Index

Project: Scalable AWS Infrastructure for Containerized and Serverless Web Applications

1. Introduction

This project demonstrates a scalable AWS infrastructure that combines containerized web applications with serverless processing capabilities. The solution uses Docker Swarm for container orchestration, CloudFormation for Infrastructure as Code, and Lambda functions for serverless data processing, deployed through automated CI/CD pipelines.

1.1 The entire project can be found at GitHub

Made you look! https://github.com/mymh13/scaleable_aws_turn-in-2

The project builds on an extensive tutorial (link at paragraph 11.2) I wrote. There I describe how you can set up a Docker Swarm project on AWS and then convert it to a CloudFormation Nested stacks solution, plus add a .NET web application. Because of that extensive in-depth guide, I decided to deliberately keep this project more as an overview.

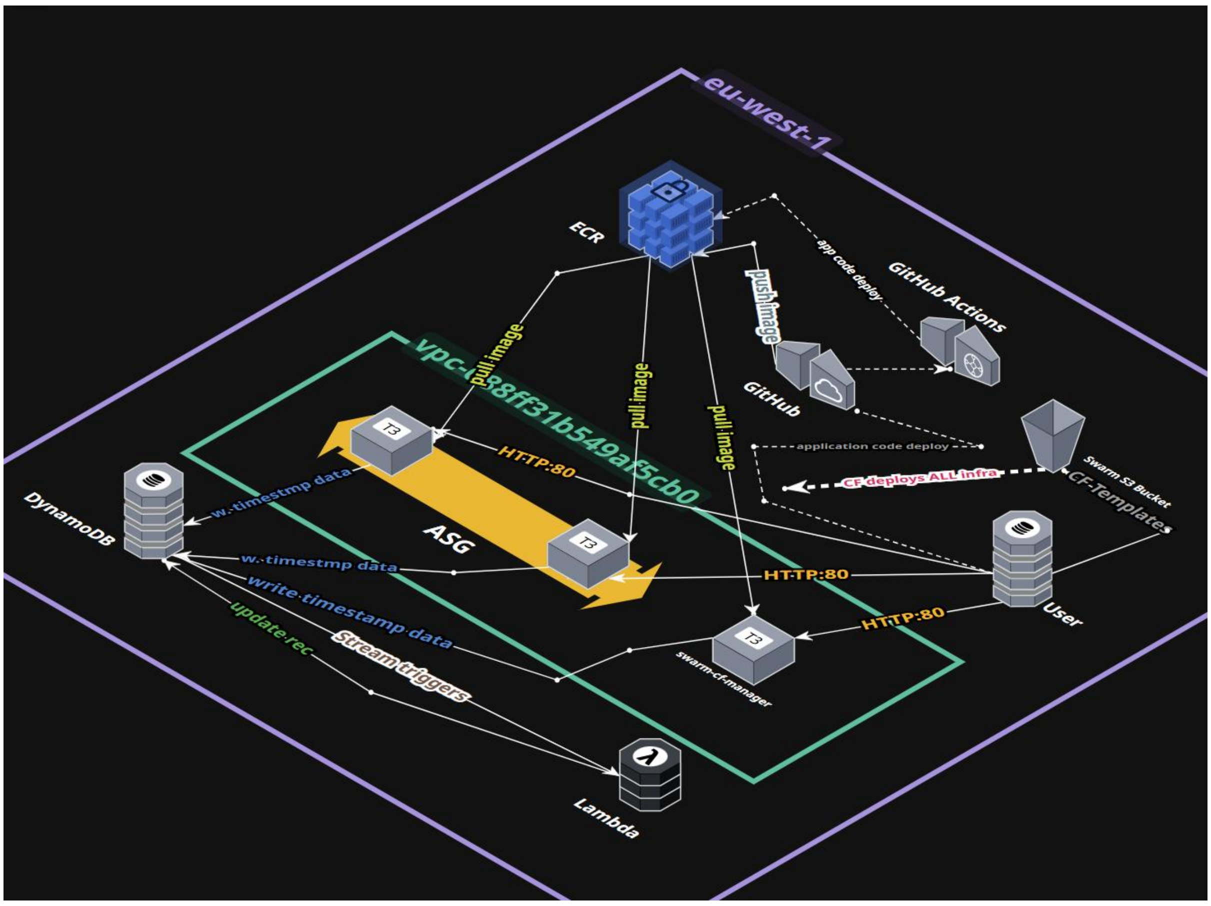
1.2 Prerequisites (knowledge, accounts and software installs)

To replicate this system, you are expected to know and have:

- Basic understanding of AWS, Docker, Git and using shell commands (Bash)
- AWS account with IAM-user and necessary policies set up (ex. S3)
- AWS-CLI installed and configured to be run (I used version 2.28.8)

2. Overview of the project

This section provides a high-level overview of the complete system architecture and its key components. Image provided that gives an overview of the infrastructure and service flow:



The following subsections break down each major element of the infrastructure, from AWS services to deployment pipelines.

