Inlämningsuppgift 2 - Rapport

Albin Stenhoff

2025-09-29

Contents

Inlämningsuppgift 2 - Rapport (Gjord i Visual Studio & AWS)

Kurs: CLO24 - Cloud Development - Skalbara Molnapplikationer (AWS)
Student: Albin Stenhoff
Inlämning: 2025-09-28
GitHub repo: https://github.com/OtrevligAbbe/CLO24-Inlamningsuppgift2-AlbinStenhoff

Översikt

Syftet med denna inlämning är att visa två olika arkitekturer för att köra en webbapplikation i AWS:

1. **Containerbaserad lösning** - .NET Minimal API körs i Docker Swarm på EC2 instanser.
2. **Serverless lösning** – .NET Lambda bakom API Gateway som returnerar JSON.

Båda alternativen kompletteras med **säkerhetsprinciper, IaC (Infrastructure as Code)** samt **CI/CD automation via GitHub Actions**.

1. Container - Docker Swarm i AWS

Molntjänster:

- **EC2**: kör Swarm noder (mannager + workers).
- **VPC & Subnets**: nätverk för isolering och åtkomstkontroll.
- **Security Groups**: definierar öppna portar (endast 80/443).
- **IAM**: roller för att hantera EC2 och minimera rättigheter.

Komponenter & syfte:

- En .NET 8 Minimal API paketerad som Docker image.
- Körs i två replicas i Swarm service för redundans.
- Inbyggd lastbalansering via Swarm overlay nätverk.

Säkerhet:

- Endast HTTP(S) trafik tillåten (80/443).
- Övriga portar stängda.
- IAM roller för minsta möjliga behörighet ("least privilege").
- Uppdatering av noder (patchning).

IaC/automation:

- Terraform skiss i `infra-docker/` som definierar:
  - Provider (AWS, region eu-north-1).
  - Variabler för resurser (VPC, subnets, EC2, SG).
- CI/CD i GitHub Actions validerar Terraform (`terraform validate`) och bygger container image.

Verifiering:

- `docker service ls` visar att tjänsten är igång.
- `docker service ps` visar att två replicas körs.
- `GET /health` returnerar `{ "status":"healthy" }`.

Docker Swarm - steg för steg

1) Bygg image: `docker build -t clo24-minapi:local .`
2) Initiera swarm: `docker swarm init`
3) Starta service (2 repliker):
   `docker service create --name clo24-minapi --replicas 2 --publish 8080:8080 clo24-minapi:local`
4) Verifiera:
   `docker service ls` (2/2), `docker service ps clo24-minapi`
5) Hälsokoll: `curl http://localhost:8080/health` -> `healthy`

Provisionering i AWS (kortfattat)

1. Skapa ett **key pair** i EC2 och en **Security Group** som endast öppnar 22, 80, 443.
2. Starta 1 **manager** och 1–2 **worker**-instanser (Amazon Linux 2/Ubuntu) i **eu-north-1**, i samma **VPC/Subnets**.
3. Installera Docker på alla noder.
4. På manager:

```
docker swarm init --advertise-addr <manager-private-ip>
docker swarm join-token worker
```

Kopiera docker swarm join ...-kommandot. 5. På workers: kör join-kommandot från steg 4. 6. Rulla ut tjänsten på Swarm:

```
docker service create --name clo24-minapi \
  --replicas 2 --publish 80:8080 <REGISTRY-IMAGE>
docker service ls && docker service ps clo24-minapi
```

2. Serverless - .NET Lambda + API Gateway

Molntjänster:

- **API Gateway**: HTTP endpoint.
- **Lambda (.NET 8)**: kör logiken och returnerar JSON.
- **DynamoDB / S3**: möjliga lagringsalternativ.
- **CloudWatch**: loggning och övervakning.

Komponenter & syfte:

- Lambda funktion returnerar en enkel JSON som svar.
- API Gateway ropar på funktionen via en GET request.
- Lokal simulering via **LocalRunner** i Visual Studio.

Säkerhet:

- IAM roller för Lambda med endast nödvändiga rättigheter.
- CORS i API Gateway.
- Hemligheter hanteras via miljövariabler eller AWS Secrets Manager.

