Elton, Hermann Owren
Rosendahl, Olaf

Styrker og svakheter ved ulike skyløsninger

Bacheloroppgave i Dataingeniør
Veileder: Holt, Alexander
Mai 2022

Bacheloroppgave

NTNU
Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for datateknologi og informatikk

Elton, Hermann Owren
Rosendahl, Olaf

Styrker og svakheter ved ulike skyløsninger

Bacheloroppgave i Dataingeniør
Veileder: Holt, Alexander
Mai 2022

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for datateknologi og informatikk

Abstract

The cloud platforms offer a large amount of services to run software. There are a lot of cloud solutions and cloud services and it has become challenging to find out which solutions are suitable in different use cases. Kantega is interested in learning more about the possibilities of the cloud and wanted an insite in the strengths and weaknesses of various cloud solutions.

Through this project, we have thus made an overview of cloud solutions and services offered by the three largest cloud providers. To analyze these solutions, we have both looked at documentation and tested them in practice. In order for the practical testing to be realistic, we have developed a web application for a customer of Kantega that has been used for the experimental testing.

The three cloud providers we have explored cloud services with are Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP), within the categories Function as a Service (FaaS), Platform as a Service (PaaS), Container as a Service (CaaS) and Kubernetes.

The results show that the different cloud solutions have relatively clear differences. Based on this, we have identified strengths and weaknesses, as well as areas of use where the various services fit well.

FaaS offers a fast way to the market with a lot of abstraction, but lacks flexibility if you want more control. It is best suited for individual side jobs.

PaaS makes it easy to deploy larger applications without having to deal with much configuration. In addition, it is quite easy to connect your own domains, as well as other functionality from the cloud provider.

CaaS gives users more control over the infrastructure and provides different services, each with its own benefits that give users many opportunities without as much configuration as with Kubernetes.

Kubernetes is a great tool with a lot of functionality. It offers a lot of control and configuration, but with this comes a steep learning curve.

Sammendrag

Skyplattformene tilbyr en stor mengde tjenster for å kjøre programvare. Det har etterhvert blitt så mange skyløsninger og skytjenester at det er blitt utfordrende å finne ut hvilke løsninger som passer til ulik bruk. Kantega er interessert i å lære mer om mulighetene i skyen og ønsket mer kunnskap om styrkene og svakhetene ved dem.

Gjennom dette prosjektet har vi dermed laget en oversikt over skyløsninger og tjenester som tilbys av de tre største skyleverandørene. For å analysere disse løsningene har vi både sett på dokumentasjon og testet dem ut i praksis. For at den praktiske testingen skulle være reell har vi utviklet en web-applikasjon for en kunde av Kantega som har blitt brukt til den eksperimentelle testingen.

De tre skyleverandørene vi har utforsket skytjenester hos er Amazon Web Services (AWS), Microsoft Azure, og Google Cloud Plattform (GCP), innenfor kategorierene Function as a Service (FaaS), Platform as a Service (PaaS), Container as a Service (CaaS) og Kubernetes.

Resultatene viser at de forskjellige skyløsningene har relativt klare forskjeller. Basert på dette har vi identifisert styrker og svakheter, samt bruksområder der de forskjellige tjenestene passer godt.

FaaS tilbyr en rask vei til markedet med mye abstraksjon, men mangler fleksibilitet om en ønsker mer kontroll. Det passer bast for enkeltstående sidejobber.

PaaS gjør det enkelt å utplassere større applikasjoner uten å måtte forholde seg til særlig mye konfigurasjon. I tillegg er det ganske enkelt å tilknytte egne domener, samt annen funksjonalitet fra skyleverandøren.

CaaS gir brukere mer kontroll over infrastrukturen og tilbyr mange forskjellige tjenester med hver sine fordeler som gir brukere mange muligheter uten like mye konfigurasjon som med Kubernetes.

Kubernetes er et stort verktøy med mye funksjonalitet. Det tilbyr mye kontroll og konfigurasjon, men med dette kommer en høy læringskurve.

Forord

Denne oppgaven er skrevet for NTNU i forbindelse med bacheloroppgaven for dataingeniør. Oppgaven er gitt av Kantega som ønsket en analyse av mulighetene innen ulike skyløsninger, samt utvikling av en web-applikasjon for en av deres kunder.

