

PGR301 EKSAMEN 2024 Couch Explorers - Bærekraftig turisme fra sofakroken !



Som nyansatt i Couch Explorers – en spennende startup som gir brukerne muligheten til å reise verden rundt fra komforten av egen sofa – er du med på å utvikle fremtidens reiseløsning! Ved hjelp av kunstig intelligens genererer tjenesten imponerende bilder av kjente landemerker og unike reisemål fra alle verdenshjørner.

Disse bildene kan brukerne dele på sosiale medier som minner fra deres "reise", helt uten å forlate hjemmet! I fremtidige versjoner av tjenesten vil KI kunne trenes på bilder av brukerne selv, slik at vi kan lage mer realistiske selfies og kanskje til og med emulere retrofotografier som analog film og polaroid – for den ultimate lo-fi reisedokumentasjonen.

Med et sterkt fokus på både KI og bærekraft, har selskapet allerede sikret finansiering for å videreutvikle denne banebrytende løsningen.

Din rolle er å legge til rette for en smidig utviklingsprosess som følger strenge DevOps-prinsipper helt fra starten. Kvaliteten på koden vil variere, og når det er nødvendig, vil du få ansvaret for å implementere forbedringer og sikre at løsningen leverer på høyeste standard.

Krav til leveransen

Eksamensoppgaven, kode og nødvendige filer er tilgjengelig i GitHub-repoet:

<https://github.com/glennbechdevops/eksamen-2024>.

Retningslinjer for innlevering

Wiseflow:

- Når du leverer oppgaven i WiseFlow, last opp et dokument som kun inneholder en lenke til ditt repository. Filen må være i PDF- eller tekstformat.

GitHub:

- Ikke lag en fork av det opprinnelige repositoryet. Kopier i stedet innholdet til et repository du selv oppretter.
- For å unngå at andre studenter ser din besvarelse, kan du gjerne jobbe i et privat repository. Gjør repositoryet offentlig rett før innleveringsfristen.
- I repositoryet ditt skal du lage en fil, **README.md** for å besvare drøfte-oppgaver og oppgavespesifikke leveranser.

Viktig! Oppgavespesifikke leveranser

I hver oppgave er det det vil være en eller flere konkrete leveranser. Dette kan for eksempel være lenker til en GitHub Actions workflow-kjøringer, et objekt i en S3 bucket osv. Dette er for å gjøre sensur mer effektivt. Pass på at du i ditt README dokument får med deg alle leveranser. Lag en tabell i slutten av dokumentet på formatet

- Oppgave 1
 - Leveranse 1.
 - Leveranse 2.
- Oppgave 2
 - Osv

Hvis du velger å ikke svare på enkeltoppgaver, setter også sensor stor pris på at du nevner dette i besvarelsen.

Spesielle hensyn knyttet til Cloud 9

Løsning på problem med diskplassmangel, eller rettigheter / IAM er beskrevet her -

https://github.com/glennbechdevops/cloud9_tools Ved tekniske problemer med miljø under eksamen, kontakt lærer.

Evaluering

- Oppgave 1. 30 Poeng. AWS Lambda og GitHub Actions
- Oppgave 2. 30 Poeng. Infrastruktur med Terraform of SQS
- Oppgave 3. 15 Poeng. Javaklient og Docker
- Oppgave 4. 15 Poeng. Metrics og overvåkning
- Oppgave 5. 10 Poeng. Drøft; Serverless, Function as a service vs Container-teknologi

Litt om AWS Bedrock

AWS Bedrock er en tjeneste fra Amazon som gir tilgang til ulike AI-modeller uten behov for å håndtere den tekniske infrastrukturen selv. I denne oppgaven skal du bli kjent med Amazons egen generative AI-modell for bilder, "Titan."

Ved å bruke denne modellen kan du sende inn et "prompt," som for eksempel "en solnedgang over fjorden med fjell i bakgrunnen." Bedrock vil deretter generere et bilde som samsvarer med beskrivelsen.