2.1 Choices, choices.. why when what how?

A rough overview would be this: Normally I will look at what problem a system solves, “what do we want to accomplish with this”, and try to build a minimalistic but scalable system that fits that bill. Then I build the infrastructure and the IaC basics for smooth deployment and testability when we move over to the actual application (that is always the last step).

This time however, I put some constraints which I had to follow: I wanted to build on the Swarm-to-CF-Nested-Stacks-tutorial I had written. This means I already had an infrastructure that needed to be adapted, it even had a simple .NET webform. That is more complicated.

My approach to this was to simply add one small feature at a time, and test it well, step by step. I wanted to add a serverless component and decided to convert the .NET webform to be a button storing a timestamp and ID so we could verify that our infrastructure is scalable – and the Lambda then was a perfect “verification tool” that would acknowledge the above.

I also wanted to introduce GitHub Actions for automated deployment. This required significant changes to the existing workflow, so I focused specifically on automating the Docker container build and ECR push processes while maintaining the CloudFormation deployment as a separate, controlled step.

2.2 General overview: Infrastructure

The infrastructure consists of a multi-tier AWS environment that combines EC2-based Docker Swarm-clusters with serverless Lambda functions, all managed through CloudFormation Nested Stacks pattern.

DynamoDB provides persistent storage and Lambda handles background processing tasks. This hybrid approach allows for both predictable containerized workloads and event-driven serverless processing, creating a flexible and scalable hosting environment.

2.3 General overview: AWS services

The solution leverages several key AWS services:

- EC2 instances provide the foundation for Docker Swarm nodes
- ASG (Auto Scaling Groups) enable automatic horizontal scaling based on demand
- DynamoDB serves as the managed NoSQL-database for persistent storage
- ECR (Elastic Container Registry) hosts the container images
- Lambda functions handle serverless processing tasks
- IAM roles and policies provide access control throughout the infrastructure
- SG (Security Group) act as virtual firewalls controlling in-/outbound traffic
- S3 bucket stores CloudFormation templates and deployment artifacts

2.4 General overview: CloudFormation + Nested Stacks pattern

The entire infrastructure is defined as Infrastructure as Code using CloudFormation (CF) Nested Stack. CF being an AWS tool, Nested Stacks being a specific pattern where a root template orchestrates multiple specialized child templates.

Each nested stack focuses on a specific component (SG, EC2, IAM, etc), promoting modularity and reusability while managing dependencies between resources.

This approach enables version-controlled infrastructure deployments and simplifies updates by isolating changes to specific stack components, though it has its risks too which I will elaborate in section 5.

2.5 General overview: CI/CD and GitHub Actions

The CI/CD pipeline is implemented using GitHub Actions, which automates the container build and deployment process whenever code changes are pushed to the repository (*main* branch).

The workflow uses GitHub's OIDC (OpenID Connect) integration with AWS to securely authenticate without storing long-lived credentials. Then it builds the .NET-application into a Docker image and pushes it to Amazon ACR.

This automation ensures consistent deployments and reduces manual intervention, though the CloudFormation infrastructure deployment remains a separate, controlled process to maintain stability (like having separate Back- and Front-end deployments).

2.6 General overview: Swarm + container setup driving the .NET

Docker Swarm orchestrates the containerized .NET MVC application across multiple EC2 Instances, providing high availability and load distribution.

The .NET application is packaged as a Docker container and deployed as a Swarm service, allowing for automatic scaling and failover capabilities when nodes become unavailable.

The Swarm cluster automatically distributes containers across available nodes and maintains the desired numbers of replicas, while the ALB routes external traffic to healthy container instances regardless of which EC2 node they're running on.

2.7 General overview: Lambda processor and its role

The Lambda function serves as the serverless component of the architecture, acting as a verification and processing tool for the containerized application's data.

When users submit timestamps through the .NET web application, the data is stored in DynamoDB and can trigger Lambda functions for background processing, validation, or acknowledgement tasks.

This serverless approach demonstrates how containerized and serverless architecture can work together, with Lambda providing event-driven processing capabilities that complement the always-running Docker Swarm containers.

2.8 General overview: LLM in this project

LLM has been used throughout this project primarily in two ways:

Either as a co-developer where I am discussing techniques or pattern alternatives. One example: I wanted to introduce the Lambda as an "announcer" that a timestamp had been recorded, and here I asked about ways to use Lambda for this kind of behaviour as I never used the Lambda before.

Or I have used LLMs to help me structure up sections where I lay down the general architecture, then ask it to fill the gaps. For example: I had the Lambda inside the same template as the DynamoDB but realized there was a conflict (more on that further on), so I asked the LLM to break out the Lambda part and make sure that references through the templates stayed intact. I read through the code before applying it.

