00-sg-swarm.yaml (Swarm Security Group)

This stack controls the Swarms SG.

```
AWSTemplateFormatVersion: '2010-09-09'
Description: Security Group for Swarm (SSH, HTTP, Viz, Swarm ports)

Parameters:
  StackNamePrefix:
    Type: String
  VpcId:
    Type: AWS::EC2::VPC::Id
  AllowedSshCidr:
    Type: String

Resources:
  SwarmSG:
    Type: AWS::EC2::SecurityGroup
    Properties:
      GroupDescription: !Sub '${StackNamePrefix}-swarm-sg'
      VpcId: !Ref VpcId
      # Only non-self-referencing rules here
      SecurityGroupIngress:
        # SSH (locked to your IP)
        - IpProtocol: tcp
          FromPort: 22
          ToPort: 22
          CidrIp: !Ref AllowedSshCidr
        # HTTP 80 (world)
        - IpProtocol: tcp
          FromPort: 80
          ToPort: 80
          CidrIp: 0.0.0.0/0
        # Visualizer 8080 (world)
        - IpProtocol: tcp
          FromPort: 8080
          ToPort: 8080
          CidrIp: 0.0.0.0/0
      # Egress is outward traffic
      SecurityGroupEgress:
```

```
        - IpProtocol: -1
          CidrIp: 0.0.0.0/0
      Tags:
        - Key: Name
          Value: !Sub '${StackNamePrefix}-swarm-sg'

  # Self-referencing rules as separate resources (avoids circular dependency)
  IngressSwarmMgmt2377:
    Type: AWS::EC2::SecurityGroupIngress
    Properties:
      GroupId: !Ref SwarmSG
      IpProtocol: tcp
      FromPort: 2377
      ToPort: 2377
      SourceSecurityGroupId: !Ref SwarmSG

  IngressGossipTCP7946:
    Type: AWS::EC2::SecurityGroupIngress
    Properties:
      GroupId: !Ref SwarmSG
      IpProtocol: tcp
      FromPort: 7946
      ToPort: 7946
      SourceSecurityGroupId: !Ref SwarmSG

  IngressGossipUDP7946:
    Type: AWS::EC2::SecurityGroupIngress
    Properties:
      GroupId: !Ref SwarmSG
      IpProtocol: udp
      FromPort: 7946
      ToPort: 7946
      SourceSecurityGroupId: !Ref SwarmSG

  IngressOverlayUDP4789:
    Type: AWS::EC2::SecurityGroupIngress
    Properties:
      GroupId: !Ref SwarmSG
      IpProtocol: udp
      FromPort: 4789
      ToPort: 4789
      SourceSecurityGroupId: !Ref SwarmSG

Outputs:
  SwarmSecurityGroupId:
    Value: !Ref SwarmSG
```

3.2.5 10-ec2-swarm.yaml (EC2 Manager + 2 Workers, Docker installed)

Here we create the EC2-Swarm, and set the User Data to install docker, like we did here.

```yaml
AWSTemplateFormatVersion: '2010-09-09'
Description: EC2 instances for Swarm (1 manager, 2 workers) with Docker
installed

Parameters:
  StackNamePrefix:
    Type: String
  PublicSubnetId:
    Type: AWS::EC2::Subnet::Id
  SecurityGroupId:
    Type: String
  KeyPairName:
    Type: AWS::EC2::KeyPair::KeyName
  InstanceType:
    Type: String
  AmiId:
    Type: AWS::EC2::Image::Id

Mappings: {}

Resources:
  Manager:
    Type: AWS::EC2::Instance
    Properties:
      ImageId: !Ref AmiId
      InstanceType: !Ref InstanceType
      KeyName: !Ref KeyPairName
      SubnetId: !Ref PublicSubnetId
      SecurityGroupIds: [ !Ref SecurityGroupId ]
      Tags:
        - Key: Name
          Value: !Sub '${StackNamePrefix}-manager'
      UserData:
        Fn::Base64: !Sub |
          #cloud-config
          runcmd:
            - dnf update -y
            - dnf install -y docker
            - systemctl enable --now docker
            - usermod -aG docker ec2-user

  Worker1:
    Type: AWS::EC2::Instance
    Properties:
```

```
      ImageId: !Ref AmiId
      InstanceType: !Ref InstanceType
      KeyName: !Ref KeyPairName
      SubnetId: !Ref PublicSubnetId
      SecurityGroupIds: [ !Ref SecurityGroupId ]
      Tags:
        - Key: Name
          Value: !Sub '${StackNamePrefix}-worker-1'
      UserData:
        Fn::Base64: !Sub |
          #cloud-config
          runcmd:
            - dnf update -y
            - dnf install -y docker
            - systemctl enable --now docker
            - usermod -aG docker ec2-user

  Worker2:
    Type: AWS::EC2::Instance
    Properties:
      ImageId: !Ref AmiId
      InstanceType: !Ref InstanceType
      KeyName: !Ref KeyPairName
      SubnetId: !Ref PublicSubnetId
      SecurityGroupIds: [ !Ref SecurityGroupId ]
      Tags:
        - Key: Name
          Value: !Sub '${StackNamePrefix}-worker-2'
      UserData:
        Fn::Base64: !Sub |
          #cloud-config
          runcmd:
            - dnf update -y
            - dnf install -y docker
            - systemctl enable --now docker
            - usermod -aG docker ec2-user

Outputs:
  ManagerPublicIp:
    Value: !GetAtt Manager.PublicIp
  Worker1PublicIp:
    Value: !GetAtt Worker1.PublicIp
  Worker2PublicIp:
    Value: !GetAtt Worker2.PublicIp
```

Note:
This intentionally stops at Docker installed. After stack completes, follow the tutorial's SSH steps:

- SSH to manager, run docker swarm init --advertise-addr <manager-private-ip>.
- SSH to each worker, run the docker swarm join ... command from the manager output.
- Deploy your docker-stack.yml exactly as before.

3.2.6 Nested stacks – big picture (TL;DR)

- You upload child templates to S3.
- root.yaml points to those S3 URLs and wires stacks in order.
- Create/Update the *root* stack only > everything else follows.
- Delete the *root* stack > everything tears down cleanly.
- Manual Swarm init/join stays for now (keeps the foundation simple); we'll automate later.

3.3 Important note on SSH, dynamic IP ranges, ingress/egress

If you like me run a VPN that alternates your IP now and then, it can be problematic to run "your own IP" as SSH-in-setting as per the 2.1.1 Inbound rules we set.

3.3.1 This is important to know re: SSH / AllowedSshCidr

- The template does not auto-detect your IP. AllowedSshCidr is a parameter you set in dev.params.json (or via CLI) before create-stack / update-stack.
- If your VPN gives you a new IP, you'll need to edit AllowedSshCidr to <new-ip>/32 and run update-stack to open SSH again.
- Existing SSH sessions remain allowed if your source IP doesn't change. If the VPN reassigns your IP mid-session, the TCP connection will drop, and you'll need to update the rule and reconnect.
- Alternatives (for later):
    - Use a stable CIDR from your VPN provider (if they publish one) instead of a single /32
    - SSM Session Manager (no inbound 22 at all) by attaching an IAM role and enabling SSM on the instances
    - A bastion host with a fixed IP/Security Group that's allowed to SSH, while nodes deny world-wide SSH

So your dev.params.json probably have this, and you need to modify it each time you swap IP:

```
    {
        "ParameterKey": "AllowedSshCidr",
        "ParameterValue": "your.id.range.here/32"
    },
```

You can find your IP by running this:

```
curl -s https://checkip.amazonaws.com
```

3.3.2 Let us also mention egress rules

- SecurityGroupEgress: - IpProtocol: -1, CidrIp: 0.0.0.0/0 = allow all outbound.
- Security groups are stateful, so responses to allowed inbound are automatically allowed out; however:
    - Your instances need to initiate outbound connections (e.g., dnf update, pulling Docker images) – important!

    - That requires an egress allow. The default is “allow all egress”; we set it explicitly for clarity. Technically it is not needed but this shows intent
- Keep it as-is for this public-subnet setup. (When/if you move to private subnets + NAT, the rule still makes sense—traffic exits via NAT. I.e good habit to declare it)

3.4 CF-Summary

Hopefully we have successfully converted our manual deploy to CloudFormation templates now. I explicitly tried not to paste alot of code in here so you (the reader) would just copy-paste, I try to explain what the code does and what. That is a bit hard to do with the CloudFormation templates, so I just assume you know the CF and AWS basics.

A bit short explanation would probably be that CF declare parameters (like the ingredients if we are reading a cooking recipe) and then we specify the instructions how to cook (resources, etc). Finally we share that information through the Outputs section.

4. Deploying our CF and validating it

We keep this fully IaC + CLI-driven: validate locally, upload the child templates to S3, then create/update/delete the single root stack.

4.1 Tiny quality of life life-hacks

We will do two sections below, the second is mandatory, the first is optional: but it makes life so much better that it will be worth setting that too.

4.1.1 Optional, tiny .env for copy-pasting shorter CLI

This .env ends up in the /infra/ directory, here it will be ignored by .gitignore. Create that .env file and add this code to it:

```
# .env (example)
export REGION="eu-west-1"
export BUCKET="cf-swarm-<user-id-here>-eu-west-1"
export PREFIX="swarm-iac/templates/"
export STACK="swarm-cf-root"
export PARAMS="infra/parameters/dev.params.json"
```

To use (export) those values in your shell session, type the following in Bash (and make sure you are on the same level as the .env file, I keep mine in the Git-root as that is usually my default):

```
source .env
```

If you want to verify that this worked (not a bad idea), run this:

```
for v in REGION BUCKET PREFIX STACK PARAMS; do echo "$v=${!v}"; done
```

It will print out the stored parameter values in your console so you can see if they are correct.

4.1.2 Create dev.params.json

Our json template needs to be in an array form for CloudFormation to be able to read it. I handled this by creating two new files: dev.params.json and dev.example.json. The example-version show what it might look like. While the params-version is the real deal, gitignored. While at it, delete the dev.json template we had as it was replaced by the above versions.

It is time to set the real values to dev.params.json so we can use those in the system. For obvious reasons I cannot post my personal information here, just replace the data with your own:

- StackNamePrefix — it is there to add a prefix to names we create from templates
- TemplateBucket/TemplatePrefix — where child templates live in S3: path routes
- VpcId/PublicSubnetId — we're using your existing (default) VPC; just point to one public subnet
- AllowedSshCidr — your IP in /32 CIDR for SSH (port 22) to the instances
- KeyPairName — existing EC2 key pair (no .pem)
- InstanceType — size for the EC2's
- AmiId — the AMI to boot (Amazon Linux 2023 in this case), you can find this when you browse the AMI alternatives (but for simplicity: use ami-097f734cebd08c39e)

This below is the example file, some data is correct but replace the rest:

```
[
  { "ParameterKey": "StackNamePrefix", "ParameterValue": "swarm-cf" },
  { "ParameterKey": "TemplateBucket",  "ParameterValue": "cf-artifacts-<account-
id>-eu-north-1" },
  { "ParameterKey": "TemplatePrefix",  "ParameterValue": "swarm-iac/templates/" },
  { "ParameterKey": "VpcId",           "ParameterValue": "vpc-xxxxxxxx" },
  { "ParameterKey": "PublicSubnetId",  "ParameterValue": "subnet-xxxxxxxx" },
  { "ParameterKey": "AllowedSshCidr",  "ParameterValue": "1.2.3.4/32" },
  { "ParameterKey": "KeyPairName",     "ParameterValue": "your-keypair" },
  { "ParameterKey": "InstanceType",    "ParameterValue": "t3.small" },
  { "ParameterKey": "AmiId",           "ParameterValue": "ami-097f734cebd08c39e" }
]
```

4.1.2.1 Important note, do not forget to set values

At this point, I did the rookie mistake by forgetting to set the values for the templates I gave you above. I forgot to replace the xxxxx after vpc and subnet with their actual values.

4.1.2.2 Where to find the VPC ID

VPC can be found at AWS Console > VPC > Your VPCs – just copy the ID and replace in the file.

4.1.2.3 Where to find the Subnet ID and which to pick and why

Subnets can be found in the menu just under "Your VPCs", the step we just visited. You will see three default Subnets, so to know which to pick you need to understand Subnet basics. You see this:

- Default VPC CIDR: 172.31.0.0/16 (a /16 = 65,536 IPs).
- Default subnets (one per AZ) like:
    - 172.31.0.0/20

- 172.31.16.0/20
- 172.31.32.0/20

A /20 is 4,096 IPs. It's a slice of the /16. The "16" and "32" offsets are just the next /20 blocks (each /20 advances by 16 in the third octet for 172.31.x.0). In the default VPC, these default subnets are public (they auto-assign public IPs and have a route to the Internet Gateway).

What to pick now?

- Pick any one default subnet in that VPC where MapPublicIpOnLaunch = true. All three are usually public in the default VPC—choose one.
- Why one subnet is fine (for now)
- Our minimal template places all 3 instances in one public subnet for simplicity. We'll spread across AZs later when we add ALB/ASG, etc.

A summary would be that the Subnets will be useful when you need to extend IP ranges and create internal solutions. We have a rudimentary network setup, so we don't need that type of internal communication, we are just solving it by opening ports and using self-reference for the Swarm.

4.1.2.4 Using CLI to solve the above

I am not going in-depths on these commands, what they do is to search for the information you were looking for manually in the steps above:

```
# 1) Default VPC in your region
aws ec2 describe-vpcs \
  --region "$REGION" \
  --filters Name=isDefault,Values=true \
  --query 'Vpcs[0].VpcId' --output text

# 2) List subnets in that VPC; pick any with PublicIpOnLaunch = true
VPC_ID="vpc-xxxxxxxx"  # <- paste from the previous command
aws ec2 describe-subnets \
  --region "$REGION" \
  --filters Name=vpc-id,Values=$VPC_ID \
  --query
'Subnets[].{SubnetId:SubnetId,Az:AvailabilityZone,Cidr:CidrBlock,PublicIpOnLau
nch:MapPublicIpOnLaunch}' \
  --output table
```

4.2 Validate template (syntax check)

- You've created the artifacts S3 bucket and given your user S3 permissions (section 3.1)
- AWS CLI is configured for the target account
- Your three templates exist:
  1. infra/templates/00-sg-swarm.yaml
  2. infra/templates/10-ec2-swarm.yaml
  3. infra/templates/root.yaml

So now we will bring forward one of our strongest weapons: a simple syntax validation check. We will run this after we make updates to our CloudFormation YAMLs, it will save us time and headache:

```
aws cloudformation validate-template --template-body file://infra/templates/00-sg-swarm.yaml
aws cloudformation validate-template --template-body file://infra/templates/10-ec2-swarm.yaml
aws cloudformation validate-template --template-body file://infra/templates/root.yaml
```

Pardon the small font, I did not want them to be cut off, it is worth listing them on a line like that. My point is this: you will want to copy paste these babies frequently. I will link to this section.

Be mindful that these validate-template commands check syntax, it does not fetch child TemplateURLs. This is why we have to do individual validations, otherwise we could have used root.

4.3 Upload the templates to S3

This is another command you will run frequently. Upload the files to S3 – child firsts, then root.

```
aws s3 cp infra/templates/00-sg-swarm.yaml   "s3://${BUCKET}/${PREFIX}00-sg-swarm.yaml"
aws s3 cp infra/templates/10-ec2-swarm.yaml  "s3://${BUCKET}/${PREFIX}10-ec2-swarm.yaml"
aws s3 cp infra/templates/root.yaml          "s3://${BUCKET}/${PREFIX}root.yaml"
```

What this does: cp copies the file from path/filename.yaml and then it ships it to s3://the-bucket-variable-we-defined/(adds-the-prefix-to-our-filename)filename.yaml. root will have the templateURL inside S3 for the child templates, this is why it is important to upload the childs first so you have the latest version pulled when you run update-stack on root.

4.4 Create the stack (daily bring-up of the stacks)

This is a command you will run everytime you want to bring up the (most recently uploaded) stacks on AWS. For these school/non-live versions I would suggest you tear down the project daily and then you re-deploy it with this command on a daily basis. Since a new session most likely will occur in your shell (Bash) when you start up, remember to run the export command first from 4.1.1. Otherwise you will need to adjust these commands too since we parametrized values! Then run this:

```
aws cloudformation create-stack \
  --region "$REGION" \
  --stack-name "$STACK" \
  --template-url
"https://s3.${REGION}.amazonaws.com/${BUCKET}/${PREFIX}root.yaml" \
  --parameters file://"$PARAMS" \
  --capabilities CAPABILITY_IAM CAPABILITY_NAMED_IAM

aws cloudformation wait stack-create-complete \
  --region "$REGION" --stack-name "$STACK"
```

Create stack in the region and stack-name you predetermined. Template-url will be the path given, parameters are taken from $PARAMS, capabilities IAM. Maybe I used this command too many times,

if you feel it needs further explanation, please reach out to me, it seems rather straightforward to me.

Verfication: you should get a return prompt that gives you the Stack ID.

4.4.1 Grab outputs (IPs, SG id)

Run this command to get a table printout of the values – since you do get a new IP and SG-ID daily:

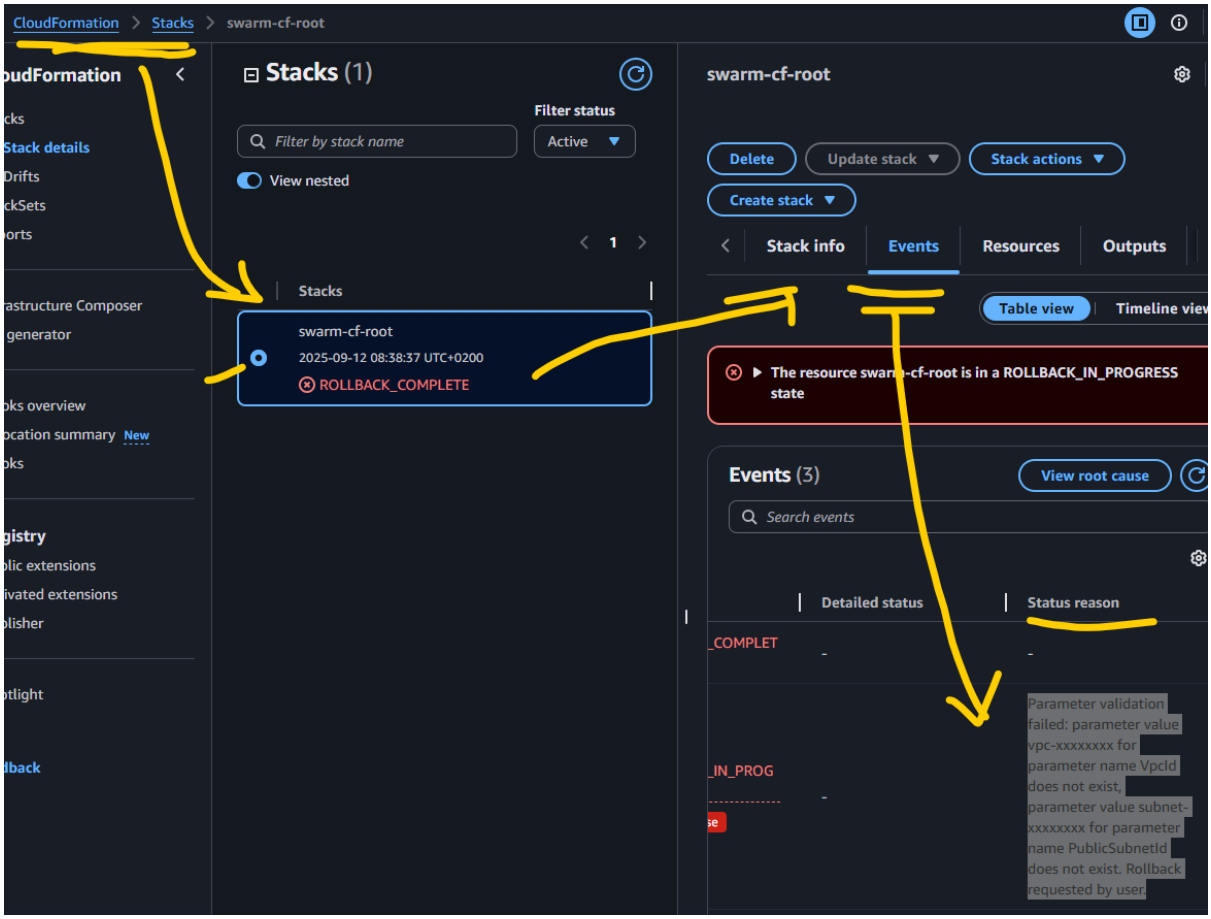
```
aws cloudformation describe-stacks \
  --region "$REGION" --stack-name "$STACK" \
  --query 'Stacks[0].Outputs[].[OutputKey,OutputValue]' --output table
```

I go more in-depth on this topic at section 4.7.