Funksjonaliteten vi trenger fra AWS Bedrock er foreløpig ikke tilgjengelig i Irland, så du vil se referanser til regionen "us-east-1" i koden. Likevel skal du bruke Irland (eu-west-1) som region for infrastrukturen din. Det er ingen problem for en Lambda-funksjon i Irland å benytte Bedrock-tjenesten i USA.

Oppgave 1 - AWS Lambda

A. Oppgave: Implementer en Lambda-funksjon med SAM og API Gateway

I klassens delte AWS-konto finnes det en S3-bucket med navnet `pgr301-couch-explorers`. Din oppgave er å ta utgangspunkt i koden som ligger i `generate_image.py` og implementere denne funksjonaliteten som en AWS Lambda-funksjon ved hjelp av AWS SAM (Serverless Application Model). Lambda-funksjonen skal eksponeres gjennom et POST-endepunkt via API Gateway.

Prøv gjerne pythonkoden først for å bli kjent med tjenesten. Bruk Cloud9 eller annet miljø med python installert, og installer avhengigheter i et nyt Python `virtual environment`

```
python3 -m venv .venv  
source .venv/bin/activate  
pip3 install boto3
```

Så kan du kjøre Pythonkoden

```
python3 generate_image.py
```

Her er noen prompts dere kan forsøke

1. "**Vintage Paris Trip**": "Show me sitting at a classic Parisian café with the Eiffel Tower in the background, a coffee and croissant on the table. The image should have an old polaroid filter, with soft faded tones and a sunbeam lightly hitting the Eiffel Tower."
2. "**Sunset Safari**": "Place me in a jeep on an African savanna with lions and elephants in the background under a golden sunset. Use an analog 1980s photo effect with warm color tones, and give it a grainy texture for an authentic safari experience."
3. "**Mountain Adventure**": "Create a realistic image of me standing on the edge of a cliff with dramatic, snow-capped mountains stretching out behind me. Add a soft orange glow in the sky to mimic early morning light, and give the image a slightly grainy, retro film effect."

Eksempelbilde



Trinn 1: Opprett en SAM-applikasjon

1. Opprett en ny mappe i repositoryet ditt for SAM-applikasjonen, for eksempel kalt `sam_lambda`.
2. Sett opp infrastrukturen for Lambda-funksjonen på en av følgende måter:
 - Bruk `sam init` til å generere en ny SAM-applikasjon i Python.
 - Alternativt, bruk filer fra et eksisterende SAM-prosjekt som utgangspunkt, og tilpass dem etter behov for denne oppgaven.

Trinn 2: Skriv Lambda-funksjonen

Implementer koden fra `generate_image.py` som en Lambda-funksjon. Funksjonen skal motta en forespørsel via et POST-endepunkt, generere et bilde- og deretter lagre det i S3-bucketten `pgr301-couch-explorers`. Bruk kandidatnummeret ditt som prefix, slik at URL-en til bildene dine blir `s3://pgr301-couch-explorers/<kandidatnr>/*`. Bildets prompt skal sendes som en HTTP body i forespørselen.

Trinn 3: Fjern hardkoding av bucket-navn fra koden

For å gjøre løsningen mer fleksibel og profesjonell, skal du fjerne hardkodingen av S3-bucket-navnet fra `generate_image.py`. Definer bucket-navnet i SAM-templatefilen og bruk en dynamisk måte for funksjonen å hente dette navnet på, slik at du unngår å legge bucket-navnet direkte inn i koden.

Trinn 4: Test og deploy SAM-applikasjonen

1. Bygg Lambda-funksjonen lokalt med SAM CLI for å sikre at den fungerer som forventet.
2. Deploy applikasjonen. Etter deploy bør du verifisere at POST-endepunktet fungerer, og at Lambda-funksjonen kan lagre filer i S3-bucketten `pgr301-couch-explorers`.