2.9 The system in action

An overview of the natural flow within the system:

2.9.1 CloudFormation deployment flowchart

1. Developer runs aws cloudformation `create-stack` with root.yaml
2. Root template uploads to S3 and initiates nested stack creation
3. Security Groups (00-sg-swarm.yaml) > IAM Roles (20-iam-ec2-role.yaml) > ECR (25-ecr.yaml)
4. GitHub OIDC (26-github-oidc.yaml) > DynamoDB (30-dynamodb.yaml) > Lambda (35-lambda-processor.yaml)
5. EC2 instances (10-ec2-swarm.yaml) provisions with dependencies
6. Stack creation completes

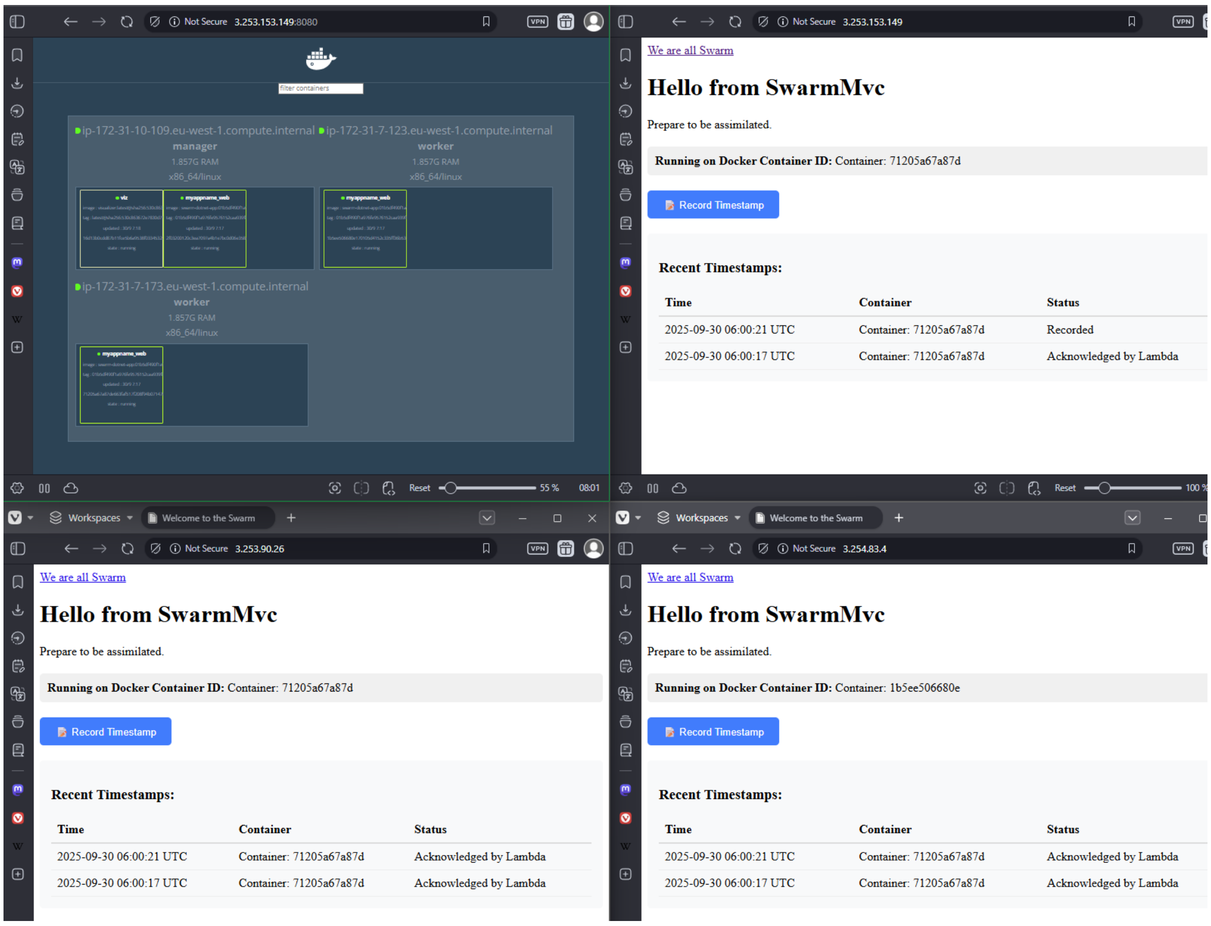
2.9.2 CI/CD push flowchart

1. Developer push or merge code to GitHub repository main branch
2. GitHub Actions workflow triggers on push event
3. Workflow authenticates to AWS using OIDC role
4. .NET application builds inside Docker container
5. Container image pushes to ECR repository
6. Docker Swarm pulls updated image and redeploys service

2.9.3 User interaction flowchart

1. User clicks "Record Timestamp" button on web interface
2. .NET application captures timestamp + container ID
3. Data saves to DynamoDB table "swarm-cf-Submissions"
4. Lambda function processes new record (acknowledgment/validation)
5. Page refreshes, displays updated timestamp list from DynamoDB
6. User sees their submission with "Acknowledged by Lambda" status

<u>Example below</u>: showing Docker Visualizer top left (if you zoom in you can see ID and match with the other tabs). Top right is the Manager public IP, bottom is Workers.



We are all Swarm

Hello from SwarmMvc

Prepare to be assimilated.