4.4.2 Error handling when creating

Sometimes you do something naughty, and you will get slapped on the fingers. The good news is the system is good at telling you what you did wrong. Logs are your best friends. One such example could be that you forget to replace the “vpc-xxxxxxxxxx” parameter in dev.params.json, like I did. But you can find the error easily.

When you upload your files (and this is why I put this error handling under section 4.4), you will get a “ROLLBACK_COMPLETE” or similar error in your CloudFormation console at AWS. If you click your stack there, and check the Events tab, it will provide information about what you did wrong. Like this:



In this case the solution was easy: just apply the proper information in dev.params.json, re-upload the root.yaml and re-run stack creation/update (depending where you are in your process).

4.5 Update the stack (apply template changes)

Whenever you have updated templates, validate, re-upload it to S3, then run update-stack like this:

```
aws cloudformation update-stack \
  --region "$REGION" \
  --stack-name "$STACK" \
  --template-url
"https://s3.${REGION}.amazonaws.com/${BUCKET}/${PREFIX}root.yaml" \
  --parameters file://"$PARAMS" \
  --capabilities CAPABILITY_IAM CAPABILITY_NAMED_IAM

aws cloudformation wait stack-update-complete \
  --region "$REGION" --stack-name "$STACK"
```

It is a similar version to the create-stack, but instead it updates the stack. Remember: root last!

4.6 Tear-down of the stacks (daily or frequent routine)

When you are done working and take a break, tear down the stack(s) (you tear it down using the STACK variable which is a mutual name for the primary stack that root commands, thus all child templates go down with it) by using this command:

```
aws cloudformation delete-stack \
  --region "$REGION" \
  --stack-name "$STACK"

aws cloudformation wait stack-delete-complete \
  --region "$REGION" --stack-name "$STACK"
```

4.7 Optional: Check the public IPs of the Instances

We will need the public IP for the temporary step in 4.7.1, but it also nice to know it if we would need to SSH in or whatever. Here is how you do that:

```
aws cloudformation describe-stacks \
  --region "$REGION" --stack-name "$STACK" \
  --query 'Stacks[0].Outputs[].[OutputKey,OutputValue]' --output table
```

```
$ aws cloudformation describe-stacks \
  --region "$REGION" --stack-name "$STACK" \
  --query 'Stacks[0].Outputs[].[OutputKey,OutputValue]' --output table
-----------------------------------------
|             DescribeStacks            |
+------------------+--------------------+
|  Worker1PublicIp |  108.130.168.64    |
|  Worker2PublicIp |  63.34.171.184     |
|  SecurityGroupId |  sg-02167463e11a80d6c  |
|  ManagerPublicIp |  54.78.33.150      |
+------------------+--------------------+
```

Quite nice! And useful. 😊

4.7.1 Bonus: Capture the three public IPs into variables

Oh snap now we are into bonus-land-deluxe, but developers are supposed to automate..

```
MANAGER_PUB=$(aws cloudformation describe-stacks --region "$REGION" --stack-name "$STACK" \
  --query "Stacks[0].Outputs[?OutputKey=='ManagerPublicIp'].OutputValue" --output text)
WORKER1_PUB=$(aws cloudformation describe-stacks --region "$REGION" --stack-name "$STACK" \
  --query "Stacks[0].Outputs[?OutputKey=='Worker1PublicIp'].OutputValue" --output text)
WORKER2_PUB=$(aws cloudformation describe-stacks --region "$REGION" --stack-name "$STACK" \
  --query "Stacks[0].Outputs[?OutputKey=='Worker2PublicIp'].OutputValue" --output text)

echo "Manager: $MANAGER_PUB"
echo "Worker1: $WORKER1_PUB"
echo "Worker2: $WORKER2_PUB"
```

This grabs the three public IPs and stores them into the variables. See 4.7.2 for :usefulness:

4.7.2 Temporary: Until we automate deploy, do it manually

1. SSH to Manager > init swarm. See section 2.3.2 and follow those steps.
2. SSH to each worker, same link as above.
3. Deploy the stack from the Manager, same as above too.

If you were a geek and did step 4.7.1 for the bonus-capture you can do like this:

```
# Manager
ssh -i ~/.ssh/tinfoil-eu-west-1.pem ec2-user@$MANAGER_PUB
sudo docker swarm init --advertise-addr "$MANAGER_PRIV"
# Copy the printed join command
# Type “exit” to leave this node
```

```
# Worker 1
ssh -i ~/.ssh/tinfoil-eu-west-1.pem ec2-user@$WORKER1_PUB
sudo docker swarm join --token SWMTKN-1-... "$MANAGER_PRIV:2377"
# Obviously, replace the ... with the actual join command
# exit

# Worker 2
ssh -i ~/.ssh/tinfoil-eu-west-1.pem ec2-user@$WORKER2_PUB
sudo docker swarm join --token SWMTKN-1-... "$MANAGER_PRIV:2377"
```

That is pretty cool. And lazy. Which we like. No more copy/pasting or writing down IPs.

4.7.3 Deploying the Docker Compose (temporary until automation)

We keep this step manual as in step 2.4.1 and 2.4.2. SSH into the Manager, create the compose file (as in 2.4.1), deploy it as in 2.4.2.

4.8 Validation

There are all kinds of ways to validate, I believe in validating small incremental steps as you build, if you validate too much at this step then there is a decent risk that you have alot of issues created along the way that is alot more messy to solve now. But, for the sake of validation, I will show you a number of tools we have at our disposal here:

4.8.1 Checking the IPs of the Swarm

Run this command in Bash:

```
MANAGER_PUB=$(aws cloudformation describe-stacks --region "$REGION" --stack-
name "$STACK" \
  --query "Stacks[0].Outputs[?OutputKey=='ManagerPublicIp'].OutputValue" --
output text)
WORKER1_PUB=$(aws cloudformation describe-stacks --region "$REGION" --stack-
name "$STACK" \
  --query "Stacks[0].Outputs[?OutputKey=='Worker1PublicIp'].OutputValue" --
output text)
WORKER2_PUB=$(aws cloudformation describe-stacks --region "$REGION" --stack-
name "$STACK" \
  --query "Stacks[0].Outputs[?OutputKey=='Worker2PublicIp'].OutputValue" --
output text)

aws ec2 describe-instances --region "$REGION" \
  --filters "Name=ip-
address,Values=${MANAGER_PUB},${WORKER1_PUB},${WORKER2_PUB}" \
  --query
'Reservations[].Instances[].[Tags[?Key==`Name`]|[0].Value,PublicIpAddress,Priv
ateIpAddress]' --output table
```

You will get an output that looks like this:

```
aws ec2 describe-instances --region "$REGION" \
  --filters "Name=ip-address,Values=${MANAGER_PUB},${WORKER1_PUB},${WORKER2_PUB}" \
  --query 'Reservations[].Instances[].[Tags[?Key==`Name`]|[0].Value,PublicIpAddress,PrivateIpAddress]' --output table
-------------------------------------------------------
|                  DescribeInstances                  |
+------------------+-----------------+----------------+
|  swarm-cf-worker-1|  108.130.168.64  |  172.31.10.241  |
|  swarm-cf-manager |  54.78.33.150    |  172.31.2.233   |
|  swarm-cf-worker-2|  63.34.171.184   |  172.31.13.18   |
+------------------+-----------------+----------------+
```

To the left is your Swarm nodes and their names. In the middle you have the public IP, the one you can visit to check these instances in a browser. To the right is their private IP. Notice how they correlate with the Subnet you set earlier.

4.8.2 Swarm health and on-instance checks

Like we did in 2.4.3. Since you are doing those commands on the manager, you can also do on-instance checks from the manager. This is overkill. But hey. Better be safe than sorry? 😊

```
sudo ss -tulpn | egrep ':80|:8080'   # ports open
sudo docker logs $(sudo docker ps -q --filter name=myappname_web -n 1) --
tail=50
```

4.8.3 Web reachability

I think you should do this two ways. First through shell / Bash:

```
# Nginx on any node (HTTP 200 OK)
curl -I http://$MANAGER_PUB
curl -I http://$WORKER1_PUB
curl -I http://$WORKER2_PUB
```

This gives you more detailed info about the state of the machines, but what you want to see primarily is a HTTP/1.1 200 OK at the start. It is nice to see the Nginx server and the Content-type: text-html there too, this means your html page is displaying properly.

Then as step two just visit the public IPs in a browser. Don't forget to test the manager's public IP with port :8080 and see if you can see the Visualizer, it should display, then you know the SGs port setting for the Visualizer is correctly set up too.

4.9 CF-Summary

This section is rather big, but at the same time it is not that many steps, much copy-paste. If everything goes well it should work immediately, if not, I recommend going through them one step at a time and trying to figure out if every value is set correct. In many of my copy-paste examples there are variables that needs to be replaced by your actual values. Be mindful!

- You validated YAML locally, uploaded children to S3, and created a single root stack that built the SG and EC2 layers in order.
- Outputs gave you public IPs; you derived private IPs for advertise-addr.
- You manually initialized/joined the swarm and deployed the same compose from 2.4.

- Result: identical outcome to the console walkthrough, but now fully reproducible via IaC + one root stack (create/update/delete).

When you are done, you have successfully converted the manual guide to Cloud Formation!

5. Introducing DynamoDB and reviewing IAM policies

Now we step out of the template-boundary. only real decision we did up until now, was to convert the manual setup to a Nested Stacks CloudFormation setup.

5.1 Design considerations

DynamoDB On-Demand (Pay per request) + strong perimeter has pros: zero capacity planning, autoscales up and down, but nasty cons: cost is proportional to traffic. A bot storm on our forms field could rack up charges! Cost-control and abuse protection is a must!

IMDSv2 – Instance MetaData Service v2 for EC2 is a local, hardened HTTP endpoint on each instance. That means each instance is attached to an IAM Role (Instance profile), with policies. Pros here is we need no hard-coded keys, credentials rotate automatically, and every Swarm node inherits the same least-privlege access. Sounds neat?

IAM strategy – Least privilege is a must, and it must function in a scalable environment. The idea is to attach one EC2 instance profile to all swarm nodes. IMDSv2 > STS (short lived credentials) > AWS SDK in the .NET app? No static keys is a win.

5.2 We have to begin somewhere. Begin the beginner

Plan: Attach an EC2 Role and enforce IMDSv2.

5.2.1 New child template: EC2 role + instance profile

Create infra/templates/20-iam-ec2-role.yaml

```yaml
AWSTemplateFormatVersion: '2010-09-09'
Description: EC2 Instance Role + Instance Profile

Parameters:
  StackNamePrefix:
    Type: String

Resources:
  AppEc2Role:
    Type: AWS::IAM::Role
    Properties:
      RoleName: !Sub '${StackNamePrefix}-ec2-app-role'
      AssumeRolePolicyDocument:
        Version: '2012-10-17'
        Statement:
          - Effect: Allow
```

```
            Principal: { Service: ec2.amazonaws.com }
            Action: sts:AssumeRole

  AppEc2InstanceProfile:
    Type: AWS::IAM::InstanceProfile
    Properties:
      InstanceProfileName: !Sub '${StackNamePrefix}-ec2-app-profile'
      Roles: [ !Ref AppEc2Role ]

Outputs:
  InstanceProfileName:
    Value: !Ref AppEc2InstanceProfile
```

Validate. Always validate.

```
aws cloudformation validate-template --template-body file://infra/templates/20-iam-ec2-role.yaml
```

5.2.2 Attach the profile + enforce IMDSv2 on our existing EC2s

We need to modify our templates to attach the profile. Edit 10-ec2-swar.yaml:

Under “Parameters” (probably line 4) you have six parameters. Add this after them:

```
  InstanceProfileName:
    Type: String
```

On each of the instances (under the Resources subsection, Manager should be on line 23, Worker1 and 2 on line 43 and 63 respectively), add these two properties:

```
      IamInstanceProfile: !Ref InstanceProfileName
      MetadataOptions:
        HttpEndpoint: enabled
        HttpTokens: required   # IMDSv2 required
```

It should look something like this:

```
16      Type: AWS::EC2::Image::Id
17    InstanceProfileName:
18      Type: String
```

```
30      SecurityGroupIds: [ !Ref SecurityGroupId ]
31      IamInstanceProfile: !Ref InstanceProfileName
32      MetadataOptions:
33        HttpEndpoint: enabled
34        HttpTokens: required   # IMDSv2 required
35      Tags:
```

Guess what comes next? You’re right. Validate!

```
aws cloudformation validate-template --template-body file://infra/templates/10-ec2-swarm.yaml
```

5.2.3 Orchestrate this in root.yaml

First, we need to add the new 20-template to the nested stacks in the root template. We do this by opening root, scrolling down above the SwarmEc2Stack and add this code around line 45:

```
  IamEc2RoleStack:
    Type: AWS::CloudFormation::Stack
    Properties:
```

```
      TemplateURL: !Sub
'https://s3.${AWS::Region}.amazonaws.com/${TemplateBucket}/${TemplatePrefix}20
-iam-ec2-role.yaml'
      Parameters:
        StackNamePrefix: !Ref StackNamePrefix
```

We need to pass the profile to the EC2 stack (the SwarmEc2Stack in root). To do so we have to update some parameters, notably we want to add ", IamEc2RoleStack" in the DependsOn line, and then add a line at the end where we give the InstanceProfileName-parameter:

```
    # Update this line
    DependsOn: [ SwarmSecurityGroupStack, IamEc2RoleStack ]
        # Add this line to the end of SwarmEc2Stack
        InstanceProfileName: !GetAtt
IamEc2RoleStack.Outputs.InstanceProfileName
```

Validate. No really, just do it. You know you want to.

```
aws cloudformation validate-template --template-body file://infra/templates/root.yaml
```

5.2.4 Upload and update the root stack

This is a bit different from the earlier uploads and updates, now we have modified our EC2 Instances and attached a profile, so we run this first:

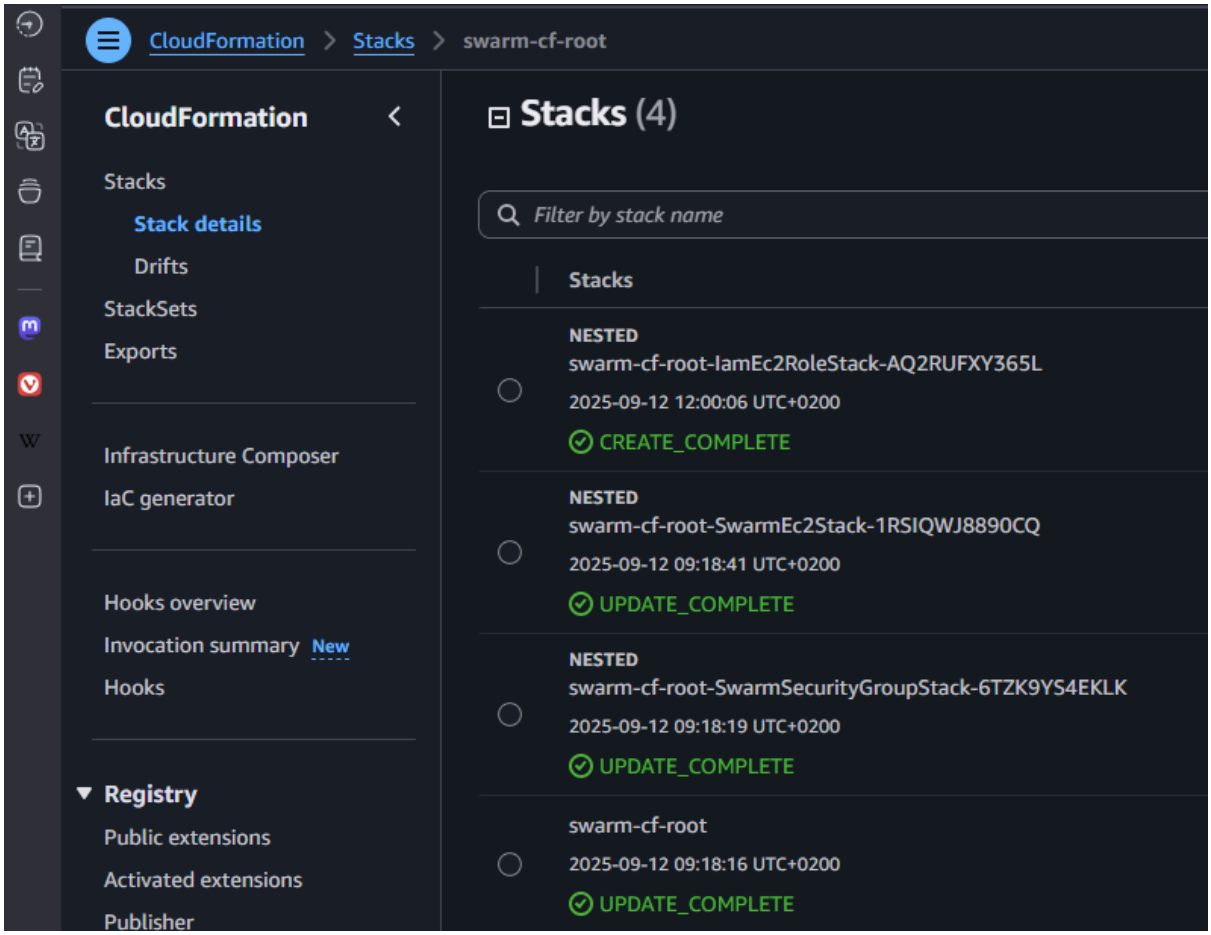
```
# Upload files to the S3 bucket
aws s3 cp infra/templates/20-iam-ec2-role.yaml "s3://$BUCKET/${PREFIX}20-iam-ec2-role.yaml" --region
"$REGION"
aws s3 cp infra/templates/10-ec2-swarm.yaml    "s3://$BUCKET/${PREFIX}10-ec2-swarm.yaml"    --region
"$REGION"
aws s3 cp infra/templates/root.yaml            "s3://$BUCKET/${PREFIX}root.yaml"            --region
"$REGION"
# Update root, instances will be replaced to attach the profile
aws cloudformation update-stack \
  --region "$REGION" --stack-name "$STACK" \
  --template-url "https://s3.${REGION}.amazonaws.com/${BUCKET}/${PREFIX}root.yaml" \
  --parameters file://"$PARAMS" \
  --capabilities CAPABILITY_IAM CAPABILITY_NAMED_IAM
```

This might lead to a change of public IPs, because we replace the EC2s. Re-pull outputs:

```
aws cloudformation describe-stacks --region "$REGION" --stack-name "$STACK" \
  --query 'Stacks[0].Outputs[].[OutputKey,OutputValue]' --output table
```

In my case I did get new IPs when doing this. I re-ran the commands from 4.7.1 to store the new IPs in the variables.