Vi valgte denne oppgaven fordi det ikke har vært undervisning rundt skyen i løpet av studiet. Denne oppgaven var dermed en gylden mulighet til å få mer innsikt i dette temaet. Det var også spennende å kunne jobbe med en reell kunde av Kantega ettersom dette kan gi verdifull erfaring for videre studier og arbeidsliv.

Under gjennomføringen har vi fått faste plasser på Kantega sine kontorer i Trondheim. Dette har vært veldig nyttig både for god kommunikasjon innad i teamet, men også mot Kantega og deres kunde. Vi ønsker å utrykke en spesiell takk til følgende personer:

- Edvard Neset Karlsen, Jørund Leknes og Magnus Haugsand fra Kantega, for veiledning, hjelp og gode innspill gjennom prosjektet
- Gründerne bak prosjektet vi har laget en web-applikasjon for. Det har vært veldig spennende å kunne være med på å videreutvikle idéen
- Alexander Holt, veileder fra Institutt for datateknologi og informatikk ved NTNU, for gode tilbakemeldinger og veileding
- Alle som har lest gjennom hovedrapport og foreslått forbedringer og endringer

Trondheim, 20.05.2022

Hermann Owren Elton

Elton, Hermann Owren

Olaf Rosendahl

Rosendahl, Olaf

Innhold

Abstract . . . iii

Sammendrag . . . iv

Forord . . . v

Innhold . . . vi

Figurer . . . x

Tabeller . . . xii

Akronymer . . . xiii

Ordliste . . . xiv

1 Introduksjon . . . **1**

1.1 Bakgrunn . . . 1

1.2 Oppgavetekst . . . 2

1.2.1 Forandringer i oppgaveformulering . . . 2

1.3 Problemstilling . . . 3

1.4 Struktur . . . 3

2 Teori om skytjenester . . . **4**

2.1 Skytjenester . . . 4

2.2 Serverløs databehandling . . . 4

2.3 Infrastructure as a Service (IaaS) . . . 5

2.4 Virtuell maskin (VM) . . . 5

2.5 Konteinerbilde . . . 5

2.6 Konteiner . . . 6

2.7 Konteiner-orkestrering . . . 7

2.8 Kubernetes . . . 7

2.9 Container as a Service (CaaS) 9
2.10 Platform as a Service (PaaS) 10
2.11 Function as a Service (FaaS) 10
2.12 Infrastructure as Code (IaC) 11
2.12.1 Deklarativ IaC 12
2.12.2 Imperativ IaC 12
3 Metode 13
3.1 Applikasjon 13
3.2 Valg av teknologi 13
3.2.1 Quarkus 13
3.2.2 NextJS 14
3.2.3 PostgreSQL 14
3.2.4 Docker 14
3.2.5 Pulumi 15
3.2.6 Eksterne tjenester 15
3.3 Valg av utviklingmetode 16
3.3.1 Versjonskontroll 16
3.3.2 Kanban 16
3.3.3 Kanban-brett 16
3.4 Forskningsmetoder 16
3.4.1 Kvalitativ metode 17
3.4.2 Kvantitativ metode 17
3.5 Parametre for sammenligning 17
3.5.1 Pris 17
3.5.2 Anvendbarhet 17
3.5.3 Ytelse og cold start 18
3.5.4 Skalering 19
3.5.5 Leverandørinnlåsing 20
3.6 Analysemetode 20
3.6.1 FaaS 21
3.6.2 PaaS 21

3.6.3 CaaS 22
3.6.4 Konteiner-orkestrering 22
3.6.5 Virtuell maskin 23
4 Resultater 24
4.1 Ingeniørfaglige resultater 24
4.1.1 Applikasjon 24
4.2 Vitenskapelige resultater 29
4.2.1 FaaS 29
4.2.2 PaaS 34
4.2.3 CaaS 40
4.2.4 Kubernetes 45
4.3 Administrative resultater 47
4.3.1 Milepæler 48
4.3.2 Timeforbruk 48
4.3.3 Versjonskontroll 49
5 Diskusjon 50
5.1 Ingeniørfaglige resultater 50
5.1.1 Sluttprodukt 50
5.1.2 Teknologier 51
5.1.3 Resultatmål 51
5.2 Vitenskapelige resultater 52
5.2.1 FaaS 52
5.2.2 PaaS 53
5.2.3 CaaS 54
5.2.4 Konteiner-orkestrering 56
5.3 Administrative resultater 57
5.3.1 Milepæler 57
5.3.2 Timeforbruk 57
5.3.3 Arbeidsmetodikk 57
5.3.4 Gruppearbeid 59
5.4 Feilkilder 60