Running on Docker Container ID: Container: 71205a67a87d

Record Timestamp

Recent Timestamps:

Time	Container	Status
2025-09-30 06:00:21 UTC	Container: 71205a67a87d	Recorded
2025-09-30 06:00:17 UTC	Container: 71205a67a87d	Acknowledged by Lambda

We are all Swarm

Hello from SwarmMvc

Prepare to be assimilated.

Running on Docker Container ID: Container: 71205a67a87d

Record Timestamp

Recent Timestamps:

Time	Container	Status
2025-09-30 06:00:21 UTC	Container: 71205a67a87d	Acknowledged by Lambda
2025-09-30 06:00:17 UTC	Container: 71205a67a87d	Acknowledged by Lambda

We are all Swarm

Hello from SwarmMvc

Prepare to be assimilated.

Running on Docker Container ID: Container: 1b5ee506680e

Record Timestamp

Recent Timestamps:

Time	Container	Status
2025-09-30 06:00:21 UTC	Container: 71205a67a87d	Acknowledged by Lambda
2025-09-30 06:00:17 UTC	Container: 71205a67a87d	Acknowledged by Lambda

There is a reason I chose to illustrate the system using container ID: this single variable is like the glue that ties together the various systems: the scalable instances and containers, the Lambda vs .NET-functionality, the Swarm and its role in orchestrating, etc.

3. Infrastructure in-depth

The infrastructure deploys EC2 instances (t3.small) running Amazon Linux with Docker, forming a Docker Swarm cluster with one manager and multiple scalable worker nodes managed by Auto Scaling Groups (min: 2, max: 10, desired: 2).

Security Groups restrict access to HTTP (80), SSH (22), and Docker Swarm internal ports (2377, 7946, 4789) from specified CIDR blocks. The Docker containers communicate internally on overlay networks and expose services through Docker Swarm's ingress load balancing directly to the EC2 instances.

DynamoDB tables use on-demand billing with DynamoDB Streams enabled. Lambda functions (Python 3.9 runtime) trigger via DynamoDB Streams with IAM execution roles providing least-privilege access. ECR repositories store container images with lifecycle policies for automated cleanup of untagged images.

GitHub OIDC integration eliminates long-lived credentials by using temporary STS tokens with assume-role permissions. CloudFormation Nested Stacks manage dependencies through cross-stack references and outputs, enabling modular infrastructure updates while maintaining referential integrity across resources.

This infrastructure uses Docker Swarms built-in ingress load balancing instead of AWS LAB, for a small-scale simple setup this is a valid solution. On a bigger scale and in an enterprise setup we might want to consider an external ALB.

3.1 Verification and troubleshooting: Infrastructure

Common issues and where to look:

- Stack creation fails - Check CloudFormation Events tab for specific resource errors, verify parameter values in dev.params.json
- EC2 instances fail to join swarm - Check SSM Parameter Store for /swarm/manager-ip and /swarm/worker-token, review EC2 instance logs via Systems Manager Session Manager
- Docker service not accessible - Verify Security Group rules allow port 80, check if Docker service is running with docker service ls on manager node
- Lambda not processing records - Check CloudWatch Logs for Lambda function, verify DynamoDB Streams are enabled and EventSourceMapping is active (see 10.1)
- GitHub Actions build fails - Verify OIDC role exists and has ECR push permissions, check GitHub repository secrets and workflow logs
- Container deployment issues - Check ECR repository exists and contains images, verify Docker Swarm can pull from ECR with correct IAM permissions
- Network connectivity problems - Confirm VPC/subnet configuration, verify Security Group self-referencing rules for Swarm ports (2377, 7946, 4789)

4. AWS services in-depth

Elaborating a bit on the different AWS services used:

EC2: Serves as the compute (the workforce) foundation for the Docker Swarm cluster. The manager node coordinates the cluster and publishes join credentials, while worker nodes automatically discover and join the swarm. All nodes run the containerized .NET application and handle user requests.

Auto Scaling Group: Provides horizontal scaling capability for worker nodes based on demand. Ensures the desired number of worker instances are always available and automatically replaces failed nodes, maintaining cluster resilience without manual intervention.

DynamoDB: Acts as the persistent data store for timestamp records submitted through the web application. Each record contains timestamp, container ID, and processing status. Streams feature enables real-time Lambda processing when new records are added.

ECR: Functions as the container registry for the .NET application images. GitHub Actions builds and pushes new images here, which Docker Swarm then pulls during service updates. Provides versioned storage and secure access to container artifacts.

Lambda: Implements the serverless processing component that responds to DynamoDB changes. When users submit timestamps, Lambda automatically processes these records and updates their status, demonstrating event-driven architecture alongside the containerized components.

IAM: Enables secure communication between all services without hardcoded credentials. EC2 instances can access ECR and SSM, Lambda can modify DynamoDB records, and GitHub Actions can push to ECR - all through role-based permissions.