I always keep a window open on AWS Console > CloudFormation > Stacks > stackname (swarm-cf-root in this case) so I can see live if the Swarms are healthy:



5.2.5 Verify IMDSv2 + role (on the Manager)

Let us start by SSHing into the Manager:

```
# SSH to the Manager

ssh -i ~/.ssh/tinfoil-eu-west-1.pem ec2-user@$MANAGER_PUB
# On the Manager, show the HTTP status
curl -o /dev/null -s -w "%{http_code}\n" http://169.254.169.254/latest/meta-data/iam/info
```

This should return a 401:

```
[ec2-user@ip-172-31-2-233 ~]$ curl -o /dev/null -s -w "%{http_code}\n" http://169.254.169.254/latest/meta-data/iam/info
401
```

401 means “unauthorized access”. We now need a token to access this information! Let us try get a token and then see if we can access the information:

```
# Grab a token
TOKEN=$(curl -sX PUT "http://169.254.169.254/latest/api/token" \
  -H "X-aws-ec2-metadata-token-ttl-seconds: 21600")

# Use the token see if you can access information now
```

```
curl -s -H "X-aws-ec2-metadata-token: $TOKEN" \
  http://169.254.169.254/latest/meta-data/iam/security-credentials/
```

That is a solid response, mine returned swarm-cf-ec2-app-role-etc! Exit out of the Manager. You can now verify by checking Instance descriptions, they should say HttpTokens: Required. Try:

```
aws ec2 describe-instances --region "$REGION" --instance-ids "$MANAGER_ID" \
  --query 'Reservations[0].Instances[0].MetadataOptions'
```

If you want you can also check if an instance profile is attached:

```
aws ec2 describe-instances --region "$REGION" --instance-ids "$MANAGER_ID" \
  --query 'Reservations[0].Instances[0].IamInstanceProfile'
```

5.2.6 Summary and an explanation of what we just did

Wow, that is alot of code spammage. We were not supposed to do that here, you promised you would explain what we are doing? Sorry, I was in a flow! Yes, you are right, so let us just do that. Normally I kept these summaries brief. This however, is a massive adaptation, so let us review:

What we added

- An IAM Role for EC2 + an Instance Profile (the “holder” that lets EC2 attach that role)
- We told all three swarm instances (manager + 2 workers) to use that profile (crucial!)
- We enforced IMDSv2 on the instances (HttpTokens: required)

How this works

- Think of the IAM Role as a keycard with permissions
- The Instance Profile is how that keycard is physically attached to each EC2 instance
- Inside each instance there is a tiny, local HTTP endpoint called IMDS (Instance Metadata Service)
- With IMDSv2, you must first ask for a short-lived token, then present that token to read metadata (like the role name) or to let the AWS SDK fetch temporary credentials for the role
- Your app (and AWS SDKs) automatically use IMDSv2 behind the scenes. You don’t code the token dance yourself—SDKs handle that. (We only used curl to prove it’s enforced)

Why we did it

- No hard-coded keys: the app will get short-lived credentials from the role. Nothing to store in code or .env
- Least privilege ready: the role currently has *no* data permissions. In the next step we’ll add just enough access to DynamoDB
- Secure by default: IMDSv2 blocks casual metadata access (and helps mitigate SSRF-style issues). Without the token, access is denied

What changed operationally

- Updating the template to attach the profile replaced the EC2 instances, so public IPs may have changed—we re-pulled outputs
- SSH still respects your AllowedSshCidr; if your VPN IP changes, update that parameter and update-stack

- From now on, any container scheduled by Swarm on any node has the same identity (the same role)—great for horizontal scaling

5.3 Let us introduce DynamoDB

The goal is here to keep it simple but viable: one table the app can user later, no IAM changes yet, we are saving that for 5.4.

5.3.1 Starting off by adding a DynamoDB child template

Create infra/templates/30-dynamodb.yaml, like this:

```yaml
AWSTemplateFormatVersion: '2010-09-09'
Description: DynamoDB table (On-Demand) for form submissions

Parameters:
  StackNamePrefix:
    Type: String
  TableName:
    Type: String
    Default: swarmcf-Submissions

Resources:
  SubmissionsTable:
    Type: AWS::DynamoDB::Table
    Properties:
      TableName: !Ref TableName
      BillingMode: PAY_PER_REQUEST
      AttributeDefinitions:
        - AttributeName: pk
          AttributeType: S
        - AttributeName: sk
          AttributeType: S
      KeySchema:
        - AttributeName: pk
          KeyType: HASH
        - AttributeName: sk
          KeyType: RANGE
      Tags:
        - Key: Name
          Value: !Sub '${StackNamePrefix}-submissions'

Outputs:
  TableName:
    Value: !Ref SubmissionsTable
```

Have you ever heard me say this before? “Validate”?

```
aws cloudformation validate-template --template-body file://infra/templates/30-dynamodb.yaml
```

5.3.1.1 Let us explain a few parameters here:

Most is human-readable here. Prefix, Tablename, BillingMode, Tags, we know or can guess what that means. However, what are those letter-attributes on the keys? That is something we could look at:

pk (partition) spreads data; sk (sort) orders within that group. Both defined as strings (S) here.
They are both tools to help sort data, to explain in a super-simple way. We will want good means to distribute and to fetch data from DynamoDB when we read/write.

5.3.2 Wire that bad boy into root.yaml

You are getting pro at this, no?! We will just add a parameter into root, like we did last section. Add this below the AmiId under the Parameters section:

```
  TableName:
    Type: String
    Default: swarmcf-Submissions
```

We will also add the nested stack below the IAM role stack (IamEc2RoleStack):

```
  DynamoDbStack:
    Type: AWS::CloudFormation::Stack
    Properties:
      TemplateURL: !Sub
'https://s3.${AWS::Region}.amazonaws.com/${TemplateBucket}/${TemplatePrefix}30
-dynamodb.yaml'
      Parameters:
        StackNamePrefix: !Ref StackNamePrefix
        TableName: !Ref TableName
```

We can also expose the name in Outputs (at the bottom):

```
  DynamoTableName:
    Value: !GetAtt DynamoDbStack.Outputs.TableName
```

What is the meaning of this? Well, Outputs means information we share within the Stack.

Validate!

```
aws cloudformation validate-template --template-body file://infra/templates/root.yaml
```

5.3.2.1 Update parameters file

We need to add a TableName to the parameter file (dev.params.json):

```
{ "ParameterKey": "TableName", "ParameterValue": "swarmcf-Submissions" }
```

5.3.3 Upload and update the Swaaarms (we are all Swarm)

I like to work incremental, small baby steps, so you recognize the routine now. We will upload the new 30-dynamodb.yaml, root.yaml (always), then run update-stack, wait for complete:

```
aws s3 cp infra/templates/30-dynamodb.yaml "s3://$BUCKET/${PREFIX}30-
dynamodb.yaml" --region "$REGION"
```

```
aws s3 cp
infra/templates/root.yaml        "s3://$BUCKET/${PREFIX}root.yaml"        --
region "$REGION"

aws cloudformation update-stack \
  --region "$REGION" --stack-name "$STACK" \
  --template-url
"https://s3.${REGION}.amazonaws.com/${BUCKET}/${PREFIX}root.yaml" \
  --parameters file://"$PARAMS" \
  --capabilities CAPABILITY_IAM CAPABILITY_NAMED_IAM

aws cloudformation wait stack-update-complete --region "$REGION" --stack-name
"$STACK"
```

5.3.4 Verify DynamoDB – there is no app yet but we can test it

We will check root outputs: it should now include DynamoTableName. Let us have a look:

```
aws cloudformation describe-stacks --region "$REGION" --stack-name "$STACK" \
  --query 'Stacks[0].Outputs[].[OutputKey,OutputValue]' --output table
```

```
$ aws cloudformation describe-stacks --region "$REGION" --stack-name "$STACK" \
    --query 'Stacks[0].Outputs[].[OutputKey,OutputValue]' --output table
  ------------------------------------------
  |             DescribeStacks             |
  +-----------------+----------------------+
  |  Worker1PublicIp |  108.130.168.64     |
  |  Worker2PublicIp |  63.34.171.184      |
  |  DynamoTableName |  swarmcf-Submissions |
  |  SecurityGroupId |  sg-02167463e11a80d6c |
  |  ManagerPublicIp |  54.78.33.150       |
  +------------------+----------------------+
```

Success! There is also the option to check if the table exist and also list tables:

```
# Check the table exists
aws dynamodb describe-table --region "$REGION" --table-name swarmcf-
Submissions --output table

# (Optional) list tables
aws dynamodb list-tables --region "$REGION" --output table
```

5.3.5 DynamoTemplate-Summary

We deploy the new template and verify its existence. Questions? 😊

5.4 Minimal IAM for DynamoDB (least privilege)

We will focus on giving the EC2 role a tiny DynomoDB right.

5.4.1 Update the IAM child

Add TableName + inline policy. Edit 20-iam-ec2-role.yaml. Add this after StackNamePrefix in the Parameters section:

```
  TableName:
    Type: String
```

Then under Resources > AppEc2Role > Properties – add Policies at the end of that segment:

```
      Policies:
        - PolicyName: !Sub '${StackNamePrefix}-dynamodb-access'
          PolicyDocument:
            Version: '2012-10-17'
            Statement:
              - Effect: Allow
                Action:
                  - dynamodb:DescribeTable
                  - dynamodb:PutItem
                  - dynamodb:GetItem
                  - dynamodb:UpdateItem
                  - dynamodb:Query
                  - dynamodb:Scan
                  - dynamodb:DeleteItem   # Optional; enable only if you want
deletes
                Resource:
                  - !Sub
'arn:aws:dynamodb:${AWS::Region}:${AWS::AccountId}:table/${TableName}'
                  - !Sub
'arn:aws:dynamodb:${AWS::Region}:${AWS::AccountId}:table/${TableName}/index/*'
```

Should look a bit like this:

```yaml
Resources:
  AppEc2Role:
    Type: AWS::IAM::Role
    Properties:
      RoleName: !Sub '${StackNamePrefix}-ec2-app-role'
      AssumeRolePolicyDocument:
        Version: '2012-10-17'
        Statement:
          - Effect: Allow
            Principal: { Service: ec2.amazonaws.com }
            Action: sts:AssumeRole
      Policies:
        - PolicyName: !Sub '${StackNamePrefix}-dynamodb-access'
          PolicyDocument:
            Version: '2012-10-17'
            Statement:
              - Effect: Allow
                Action:
                  - dynamodb:DescribeTable
                  - dynamodb:PutItem
                  - dynamodb:GetItem
                  - dynamodb:UpdateItem
                  - dynamodb:Query
                  - dynamodb:Scan
                  - dynamodb:DeleteItem   # Optional; enable only if you want deletes
                Resource:
                  - !Sub 'arn:aws:dynamodb:${AWS::Region}:${AWS::AccountId}:table/${TableName}'
                  - !Sub 'arn:aws:dynamodb:${AWS::Region}:${AWS::AccountId}:table/${TableName}/index/*'
```

5.4.1.1 Brief explanation of the above

Note what we did above: we added actions (Describe, Put, Get, Update, Query, Scan, Delete). The resource links to our Table. CloudFormation is rather easy to read in plain text as long as we do not use abbreviations and know their services.

Now we just need to do the following:

1. Validate

```
aws cloudformation validate-template --template-body file://infra/templates/20-iam-ec2-role.yaml
```

2. Do the following:

5.4.2 Pass TableName from root > IAM child

Under root.yaml > Resources > IamEc2RoleStack > Parameters, add this at the end:

```yaml
        TableName: !Ref TableName
```

Remember, root is the orchestrator, it is the spider in the web. In this case, we added the TableName earlier but it needs to be passed to the IamEc2RoleStack from root.

Validate root too just for the fun of it (or wait, you should always validate..):
`aws cloudformation validate-template --template-body file://infra/templates/root.yaml`

5.4.3 Lightweight verification to confirm the table exists

Run this to see if you can see the table:

```
aws dynamodb describe-table --region "$REGION" --table-name swarmcf-
Submissions --output table
```

If you see the table, you're set.

5.4.4 IAM-child-Summary

We gave the EC2 role just enough permission to talk to one DynamoDB table, passed that table name from the root, updated the stack (no instance replacement), and confirmed the role can call DynamoDB using IMDSv2-sourced, short-lived credentials.

6. Automate the docker setup

Basicly what we want to do now is to build a solution that automatically installs and initializes Docker on our Instances on startup. One challenge we have here is that we are taking down the setup on a daily basis so the IPs and restart them using a public VPC each time = no static IPs. We need a solution that can handle this installation and communication dynamically.

6.1 Why SSM Parameter Store (vs ALB) for automation

SSM (Parameter Store, for keys/values, there is also a Session Manager) and cloud-init (via User Data) is one such solution. Another could be to use an ALB (Application Load Balancer). SSM has no fee for starting a session, only optional things like storing session logs in CloudWatch/S3 or advanced features carry a cost. ALB on the other hand, is not free, it has a (small) hourly charge + usage.

ALB is great for stable DNS and loadbalancing, but for this project, we will be fine using SSM.

6.2 Add minimal permissions to the existing EC2 role

In 20-iam-ec2-role.yaml, Resources > Policies > add this -PolicyName section after the DynamoDB -PolicyName-section:

```
        - PolicyName: !Sub '${StackNamePrefix}-swarm-ssm-s3'
          PolicyDocument:
            Version: '2012-10-17'
            Statement:
              # Read/write the two parameters we use as rendezvous
              - Effect: Allow
                Action:
                  - ssm:GetParameter
                  - ssm:PutParameter
                Resource:
                  - !Sub
'arn:aws:ssm:${AWS::Region}:${AWS::AccountId}:parameter/swarm/worker-token'
                  - !Sub
'arn:aws:ssm:${AWS::Region}:${AWS::AccountId}:parameter/swarm/manager-ip'
              # Fetch compose file from your artifacts bucket
              - Effect: Allow
                Action:
                  - s3:GetObject
```

```
                Resource:
                  - !Sub
'arn:aws:s3:::${TemplateBucket}/${TemplatePrefix}artifacts/docker-stack.yml'
              # ListBucket is needed for some S3 clients (optional, safe to
include)
              - Effect: Allow
                Action: s3:ListBucket
                Resource: !Sub 'arn:aws:s3:::${TemplateBucket}'
```

Note that you are not removing anything, what we do here is that we add a "PolicyDocument" which sets Actions for certain Resources. So in plain English, this is what we allow:

We let the EC2 role write /swarm/worker-token and /swarm/manager-ip to SSM Parameter Store (the manager publishes the join token and its private IP), and let workers read them so they can auto-join the swarm without SSH. We also allow s3:GetObject for artifacts/docker-stack.yml so the manager can pull the compose file and run docker stack deploy during boot—no hand-edited files on the instance.

The s3:ListBucket on your bucket doesn't list *all* buckets; it permits listing/metadata of objects inside that one bucket, which some clients do before fetching a file (it's optional but harmless). Together, these permissions enable a true one-command deploy: create/update the stack and the cluster comes up, joins, and deploys itself.

Note: Your workstation might see "ParameterNotFound" if your IAM User lacks ssm:GetParameter on the parameters – even if they exist. Instances however, can read the (role has permissions).

6.2.1 Make sure the template also pass the parameters in

At the top of the 20-iam-ec2-role.yaml file, you have Parameters with StackNamePrefix and TableName. Add these two below the other ones:

```
  TemplateBucket:
    Type: String
  TemplatePrefix:
    Type: String
```

Repeat the same process with 10-ec2-swarm.yaml. It needs those parameters passed too. And while we are at it, add this to root.yaml > IamEc2RoleStack > Parameters. You had StackNamePrefix and TableName here too, so add the two new ones too:

```
        TemplateBucket:   !Ref TemplateBucket
        TemplatePrefix:   !Ref TemplatePrefix
```

Repeat that same process with the SwarmEc2Stack, the same two lines at the end as you did above.

And one final addition that is important, on each Worker, under "Type", add DependsOn:

```
  Worker1:
    Type: AWS::EC2::Instance
    DependsOn: Manager
```

When adding things like this in the .yaml files, be mindful of the indentation! If you are using Vs Code or a modern IDE it probably warn you, but this is also what is good with the syntax validation we do.

6.2.2 Upload the docker-compose

Remember, in section 2.4.1 we wrote the docker-stack file straight onto the manager, using the cat command. Now, we need to do the same thing but work with a file, so do this:

1. Create the file: infra > artifacts > docker-stack.yml (matches S3 path)
2. Add the code from 2.4.1 minus the “cat > docker-stack.yml << 'EOF' ... EOF” in the file
3. Upload the file to S3 using this command:

```
aws s3 cp infra/artifacts/docker-stack.yml
"s3://$BUCKET/${PREFIX}artifacts/docker-stack.yml" --region "$REGION"
```

6.3 Update the Mangers User Data, add init + publish + deploy

In 10-ec2-swarm.yaml under the Manager instance (should be the first Resources section, around line 38 you will) find UserData. We used to just run a few lines of code there, replace with this:

```
          #cloud-config
          runcmd:
            - dnf -y update
            - dnf -y install docker awscli
            - systemctl enable --now docker
            - usermod -aG docker ec2-user
            - 'echo "export AWS_REGION=${AWS::Region}; export
AWS_DEFAULT_REGION=${AWS::Region}" | tee -a /etc/profile.d/awsregion.sh'
            - |
              bash -lc '
                # wait briefly for Docker
                for i in {1..10}; do docker info >/dev/null 2>&1 && break ||
sleep 2; done

                # IMDSv2 token + private IP
                TOKEN=$(curl -sX PUT "http://169.254.169.254/latest/api/token"
\
                  -H "X-aws-ec2-metadata-token-ttl-seconds: 21600")
                MAN_IP=$(curl -s -H "X-aws-ec2-metadata-token: $TOKEN" \
                  http://169.254.169.254/latest/meta-data/local-ipv4)

                # init + publish + deploy
                docker swarm init --advertise-addr "$MAN_IP"
                TOK=$(docker swarm join-token -q worker)
                aws ssm put-parameter --name /swarm/manager-ip  --type String
--overwrite --value "$MAN_IP" --region ${AWS::Region}
                aws ssm put-parameter --name /swarm/worker-token --type String
--overwrite --value "$TOK"     --region ${AWS::Region}
```

```
                aws s3 cp
s3://${TemplateBucket}/${TemplatePrefix}artifacts/docker-stack.yml
/tmp/docker-stack.yml --region ${AWS::Region}
                docker stack deploy -c /tmp/docker-stack.yml myappname
              '
```

Let us update the Worker 1 and Worker 2 too:

```
          #cloud-config
          runcmd:
            - dnf -y update
            - dnf -y install docker awscli
            - systemctl enable --now docker
            - usermod -aG docker ec2-user
            - 'echo "export AWS_REGION=${AWS::Region}; export
AWS_DEFAULT_REGION=${AWS::Region}" | tee -a /etc/profile.d/awsregion.sh'
            - |
              bash -lc '
                MIP=$(aws ssm get-parameter --name /swarm/manager-ip  --query
Parameter.Value --output text --region ${AWS::Region})
                TOK=$(aws ssm get-parameter --name /swarm/worker-token --query
Parameter.Value --output text --region ${AWS::Region})
                # Wait until worker-token and manager-ip values exist
                for i in {1..60}; do
                  MIP=$(aws ssm get-parameter --name /swarm/manager-ip  --
query Parameter.Value --output text --region ${AWS::Region} 2>/dev/null) ||
true
                  TOK=$(aws ssm get-parameter --name /swarm/worker-token --
query Parameter.Value --output text --region ${AWS::Region} 2>/dev/null) ||
true
                  if [ -n "$MIP" ] && [ -n "$TOK" ]; then break; fi
                  sleep 2
                done
                docker swarm join --token "$TOK" "$MIP:2377"
              '
```

The TL:DR-version in plain English is this: We run the previous commands to update and install, then we query to get the IP and token so we can join, and we complicate the code a bit to have retries in there. My experience is that sometimes the computations from the computer can go faster than this kind of traffic/installs, so having a rudimentary if-else statement or some kind of script running retries is a helpful tool that saves you headache and glitches.