6 Konklusjon og videre arbeid 61
6.1 Konklusjon 61
6.2 Videre arbeid 62
6.3 Samfunnspåvirkning 63
Referanser 64
A Visjonsdokument 74
B Kravdokumentasjon 88
C Systemdokumentasjon 95
D Prosjekthåndbok 105

Figurer

2.1 Oppbygning av et konteinerbilde 6
2.2 Flere konteiner-applikasjoner på samme infrastruktur 6
2.3 Eksempel på fordeling av pods i noder 8
2.4 Eksempel på en Kubernetes-klynge 9
2.5 IaaS vs CaaS vs PaaS vs FaaS 11

3.1 Strategi for last-testing. X-akse viser tid i timer, y-akse viser antall bot's 18
3.2 Eksempel på deler av resultat fra Loadster 18
3.3 Vertikal og horisontal skalering 20

4.1 Flyt for innlogging og registrering med Auth0 25
4.2 Administrering av brukere og deres roller 26
4.3 Intern kommunikasjonskanal 27
4.4 Personlig huskeliste 27
4.5 Telefonkatalog 28
4.6 Menylinje for eksterne systemer 28
4.7 Cold starts hos de ulike leverandørene 31
4.8 AWS Elastic Beanstalk på Elastic Container Service 35
4.9 Eksempler på konfigurasjon av Google App Engine med app.yaml 37
4.10 Azure App Service - triggere for autoskalering 38
4.11 Azure App Service - skalering basert på tid 38
4.12 AWS Elastic Beanstalk - triggere for autoskalering 39
4.13 AWS Elastic Beanstalk - skalering basert på tid 39

4.14 Azure Container Group med flere Container Instances, hentet fra [105] 41
4.15 Timeforbruk fordelt på person og uke gjennom prosjektet . . 48

5.1 Skjermbilde fra Kanban-brettet under utviklingen av applikasjonen 58
5.2 Eksempel på kommentar i pull request fra GitHub 59

Tabeller

3.1 FaaS-løsninger som har blitt sammenlignet 21
3.2 PaaS-løsninger som har blitt sammenlignet 22
3.3 CaaS-løsninger som har blitt sammenlignet 22
3.4 Kubernetes-løsninger 23

4.1 Kjøretidsmiljøer FaaS 29
4.2 Pris FaaS-løsninger 32
4.3 Kjøretidsmiljøer PaaS 34
4.4 Gjennomsnittlig responstid for PaaS-løsninger med differanse mellom opplasting av kildekode i forhold til konteiner 39
4.5 Estimert pris pr. måned for PaaS-løsninger [100]–[102] 40
4.6 Estimert pris pr. måned for CaaS-løsninger for instans med 1 CPU-kjerne og 4 GB minne [138]–[142] 44
4.7 Estimert pris pr. måned for Kubernetes-løsninger uten rabatt . 47

Akronymer

API Programmeringsgrensesnitt. 10, 13, 18, 24, 25, 28, 30, 42, 46, 52

AWS Amazon Web Services. 1, 2, 20, 30, 32, 33, 36, 40, 43, 45, 47, 52, 54, 56

CaaS Container as a Service. 9, 22, 43, 45, 56

FaaS Function as a Service. 2, 10, 13, 21, 29, 33, 52, 60, 61

GCP Google Cloud Platform. 1, 2, 20, 31–33, 37, 40, 43, 53–56

IaaS Infrastructure as a Service. 5

IaC Infrastructure as Code. 11, 12, 15, 63

PaaS Platform as a Service. 10, 17, 40, 53, 54, 56, 60–62

SSG Static Site Generation. 14

SSR Server-side Rendering. 14

VM Virtuell maskin. 5, 11, 23, 40, 45

Ordliste

Cold start: Tiden det tar for en applikasjon å starte opp til den er klar til å motta forespørsler. vii, x, 18, 19, 21, 22, 31, 32, 43, 45, 52, 53, 60, 61