Tips og anbefalinger

- **Timeout:** Husk at Lambdafunksjoner har en konfigurerbar timeout
- **IAM-rolle:** Sørg for at Lambda-funksjonen har nødvendige tillatelser til å skrive til S3-bucketene, og kalle tjenesten AWS Bedrock
- **Regionkonfigurasjon:** Husk at regionen for infrastrukturen skal være **eu-west-1**, selv om ressurser som AWS Bedrock kan ligge i andre regioner.

Leveranser

- HTTP Endepunkt for Lambdafunksjonen som sensor kan teste med Postman

B: Opprett en GitHub Actions Workflow for SAM-deploy

Lag en GitHub Actions workflow som automatisk deployer Lambda-funksjonen hver gang det skjer en push til **main**-branchen i GitHub-repositoryet ditt. Denne workflowen skal automatisere hele prosessen slik at funksjonen blir oppdatert og tilgjengelig uten manuell deploy hver gang det gjøres endringer i koden.

Steg for å lage workflow:

1. Opprett en ny YAML-fil i **.github/workflows**-mappen i repositoryet ditt (for eksempel **deploy_lambda.yml**).
2. Definer workflowen slik at den bygger og deployer SAM-applikasjonen:
 - Pass på at workflowen har tilgang til nødvendige AWS Access Keys/Credentials (via GitHub Secrets) slik at deploy-prosessen kan fullføres.

Leveransekrav:

- **Lenke til kjørt GitHub Actions workflow:** Lever en lenke til en vellykket kjøring av GitHub Actions workflow som har deployet SAM-applikasjonen til AWS.

Oppgave 2 - Infrastruktur med Terraform og SQS

A. Infrastruktur som kode

Utviklingsteamet innser raskt at asynkronitet er avgjørende for å bygge løsninger som skalerer godt. Under en privat beta med kun 1000 brukere gikk den SAM-baserte Lambda-funksjonen på en skikkelig smell – brukerne opplevde timeouts og feil i mobilappene. Hovedproblemet er at det tar opptil 10 sekunder å generere et bilde, og når mange brukere er på samtidig, oppstår det alvorlige ytelsesproblemer.

Den tidligere systemarkitekten, Frank, hadde allerede forutsett denne utfordringen før han trakk seg ut og tok med seg en stor del av venturekapitalen til Sør-Amerika. Han utviklet en løsning som bruker SQS til å håndtere asynkrone meldinger mellom klientene og bildegenereringskoden. Ved å introdusere en SQS-kø mellom klientene og bildeprosesseringskoden, oppnår vi en mye mer skalerbar løsning. Når belastningen øker, blir konsekvensen kun at det tar litt lengre tid før bildet er klart, uten at systemet må håndtere alle forespørsler samtidig. Denne løsningen ligger i filen **Lambda_sqs** og inneholder en Lambda-funksjon som mottar meldinger fra SQS, noe som gir en effektiv og skalerbar tilnærming til bildeprosesseringen.

Siden denne Lambda-funksjonen er relativt enkel og skrevet i Python, vurderer du at Terraform kan være et godt valg for å sette opp både Lambda og infrastruktur.

Oppgave

I en ny mappe, for eksempel `infra` - Skriv Terraformkode for å konfigurere:

- Lambda-funksjonen med SQS-integrasjon (bruk koden i `lambda_sqs.py`)
- Din egen SQS-kø og nødvendig integrasjon for AWS Lambda
- Nødvendige IAM-ressurser mm.

Bruk AWS-provider og en Terraform state-fil som lagres i S3-bucketen `pgr301-2024-terraform-state`. Konfigurer Terraform-provideren til å kreve Terraform-versjon over 1.9 og AWS-provider versjon 5.74.0.

Skriv om koden Lambdakoden skal lagre filer på samme lokasjon som SAM applikasjonen `s3://pgr301-couch-explorers//*`

Tips og anbefalinger

- **Timeout:** Husk at Lambdafunksjoner har en konfigurerbar timeout

B. Opprett en GitHub Actions Workflow for Terraform

Lag en GitHub Actions workflow som håndterer deploy av infrastrukturen til AWS ved å kjøre Terraform-koden. Workflowen skal kjøre forskjellige Terraform-kommandoer basert på hvilken branch det pushes til, slik at endringer kan testes og gjennomgås før de gjøres permanent på hovedinfrastrukturen.