6.4 Upload, update, profit?

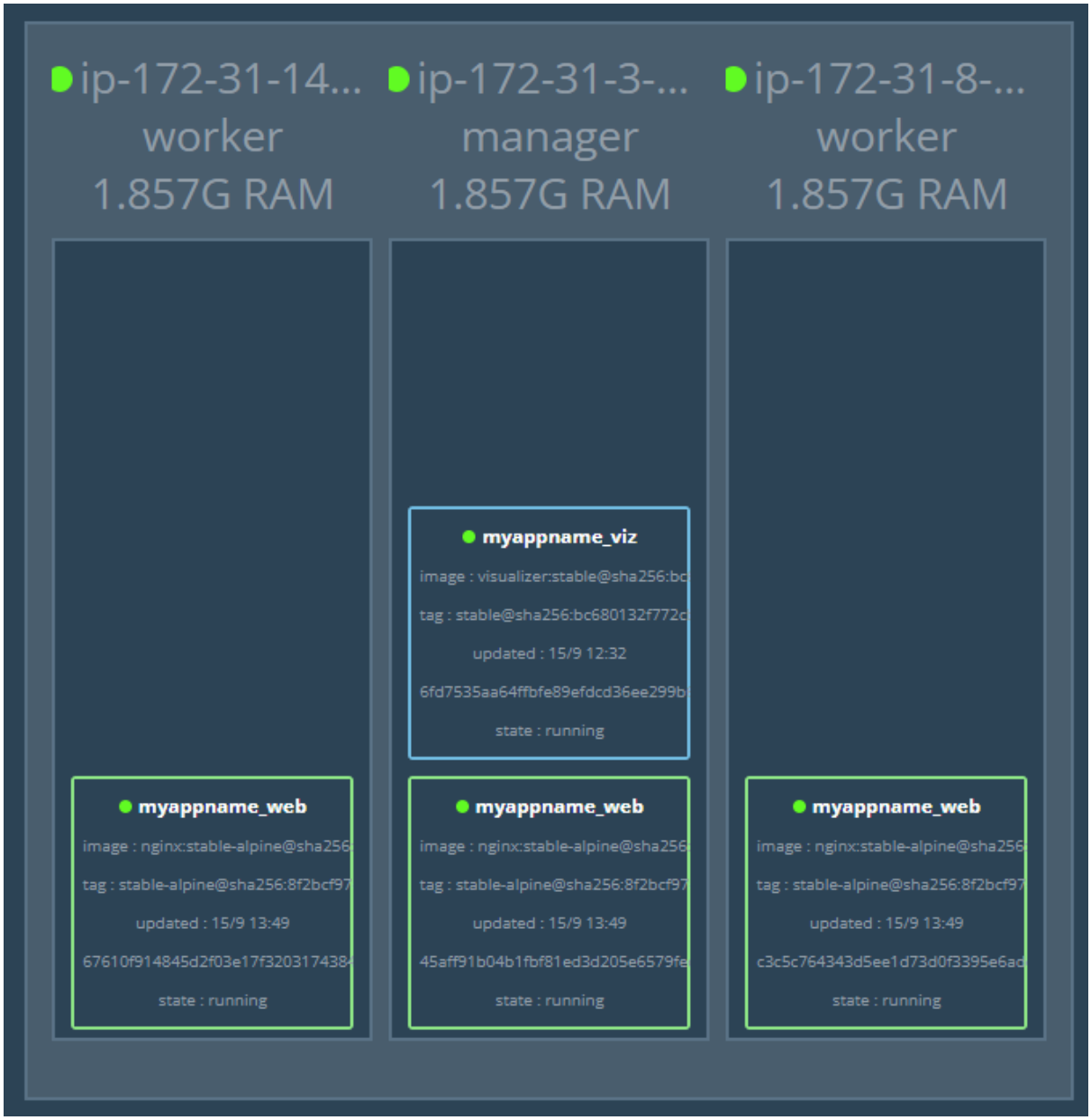
Validate templates you say? Oh that cannot be.. we never did that before? Then upload the files, remember root last, and after all three are uploaded then let us update-stack:

```
aws cloudformation validate-template --template-body
file://infra/templates/20-iam-ec2-role.yaml
```

```
aws cloudformation validate-template --template-body
file://infra/templates/10-ec2-swarm.yaml
aws cloudformation validate-template --template-body
file://infra/templates/root.yaml

aws s3 cp infra/templates/10-ec2-swarm.yaml "s3://$BUCKET/${PREFIX}10-ec2-
swarm.yaml" --region "$REGION"
aws s3 cp infra/templates/20-iam-ec2-role.yaml "s3://$BUCKET/${PREFIX}20-iam-
ec2-role.yaml" --region "$REGION"
aws s3 cp
infra/templates/root.yaml          "s3://$BUCKET/${PREFIX}root.yaml"
--region "$REGION"

aws cloudformation update-stack \
  --region "$REGION" --stack-name "$STACK" \
  --template-url
"https://s3.${REGION}.amazonaws.com/${BUCKET}/${PREFIX}root.yaml" \
  --parameters file://"$PARAMS" \
  --capabilities CAPABILITY_IAM CAPABILITY_NAMED_IAM
```

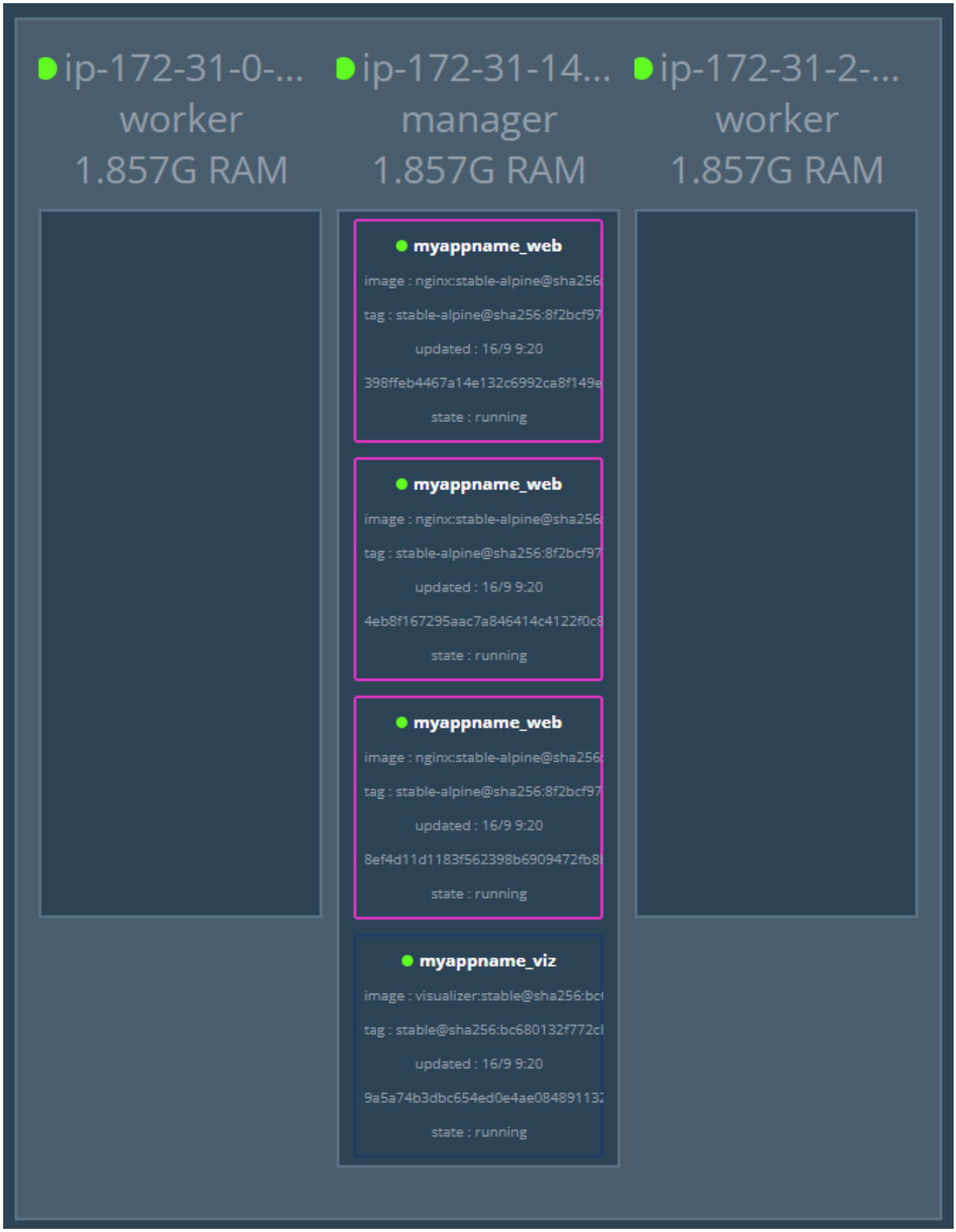
What to expect:

EC2s will replace. I.e. new IPs. On completion, the swarm should already be initialized, workers joined, and the Nginx + Visualizer stack deployed. Hit http://<any-node-ip> and http://<manager-ip>:8080 to verify that everything is working as it should. http://18.202.238.106:8080 =



6.4.1 A tiny correction added the day after

I took down all stacks yesterday, ran source .env and create-stack, all looks green and the new Instance-IPs work fine, so at this point the guide should be working fine for you? I had a problem though. See below:



Basically what happens is that Swarm does not auto-balance tasks when new nodes join. My Stack deploys on the Manager first, Workers join a bit later, so all replicas land on the Manager and stay there until we update or scale.

One solution would be to manually or automatically try to run an update (docker service update) or to run a command to scale up or down. Since this whole system is built to be scaleable and

automatic, that would be a poor solution. Instead, I added a check to the Managers User Data. Review the bottom half, after the section with TOKEN and MAN_IP, replace your code with this:

```
                # Init the Swarm
                docker swarm init --advertise-addr "$MAN_IP"

                # Publish join info
                TOK=$(docker swarm join-token -q worker)
                aws ssm put-parameter --name /swarm/manager-ip  --type String --overwrite --value
"$MAN_IP" --region ${AWS::Region}
                aws ssm put-parameter --name /swarm/worker-token --type String --overwrite --value
"$TOK"     --region ${AWS::Region}

                # Wait for workers to join (expecting 2; adjust if your size changes)
                until [ "$(docker node ls --format "{{.ManagerStatus}} {{.Status}} {{.Availability}}"
\
                            | awk '\''
                              $1=="" && $2=="Ready" && $3=="Active"{c++}
                              END{print c+0}'\'')" -ge 2 ]; do
                  sleep 5
                done

                # Pull compose and FIRST deploy
                aws s3 cp s3://${TemplateBucket}/${TemplatePrefix}artifacts/docker-stack.yml
/tmp/docker-stack.yml --region ${AWS::Region}
                docker stack deploy -c /tmp/docker-stack.yml myappname

                # Small buffer to let tasks start
                sleep 10

                # **Option A automated**: clean recreate of the app service (brief blip)
                docker service rm myappname_web || true
                docker stack deploy -c /tmp/docker-stack.yml myappname
              '
```

This means we now wait a little bit after deploy and adds a small buffer to let tasks start.

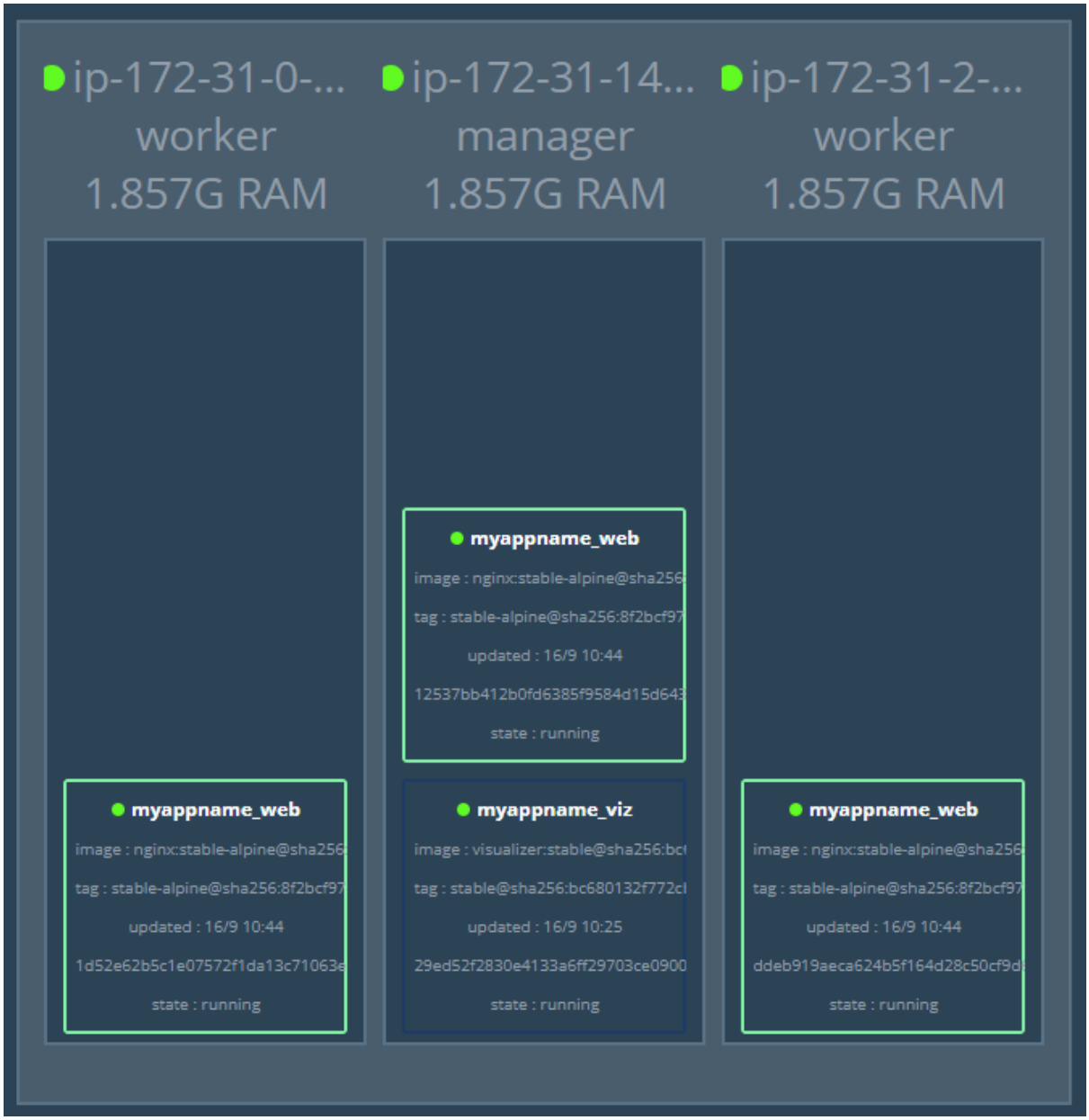
Also, update the docker-stack.yml you should have the services > web > deploy > placement path. Below “placement” and above “preferences”, should be line 8, add this:

```
        max_replicas_per_node: 1  # ensures 1 node before doubling it
```

need. Validate 10-ec2-etc, upload 10-ec2-etc plus the docker-stack.yml, run update-stack. Remember the docker-stack has a different path:

```
aws s3 cp infra/artifacts/docker-stack.yml
"s3://$BUCKET/${PREFIX}artifacts/docker-stack.yml" --region "$REGION"
```

Back on track again:



6.5 Automation-Summary

- Manager UserData: get IMDSv2 token > fetch MAN_IP > swarm init > publish /swarm/* > pull compose > docker stack deploy.
- All nodes UserData: set default region
  echo "export AWS_REGION=${AWS::Region}; export AWS_DEFAULT_REGION=${AWS::Region}" | tee -a /etc/profile.d/awsregion.sh
- Workers: DependsOn: Manager + small wait loop for /swarm/manager-ip and /swarm/worker-token.

Workers depend on the Manager instance; the manager writes the rendezvous values in SSM Parameter Store. IMDSv2 is used to securely fetch the manager's private IP and to issue short-lived credentials to the instances via the role.

When I created this guide I had a problem with the regions sitting as blank info inside the Instances so I set the regional value within the manager and export that.

At this point I would love to set up SSM Session Manager and ditch SSH/22 but let us get rolling with a .NET app instead, the SSM is more cosmetic at this point than adding value. .NET completely transforms this Swarm to a proper application.

7. .NET, .NET, my Java-library for a .NET(-app)!

There is the shorter route to build this: run a "dotnet new mvc" in your IDEs terminal and just use Microsofts own MVC-template. I do not work that way. Yes, unfortunately this is going to be more complicated here, but in return – we will learn more, and you will have more control with less bloat.

7.1 Minimal MVC to get started

Here is very important for us developers with limited experience: Be mindful with LLMs. They will want to give you big blocks of code, multiple classes and they will not put things into context. I am not sure if you are familiar with the Model-View-Controller (MVC) concept? We will not go into that here, but it is worth mentioning the basics and why we use that. Read more in-depth at X.3.

7.1.1 Separations of concern and modularity

The MVC pattern in .NET is well-suited for web applications because it cleanly separates concerns: the Model manages data and business logic, the View handles the user interface, and the Controller coordinates communication between them. This structure makes the codebase more maintainable, testable, and scalable, while also simplifying integration with backends and APIs by keeping data handling and presentation clearly separated.

I use similar arguments to run the Nested Stacks pattern in CloudFormation.

7.1.2 BIY/DIY – Build/Do It Yourself – the plan

We will begin by building the basic skeleton for a MVC pattern in .NET:

- A minimal Program.cs (the orchestrator / engine)
- HomeController to handle root/landing page results
- An Index.cshtml landing page (replace our Nginx with)
- Slimmed Dockerfile (setup)

Basically we will just make a web front, and then make sure our Swarm nodes load that web front instead of Nginx. So we will have to adjust our Docker compose to display this new front. We do not build interactive elements (a webform) on the .NET app yet, so no connection to DynamoDB/S3 yet.