Cron: Et verktøy som brukes til å sette opp jobber som skal kjøres periodisk på spesifikke tidspunkter, datoer eller intervaller [1]. 37, 54, 61

Dapr: Reduserer kompleksiteten i kommunikasjon for distribuerte mikrotjenesteapplikasjoner ved hjelp av APIer [2]. 42, 55

Instans: En maskinenhet som brukes til å kjøre for eksempel en applikasjon. Instanser kan være av forskjellig størrelser når gjelder både CPU-kraft og minnestørrelse. 7, 17

Kommandolinjeverktøy: Et brukergrensesnitt der brukeren kommuniserer med et dataprogram eller operativsystem ved å skrive kommandoer. Tilbakemeldinger mottas gjennom tekst [3]. 11, 30, 31, 34, 36, 37, 42, 46, 51

Last-test: Testing av en applikasjons ytelse og evne til å håndtere trafikk ved å simulere at mange brukere benytter applikasjonen samtidig. 18, 21, 39

Pull request: En forespørsel om flette ny kode inn i en annen gren (branch) i et Git-repository hos GitHub. Brukes gjerne til å kvalitetskontroll av kode under utvikling [4]. 16, 28, 49, 58, 59

Skyleverandør: En leverandør av skytjenester. Azure, AWS og GCP er eksempler på skyleverandører. 4, 18, 20, 51, 56, 62, 63

Skyløsning: En samling tjenester som bruker samme konsept for å løse et problem i skyen. xiv, 4, 5, 9, 10, 13–15, 17, 20, 24, 28, 50–52, 54, 57, 60–63

Skytjeneste: «Skytjenester (cloud computing) er en samlebetegnelse på alt fra dataprosessering og datalagring til programvare på servere som er tilgjengelig fra eksterne serverparker tilknyttet internett.» [5] Skytjenester kan kategoriseres i skyløsninger. xiv, 1, 4, 10, 13, 15, 18, 20, 22, 23, 47, 50, 51, 54, 56, 60–63

SSL-sertifikat: «Et digitalt sertifikat som autentiserer nettstedets identitet og muliggjør kryptert tilkobling. SSL står for «Secure Sockets Layer», en sikkerhetsprotokoll som oppretter en kryptert forbindelse mellom en webserver og en nettleser.» [6]. 18, 22, 34, 35, 37, 41–43, 46, 53–55

VPC: Et isolert virtuelt nettverk der flere forskjellige tjenester kan befinne seg. Samlokaliserte tjenester i et VPC kan kommunisere med hverandre uten at tjenestene er eksponert offentlig [7]. 18, 42, 43, 55

Åpen kildekode: Allment tilgjengelig kildekode som distribueres med en lisens som gir brukere rettighet til å bruke koden. Lisensen bestemmer graden av frihet og bruksrettigheter [8]. 7, 13, 14

Kapittel 1

Introduksjon

1.1 Bakgrunn

Skyen lar bedrifter og personer benytte skytjenester som lagring, nettverk og datakraft uten å selv eie disse fysisk. Den lar flere dele de samme ressursene samtidig gjennom virtualisering og lar brukere koble seg til tjenestene gjennom nettverk.

I 1967 lanserte IBM det første operativsystemet for virtuelle maskiner. Dette la grunnlaget for at flere kunne dele samme ressurser på samme tid [9]. To år etter, i 1969, ble ARPANET tatt i bruk av det amerikanske forsvarsdepartementet [10]. ARPANET er forløperen til dagens internett som er en fundamental komponent for skytjenester.

Lanseringen av World Wide Web i 1991 [11] er et symbol på hvordan teknologi som nettverk, lagring, virtualisering og datakraft nådde en viss modningsgrad på 90-tallet. Dotcom-revolusjonen fulgte dette og flere nettsider viste en front-end til brukere der backend-logikken var på servere plassert et annet sted i World Wide Web. Den første kjente bruken av «cloud» skal ha vært i et internt dokument hos selskapet Compaq i 1996 [12].

