Hvordan kjøre koden(guide)

#Antagelsen her er at personen har docker desktop installert eller tilgjengelig

Docker Desktop må være installert og kjørende

Hvordan kjøre prosjektet

For å kjøre dette prosjektet lokalt trenger du kun Docker Desktop. Det er ikke nødvendig å installere Python eller andre avhengigheter direkte på maskinen.

Steg 1: Klon repository

Klon prosjektet fra GitHub

git clone https://github.com/TharusanJulian/mcp-konsulent-staffing.git

Gå inn i prosjektmappen

cd mcp-konsulent-staffing

Steg 2: Start tjenestene

Start begge mikrotjenestene med Docker Compose, docker bygger images og starter containerne automatisk

docker compose up –build

Steg 3: Tjenester som startes

konsulent-api kjører på http://localhost:8001

llm-verktoy-api kjører på http://localhost:8002

Steg 4: Test at løsningen fungerer

Hent listen med konsulenter:

curl http://localhost:8001/konsulenter

Henting av sammendrag basert på behov:

curl http://localhost:8002/tilgjengelige-konsulenter/sammendrag?min_tilgjengelighet_prosent=50&paakrevd_ferdighet=python

Steg 5: API-dokumentasjon

Begge tjenestene har automatisk generert dokumentasjon via FastAPI

Konsulent-API: http://localhost:8001/docs

LLM-verktøy-API: http://localhost:8002/docs

Steg 6: Stoppe prosjektet

Trykk Ctrl + C i terminalen der prosjektet kjører

Hvordan jeg gikk frem for å løse caset gitt av Append Consulting

Da jeg fikk denne case-oppgaven fra Append Consulting, startet jeg med å roe helt ned og lese oppgaven flere ganger før jeg begynte å kode. For meg er det viktig å forstå hva som faktisk blir spurt om, før jeg hopper rett inn i tekniske detaljer. Derfor brukte jeg først tid på å kartlegge hva oppgaven egentlig går ut på, hvilke krav som er absolutte, og hva som eventuelt bare er rammer rundt løsningen.

Grunnen til at jeg bevisst valgte å notere og dokumentere tankegangen min underveis, er at jeg selv liker å kunne se tilbake på hvordan jeg resonnerte da jeg løste oppgaven. Det gir meg også en bedre måte å forklare valgene mine på i etterkant, for eksempel i et intervju.

Oppgaven beskriver en MCP-basert løsning for konsulent-staffing, der en intern AI-assistent skal kunne hente informasjon om konsulenter og gi et forståelig sammendrag basert på behov. Når jeg brøt dette ned til noe mer konkret, tolket jeg oppgaven som: hente konsulentdata, filtrere dem basert på noen kriterier, og forklare resultatet på en enkel og menneskelig måte.

For å gjøre dette lettere å forstå – både for meg selv og for andre – valgte jeg å se på løsningen som to tydelige roller. Den første rollen er veldig enkel. Dette er i praksis et arkiv. Den gjør ingenting annet enn å gi fra seg en liste med konsulenter når den blir spurt. Den tar ingen avgjørelser, vurderer ingen krav og inneholder ingen logikk utover å returnere data. Denne rollen er implementert som konsulent-api.

Den andre rollen er der den faktiske "jobben" skjer. Denne tjenesten henter konsulentlisten fra arkivet, regner ut hvor tilgjengelig hver konsulent er basert på belastning, filtrerer listen basert på minimum tilgjengelighet og ønsket ferdighet, og setter deretter sammen et ferdig sammendrag i naturlig språk. Dette er den delen en AI-assistent i praksis ville kalt for å få et svar den kan bruke direkte. Denne rollen er implementert som llm-verktoy-api.

Begge disse rollene kjører i hver sin Docker-container. Jeg valgte dette både fordi oppgaven eksplisitt ba om det, men også fordi det gir en veldig tydelig separasjon av ansvar. Tjenestene kan snakke sammen, men de er ikke avhengige av hverandre på en måte som gjør løsningen sårbar eller rotete. En viktig del av testen her er nettopp at kommunikasjonen mellom tjenestene fungerer. Hvis konsulent-api ikke leverer data, har llm-verktøy-api ingenting å jobbe med, og da faller hele løsningen sammen.