7.1.3 app/src/SwarmMvc/SwarmMvc.csproj – warning about the risk with LLM

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net9.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>
</Project>
```

Note: I ask LLMs to generate these code sections for me, but I read through them and I adjust. Here

it wanted me to use net8.0 – its data is old, even though I use the latest version LLMs. It even told me I am wrong when I adjust the code to 9.0, until I pointed out its mistake. This is written in September 2025, and the latest version of .NET is 9.0, with 10.0 coming in less than two months from now. It is probably a good idea to not generate code you do not understand.

7.1.3.1 Briefly on SDK

A SDK (Software Development Kit) is a collection of tools, libraries and APIs that simplify building applications by providing prebuilt functionality for a specific platform. In this case we use the NET SDK, further down the road we will use the AWS SDK too for interacting with AWS services.

7.1.4 app/src/SwarmMvc/Program.cs

Then we just need a basic program that sets functionality we will use and runs them (app.Run):

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddControllersWithViews();

var app = builder.Build();
app.UseStaticFiles();
app.UseRouting();

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();
```

- We initiate a builder, and the builder includes the Services.AddControllerWithViews toolkit.
- Then we create an “app” which the builder build/create.
- We tell the app to use static files and routing
- Then we define the routing for the mapcontrollerroute specifically
- Finally we run the app

7.1.5 app/src/SwarmMvc/Controllers/HomeController.cs

Let us build the first Controller, this one will “control” our Home.

```
using Microsoft.AspNetCore.Mvc;

namespace SwarmMvc.Controllers;

public class HomeController : Controller
{
    public IActionResult Index() => View();
}
```

This code defines what action we will do when Index is triggered (it will use View).

7.1.6 Then let us create the V’s (the Views, what we see at the web front)

app/src/SwarmMvc/Views/_ViewImports.cshtml –

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Naming convention for Views dictate a _underscore in the name. The only job for the ViewImports will initially be just to add a TagHelper.

app/src/SwarmMvc/Views/_ViewStart.cshtml –

```
@{
    Layout = "_Layout";
}
```

This defines the Layout. So what is the layout? We will create that next.
app/src/SwarmMvc/Views/Shared/_Layout.cshtml –

```
<!doctype html>
<html lang="en">

<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <title>Welcome to the Swarm</title>
</head>

<body>
    <header><a href="/">We are all Swarm</a></header>
    <main>
        @RenderBody()
    </main>
</body>

</html>
```

That is mostly HTML, but a “cshtml”-file means the Razor syntax, so we can mix HTML with .NET-code. It is not convention for a fully interactive frontend, but it is quick to throw together. 😊 You can see that inside the “main” body, we call the RenderBody. You might also notice the path to this file was put inside “/Shared/”, well this will be the template html-file that is shared among the views.

But we need an actual landing site too, so let us do Index-page:
app/src/SwarmMvc/Views/Home/Index.cshtml

```
<h1>Hello from SwarmMvc</h1>
<p>Prepare to be assimilated.</p>
```

Unlike a regular .html page, we do not have to build all the hypertext. We work with separations of concern and modularity here as well, so this Index-file will be called upon by RenderBody.

7.1.7 Dockerize (no AWS SDK yet)

We want a basic Dockerfile, put it at the repo root. Just name it Dockerfile, no extension:

```
# build
FROM mcr.microsoft.com/dotnet/sdk:9.0 AS build
```

```
WORKDIR /src
COPY app/src/SwarmMvc/ ./SwarmMvc/
RUN dotnet restore SwarmMvc/SwarmMvc.csproj
RUN dotnet publish SwarmMvc/SwarmMvc.csproj -c Release -o /out

# runtime
FROM mcr.microsoft.com/dotnet/aspnet:9.0
WORKDIR /app
COPY --from=build /out .
EXPOSE 8080
ENV ASPNETCORE_URLS=http://+:8080
ENTRYPOINT ["dotnet","SwarmMvc.dll"]
```

Workdir points that the directory we work with is /src. Then it copies the files from the path. It restores and publishes based on the setup in the .csproj-file, to the /out directory. Then it copies from /out, exposes port 8080. We pass ASPNETCORE_URLS via environment variables (in Dockerfile and docker-compose), not via a .env file yet, and set it to http://+:8080.

7.1.7.1 Note on ports

The app publishes port 80 on all nodes > container 8080 (80:8080). The Visualizer publishes port 8080 on the manager > container 8080 (8080:8080). Both containers listen on 8080 internally, which is normal, and there's no conflict because the published ports differ.

7.1.8 Swap Nginx to the new app in compose

In the docker-stack.yml file, replace the web-code block under services: with this:

```
  web:
    image: __ECR_IMAGE__
    environment:
      - ASPNETCORE_URLS=http://+:8080
    ports: ["80:8080"]
    networks: [webnet]
    deploy:
      replicas: 3
      placement:
        max_replicas_per_node: 1
        preferences:
          - spread: node.id
```

As you can see, we use similar setup as the Nginx did for the replicas/nodes and ports. What we are essentially doing here is we point to another image/program instead.

7.1.9 Our basic .NET-skeleton-Summary

- Program.cs wires minimal MVC + routing; HomeController returns Index.cshtml.
- Dockerfile targets .NET 9, exposes 8080 inside the container.
- Compose publishes 80>8080 for the web service; Visualizer remains on 8080 (manager only).

7.2 Build > push > upload compose > redeploy

We need a place for Swarm to pull the container image from. Text files (our YAML) can live in S3, but nodes run images, not source. We considered four options:

- Amazon ECR + IAM (chosen): AWS-native, no personal tokens, pulls authorized by the EC2 instance role you already use.
- GitHub Container Registry (GHCR): needs a GitHub token you'd have to manage.
- Private registry inside Swarm: more ops to run/secure.
- Pre-loading images on each node: brittle and manual.

Decisions: Use ECR + IAM for least secrets and best fit with our CloudFormation/IAM setup.

7.2.1 ECR + IAM – progression

Let us go back to the infra/templates section and create 25-ecr.yaml:

```yaml
AWSTemplateFormatVersion: '2010-09-09'
Description: ECR repository for the Swarm .NET app

Parameters:
  RepositoryName:
    Type: String
    Description: ECR repo name (no slash)
    Default: swarm-dotnet-app
  ImageTagMutability:
    Type: String
    AllowedValues: [MUTABLE, IMMUTABLE]
    Default: MUTABLE
  ScanOnPush:
    Type: String
    AllowedValues: [true, false]
    Default: true
  KeepUntagged:
    Type: Number
    Default: 10
    Description: Keep last N untagged images (lifecycle)

Conditions:
  EnableScan: !Equals [!Ref ScanOnPush, "true"]

Resources:
  AppRepository:
    Type: AWS::ECR::Repository
    Properties:
      RepositoryName: !Ref RepositoryName
      ImageTagMutability: !Ref ImageTagMutability
      ImageScanningConfiguration:
        ScanOnPush: !If [EnableScan, true, false]
      EncryptionConfiguration:
        EncryptionType: AES256
```

```
      LifecyclePolicy:
        LifecyclePolicyText: !Sub |
          {
            "rules": [
              {
                "rulePriority": 1,
                "description": "Expire untagged images beyond
${KeepUntagged}",
                "selection": {
                  "tagStatus": "untagged",
                  "countType": "imageCountMoreThan",
                  "countNumber": ${KeepUntagged}
                },
                "action": { "type": "expire" }
              }
            ]
          }

Outputs:
  RepositoryUri:
    Value: !GetAtt AppRepository.RepositoryUri
    Export:
      Name: !Sub "${AWS::StackName}-RepositoryUri"
  RepositoryArn:
    Value: !GetAtt AppRepository.Arn
```

ECR is “Elastic Container Registry”, what we have created here is actually a “small” file, but we have to set quite a few values for it. This basically works like a dynamic storage for us. Next up, we have to add the child template to root.yaml. Below all other <Name>Stack, place this:

```
  EcrStack:
    Type: AWS::CloudFormation::Stack
    Properties:
      TemplateURL: !Sub
"https://${TemplateBucket}.s3.${AWS::Region}.amazonaws.com/${TemplatePrefix}25
-ecr.yaml"
      Parameters:
        RepositoryName: !Ref RepositoryName
        ImageTagMutability: MUTABLE
        ScanOnPush: true
        KeepUntagged: 10
```

Also add this at the bottom of the Parameters at the start of the document:

```
  RepositoryName:
    Type: String
```

We also have to add this to the dev.params.json so we have the parameters set:

```
    {
```

```
        "ParameterKey": "RepositoryName",
        "ParameterValue": "swarm-dotnet-app"
    }
```

7.2.1.1 Update the Manager User Data (deploy from ECR)

Visit the Resources > Manager. Under UserData, replace the code with this:

```
      UserData:
        Fn::Base64: !Sub |
          #cloud-config
          runcmd:
            - dnf -y update
            - dnf -y install awscli

            # 1) Ensure Docker is installed and running (robust for AL2023
variants)
            - |
              bash -lc '
                if ! command -v docker >/dev/null 2>&1; then
                  dnf -y install docker || dnf -y install moby-engine docker-
cli
                fi
                systemctl enable --now docker
                usermod -aG docker ec2-user || true
                for i in {1..10}; do docker info >/dev/null 2>&1 && break ||
sleep 2; done
              '

            # 2) Init swarm, publish tokens, substitute ECR image, ECR login,
deploy
            - !Sub |
              bash -lc '
                echo "export AWS_REGION=${AWS::Region}; export
AWS_DEFAULT_REGION=${AWS::Region}" | tee -a /etc/profile.d/awsregion.sh

                # IMDSv2 private IP
                TOKEN=$(curl -sX PUT "http://169.254.169.254/latest/api/token"
-H "X-aws-ec2-metadata-token-ttl-seconds: 21600")
                MAN_IP=$(curl -s -H "X-aws-ec2-metadata-token: $TOKEN"
http://169.254.169.254/latest/meta-data/local-ipv4)

                # Init + publish join info
                docker swarm init --advertise-addr "$MAN_IP"
                TOK=$(docker swarm join-token -q worker)
                aws ssm put-parameter --name /swarm/manager-ip  --type String
--overwrite --value "$MAN_IP" --region ${AWS::Region}
                aws ssm put-parameter --name /swarm/worker-token --type String
--overwrite --value "$TOK"     --region ${AWS::Region}
```

```
                # Optional gate: wait for two Ready/Active workers
                for i in {1..60}; do
                  c=$(docker node ls --format "{{.ManagerStatus}} {{.Status}}
{{.Availability}}" | awk '\''$1=="" && $2=="Ready" && $3=="Active"{c++}
END{print c+0}'\'')
                  [ "$c" -ge 2 ] && break || sleep 5
                done

                # Pull generic compose
                aws s3 cp
s3://${TemplateBucket}/${TemplatePrefix}artifacts/docker-stack.yml
/tmp/docker-stack.yml --region ${AWS::Region}

                # substitute placeholder with real ECR image
                IMAGE="${AWS::AccountId}.dkr.ecr.${AWS::Region}.amazonaws.com/
swarm-dotnet-app:dev"
                sed -i "s|__ECR_IMAGE__|$IMAGE|g" /tmp/docker-stack.yml

                # login to ECR via instance role, then deploy with creds
shared to workers
                aws ecr get-login-password --region ${AWS::Region} \
                  | docker login --username AWS --password-stdin
"${AWS::AccountId}.dkr.ecr.${AWS::Region}.amazonaws.com"

                docker stack deploy -c /tmp/docker-stack.yml --with-registry-
auth myappname

                # Optional nudge for even spread (passes creds)
                sleep 10
                docker service rm myappname_web || true
                docker stack deploy -c /tmp/docker-stack.yml --with-registry-
auth myappname
              '
```

That is a massive code change! ..actually not that much, it is mainly tweaks, but in this case it was easier to present the block than trying to direct to all tiny changes, using text to explain.

We first make sure Docker is installed and the daemon is ready. Then we initialize the swarm, publish /swarm/manager-ip and /swarm/worker-token to SSM, fetch the generic compose from S3, replace __ECR_IMAGE__ with the real ECR URI using CloudFormation pseudo-parameters, log in to ECR using the instance role, and deploy the stack with --with-registry-auth so workers can pull the image without manual login.

7.2.1.2 Update Worker UserData (join Swarm)

Apply the same two-block pattern as above to both Workers. Update their UserData with this:

```
      UserData:
        Fn::Base64: !Sub |
```

```
          #cloud-config
          runcmd:
          - dnf -y update
          # ensure awscli present (needed to read SSM params)
          - dnf -y install awscli

          # 1) Make sure Docker is installed and running
          - |
            bash -lc '
              if ! command -v docker >/dev/null 2>&1; then
                dnf -y install docker || dnf -y install moby-engine docker-
cli   # CHANGED
              fi
              systemctl enable --now docker
              usermod -aG docker ec2-user || true
              for i in {1..10}; do docker info >/dev/null 2>&1 && break ||
sleep 2; done   # CHANGED: wait for daemon
            '

          # 2) Join the swarm (wait for SSM rendezvous values)
          - |
            bash -lc '
              echo "export AWS_REGION=${AWS::Region}; export
AWS_DEFAULT_REGION=${AWS::Region}" | tee -a /etc/profile.d/awsregion.sh
              for i in {1..60}; do
                MIP=$(aws ssm get-parameter --name /swarm/manager-ip  --query
Parameter.Value --output text --region ${AWS::Region} 2>/dev/null) || true
                TOK=$(aws ssm get-parameter --name /swarm/worker-token --query
Parameter.Value --output text --region ${AWS::Region} 2>/dev/null) || true
                [ -n "$MIP" ] && [ -n "$TOK" ] && break
                sleep 2
              done
              docker swarm join --token "$TOK" "$MIP:2377"
            '
```

We ensure Docker is installed/running, then poll SSM for the manager IP and worker token and join the swarm. Workers do **not** log in to ECR or deploy anything; they just become eligible to run tasks the manager schedules.

Validate. Vaaaaaaaalidddddaaatteeee. It has been some time since the last one! 😊

```
aws cloudformation validate-template --template-body file://infra/templates/25-ecr.yaml
aws cloudformation validate-template --template-body file://infra/templates/10-ec2-swarm.yaml
aws cloudformation validate-template --template-body file://infra/templates/root.yaml
```

Validation is really useful. Here I got a validation warning because I had used brackets around {IMAGE} in the User Data, which works most of the time, but if it is inside a !Sub block, then CloudFormation will try substitute it. Thus we just remove the brackets inside one line (I have already

corrected the code above for your convenience). Validated again and then it worked. Remember, that validate-template is mostly a syntax checker, but it does catch tiny mistakes before we move too far!

7.2.2 Create ECR repository

This is not really a big section, but the previous one grew quite big so let us separate the upload and update, to here. Run the regular template uploads, plus the docker-stack.yml:

```
# templates (children)
aws s3 cp infra/templates/25-ecr.yaml \
  "s3://${BUCKET}/${PREFIX}25-ecr.yaml"

aws s3 cp infra/templates/10-ec2-swarm.yaml \
  "s3://${BUCKET}/${PREFIX}10-ec2-swarm.yaml"

# artifacts: upload the GENERIC compose (with __ECR_IMAGE__)
aws s3 cp infra/artifacts/docker-stack.yml \
  "s3://${BUCKET}/${PREFIX}artifacts/docker-stack.yml" --region $REGION

# template (root) last
aws s3 cp infra/templates/root.yaml \
  "s3://${BUCKET}/${PREFIX}root.yaml"
```

Normally at this point, we would run update-stack, but we have an important step to pass before we can do that. The Managers User Data now calls `aws ecr get-login-password` and Swarm will pull from ECR. That will fail unless the EC2 Instance has the ECR permissions, which we set in 7.2.3..:

7.2.3 IAM: allow EC2-Instances to pull (and occasionally push) from ECR

In section 5.2.1 we created the 20-iam-ec2-role.yaml. It is time to edit that one to add policies. Scroll down to Resources > AppEc2Role > Properties > Policies > - PolicyName: !Sub '${StackNamePrefix}-swarm-ssm-s3' – you should find this around line 42. After the ssm:, arn: and s3: policies, add these ecr:-policies at the end.

```
              # Read/write – keep the code you had
                   ..<snip code snip snip>..
              # Fetch – keep the code you had
                   ..<snip code snip snip>..
              # ListBucket – keep the code you had
                   ..<snip code snip snip>..
                   ..<ADD THE CODE BELOW>..
              # ECR: allow fetching a registry auth token (required for login)
              - Effect: Allow
                Action:
                  - ecr:GetAuthorizationToken
                Resource: "*"
              # ECR: allow pulling images from this repo
              - Effect: Allow
```

```
                Action:
                  - ecr:BatchCheckLayerAvailability
                  - ecr:GetDownloadUrlForLayer
                  - ecr:BatchGetImage
                Resource: !Sub
'arn:aws:ecr:${AWS::Region}:${AWS::AccountId}:repository/swarm-dotnet-app'
```

This is the CloudFormation way of handing roles the way we did things manually in the IAM console, early in this guide. This is helpful if we are working with specific rights and access on specific nodes, which we should want to do, rather than setting generalised roles – or having to create a lot of roles manually. Now we can handle this through template uploads.

At this point, we are ready to validate the 20-iam-ec2-role.yaml file, then upload:

```
# validate
aws cloudformation validate-template --template-body
file://infra/templates/20-iam-ec2-role.yaml

# upload
aws s3 cp infra/templates/20-iam-ec2-role.yaml  "s3://${BUCKET}/${PREFIX}20-
iam-ec2-role.yaml"
```

I had a tiny syntax error in mine, which I adjusted (both the template code for you here above, and in my actual files). Adjusted it, re-validated the file, uploaded it and ran create-stack. If you follow these steps and always validate (both syntax validation but also check health endpoints and IPs etc), you will always notice immediately if a step breaks. Checking CloudFormation stack logs (events, like I showed earlier in this guide) is super helpful, you can usually pinpoint the error immediately.

Wait a second now with validation (! Yes, really). We need to do next section first.

7.2.4 Build and push the image (ECR)

In older setups you might use CodePipeline/CodeBuild to build and push images or publish to Docker Hub / GHCR with personal tokens. Here we keep it simpler and more secure for this prototype: build locally > push to Amazon ECR > pull at boot via IAM. No GitHub tokens, no extra pipeline to maintain; EC2 instances use their instance role to authenticate pulls automatically.