AWS lanserte «Simple Storage Service» (S3) og beta-versjon av «Amazon Elastic Compute Cloud» (EC2) i 2006. Med det var de den første tilbyderen på markedet [13]. S3 tillot brukere å lagre filer i skyen, mens EC2 ga kunder mulighet til å leie virtuelle maskiner for kun 10 cent per time. Amazon fant behovet for å lage AWS etter å ha sett selv hvor mye tid de brukte innad i Amazon.com på å sette opp infrastruktur for nye produkter [14]. Dagens hovedkonkurrenter, Google Cloud Platform (GCP) og Microsoft Azure, kom noe senere på markedet. Google lanserte *Google App Engine* i april 2008, mens Microsoft lanserte Azure i februar 2010.

Det er naturlig å se på skytjenestenes utvikling frem til 2013 som en form for første generasjon skytjeneste. I 2013 ble konteiner-tjenesten Docker lansert som åpen kildekode [15]. Etter dette har antall løsninger vokst i

stor fart på grunn av nye muligheter i det som kan kalles generasjon to. AWS lanserte i 2014 *Elastic Container Service* (ECS) for å støtte kjøring av konteinere. I tillegg lanserte AWS sin Function as a Service (FaaS) tjeneste *AWS Lambda*, som lar kunder enkelt sette opp funksjoner som skal kjøres basert på spesifikke triggere. Konteiner-orkestrering-plattformen Kubernetes er en annen viktig teknologi som har hatt en voldsom popularitetsvekst siden 2017.

Professor John McCarthy sa på MITs hundreårsfeiring i 1961 at «Computing may someday be organized as a public utility just as the telephone system is a public utility» [16]. Sitatet fremstår i dag som en visjonær fremstilling av fremtiden som pr. dags dato i stor grad stemmer.

1.2 Oppgavetekst

Dette er oppgaveteksten vi fikk fra oppdragsgiver Kantega:

«De tre store skyplattformene (Google Cloud Platform (GCP), Microsoft Azure og Amazon AWS) tilbyr en stor mengde produkter og løsninger for å rulle ut og kjøre programvare. For å nevne ett eksempel tilbyr hver plattform ulike proprietære løsninger for såkalt serverless hosting, men dette er bare en av mange alternativ. For utviklere og arkitekter er det utfordrende å orientere seg i dette store landskapet av muligheter.

Hovedmålet med dette prosjektet er å gjøre en komparativ analyse av ulike alternativ for å kjøre programvare på de store skyplattformene. Analysen bør se på fordeler og ulemper med tanke på ulike faktorer slik som kost, fleksibilitet, skalerbarhet, resiliens og robusthet, bruk av åpne vs. proprietære produkter og såkalt developer experience.

Som et motiverende case for denne analysen, men også som et mål i seg selv, ønsker vi at studentene skal utvikle en prototype på en web-applikasjon for en start-up som er kunde av Kantega. I tillegg til å eksperimentere med ulike alternativ for å kjøre denne applikasjonen i skyen, kan det bli aktuelt å intervjue Kantega-konsulenter som jobber på ulike prosjekter hvor man bruker skyteknologi.»

1.2.1 Forandringer i oppgaveformulering

Etter planlegging og diskusjon med oppdragsgiver har oppgaveteksten endret seg noe. For å hindre at oppgaven ikke ble for omfattende har de planlagte sammenligningsfaktorene blitt redusert til pris, anvendbarhet, ytelse og cold start, skalering og leverandørinnlåsing. Robusthet ble vurdert som vanskelig å måle ettersom man potensielt kan leie store maskiner som tåler veldig høy belastning. Developer experience kan fort bli subjektivt og en analyse av brukergrensesnitt, noe vi kom frem til at ikke var like relevant som de andre faktorene. Sikkerhet rundt de forskjellige løsningene og muligheter for å kjøre på både leverandørens og sine

egne servere ble også vurdert å se nærmere på, men ikke prioritert på grunn av tid. Det har heller ikke blitt prioritert å bruke tid på å intervjue Kantega-konsulenter.

Nærmere beskrivelse av krav til applikasjonen er definert i Vedlegg A Visjonsdokument og Vedlegg B Kravdokumentasjon.