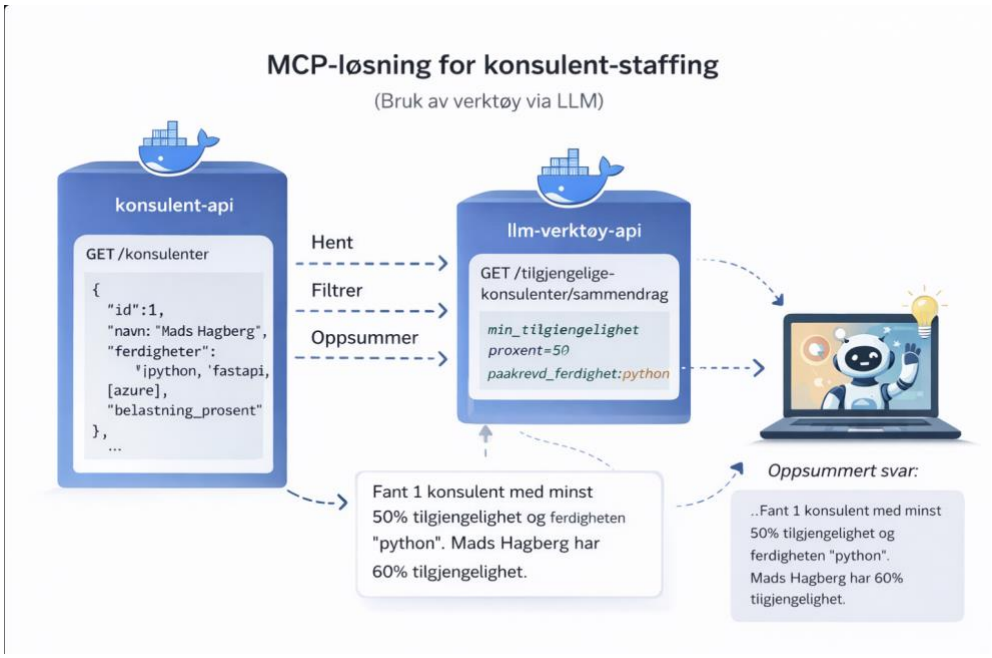
Oppgaven bruker begrepet MCP, Model Context Protocol. Min forståelse er at dette i praksis handler om å bygge et verktøy som en språkmodell kan bruke via function calling eller tool use. I denne løsningen fungerer konsulent-api som en ren datakilde, mens llm-verktøy-api fungerer som et mellomlag som kombinerer data og forretningslogikk og returnerer et ferdig strukturert svar. Sammendraget genereres deterministisk, men løsningen er lagt opp slik at det enkelt kan utvides med en faktisk språkmodell, for eksempel via OpenRouter, dersom man ønsker det.

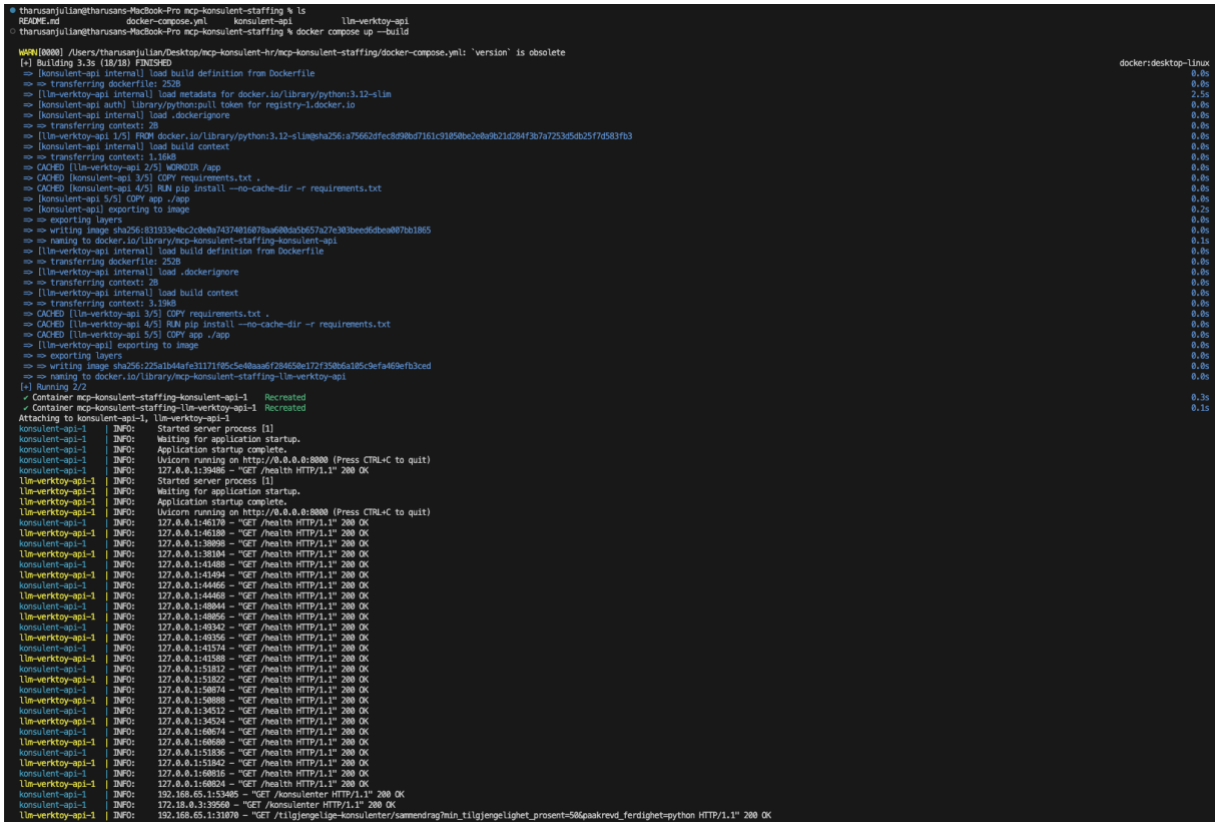
Underveis i arbeidet brukte jeg hjelpemidler som ChatGPT, Claude, Perplexity og Gemini for å stille grunnleggende spørsmål, friske opp kunnskap rundt FastAPI og Docker, og dobbeltsjekke arkitekturvalg. For meg fungerer disse

verktøyene best som en sparringspartner, ikke som en fasit, og jeg brukte dem hovedsakelig til å strukturere tankene mine og komme raskere videre når jeg sto fast.