Security Groups: Control network access at the instance level, allowing public HTTP traffic while restricting SSH access to specific IP ranges. Internal Swarm communication ports are secured through self-referencing rules within the security group.

S3: Hosts the CloudFormation templates during deployment, enabling the nested stack pattern where the root template references child templates stored in S3 buckets.

4.1 Verification and troubleshooting: AWS

- IAM permission errors > Verify EC2 instance profile has SSM and ECR permissions, check Lambda execution role can access DynamoDB table and streams
- ECR access denied > Confirm GitHub OIDC role has ECR push permissions, verify EC2 instances can assume role for ECR pulls
- DynamoDB stream not triggering Lambda > Check EventSourceMapping is active and Lambda has stream read permissions on table ARN
- SSM Parameter Store access fails > Ensure EC2 instance profile includes SSM parameter read/write permissions for /swarm/* paths
- Security Group blocking traffic > Verify HTTP port 80 allows 0.0.0.0/0, SSH restricted to your IP, Swarm ports allow self-referencing

- S3 template access issues > Confirm CloudFormation can read templates from S3 bucket, check bucket policies if using custom S3 bucket

5. CloudFormation and Nested Stacks in-depth

Infrastructure as Code (IaC)-implementation using AWS CloudFormation's nested stack architecture for modular, maintainable deployments.

5.1 CloudFormation

CloudFormation enables IaC by defining AWS resources in YAML templates that can be version-controlled, reviewed, and deployed consistently. Templates describe the desired state of infrastructure, and CloudFormation handles the creation, updating, and deletion of resources in the correct order based on dependencies.

This approach eliminates manual resource provisioning, reduces configuration drift, and ensures reproducible deployments across environments. The declarative nature means you specify what you want, not how to achieve it, while CloudFormation manages the underlying API calls and handles rollback scenarios when deployments fail.

5.2 Nested Stacks pattern

The nested stack pattern breaks down complex infrastructure into smaller, focused templates that are orchestrated by a root template. Each child template handles a specific domain (SG, IAM roles, etc) and can be developed, tested, and updated independently.

The root template coordinates dependencies by passing outputs from one stack as parameters to another, creating a dependency graph that CloudFormation respects during deployment. Cross-stack references ensure resources are created in the correct order - for example, DynamoDB must exist before Lambda can reference its stream ARN. As we can see.

5.3 Verification and troubleshooting: CF

- Template dependency issues > Carefully order nested stack creation - DynamoDB before Lambda, IAM roles before EC2 instances. Review stack outputs and parameter dependencies in root template
- Update-stack failures > Nested stacks often require delete-stack followed by create-stack instead of update-stack, especially when changing resource types or dependencies. This creates significant deployment time overhead and is hard to troubleshoot as there is nothing “wrong” with the settings!

- Parameter passing errors > Verify outputs from child stacks match parameter expectations in dependent stacks. Check parameter types and naming consistency
- Stack rollback scenarios > When one nested stack fails, entire deployment rolls back. Check CloudFormation Events tab in failed stack to identify root cause
- Resource limit conflicts > Ensure no naming conflicts between stacks, verify IAM role names and resource tags are unique across nested deployments

Brief: CloudFormation > Stacks > root stack and Event-log tab is your best friend.

Refer to the guide at https://github.com/mymh13/swarm-dotnet-test for in-depth tutorial on how to set up a core Swarm system using a Nested Stacks pattern.

6. CI/CD in-depth

The CI/CD pipeline automates the application deployment process using GitHub Actions, which triggers on code pushes (or merge, to main branch) to build, containerize, and deploy the .NET application. The workflow authenticates to AWS using OpenID Connect (OIDC), eliminating the need for long-lived access keys stored as secrets.

When code changes are pushed, GitHub Actions builds the .NET application inside a Docker container, tags it appropriately, and pushes the image to Amazon ECR. The Docker Swarm cluster then pulls the updated image and performs a rolling update of the service, ensuring zero-downtime deployments.

This approach separates application deployment (automated via CI/CD) from infrastructure provisioning (manual CloudFormation deployment), providing controlled infrastructure changes while enabling fast application iterations. The OIDC integration creates temporary credentials that enhance security by providing just-in-time access with limited permissions: specifically tailored for ECR operations.

6.1 Verification and troubleshooting

- GitHub Actions authentication fails > Verify OIDC provider exists in AWS IAM, check GitHub repository has correct AWS account ID and role ARN configured
- ECR push permission denied > Confirm OIDC role has ECR GetAuthorizationToken, BatchCheckLayerAvailability, and BatchGetImage permissions for the specific repository
- Docker build failures > Check Dockerfile syntax, verify .NET project builds locally, review GitHub Actions logs for missing dependencies or build context issues
- Image push timeouts > Large images may exceed GitHub Actions timeout limits, consider multi-stage builds to reduce image size or increase workflow timeout
- Swarm service update delays > Docker Swarm pulls images on-demand, verify EC2 instances have ECR access and sufficient bandwidth for image downloads

- Workflow not triggering > Check GitHub Actions workflow file syntax, verify push events target correct branches, confirm workflow file is in .github/workflows/ directory – and something I did that was GREAT was to add a log step in the workflow file! If you make sure it prints out the various steps it goes through, you will have an excellent logfile to dig through, easily visible at your Repo!