7.2.4.1 Update your .env and use the values set there

For this section we uses variables set in our .env. I updated my .env to use the “export” command, as I do “source .env” at the start of each session to load in variables. It looks like this:

```
# .env (example)
export REGION="eu-west-1"
export BUCKET="cf-swarm-<account-id>-eu-west-1"
export PREFIX="swarm-iac/templates/"
export STACK="swarm-cf-root"
export PARAMS="infra/parameters/dev.params.json"
export ACCOUNT="<account-id>"
export REPO="swarm-dotnet-app"
```

```
# Derived (use inline when needed)
# IMAGE="$ACCOUNT.dkr.ecr.$REGION.amazonaws.com/$REPO:dev"
```

Note: You will have to replace the <account-id> sections with your actual 12-number account id. When you have these variables available, you can run this in Bash:

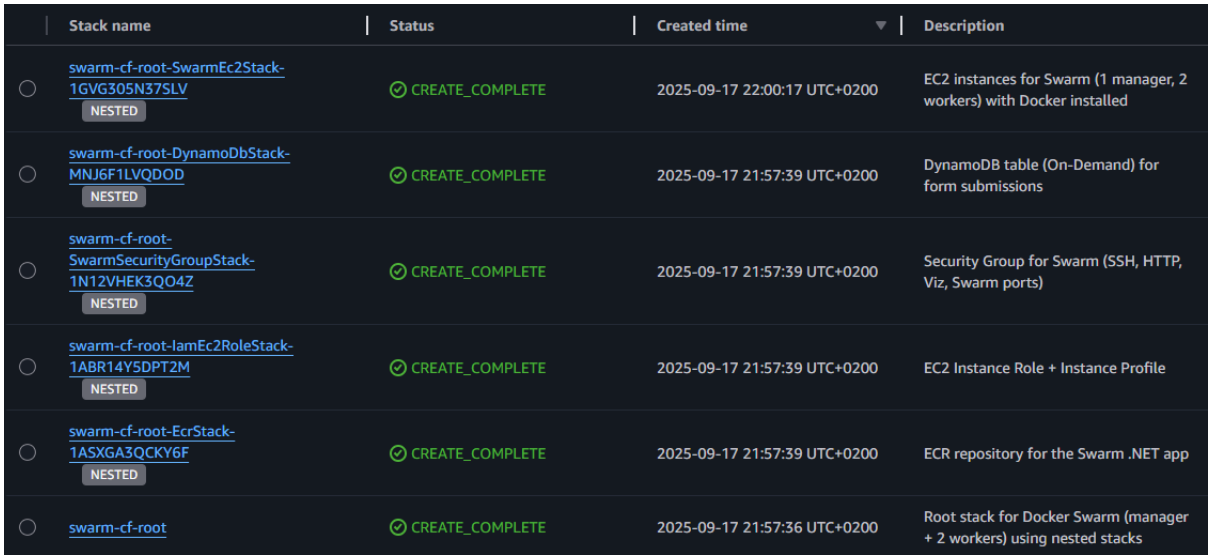
```
# Using .env: ACCOUNT, REGION, REPO
export IMAGE="$ACCOUNT.dkr.ecr.$REGION.amazonaws.com/$REPO:dev"

aws ecr get-login-password --region "$REGION" \
| docker login --username AWS --password-stdin
"$ACCOUNT.dkr.ecr.$REGION.amazonaws.com"

docker build -t "$IMAGE" -f Dockerfile .
docker push "$IMAGE"
```

I quite like using parameters instead of hardcoding values, obviously hardcoding is not to be recommended, but this also helps me to re-use commands and automize things. What we do above? We export an image of our package, then we run Dockerfile to build with the instructions there. Finally, we push the built copy to the IMAGE.

Vital: Because the image exists in ECR before we create the stack, the manager's UserData can pull it immediately and deploy the service on first boot (no manual steps).

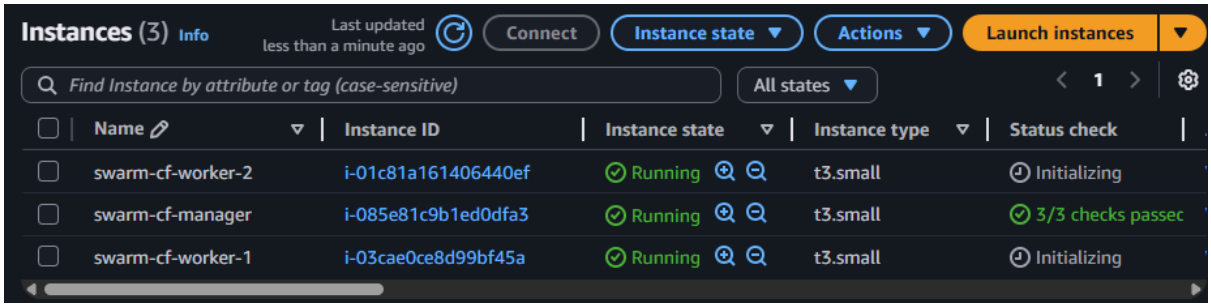
7.2.4.2 Verify that the ECR system and policies are green

Now, we are ready to create-stack or update-stack. Do it. See if everything stays green and healthy before moving on to the next step. Hopefully you can see this at AWS CloudFormation console:

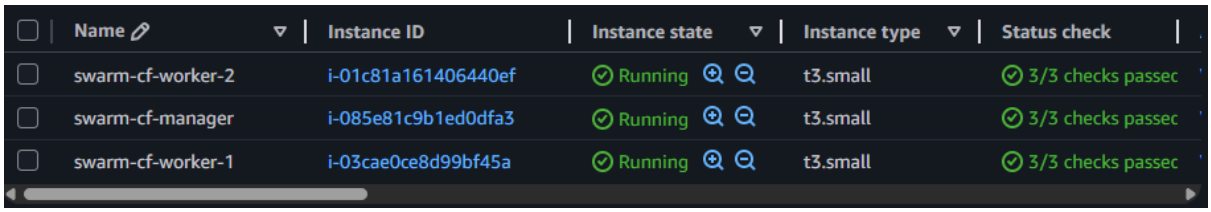


Stack name	Status	Created time	Description
swarm-cf-root-SwarmEc2Stack-1GVG305N37SLV NESTED	CREATE_COMPLETE	2025-09-17 22:00:17 UTC+0200	EC2 instances for Swarm (1 manager, 2 workers) with Docker installed
swarm-cf-root-DynamoDbStack-MNJ6F1LVQDOD NESTED	CREATE_COMPLETE	2025-09-17 21:57:39 UTC+0200	DynamoDB table (On-Demand) for form submissions
swarm-cf-root-SwarmSecurityGroupStack-1N12VHEK3QO4Z NESTED	CREATE_COMPLETE	2025-09-17 21:57:39 UTC+0200	Security Group for Swarm (SSH, HTTP, Viz, Swarm ports)
swarm-cf-root-IamEc2RoleStack-1ABR14Y5DPT2M NESTED	CREATE_COMPLETE	2025-09-17 21:57:39 UTC+0200	EC2 Instance Role + Instance Profile
swarm-cf-root-EcrStack-1ASXGA3QCKY6F NESTED	CREATE_COMPLETE	2025-09-17 21:57:39 UTC+0200	ECR repository for the Swarm .NET app
swarm-cf-root	CREATE_COMPLETE	2025-09-17 21:57:36 UTC+0200	Root stack for Docker Swarm (manager + 2 workers) using nested stacks

Important note: Since I added that little bit of delay on the initialization of the Instances, and I wanted the Manager to run first, you might end up seeing something like this in your EC2 Instances:



That is normal, and it might actually take several minutes for all Instances to go green. I think it takes 2-5 minutes for me normally after the entire stack is up, before the full group goes green:



At this point, you can check their IP in a browser like we did before.

7.2.4.3 Keeping it simple and testable

I had both Workers and the Manager showing this:

As you notice this page is super simple. The reason is this: if I spend alot of energy working the .NET app before I have the infrastructure working, then there is also risk that things in the .NET app would be broken without me noticing. If Instead I focus on having a simple landing-/health-page up, then I can focus on testing the infrastructure first.

This page above is working, and I even put a mini-dynamic test there by making sure the link ("We are all Swarm") leads dynamically to its own IP. Since every EC2 Instance uses the same .NET-platform, we need a way to make sure we can see that each node/Instance is unique. I do this by having some data dynamically: normally I would display a simple message "This is instance <dynamic-message-using-instance-ID-or-IP>.

When the entire infrastructure is running smooth, and the takedown and re-deploy (create-stack) of stacks is working as intended, then I will work on the .NET app. At that point, I will also be able to constantly see immediately if something breaks while working on it. Keep things simple, testable.

7.2.5 Upload the updated compose to S3

Upload the generic docker-stack.yml (still containing image: _ECR_IMAGE_) to the artifacts location in S3. The manager will fetch this file at boot, substitute the placeholder with the real ECR URI using ${AWS::AccountId} and ${AWS::Region}, log in to ECR, and deploy the stack.

Command to upload the docker-stack to your S3 (re-use your .env variables):

```
aws s3 cp infra/artifacts/docker-stack.yml \
  "s3://${BUCKET}/${PREFIX}artifacts/docker-stack.yml" \
  --region "$REGION"
```

You only need to re-upload this file if the service definition itself changes (ports, env vars, etc.).

7.2.5.1 Important note on versioning

You want to consider what happens if you do not know which version of the docker-stack.yml we upload. For this small project this is not super important, we can keep track as we just build this step by step and then we will throw away the entire concept. If things break, we backtrack one step only.

But, that said, it is important to understand the concept of versioning here. What we can do is to keep a stable S3 key (our ../artifacts/docker-stack.yml) and turn on S3 Versioning for the buckets. That way you get a tool for rollbacks, without changing URLs. We could also look into handling this through GitHub versioning as well.

7.2.5.2 When to re-upload?

Re-upload the compose ONLY when the service definition changes. Examples:

- Yes, re-upload when you change:
    - Ports, env vars, secrets/configs, networks, volumes
    - Replicas/placement/constraints/update_config/restart_policy
    - Add/remove a service (e.g., introduce a proxy, Redis, etc.)
    - Service names or network names
- No, re-upload needed when you only change the container image contents:
    - Push a new image to the same tag (e.g., :dev) and redeploy
    - The compose still has __ECR_IMAGE__; the manager substitutes the full URI at deploy time
- Changing the image TAG policy:
    - In our setup the tag (:dev) is built in UserData. To move to :v0.1.0, update UserData (manager template) and redeploy the stack. You still don't need to re-upload the compose
    - If you later parameterize the tag via SSM/env, you can switch tags without touching templates or compose

Tip: if you scale via CLI (docker service scale) during testing, capture that in compose later so desired state persists across fresh stack creations.

7.2.6 Redeploy

Fresh stacks will auto-deploy from UserData when you run create-stack.
For an already running stack, you can re-deploy the compose to make Swarm pick up new images or updated service definitions. On the Manager node:

```
aws s3 cp s3://${BUCKET}/${PREFIX}artifacts/docker-stack.yml /tmp/docker-stack.yml --region "$REGION"
docker stack deploy -c /tmp/docker-stack.yml --with-registry-auth myappname
```

Note that this is a manual redeploy, but we should not need/want to do a live update more than during development phases, so I am leaving this comment here so we know how to do this manually, but we will focus on automating the core product primarily.

This does not rebuild images: it simply refreshes the Swarm services against the compose file. Our setup with replicas + max_replicas_per_node + spread ensures tasks remain evenly distributed across nodes.

If you only changed the container image (same tag, e.g., :dev), you can skip S3 and just nudge Swarm to pull the latest:

```
docker service update --force myappname_web
```

That forces the service to restart its tasks using the current ECR image.

7.2.7 Verify in the browser

Verification has two parts:

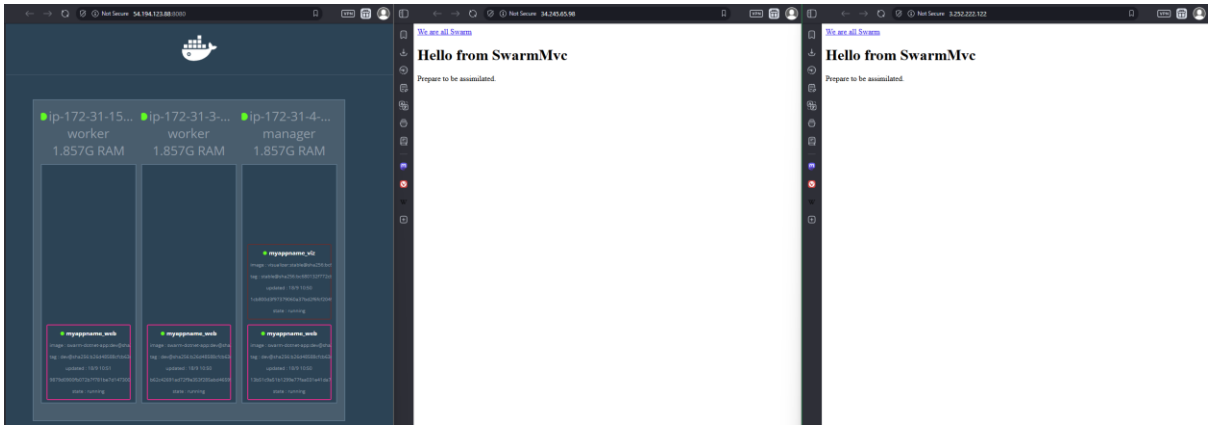
1) Visualizer

Open the Visualizer on the Manager node:
http://<manager-public-ip>:8080
You should see myappname_web running with 3 replicas, spread evenly across Manager and Workers.

2) Landing page

Open a browser to any node's public IP on port 80:
http://<manager-public-ip>
http://<worker1-public-ip>
http://<worker2-public-ip>

Each should serve the simple .NET landing page. If replicas are healthy and load balancing is working, you'll get the page no matter which node you hit.



Good job! What does not show in this picture is that the links are showing the IP to the actual Instance they are displaying on, I mentioned this earlier in the guide. It is important that our .NET-application shares data across the nodes (so they all can get the same state from the backend), but they also need to be able to carry their own webservice stand-alone.

7.2.8 Mini-troubleshooting

If verification fails, check these basics:

- Docker not found > ensure dnf install docker ran in UserData or install manually
- Worker won't join > confirm Manager published /swarm/manager-ip and /swarm/worker-token in SSM
- Service not running > run `docker service ls` and `docker service ps myappname_web` on the Manager
- No page on port 80 > verify Security Group allows inbound HTTP (80) from your IP
- Image not found > confirm you built and pushed swarm-dotnet-app:dev to ECR

These cover many of the errors that might have slipped through your decidated syntax-verification and visual verification steps in the browser, along the way.

7.2.9 .NET-deploy-Summary

We now have a working flow for deploying our .NET app via Docker Swarm:

- Built a basic .NET 9 MVC app with Dockerfile and docker-stack.yml
- Created an ECR repository and IAM role permissions for Manager + Workers
- Updated EC2 UserData to auto-install Docker, initialize/join Swarm, fetch compose, and deploy from ECR
- Pushed our app image to ECR using AWS IAM (no GitHub/DockerHub tokens)
- Uploaded compose with placeholder to S3; Manager substitutes the real image URI on boot
- Verified services with Visualizer and landing page
- Documented redeploy and troubleshooting steps

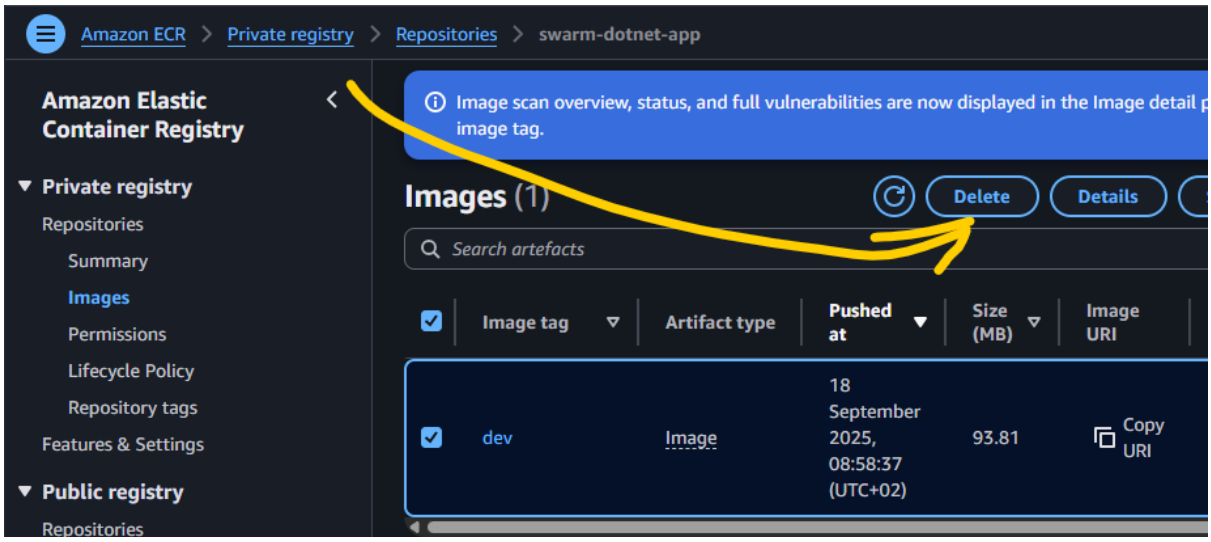
This completes the infrastructure loop: build > push > deploy > verify.

7.3 Problems with delete-stack!

Here we had problems with delete-stack: my root stack got caught in DELETE_FAILED because the ECR repository still has images, thus CloudFormation refused to delete the ECR stack.

7.3.1 Cleanup-Image-command

The graphical path:



The shell-command path:

```
# Delete all images by digest
for DIGEST in $(aws ecr list-images \
  --region eu-west-1 \
  --repository-name swarm-dotnet-app \
  --query 'imageIds[].imageDigest' \
  --output text); do
  aws ecr batch-delete-image \
    --region eu-west-1 \
    --repository-name swarm-dotnet-app \
    --image-ids imageDigest=$DIGEST
done
```

1. We run a command that will list all the ECR images and save them in DIGEST
2. Then we delete the images that are listed in the DIGEST parameter

After that run the normal delete-stack command. This worked perfectly for me, stacks are deleted. But we need to make sure this does not happen again next time! So, we will adjust 25-ecr-yaml. Under Resources > AppRepository > Properties you want this:

```
Resources:
  AppRepository:
    Type: AWS::ECR::Repository
    Properties:
```

```
      RepositoryName: !Ref RepositoryName
      # This below is added
      EmptyOnDelete: true     # <-- CFN will force-delete images on stack
deletion
```

I know code snippets can be confusing, so I just wanted to put it in context. You are literally just adding one line, immediately after the RepositoryName row. That line just sets a Boolean flag that empties the repository on delete.

There is also the option to add the --force flag to the delete-stack, but I try to do as little force as possible. It is just an emergency tool and should be careful with.

8. We need SSM and a CI/CD to handle Docker

I had the plan to introduce Docker and GitHub Actions later, which in a sense is odd, because I normally build the infrastructure and the IaC-aspects first. But, in this assignment, the CI/CD was not part of the concept – and the system is supposed to "expand from the bottom" (the Swarms on the Instances) so it was not natural to introduce it initially.

In short: We now have Swarm + CloudFormation, but we need automation for building, pushing, and updating our .NET Docker image.