Spesifikasjoner for workflowen:

1. Oppsett av jobber i workflowen:

- Opprett en ny YAML-fil i `.github/workflows`-mappen i repositoryet ditt, for eksempel `terraform_deploy.yml`.
- Pass på at workflowen har tilgang til nødvendige AWS Access Keys/Credentials (GitHub repository Secrets) for å kunne kjøre Terraform-kommandoene og oppdatere infrastrukturen i AWS.

2. Branch-spesifik oppførsel:

- Når det gjøres en push til `main`-branchen, skal workflowen automatisk kjøre `terraform apply` for å oppdatere infrastrukturen med eventuelle endringer. Dette sørger for at alle oppdateringer til `main` direkte reflekteres i den live AWS-infrastrukturen.
- For push til andre brancher enn `main`, skal workflowen kjøre `terraform plan`. Dette lar utviklingsteamet se en plan for hvilke endringer som ville blitt gjort, uten å faktisk oppdatere infrastrukturen. Det gir mulighet for gjennomgang og testing av infrastrukturendringer før de merges til `main`.

Leveransekrav:

- **Lenke til kjørt GitHub Actions workflow:** Lever en lenke til en vellykket kjøring av workflowen der infrastrukturkoden har blitt deployet ved bruk av `terraform apply` på en push til `main`.
- **Lenke til en fungerende GitHub Actions workflow (ikke main):** Lever en lenke til en vellykket kjøring av workflowen der infrastrukturkode bare gjør `terraform plan`

- **SQS-Kø URL:** Http Url for SQS kø. Sensor vil teste å sende melding til denne.

Oppgave 3. Javaklient og Docker

Teamet har kommet fram til at det kan være lurt å ha en klient som tester SQS-løsningen ved å sende meldinger til køen. "Frank" har allerede skrevet koden for dette – men han glemte kanskje at resten av utviklerne er typen som foretrekker Clojure og Lisp. Koden ligger i mappen **java_sqs_client**

Du kan teste Java-koden ved å kjøre den som en JAR-fil. Ved å lese gjennom koden vil du se at den henter **SQS_QUEUE_URL** fra en miljøvariabel og tar et bildeprompt som argument fra kommandolinjen. Deretter legger den en melding på SQS-køen og skriver ut meldings-ID-en.

```
mvn package  
export SQS_QUEUE_URL=https://<din SQS kø HTTPS url>  
java -jar target/imagegenerator-0.0.1-SNAPSHOT.jar "Me on top of K2"
```

Din oppgave er nå å lage et Docker-image av Java-koden, slik at andre på teamet kan bruke den uten å måtte ha Java installert lokalt.

A. Skriv en Dockerfile

Lag en **Dockerfile** for Java-koden som både bygger og kjører applikasjonen i et effektivt og kompakt image. Et tips er å bruke en multi-stage Dockerfile for å først kompile koden og deretter kjøre den i et minimalt runtime-miljø. Dette reduserer imagestørrelsen og gjør det enklere å distribuere.

B. Lag en GitHub Actions workflow som publiserer container image til Docker Hub

Lag en GitHub Actions workflow som bygger og publiserer Docker-imaget til din konto på Docker Hub (<https://hub.docker.com/>) hver gang det gjøres en push til **main**-branchen. Denne workflowen skal automatisere prosessen, slik at teamet alltid har tilgang til den nyeste versjonen av klienten. Lag en konto på Docker Hub om du ikke allerede har en.

Spesifikasjoner for workflowen:

1. Opprett en workflow-fil:

- Lag en ny YAML-fil i **.github/workflows**, for eksempel **docker_publish.yml**.

2. Konfigurer jobber i workflowen:

- Inkluder steg for å sjekke ut koden, logge inn på Docker Hub (ved hjelp av credentials lagret i GitHub Secrets), bygge Docker-imaget, tagge det, og deretter pushe det til Docker Hub-kontoen din.