IaC/automation:

- SAM template i `infra-serverless/template.yaml`.
- Beskriver resurser: Lambda funktion, API Gateway event.
- CI/CD: GitHub Actions validerar SAM (`sam validate`) och bygger .NET Lambda projektet.

Terraform (urklipp)

```
provider "aws" {
  region = var.region
}

variable "region" { default = "eu-north-1" }

resource "aws_security_group" "web" {
  name        = "clo24-web"
  description = "Allow HTTP/HTTPS/SSH"
  ingress = [
    { from_port = 80,  to_port = 80,  protocol = "tcp", cidr_blocks = ["0.0.0.0/0"] },
    { from_port = 443, to_port = 443, protocol = "tcp", cidr_blocks = ["0.0.0.0/0"] },
    { from_port = 22,  to_port = 22,  protocol = "tcp", cidr_blocks = ["<DIN-IP/32>"] }
  ]
  egress = [{ from_port = 0, to_port = 0, protocol = "-1", cidr_blocks = ["0.0.0.0/0"] }]
}
```

Verifiering:

- Lokal körning (`dotnet run --project LocalRunner/LocalRunner.csproj`) returnerar:

  {"status":"ok","service":"dotnet-lambda","message":"Hej från Lambda (.NET)!"}

Serverless (SAM) - steg för steg

1) Kod i .NET Lambda (`Function.cs`)
2) IaC i `infra-serverless/template.yaml` (AWS, Serverless, Function + API event)
3) Lokal test: `dotnet run --project app-lambda-dotnet/LocalRunner/LocalRunner.csproj`
4) CI: `sam validate -t infra-serverless/template.yaml --region eu-north-1`

3. GitHub & CI/CD

Repository:

- All kod finns publikt på GitHub.
- Strukturerad mappstruktur med `app-dotnet`, `app-lambda-dotnet`, `infra-docker`, `infra-serverless`, `docs`.
- README.md beskriver arkitekturen och hur projektet körs.

CI/CD Workflow:

- Kör automatiskt vid push:
  - Bygger .NET projekten.
  - Bygger Docker image.
  - Validerar Terraform och SAM.
- Säkerställer att koden alltid är i körbart skick.

Exempel på GitHub Actions (förenklad version)

```yaml
name: CI
on: { push: { branches: [main] } }
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-dotnet@v4
        with: { dotnet-version: '8.0.x' }

      - run: dotnet build app-dotnet/app-dotnet.csproj -c Release
      - run: docker build -t clo24-minapi:ci -f app-dotnet/Dockerfile app-dotnet

      - uses: hashicorp/setup-terraform@v3
      - run: |
          cd infra-docker
          terraform init -backend=false
          terraform validate

      - run: pipx install aws-sam-cli || pip install --user aws-sam-cli
      - if: ${{ secrets.AWS_ACCESS_KEY_ID && secrets.AWS_SECRET_ACCESS_KEY }}
        uses: aws-actions/configure-aws-credentials@v4
        with:
          aws-access-key-id:     ${{ secrets.AWS_ACCESS_KEY_ID }}
          aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          aws-region:            ${{ secrets.AWS_REGION || 'eu-north-1' }}
      - if: ${{ secrets.AWS_ACCESS_KEY_ID && secrets.AWS_SECRET_ACCESS_KEY }}
        run: sam validate -t infra-serverless/template.yaml --region ${{ secrets.AWS_REGION || 'eu-north
```

CloudWatch & säkerhet

- CloudWatch: Lambda loggar i CloudWatch Logs (felsökning + larm/metrics möjliga).
- Secrets: GitHub Secrets för AWS-nycklar i CI, inga hemligheter i repo.
- Nätverk: Security Groups öppnar endast 80/443, övrigt stängt.
  Observability för Docker Swarm
- Swarm loggar kan skickas till CloudWatch via `awslogs` driver i service-definitionen.
- Alternativt central loggning med ELK eller Grafana Loki.
- Hälsa och repliker övervakas med `docker service ps`, larm kan sättas via CloudWatch Metrics.