8.1 The solutions: Docker-automation and enabling SSM

Automation:

- Problem: Before, we built/pushed the .NET app manually. That broke the automation when we ran create-stack
- Solution: Add a GitHub Actions workflow (build-and-push.yaml) that builds the Docker image and pushes it to ECR on every commit to main
- Why: This ensures fresh images are always available to the Swarm manager without manual intervention

Instance management:

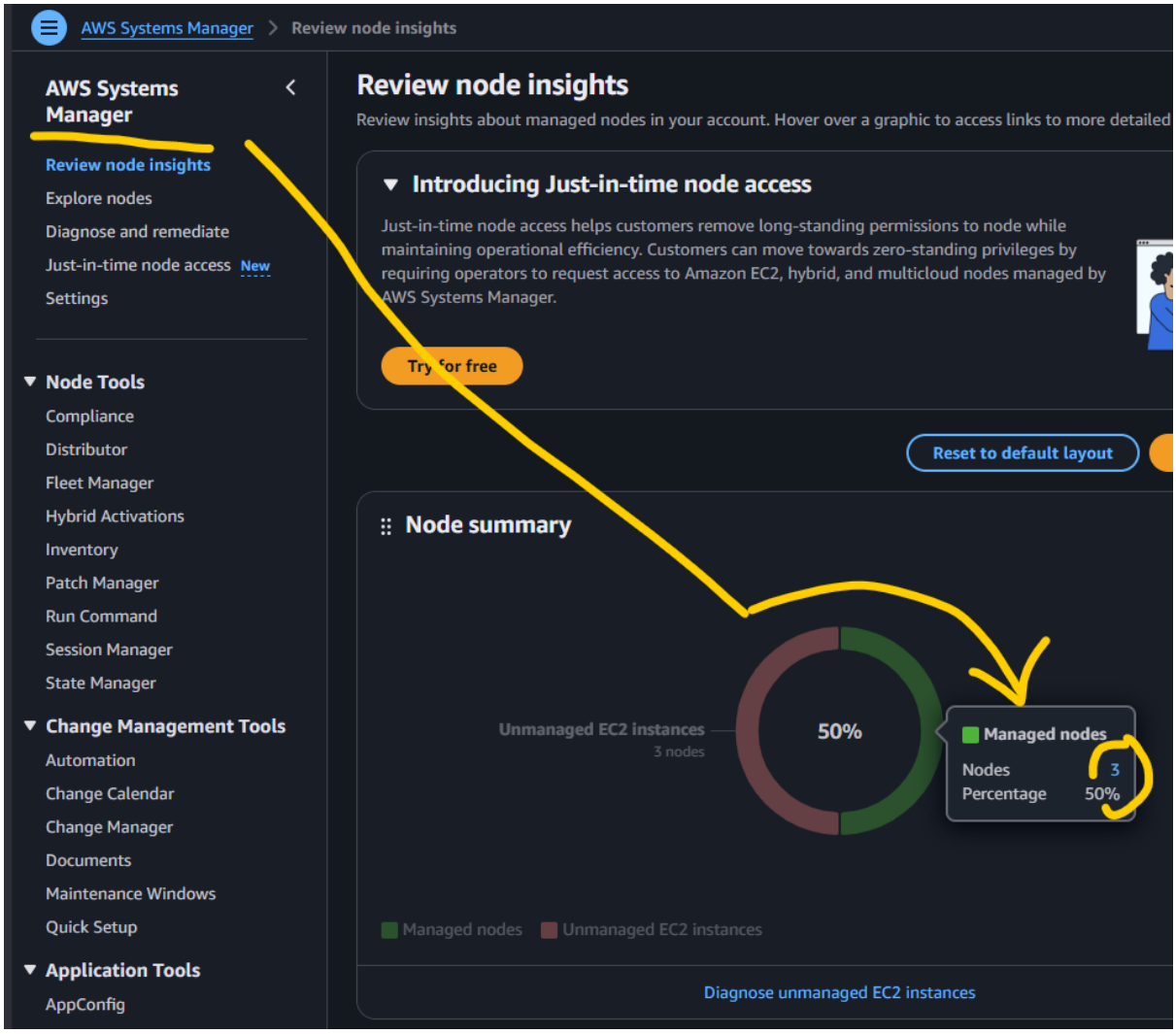
- Problem: Initially, our EC2 instances didn't show up as Managed instances, and a good solution would be to use SSM commands – but we needed to unlock those tools
- Solution (see below for in-depth guide):
    a. Enable SSM in the AWS Console
    b. Attach the policy AmazonSSMManagedInstanceCore to the EC2 IAM role (swarm-cf-ec2-app-role)
    c. Confirm instances now show as *Managed* in Systems Manager > Fleet Manager
- Why: We need this so GitHub Actions (via SSM) can send commands like docker service update –force

8.2 How to enable and implement SSM

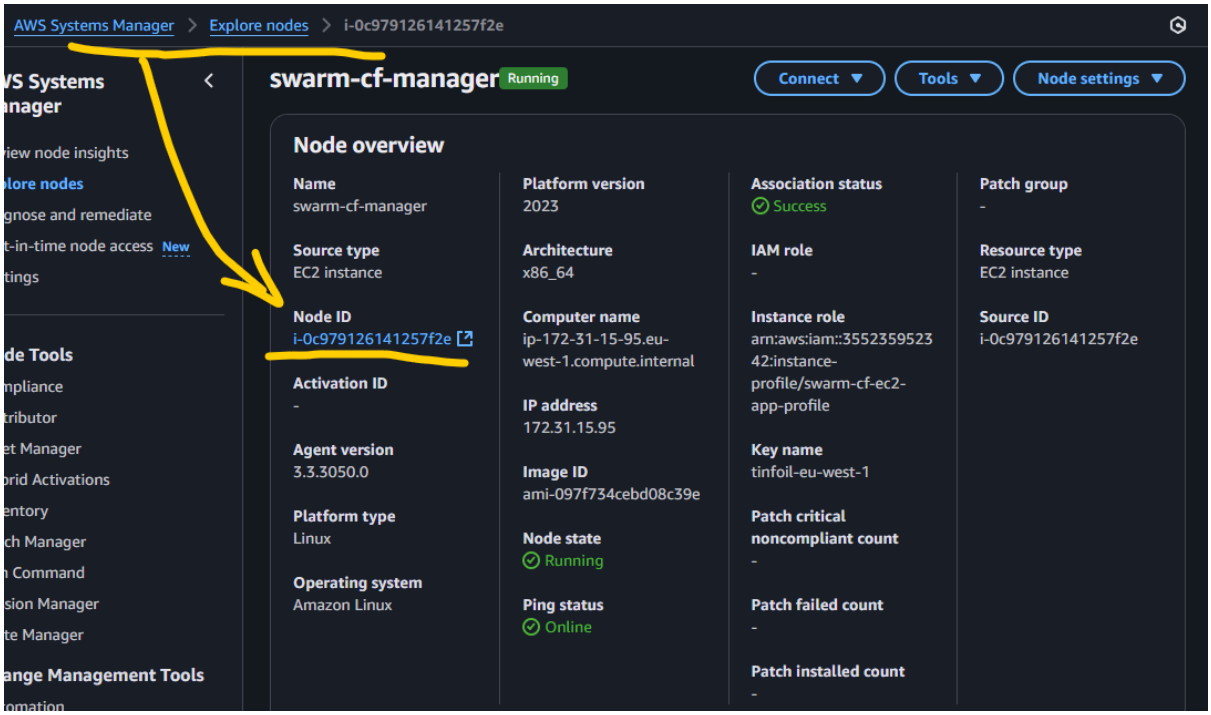
Using SSH is a strength when we develop and maintain, but it is manual, so if we want to automate (or look at Ansible etc) it is nice to "get rid of" SSH and find a tool to automate the control tooling.

AWS Systems Manager (SSM) gives us centralized tooling to manage EC2 instances remotely. This include run commands, patching, inventory, and automation, without SSH access. Nice!

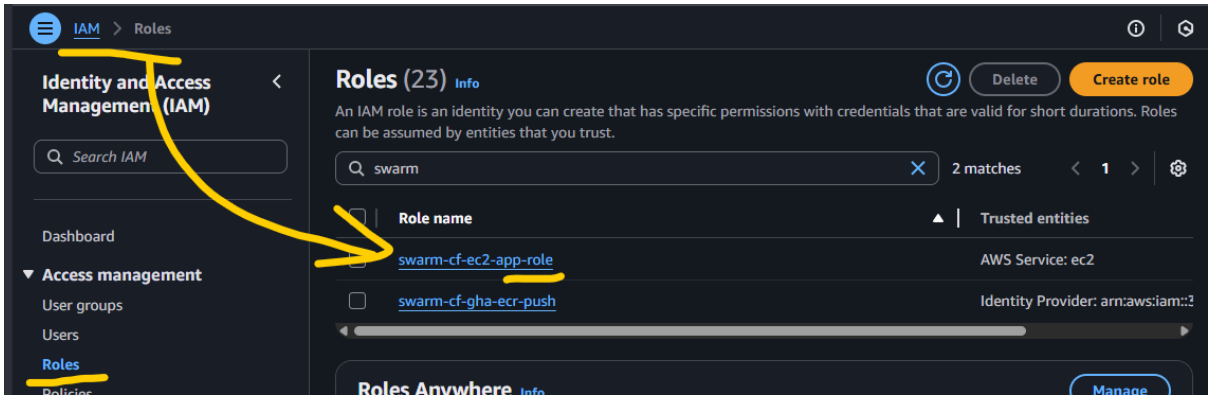
If you have not used SSM yet, search for Systems Manager on AWS. If will have a yellow button saying “enable Systems Manager”. Click that one and wait for a while, should not take more than 5-10 minutes but you will see something like this after a while:



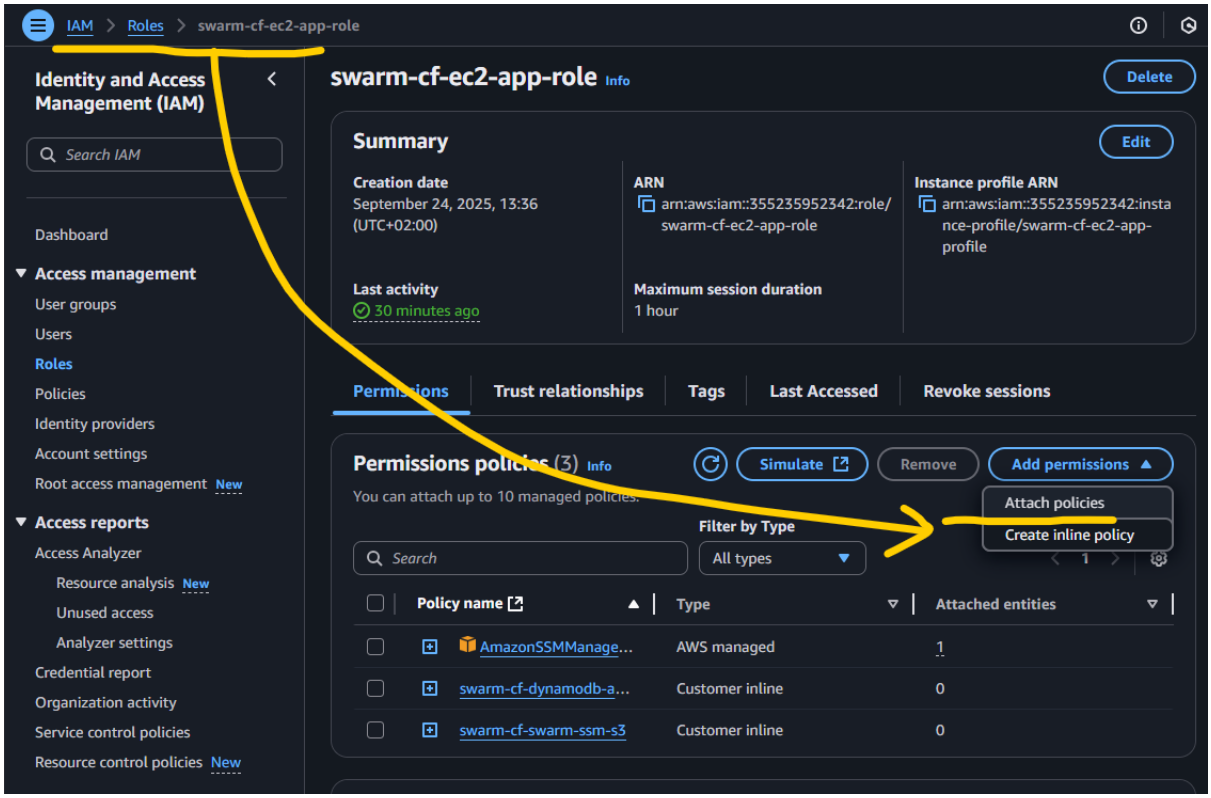
When SSM is activated, you now get tooling to manage the nodes. If you click that “managed nodes” number, you will see your instances. This is one path to reach your nodes managed identity. You will want this ID from the manager node:



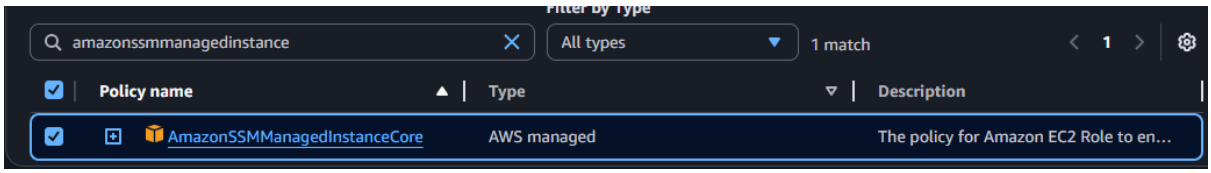
To be able to use these in our systems, we also need to add the SSM policies to our IAM. See the below image, go to your IAM > Roles and click the swarm-cf-ec2-app-role:



Here you can attach policies, see below:



Just search for AmazonSSMManagedInstanceCore in the list and add/apply that policy:



With SSM enabled and the IAM policy attached, our CF-managed EC2 instances can now register as managed nodes and receive remote commands through SSM (like, for example: ssm:SendCommand).

8.3 IAM + OIDC role for GitHub Actions

- Problem: GitHub Actions needs AWS credentials. We don't want long-lived keys
- Solution:
  a. Added 26-github-oidc.yaml (nested stack)
  b. Creates an OIDC provider for token.actions.githubusercontent.com
  c. Defines an IAM role (swarm-cf-gha-ecr-push) that GitHub Actions can assume to push images to ECR
  d. Parameters allow customizing GitHubOrg, GitHubRepo, GitHubRef, and OidcThumbprint.
- Why: Secure short-lived authentication via OIDC, no secrets stored

8.3.1 Create OIDC nested stack

Create the document 26-github-oidc.yaml and paste this code:

```
AWSTemplateFormatVersion: '2010-09-09'
Description: OIDC role for GitHub Actions to push images to ECR
```

```yaml
Parameters:
  StackNamePrefix: { Type: String }
  RepositoryName:  { Type: String, Default: swarm-dotnet-app } # ECR repo
  GitHubOrg:       { Type: String, Default: mymh13 }
  GitHubRepo:      { Type: String, Default: swarm-dotnet-test }
  GitHubRef:       { Type: String, Default: refs/heads/main }
    # allow main branch (adjust if needed)
  OidcThumbprint:  { Type: String,  Default:
6938fd4d98bab03faadb97b34396831e3780aea1 }
    # value can be modified if you want, instead of hardcoding it in the
Resources section

Resources:
  GitHubOidcProvider:
    Type: AWS::IAM::OIDCProvider
    Properties:
      Url: https://token.actions.githubusercontent.com
      ClientIdList: [ sts.amazonaws.com ]
      ThumbprintList: [ !Ref OidcThumbprint ]     # GitHub OIDC root CA

  GitHubEcrPushRole:
    Type: AWS::IAM::Role
    Properties:
      RoleName: !Sub '${StackNamePrefix}-gha-ecr-push'
      AssumeRolePolicyDocument:
        Version: '2012-10-17'
        Statement:
          - Effect: Allow
            Principal:
              Federated: !Ref GitHubOidcProvider
            Action: sts:AssumeRoleWithWebIdentity
            Condition:
              StringEquals:
                token.actions.githubusercontent.com:aud: sts.amazonaws.com
              StringLike:
                token.actions.githubusercontent.com:sub:
                  - !Sub 'repo:${GitHubOrg}/${GitHubRepo}:ref:${GitHubRef}'
      Policies:
        - PolicyName: !Sub '${StackNamePrefix}-ecr-push'
          PolicyDocument:
            Version: '2012-10-17'
            Statement:
              # Needed to log in to registry
              - Effect: Allow
                Action: ecr:GetAuthorizationToken
                Resource: '*'
              # Push/pull to the specific repo
              - Effect: Allow
```

```
                Action:
                  - ecr:BatchCheckLayerAvailability
                  - ecr:CompleteLayerUpload
                  - ecr:UploadLayerPart
                  - ecr:InitiateLayerUpload
                  - ecr:PutImage
                  - ecr:BatchGetImage
                  - ecr:GetDownloadUrlForLayer
                Resource: !Sub
'arn:aws:ecr:${AWS::Region}:${AWS::AccountId}:repository/${RepositoryName}'
              # Allow CI to send SSM RunCommand to the Manager
              - Effect: Allow
                Action:
                  - ssm:SendCommand
                  - ssm:GetCommandInvocation
                  - ssm:ListCommandInvocations       # (optional)
                  - ssm:ListCommands                 # (optional)
                  - ssm:DescribeInstanceInformation  # (optional, for
debugging)
                Resource: "*"
                # (Optional tighter guard—only allow targeting instances
tagged Name=<prefix>-manager)
                Condition:
                  StringEquals:
                    ssm:resourceTag/Name: !Sub '${StackNamePrefix}-manager'
              # Allow CI to look up the Manager instance ID
              - Effect: Allow
                Action: ec2:DescribeInstances
                Resource: "*"

Outputs:
  GitHubEcrPushRoleArn:
    Value: !GetAtt GitHubEcrPushRole.Arn
```

Let us explain three concepts briefly:

- OIDC lets GitHub Actions prove its identity to AWS without long-lived keys, so the workflow can assume a role and push to ECR using short-lived creds
- GitHubEcrPushRole > Policies define the IAM-roles we allow
- OidcThumbprint can be set in the parameter-file if you want, we could also hardcode the value in the Resources section, but this way we leave this open for a more secure solution

8.3.2 Multiple file changes

We need to tweak our files a bit. I do subsections for all changes so it is easier to track. Add this to your root.yaml:

8.3.2.1 root.yaml - under "Parameters" at the top, add the following

```
  GitHubOrg:
    Type: String
```

```
    Default: mymh13
  GitHubRepo:
    Type: String
    Default: swarm-dotnet-test
  GitHubRef:
    Type: String
    Default: refs/heads/main
  OidcThumbprint:
    Type: String
    Default: 6938fd4d98bab03faadb97b34396831e3780aea1
```

Then go to the nested stacks below, let us add a nested stack definition:

8.3.2.2 root.yaml - after the EcrStack/before Outputs, add this:

```
  OidcGitHubStack:
    Type: AWS::CloudFormation::Stack
    DependsOn: EcrStack
    Properties:
      TemplateURL: !Sub
'https://s3.${AWS::Region}.amazonaws.com/${TemplateBucket}/${TemplatePrefix}26
-github-oidc.yaml'
      Parameters:
        StackNamePrefix: !Ref StackNamePrefix
        RepositoryName:  !Ref RepositoryName
        GitHubOrg:       !Ref GitHubOrg
        GitHubRepo:      !Ref GitHubRepo
        GitHubRef:       !Ref GitHubRef
        OidcThumbprint:  !Ref OidcThumbprint
```

And since you want to be able to share information:

8.3.2.3 root.yaml - add this to Outputs:

```
  GitHubEcrPushRoleArn:
    Description: ARN of the IAM Role that GitHub Actions assumes via OIDC
    Value: !GetAtt OidcGitHubStack.Outputs.GitHubEcrPushRoleArn
    Export:
      Name: !Sub '${AWS::StackName}-GitHubEcrPushRoleArn'
```

This will bubble up the ARN role output so we can fetch it from CLI.