Til slutt endte jeg opp med en løsning som jeg bevisst har holdt enkel og ryddig. Målet har ikke vært å lage noe unødvendig avansert, men heller å vise at jeg forstår problemet, kan dele det opp i riktige ansvarsområder, og levere en løsning som er lett å forklare og lett å bygge videre på. For meg kan hele oppgaven oppsummeres ganske enkelt slik:
«Dette handler om å lage en enkel digital assistent som finner riktige konsulenter og forklarer hvorfor på en måte folk faktisk forstår.»

Caseoppgaven forklart visuelt:





Sender inn en spørring

7. Endelig prosjektstruktur

Den endelige løsningen fikk følgende struktur:

Hvordan jeg har bygget opp løsningen

Dette caset handler egentlig ikke bare om kode, men om å vise hvordan jeg tenker rundt struktur, ansvar og samarbeid mellom ulike deler av et system. Derfor valgte jeg å dele løsningen opp i to tjenester som har helt ulike roller, men som likevel jobber tett sammen.

Prosjektet består av én tjeneste som kun har ansvar for data, og én tjeneste som gjør selve «tenkingen».

Overordnet struktur

mcp-konsulent-staffing/

```
├── README.md
├── docker-compose.yml
├── konsulent-api/
│   ├── Dockerfile
│   ├── requirements.txt
│   └── app/main.py
└── llm-verktoy-api/
    ├── Dockerfile
    ├── requirements.txt
    └── app/
        ├── models.py
        ├── client.py
        └── main.py
```

Jeg har bevisst prøvd å holde strukturen enkel og oversiktlig, slik at det er lett å forstå hva som skjer hvor, også for noen med lite teknisk bakgrunn.

---

docker-compose.yml – limet som holder alt sammen

Denne filen er i praksis «startknappen» til hele prosjektet. Den beskriver at det finnes to tjenester som skal kjøre samtidig, hvilke porter de bruker, og at de skal kunne kommunisere med hverandre.

Når man kjører docker compose up --build, er det denne filen som gjør jobben med å starte hele systemet på én gang.

---

konsulent-api – den enkle delen

Denne tjenesten er bevisst holdt veldig enkel. Den kan sees på som et arkiv eller et lite register.

konsulent-api/app/main.py

Her ligger en hardkodet liste med konsulenter. Hver konsulent har:

et id-nummer

et navn

en liste med ferdigheter

en belastning i prosent

Når noen kaller endepunktet GET /konsulenter, returnerer denne tjenesten bare hele listen i JSON-format. Den gjør ingen vurderinger og ingen filtrering. Poenget er at denne tjenesten kun skal gi fra seg data, ikke ta avgjørelser.

Dette er gjort med vilje for å holde ansvarene tydelig adskilt.

---

llm-verktøy-api – der «tenkingen» skjer

Dette er hoveddelen av caset, og den delen som er ment å brukes av en AI-assistent.

models.py

Her har jeg samlet modellene som beskriver hvordan dataene ser ut. Det gjør det lettere å forstå hva en konsulent faktisk inneholder, og gjør koden mer lesbar og trygg.

client.py

Denne filen har én jobb: å hente konsulentlisten fra konsulent-api. Jeg skilte dette ut i en egen fil for å holde resten av koden ryddigere og lettere å følge.

main.py

Dette er hjertet i løsningen.

Når noen kaller:
GET /tilgjengelige-konsulenter/sammendrag

så skjer følgende:

Tjenesten henter alle konsulentene fra konsulent-api

Den regner ut hvor tilgjengelig hver konsulent er basert på belastning

Den filtrerer konsulentene basert på kravene som er sendt inn

Til slutt lager den et kort, menneskeleselig sammendrag som kan vises direkte til en bruker

Målet er at en AI-assistent skal kunne bruke dette som et verktøy, uten å måtte forholde seg til rå JSON-data eller komplisert logikk.

Hvorfor jeg valgte denne løsningen

Jeg har bevisst valgt å dele ansvarene tydelig:

Én tjeneste eier data

Én tjeneste eier logikk og forklaring

Ved å kjøre dem i separate Docker-containere er de isolert, men kan likevel kommunisere. Hvis datatjenesten er nede, kan ikke assistenten gi svar – og det er helt bevisst. Det viser tydelig avhengigheten mellom rollene.

Kort sagt er dette ment å være et enkelt, men realistisk eksempel på hvordan et MCP-lignende oppsett kan fungere i praksis.