CloudWatch verifiering
För att säkerställa att tjänsterna fungerar som de ska har jag kopplat både Lambda och Docker Swarm till **Amazon CloudWatch**.

- Lambda funktionerna loggar automatiskt till CloudWatch Logs, vilket gör att jag kan se varje request/response.
- För Docker Swarm använde jag `awslogs` drivern för att skicka containerloggar till CloudWatch.
- På så sätt kan jag övervaka status, få felmeddelanden och vid behov sätta upp larm/alerts.
  Detta gör att jag inte bara kan starta upp tjänsterna, utan även följa hur de beter sig över tid i AWS miljön.

Säkerhet och hantering av secrets

I projektet har jag medvetet undvikit att lägga känsliga uppgifter i källkod. AWS uppgifter lagras som **GitHub Secrets** och injiceras bara i GitHub Actions när de behövs (exempel `sam validate`). Detta minskar risken för läckor i repo historik. IAM följer principen **least privilege** (endast de rättigheter som krävs). Applikationshemligheter hanteras via miljövariabler (och kan vid behov flyttas till exempel AWS Secrets Manager). I API lagret hanteras **CORS** i API Gateway. För containerdelen öppnas endast **80/443** i Security Groups. För Terraform körs `init -backend=false` och `validate` i CI för att validera IaC syntax utan att skapa resurser, nästa steg i en verklig miljö vore att konfigurera remote state och en godkännandekedja.
Exempel minimal IAM-policy för EC2 läsrättigheter (för CI)
{ "Version": "2012-10-17", "Statement": [{ "Effect": "Allow", "Action": ["ec2:DescribeInstances", "ec2:DescribeSecurityGroups"], "Resource": "*" }] }

Screenshots

1. Bygga projektet med .NET

[dotnet build success](docs/Screenshots/dotnet build success.png)
Projektet byggdes framgångsrikt med `dotnet build` i PowerShell.

2. Köra projektet lokalt

[dotnet run local](docs/Screenshots/dotnet run local.png)
Applikationen startas med `dotnet run` och lyssnar på port 8080.

3. Testa API i webbläsare

[browser health check](docs/Screenshots/browser health check.png)
Webbläsaren visar JSON-svar från APIett på `http://localhost:8080`.

4. Bygga Docker-image

[docker image built](docs/Screenshots/docker image built.png)
Docker bygger upp en image för applikationen.

5. Container körs i Docker Desktop

[docker container running](docs/Screenshots/docker container running.png)
Containern `clo24-minapi:local` körs i Docker Desktop och exponerar port 8080. **Screenshots** hittas i mappen docs - screenshots.

Skalbarhet - verifiering

- Containers: Swarm service med `--replicas 2` (lastbalansering via Swarm).
- Serverless: Lambda skalar per förfrågan, API Gateway hanterar samtidiga anrop (ingen server att underhålla).

Reflektion kring arkitekturer

Arbetet tydliggjorde skillnaden mellan containerbaserad drift (Docker Swarm på EC2) och serverless (Lambda + API Gateway). Swarm ger mig full kontroll över OS, nätverk och skalning, men kräver mer ansvar, patchning av noder, säkerhetsuppdateringar och egen lastbalansering. Serverless tar bort serverskötsel helt, skalar automatiskt och passar bra för händelsedrivna och korta anrop men jag avstår viss kontroll (kallstart, limits och leverantörslåsning). För mindre, API fokuserade arbetsflöden är Lambda snabbt och kostnadseffektivt. För långkörande tjänster, specialberoenden eller krav på låg nivå kontroll passar containers bättre. Slutsatsen är att valet bör styras av krav på **kontroll vs enkelhet**, trafikmönster och kostnad.

Sammanfattande slutsats

Projektet visar praktiskt hur två arkitekturmönster i AWS kan realiseras och automatiseras med CI/CD. Jag har byggt och kört en .NET Minimal API i container, samt en .NET Lambda bakom API Gateway. Med GitHub Actions valideras kod och IaC (Docker build/compose check, Terraform validate, SAM validate). Det viktigaste jag tar med mig är helhetsflödet, från kod -> container/serverless -> säkerhet -> automation.