8.3.2.4 docker-stack.yml – compose deploy section

Here we remove max_replicas_per_node as it causes issues when we redeploy or alter the timing, sometimes I ended up with only a Manager node as I only had “max 1 replica”. We alter node.id spread to node.labels.role, now with SSM we can define a role and soft spread through that.

We also add the update_config section:

```
    deploy:
      replicas: 3
      placement:
```

```
        # max_replicas_per_node: 1
        preferences:
          # - spread: node.id - optional old alternative
          - spread: node.labels.role
      update_config:
        order: start-first
        parallelism: 1
        failure_action: rollback
```

8.3.2.5 10-ec2-swarm.yaml – Manager User Data

After docker swarm init, we label the Manager node with role=manager. Workers join with SSM tokens, but do not self-label, only the Manager label nodes. Swarm can soft-spread across roles/AZs via compose preferences. Neat, huh? This is helpful, so let us alter the "Init + publish" section:

```
                # Init + publish join info
                docker swarm init --advertise-addr "$MAN_IP"
                docker node update --label-add role=manager "$(hostname)"
                AZ=$(curl -s http://169.254.169.254/latest/meta-
data/placement/availability-zone)
                docker node update --label-add az="$AZ" "$(hostname)"
                TOK=$(docker swarm join-token -q worker)
                aws ssm put-parameter --name /swarm/manager-ip  --type String
--overwrite --value "$MAN_IP" --region ${AWS::Region}
                aws ssm put-parameter --name /swarm/worker-token --type String
--overwrite --value "$TOK"     --region ${AWS::Region}
```

Remember, that is the User Data from the Manager, not Workers. Should be three new lines added to the old block you had for this section.

8.3.3 Validate, upload and update-stack

Run the regular validation, upload, and update-stack:

```
aws cloudformation validate-template --template-body
file://infra/templates/26-github-oidc.yaml
aws cloudformation validate-template --template-body
file://infra/templates/root.yaml

aws s3 cp infra/templates/26-github-oidc.yaml   "s3://${BUCKET}/${PREFIX}26-
github-oidc.yaml"
aws s3 cp infra/templates/10-ec2-swarm.yaml   "s3://${BUCKET}/${PREFIX}10-ec2-
swarm.yaml"
aws s3 cp infra/templates/root.yaml   "s3://${BUCKET}/${PREFIX}root.yaml"
aws s3 cp infra/artifacts/docker-
stack.yml   "s3://${BUCKET}/${PREFIX}artifacts/docker-stack.yml"

aws cloudformation update-stack \
  --region "$REGION" \
  --stack-name "$STACK" \
```

```
  --template-url
"https://s3.${REGION}.amazonaws.com/${BUCKET}/${PREFIX}root.yaml" \
  --parameters file://"$PARAMS" \
  --capabilities CAPABILITY_IAM CAPABILITY_NAMED_IAM
```

8.3.4 Let us add a GitHub workflow file

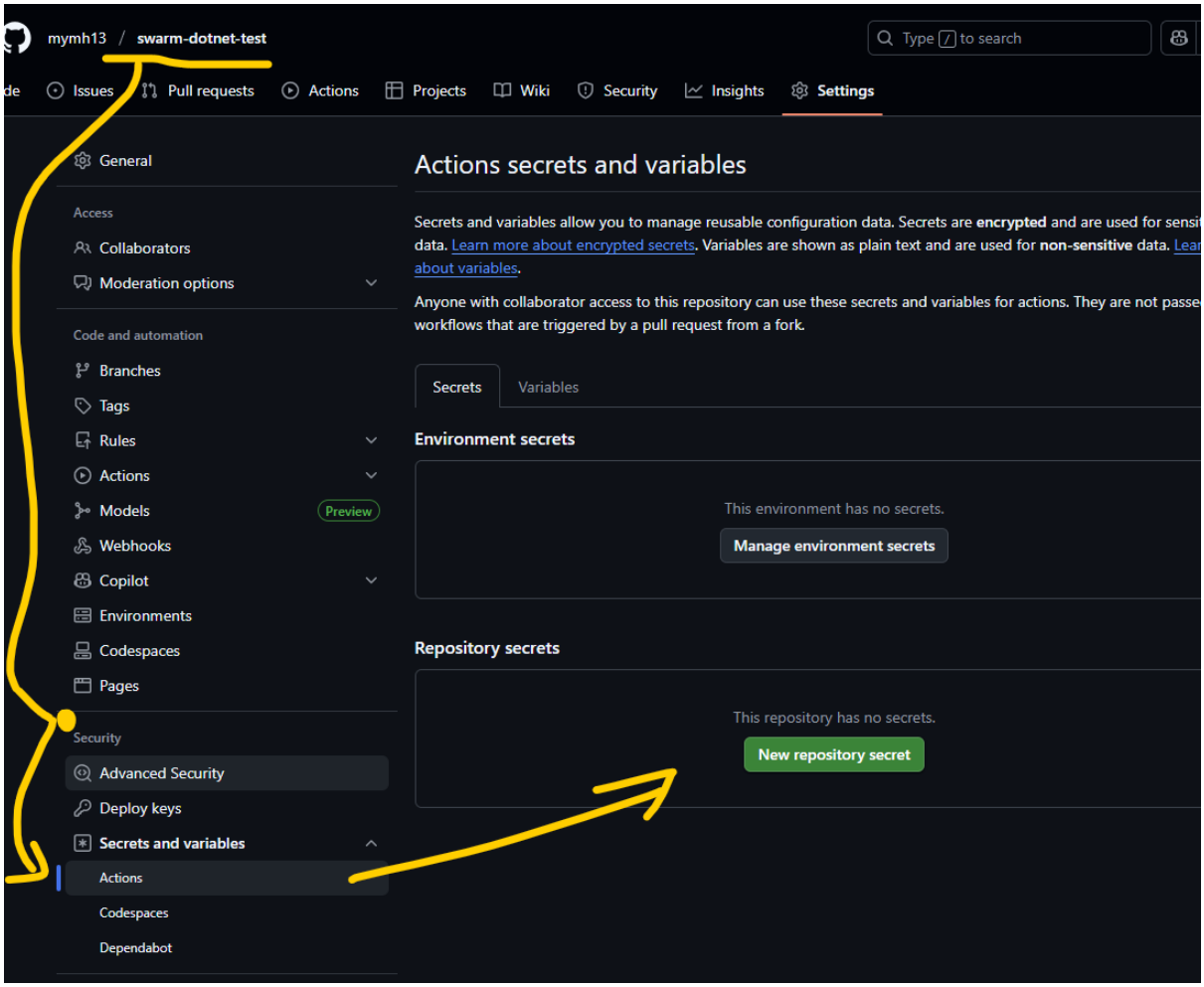
Create .github/workflows/build-and-push.yaml in the repo, and add this:

```yaml
name: Build and push Docker image to ECR

on:
  push:
    branches: [ "main" ]
  workflow_dispatch: {}

permissions:
  id-token: write
  contents: read

env:
  AWS_REGION: ${{ secrets.AWS_REGION }}
  ECR_REPOSITORY: ${{ secrets.ECR_REPOSITORY }}
  IMAGE_TAG: ${{ secrets.IMAGE_TAG }}

jobs:
  build-and-push:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Configure AWS credentials (OIDC)
        uses: aws-actions/configure-aws-credentials@v4
        with:
          role-to-assume: ${{ secrets.AWS_ROLE_TO_ASSUME }}
          aws-region: ${{ env.AWS_REGION }}

      - name: Login to Amazon ECR
        id: ecr
        uses: aws-actions/amazon-ecr-login@v2

      - name: Build and push
        uses: docker/build-push-action@v6
        with:
          context: .
          push: true
          tags: ${{ steps.ecr.outputs.registry }}/${{ env.ECR_REPOSITORY
}}:${{ env.IMAGE_TAG }}
```

I leave this pretty much un-commented: it is rather self-explanatory if you ever used with workflows before. You can see descriptive “name” sections, the rest is declaring what to run and what to use.

8.3.5 Then we need some GitHub Secrets

To secure things down a little bit, we will use GitHub Secrets and store one role and three variables.



Go to your GitHub repo > Settings > Secrets and Variables > Actions and add this:

- AWS_ROLE_TO_ASSUME = The role ARN output from the stack ** see below
- AWS_REGION = eu-west-1 in our example code, set yours here
- ECR_REPOSITORY = swarm-dotnet-app (the name we use)
- IMAGE_TAG = dev in our case, just set it to match yours

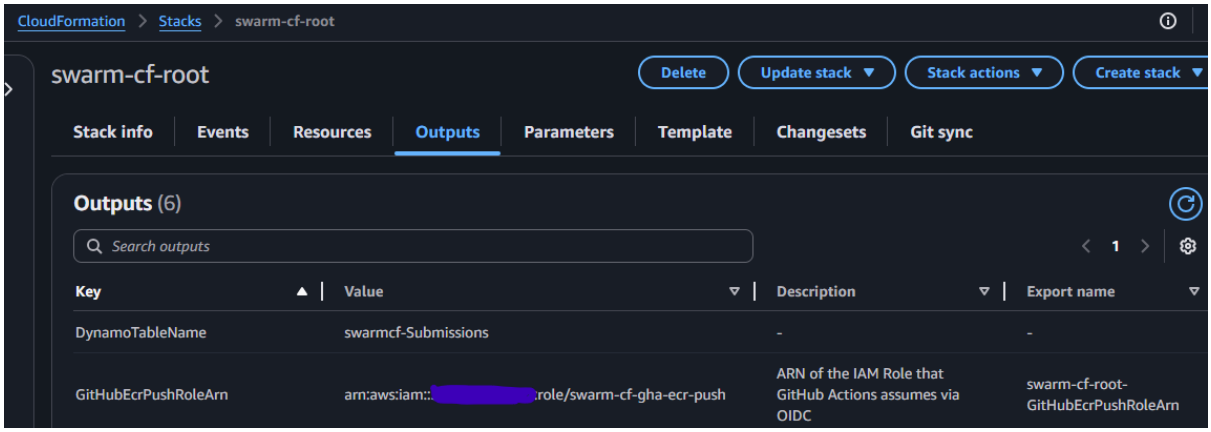
Neither of these above are super-sensitive. Why do we bother? Because having the habit to set variables and values, even if they are often the same (like a port), there can be situations where you set custom values and want to hide them. If you have the habit of always securing data, there’s less risk you miss out when it is important.

Plus: you only have to change the data in -one- place instead of all the hardcoded sections. 😊

8.3.5.1 ARN?

The ARN is the account ID plus role, you can find it like this:

Either visit AWS Console > CloudFormation > Stacks > swarm-cf-root > Outputs
You are looking for the key “GitHubEcrPushRoleArn” that we just set. It will look something like this:



The blue section is your 12-number account ID. You can see it on the top right in the AWS console. Remember to write it without using the dashes -.

Another alternative route to go to grab your ARN is through CLI:

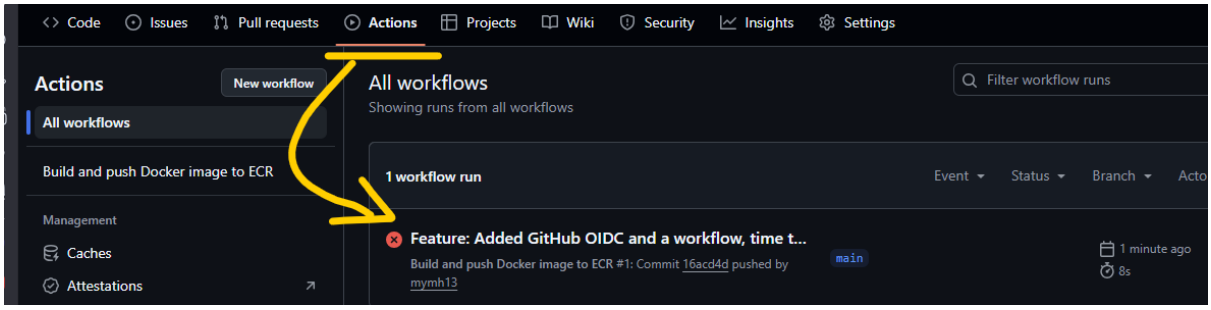
```
aws cloudformation describe-stacks \
  --region eu-west-1 \
  --stack-name swarm-cf-root \
  --query "Stacks[0].Outputs[?OutputKey=='GitHubEcrPushRoleArn'].OutputValue"
\
  --output text
```

This will return the string you need, it will be something like this:
`arn:aws:iam::<ACCOUNT-ID>:role/swarm-cf-gha-ecr-push`

Perfect, your secrets should be set, ECR repo is managed by CloudFormation, OIDC role set for GitHub Actions, workflow file ready, new stack template and root uploaded and updated.

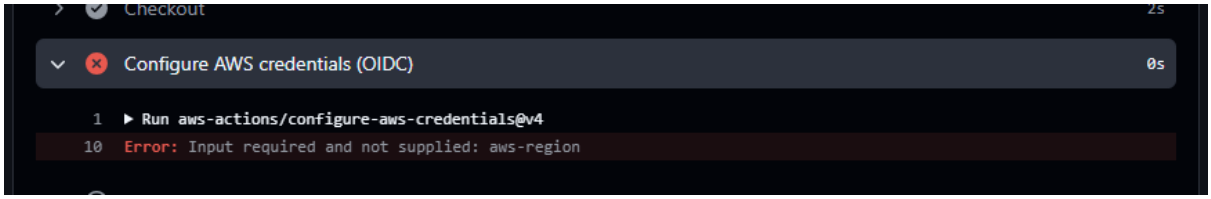
8.3.6 Run run ruuuuun the workflow, ruuun like the wind!

This step is super easy or very frustrating. We will deploy our files to our Git repo and see if the workflow works or fails. Gulp. I assume you have done this before, but let us take the rough basics: deploy to GitHub main – or merge into main. In your repo, under the Actions tab you should see what is going on:



8.3.6.1 Error handling and troubleshooting

Oops, that did not go too well. I do include a few mistakes I make in this guide, hopefully it can help you the reader to identify what happens in case you make similar mistakes. Click the commit name, then click the workflow file name (build-and-push) and you will get a sequential list where it runs its operations. You will see immediately where it failed. Mine was obvious:



Easy solution to that one. The problem was I forgot to update my deploy file from "vars.AWS_REGION" to "secrets.AWS_REGION" when we created the GitHub Secrets. So, the workflow is reading repo variables (vars.*) instead of secrets. I have fixed this in your template above, so this should not have been an issue for you / the reader. But it shows how you can do error handling if they should arise. Check logs, logs are your best friend.

8.3.6.1 Error handling re: versioning

I had a point further down in this document,called "versioning", basically I was looking at systems for version control of the image, because:

- Problem: When Swarm see "same tag, same reference" (like when we push a new image but do not change the tag), it assume no changes so there will be no pull on workers
- Solution: changing IMAGE_TAG: secrets.IMAGE_TAG to IMAGE_TAG: github.sha means our tag is automatically set to the commit SHA of whatever we push to main

SHA stands for Secure Hash Algorithm and it is GitHubs unique cryptographic hash that is generated for each commit we do. We only have to make two changes to build-and-push.yaml –

```
  # replace this IMAGE_TAG: ${{ secrets.IMAGE_TAG }}
  # with this:
  IMAGE_TAG: ${{ github.sha }}
```

You do not have to remove the IMAGE_TAG from the Secrets, it can sit there. But we also need to adjust the last section in this document, the update of Swarm via SSM. Replace with this:

```
      - name: Update Swarm service to new image via SSM
        run: |
          FULL_IMAGE="${{ steps.ecr.outputs.registry }}/${{ env.ECR_REPOSITORY
}}:${{ env.IMAGE_TAG }}"
          aws ssm send-command \
            --region "${{ secrets.AWS_REGION }}" \
            --targets "Key=tag:Name,Values=${{ vars.STACK_NAME_PREFIX }}-
manager" \
            --document-name "AWS-RunShellScript" \
            --comment "CI: set image tag to new build; Swarm will pull and
roll" \
            --parameters "commands=[\"docker service update --image
${FULL_IMAGE} --with-registry-auth --update-order start-first --update-
parallelism 1 myappname_web\"]"
```

Now your deploys should have a version control system in place tied to your tags.

8.4 CI/CD-Summary

Goal: build > push > roll out a new app version with no SSH or manual steps.

Checklist

1. Enable SSM and attach AmazonSSMManagedInstanceCore to the EC2 role so instances become Managed
2. Add 26-github-oidc.yaml (nested) to create:
    a. OIDC provider (token.actions.githubusercontent.com)
    b. IAM role the workflow assumes (ECR push + SSM send-command)
3. Update root.yaml (parameters, nested stack include, output ARN). Upload and update-stack
4. Commit .github/workflows/build-and-push.yaml
5. Add GitHub Secrets: AWS_ROLE_TO_ASSUME, AWS_REGION, ECR_REPOSITORY, IMAGE_TAG
6. Push to main > workflow builds & pushes the image to ECR
7. (Optional) Use SSM from CI to run docker service update --force myappname_web on the manager for a zero-SSH rolling update

Why this works: ***GitHub Actions authenticates with short-lived OIDC creds (no long-lived keys), ECR stores the image, and SSM lets us refresh the Swarm service remotely. The manager's UserData pulls the compose from S3 and substitutes the ECR image automatically on first boot.***

The above section is really the entire essence of our workflow.

9. Let us Formalize the .NET app

For the final part of Section 1, we will add a basic webform to the app, add rudimentary security measures, and tie this to the DynamoDB.

10. Bring on Section 2! - Introduction

The above completes the task we were assigned to do. For the following sections, we will look at the following:

1. Include a Lambda and a serverless functionality
2. We need to secure that we are working with HTTPS rather than HTTP

Then after that we have a section "room for improvement"

11. Room for improvement

Things we can consider, to take this system to new heights (or secure it better):

- VPC Gateway Endpoints for DynamoDB (keep traffic in AWS backbone)
- .NET wants a form / file handling
- HTTPS – CloudFront/S3?
- ALB/ASG and a Bastion Host
- Private VPC, routing tables and multiple AZs
- Replacing DynamoDB with a proper DB or a storage contained in an Instance, should be more cost effective?
- Terraform instead of CloudFormation?
- Versioning (history, rolling updates)
- Obviously: CloudWatch, logging and metrics

Lorum Ipsum ad Infinitum

X. References

X.1 educ8.se

The basics of this guide is based on study material and tutorials handed out during the Cloud Developer YH-program (school of applied knowledge) at Campus Mölndal in 2025.

Copyright to that material belongs to our teacher Lars Appel, this guide has been modified (and I have taken personal design choices) to reflect this. Lars' guides are more handing out the basics, this one will try to explain why we chose what. See more at: https://educ8me.se

X.2 LLM support

I did use LLMs, not to write this text, it is my words, my explanations, and my architecture/solution with the nested stacks setup in CF.

However, when I converted the manual setup from section 2 to CloudFormation, I asked it to give me a template skeleton for CloudFormation to match the sections I wrote in part 2. I have built on that, but that saved me quite a lot of time of writing CF-templates.

X.3 ASP.NET MVC at Microsoft Learn

Link to Microsoft Learns guide on ASP.NET MVC: https://learn.microsoft.com/en-us/aspnet/mvc/