1.3 Problemstilling

Utvalget av skyløsninger og tilhørende tjenester er stort og kan oppleves uoversiktlig. Dermed kan det være vanskelig å vite hvor en skal starte når man skal finne en passende skyløsning for en applikasjon. Med en oversikt over hva løsningene tilbyr og løser, samt fordeler og ulemper ved dem kan gjøre det lettere å velge en passende tjeneste. I lys av dette har vi kommet frem til følgende problemstilling:

> «Hvilke egenskaper, styrker og svakheter finnes ved de forskjellige alternativene for å kjøre web-applikasjoner hos de store skyplattformene?»

1.4 Struktur

Denne oppgaven er delt inn i 6 kapitler, samt referanser og vedlegg:

- **Kapittel 1 - Introduksjon** presenterer oppgaven med oppgavetekst og problemstilling, samt en kort introduksjon til skyens historie.
- **Kapittel 2 - Teori om skytjenester** forklarer teori og konsepter som er relevant for oppgaven.
- **Kapittel 3 - Metode** inneholder valg av teknologier og utviklingsmetoder for applikasjonen, samt fremgangsmåte for innhenting av vitenskapelige resultater og forskningsmetode.
- **Kapittel 4 - Resultater** presenterer de ingeniørfaglige, vitenskapelige og administrative resultatene.
- **Kapittel 5 - Diskusjon** inneholder drøfting og refleksjon rundt resultatene i kapittel 4.
- **Kapittel 6 - Konklusjon og videre arbeid** svarer på problemstillingen og presenterer muligheter for videre arbeid.
- **Referanse** presenterer benyttede kilder.
- **Vedlegg** inneholder rapportens vedlegg.

Kapittel 2

Teori om skytjenester

Dette kapittelet omhandler relevant teori innen skytjenester og systemutvikling. De ulike skyløsningene beskrives med forklaring av hvordan de fungerer. Teorien her danner grunnlaget for å kunne besvare problemstillingen definert i kapittel 1.3. Ved bruk av begrepet «bruker» menes det person som kjøper og administrerer skytjenester, for eksempel en skyansvarlig eller en utvikler.

2.1 Skytjenester

> «Skytjenester (cloud computing) er en samlebetegnelse på alt fra dataprosessering og datalagring til programvare på servere som er tilgjengelig fra eksterne serverparker tilknyttet internett.» [5]

Bruk av skytjenester kan senke utgifter, forenkle testing, skalering, tilgjengelighet og vedlikehold sammenlignet med å håndtere infrastruktur selv [17]. I de fleste tilfeller er det enkelt å skalere opp og ned ved å endre konfigurering hos skyleverandøren [18]. Skyleverandører tilbyr mange forskjellige skytjenester som kunder kan benytte seg av. Skytjenester kan deles inn i skyløsninger basert på funksjonalitet og virkemåte. Det innebærer at kunder med noen skyløsninger har mer ansvar for å sette opp skytjenesteene selv. De forskjellige skyløsningene vil forklares i de påfølgende underkapitlene.

2.2 Serverløs databehandling

Serverløs databehandling (serverless computing), er en skyløsning som skaper en lettere og mer kostnadseffektiv måte å bygge og håndtere skyapplikasjoner. Det fjerner ansvaret for driftsoppgaver fra brukeren og over

til skyplattformleverandøren. Som en konsekvens av dette kan utviklere bruke tid på å utvikle applikasjonen istedenfor å drifte den. Selv om navnet «serverløs» tyder på at det ikke benyttes en server, kjører applikasjonen på servere. Serverløs refererer heller til opplevelsen ved bruk av en tjeneste. Bruker skal oppleve at ting fungerer uten å måtte tenke på oppsett av servere [19].

2.3 Infrastructure as a Service (IaaS)

Infrastructure as a Service (IaaS) er en skyløsning der en IaaS-leverandør leverer maskin, nettverk og lagringsressurer. Betalingsmodellen er gjerne «pay-as-you-go», men noen leverandører tilbyr også forhåndsleie til en lavere pris. Ettersom en IaaS-leverandør sitter på store mengder maskinvareressurser kan brukere som regel enkelt skalere og ta i bruk nye ressurser etter behov. IaaS kan sammenlignes med det tidligere «timeshare»-konseptet, der en kunne låne datakraft av store bedrifter som hadde mer enn de trengte [20]. Ved bruk av en IaaS-modell får brukeren full tilgang til infrastruktur som deretter kan benyttes som om den var fysisk ved brukeren. Eksempeler på IaaS-tjenester er virtuelle maskiner, fillagring og last-balanserere [21].