7. Docker Swarm, container and .NET in-depth

Docker Swarm orchestrates the containerized .NET MVC application across multiple EC2 instances, providing high availability and automatic load distribution. The Swarm cluster consists of one manager node that handles cluster coordination and multiple worker nodes that run application containers.

The .NET application is built using a multi-stage Dockerfile that compiles the application and creates a lightweight runtime image based on the ASP.NET Core runtime. When deployed as a Docker service, Swarm automatically distributes container replicas across available nodes and maintains the desired replica count even if individual nodes fail.

The application uses Docker's overlay networking to communicate between containers and leverages Swarm's ingress load balancing to distribute incoming HTTP requests across all healthy container instances.

Each container runs independently but shares the same DynamoDB backend, allowing users to see their timestamp submissions regardless of which container processed their request. This demonstrates how containerized applications can achieve both horizontal scaling and fault tolerance while maintaining state consistency through external data stores.

7.1 Verification and troubleshooting

- Swarm initialization fails > Check if Docker daemon is running, verify network connectivity between manager and worker nodes, ensure required ports (2377, 7946, 4789) are open
- Worker nodes fail to join > Verify SSM Parameter Store contains manager IP and join token, check worker UserData execution logs, confirm Security Group allows Swarm ports
- Service deployment issues > Check if ECR image exists and is accessible, verify service definition in docker-stack.yml, ensure sufficient resources on worker nodes
- Container networking problems > Verify overlay network creation, check if ingress load balancer is distributing traffic, test direct container access via node IP
- Application not responding > Check container logs for .NET runtime errors, verify DynamoDB connectivity from containers, ensure environment variables are properly set

- Service scaling problems > Verify Auto Scaling Group can launch new instances, check if new workers successfully join swarm, monitor resource utilization on existing nodes

Refer to the guide at https://github.com/mymh13/swarm-dotnet-test for in-depth tutorial on how to set up a core Swarm system.

8. Lambda processor in-depth

The Lambda function serves as the serverless processing component that bridges the containerized application with event-driven architecture. Written in Python 3.9, it responds to DynamoDB Streams events whenever new timestamp records are inserted by the .NET application.

The function operates in two modes: stream processing for production workflow and manual testing for validation. When triggered by DynamoDB Streams, it processes INSERT events for timestamp records, updating them with acknowledgment status and processing timestamps. This demonstrates how serverless functions can augment containerized applications by providing event-driven processing without requiring additional infrastructure.

The Lambda uses IAM roles for secure access to DynamoDB, following least-privilege principles with permissions only for the specific table and its streams. The event source mapping configuration ensures reliable message processing with configurable batch sizes and error handling.

Personally: I found the Lambda system with “pk” and “sk” etc confusing, I like the SQL mechanics with clear keys. This can be confusing, and I did one or two small errors. My recommendation here would be to take the normal incremental small step approach.

I feel I have only really tapped into the real potential here though, having built-in-event-driven tools like this is more secure than running custom client-side JavaScript event handling. It is nice that the Lambda operates in a layer between the Front- and Backend, I quite like it for the tiny feature I used it for.

8.1 Verification and troubleshooting

- Lambda not triggering on DynamoDB changes > Verify DynamoDB Streams are enabled, check EventSourceMapping status is active, confirm Lambda has stream read permissions
- Function execution errors > Check CloudWatch Logs for runtime exceptions, verify TABLE_NAME environment variable is set correctly, ensure DynamoDB table exists
- Permission denied errors > Confirm Lambda execution role has DynamoDB table and stream permissions, verify IAM policy includes correct resource ARNs

- Stream processing delays > Check EventSourceMapping batch size and parallelization settings, monitor Lambda concurrent executions and throttling metrics
- Manual test invocations fail > Verify Lambda function code handles both stream events and manual test scenarios, check function timeout settings for scan operations
- Record update failures > Ensure Lambda has UpdateItem permissions on DynamoDB table, verify record key structure matches expected pk/sk format

9. Where to go (improvement possibilities)

There are quite a lot of roads leading to Rome, but let us look at a few..

9.1 Security

Strengths: IAM roles with least-privilege access, IMDSv2 enforcement on EC2 instances, Security Groups with restricted SSH access, GitHub OIDC eliminating long-lived credentials, GitHub Secrets guarding credentials.

Improvements: Implement HTTPS with SSL/TLS certificates, add Bastion Host for secure SSH access instead of direct internet exposure, enable VPC Flow Logs for network monitoring, and as long as I have a simple button (or webform) I really should protect against traffic.

9.2 CI/CD analysis

Strengths: Automated container builds and ECR deployment, secure OIDC authentication, zero-downtime rolling updates.

Improvements: Integrate CloudFormation deployment into CI/CD pipeline for full infrastructure automation, add automated testing stages (unit tests, security scans), implement blue-green deployment strategies. Consider Terraform/Ansible as a long-term and not platform-dependant solution.