3. Triggering:

- Sett opp workflowen til å kjøre automatisk ved hver push til **main**-branchen, slik at endringer i koden alltid blir reflektert i det publiserte imaget.

Du står fritt til å velge en strategi for hvordan du vil tagge container image på DockerHub. Men i denne oppgaven sin leveranse skal du gjøre rede for valget ditt.

Leveranser

- **Beskrivelse av taggestrategi:** Forklar kort hvordan du har valgt å tagge container imagene dine, og gi en begrunnelse for denne strategien.
- **Container image + SQS URL:** Lever kun image-navnet fra din Dockerhub-konto (for eksempel **glennbech/couchexplore-client**). Sensor vil bruke dette navnet til å kjøre container imaget ditt ved hjelp av en Docker-kommando.

```
docker run -e AWS_ACCESS_KEY_ID=xxx -e AWS_SECRET_ACCESS_KEY=yyy -e SQS_QUEUE_URL=<SQS_QUEUE_URL> _dockerhub_user/repo:tag_ "me on top of a pyramid"
```

Oppgave 4: Metrics og overvåkning

Selv med en asynkron løsning er utviklingsteamet bekymret for responstider og brukeropplevelsen. Et sentralt DevOps-prinsipp er Feedback, slik at vi raskt kan fange opp problemer – helst før brukerne merker dem. Noen brukere har allerede gitt dårlige anmeldelser i Apple Store og klaget over at det tar "lang" tid før bildene vises, ofte flere minutter.

For å overvåke løsningen foreslår du å sette opp en CloudWatch-alarm på SQS-metrikkene **ApproximateAgeOfOldestMessage**. Denne metrikken måler (i sekunder) hvor gammel den eldste meldingen i køen er, noe som kan indikere forsinkelser i behandlingen av forespørsler.

A. Infrastrukturkode

Utvid Terraform-koden fra Oppgave 2 med en CloudWatch-alarm som triggges når verdien for **ApproximateAgeOfOldestMessage** blir for høy. Du kan selv velge passende terskler for alarmen. Når alarmen utløses, skal en e-post sendes til en adresse spesifisert som en variabel i Terraform-koden.

Oppgave 5. Serverless, Function as a service vs Container-teknologi

Drøft implikasjonene ved å implementere et system basert på en serverless arkitektur med Function-as-a-Service (FaaS)-tjenester som AWS Lambda og meldingskøer som Amazon SQS, sammenlignet med en mikrotjenestebasert arkitektur. Legg merke til at komponentantallet ofte vil variere mellom disse tilnærmingene; en mikrotjeneste kan ofte brytes ned til flere mindre, selvstendige funksjoner i en serverless arkitektur, noe som fører til flere komponenter med egne livssykluser og behov for utrulling, versjonskontroll, overvåkning osv.

Drøft og vurder implikasjonene i lys av følgende DevOps-prinsipper:

1. **Automatisering og kontinuerlig levering (CI/CD):** Hvordan påvirker serverless-arkitektur sammenlignet med mikrotjenestearkitektur CI/CD-pipelines, automatisering, og utrullingsstrategier?
2. **Observability (overvåkning):** Hvordan endres overvåkning, logging og feilsøking når man går fra mikrotjenester til en serverless arkitektur? Hvilke utfordringer er spesifikke for observability i en FaaS-arkitektur?
3. **Skalerbarhet og kostnadskontroll:** Diskuter fordeler og ulemper med tanke på skalerbarhet, ressursutnyttelse, og kostnadsoptimalisering i en serverless kontra mikrotjenestebasert arkitektur.

4. **Eierskap og ansvar:** Hvordan påvirkes DevOps-teamets eierskap og ansvar for applikasjonens ytelse, pålitelighet og kostnader ved overgang til en serverless tilnærming sammenlignet med en mikrotjeneste-tilnærming?

I besvarelsen bør du trekke frem styrker og svakheter ved hver tilnærming, med tanke på de nevnte DevOps-aspektene.

LYKKE TIL OG HA DET GØY MED OPPGAVEN!