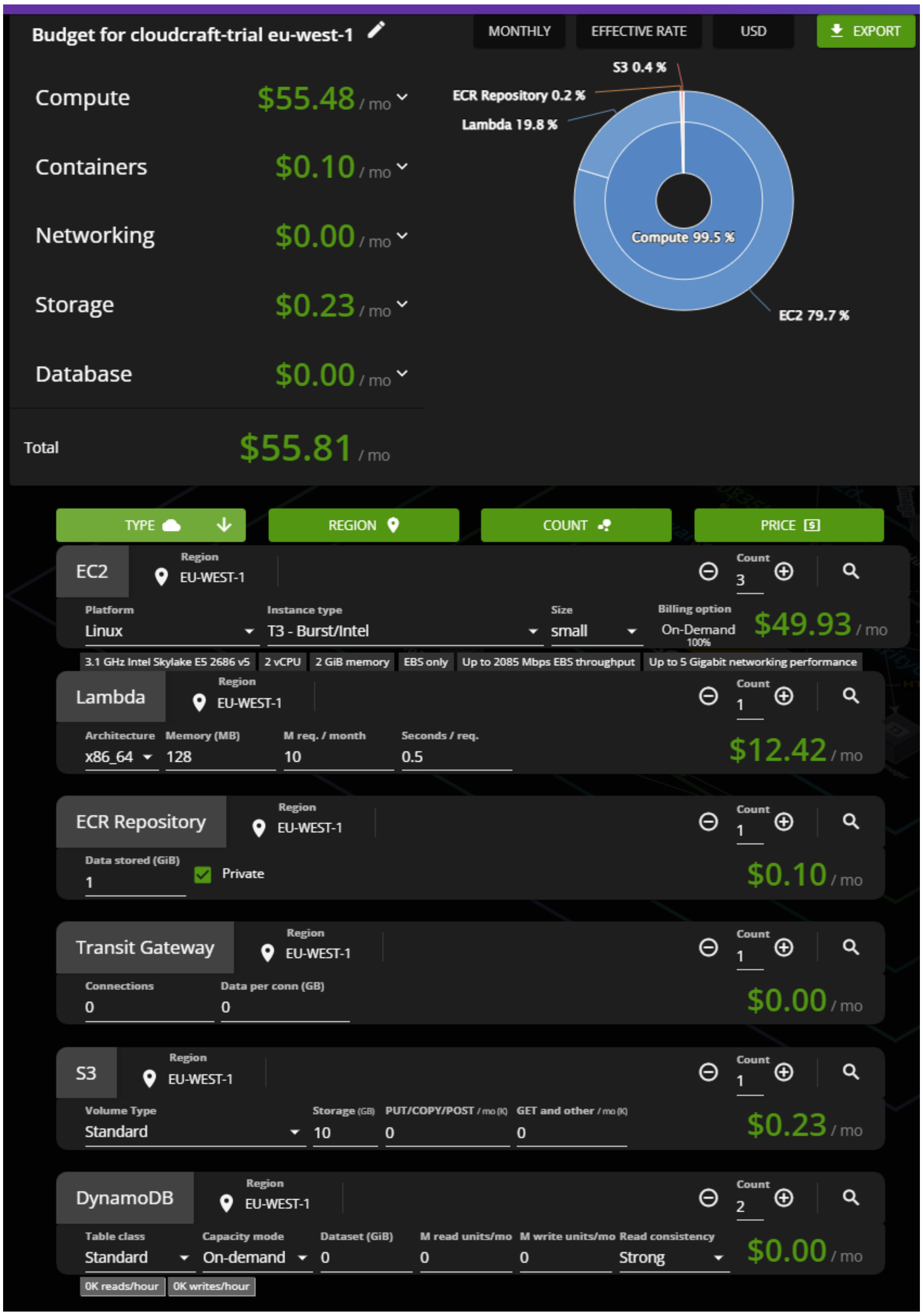
9.3 Scalability analysis

Strengths: Auto Scaling Groups handle horizontal scaling, Docker Swarm distributes load across nodes, DynamoDB scales automatically.

Limitations: Single manager node creates potential bottleneck, no multi-AZ deployment, lack of Application Load Balancer limits advanced routing options, Docker Swarm less feature-rich than Kubernetes for complex scaling scenarios.

9.4 Cost optimization possibilities

Current costs: EC2 instances run continuously (biggest expense), DynamoDB on-demand pricing, ECR storage costs. We can take down EC2 via delete-stack but that means they are unavailable instead.



Budget for cloudcraft-trial eu-west-1

Compute	$55.48 / mo
Containers	$0.10 / mo
Networking	$0.00 / mo
Storage	$0.23 / mo
Database	$0.00 / mo
Total	$55.81 / mo

EC2 — Region EU-WEST-1 — Count 3
Platform Linux; Instance type T3 - Burst/Intel; Size small; Billing option On-Demand 100% — $49.93 / mo
3.1 GHz Intel Skylake E5 2686 v5 | 2 vCPU | 2 GiB memory | EBS only | Up to 2085 Mbps EBS throughput | Up to 5 Gigabit networking performance

Lambda — Region EU-WEST-1 — Count 1
Architecture x86_64; Memory (MB) 128; M req. / month 10; Seconds / req. 0.5 — $12.42 / mo

ECR Repository — Region EU-WEST-1 — Count 1
Data stored (GiB) 1; ✓ Private — $0.10 / mo

Transit Gateway — Region EU-WEST-1 — Count 1
Connections 0; Data per conn (GB) 0 — $0.00 / mo

S3 — Region EU-WEST-1 — Count 1
Volume Type Standard; Storage (GB) 10; PUT/COPY/POST / mo (K) 0; GET and other / mo (K) 0 — $0.23 / mo

DynamoDB — Region EU-WEST-1 — Count 2
Table class Standard; Capacity mode On-demand; Dataset (GiB) 0; M read units/mo 0; M write units/mo 0; Read consistency Strong — $0.00 / mo
0K reads/hour | 0K writes/hour

Optimizations: Use Spot instances for worker nodes, implement scheduled scaling during low-traffic periods, optimize container images to reduce ECR storage, consider Reserved Instances for predictable workloads.

9.5 CloudWatch logs and metrics

Essential missing component for production readiness. Should implement centralized logging for all EC2 instances, container logs, and Lambda executions. Add custom metrics for application performance, set up CloudWatch alarms for system health, and create dashboards for operational visibility.

This would be critical for troubleshooting and performance optimization. It would be the very first implementation I would add if I was to build further on this project.

9.6 Other areas that might be worth considering

Backup/disaster recovery strategy, compliance/audit logging, container image vulnerability scanning. Protection against DDOS (Denial of Service, overflow of traffic) like I said before, this system is quite vulnerable against it because it has that exposed button that does not even have a limit on how often it can be clicked.

10. Challenges and lessons learned

Already knew this, but it is worth saying for a third time in this text: setting up the IaC, the infrastructure and the CI/CD first, before building the actual application, means the world.

10.1 Streams vs EventSourceMapping

I had some issues with this this particular topic. I could interact manually both with Streams and Event Source, but as soon as I tried to implement this into the template, I got stack deploy errors. Eventually I figured out this is a timing issue:

Lambda is dependant on the DynamoDB in this case, so my solution was that I simply split the template into two and made sure to deploy DynamoDB before Lambda.

10.2 Mixing cloud- and shellscript in User Data

The EC2 UserData scripts presented several challenges when combining cloud-init directives with shell scripting. Initially, metadata service calls failed because I was using the older IMDSv1 approach instead of the required IMDSv2 token-based system. The solution involved proper token retrieval:

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

```
WORKER_IP=$(curl -H "X-aws-ec2-metadata-token: $TOKEN" -s
http://169.254.169.254/latest/meta-data/local-ipv4)
```

Another issue was service timing - attempting Docker commands before the Docker daemon was fully started. Adding proper error handling (set -euxo pipefail) helped catch these failures early. The worker nodes also needed retry logic to wait for the manager to publish SSM parameters, requiring loops with timeouts instead of assuming immediate availability.

These UserData debugging challenges were particularly frustrating because failures often occurred silently during instance launch, making troubleshooting difficult without proper logging.

10.3 In general

There was no individual problem that held me back, and most of my progress slowly progressed. Even the Lambda vs DynamoDB that I mention in 11.1 took me no more than an hour to solve.

All these smaller problems I had, I reverted to a super simple code base and slowly, incrementally added one small step at a time. This way I could overcome every issue I faced.

That said, the biggest time sink and the most issues caused was when I implemented the CI/CD pipeline through GitHub Actions. Under normal circumstances, I would have created the infrastructure and the workflow before I created any of the software solutions. But now I am sitting here with the entire CloudFormation stack and the .NET application already ready when I introduced the CI/CD.

A summary of those problems related to the migration is “pathing”, “naming” and “make sure everything comes in the right order”. The Lambda issue was one of those cases that occurred, it worked in the CF Nested Stacks but did not when after the workflow introduction. I had to think through which order I wanted to run separate code sections.

Another issue was naming, I had forgot what I set as variables in GitHub Secrets in my old repo and then I moved it to this one – I guessed wrong on one of the variables which caused a few errors.

Combined, all these small errors took a fair bit of time to solve.

11. References and links

11.1 Our class’ studyguide/tutorial

https://cloud-developer.educ8.se/clo/3.-scalable-cloud-applications/1.-tutorials/1.-provision-a-vm-on-aws-using-cloudformation/index.html

11.2 How to set up Docker Swarm and convert it to CloudFormation Nested Stacks:

https://github.com/mymh13/swarm-dotnet-test

11.3 LLM: Partners (and enemies) in crime

ChatGPT and Claude

11.4 Image provider: CloudCraft

Image at section 2. Overview and section 9.4 provided with the service at cloudcraft.co.