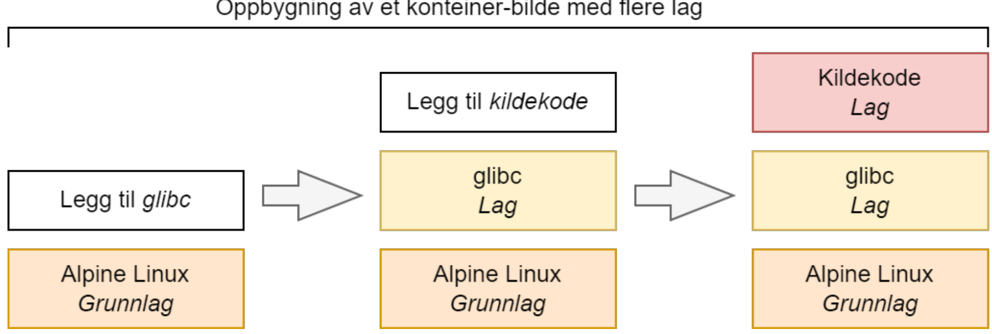
2.4 Virtuell maskin (VM)

En Virtuell maskin (VM) er et virtuelt miljø som fungerer som et en vanlig datamaskin med sin egen CPU, minne, nettverksgrensesnitt og lagring. Denne maskinen kjøres på faktisk fysisk maskin som virtualiserer tilgang til faktiske ressurser. Dermed kan én fysisk maskin kjøre flere isolerte virtuelle maskiner samtidig, uten at disse kan påvirke eller snakke med hverandre, ettersom hver VM er isolert fra resten av systemet. Den fysiske maskinen kan være lokalisert enten hos bruker eller i skyen. Maskinen som har den faktiske maskinvaren og ressursene blir ofte kalt vert-maskinen. VMene som tar i bruk ressursene til en vert, blir kalt gjeste-maskiner [22].

2.5 Konteinerbilde

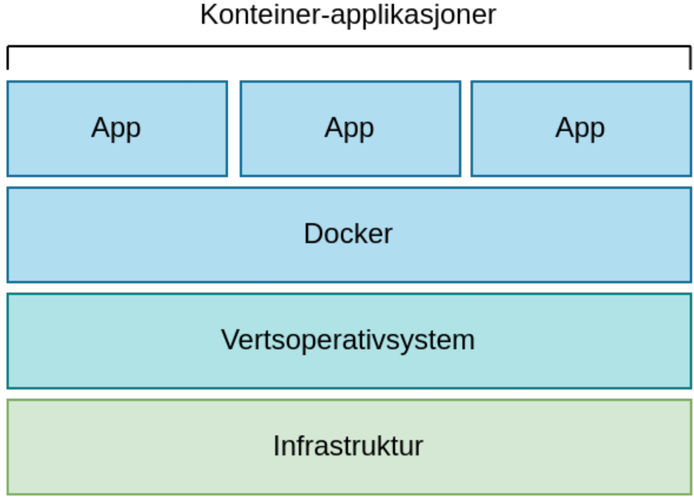
Et konteinerbilde er en uforanderlig, statisk, eksekverbar fil, litt som en .exe-fil. Den inneholder kildekode, avhengigheter, systembiblioteker og andre verktøy som et program behøver for å kjøre på en konteiner-plattform (se 2.6), slik som for eksempel Docker. Når et konteinerbilde kjøres, benytter det kjernen til operativsystemet på maskinen den kjører på. Dermed trenger ikke bildet inneholde et helt OS. Bildet kan sees på som en oppskrift på hvordan man gjenskaper en tilstand med programvare, avhengigheter og porter som skal eksponeres. Oppskriften består av opptil

flere lag på toppen av et opprinnelig lag. Hver gang en gjør endringer på et lag, lages det en ny mal som har et nytt lag på toppen av de forrige. Bilder kan dermed bestå av en rekke forskjellige lag. Ved å pakke kode lagvis, er det mulig å gjenbruke lag fra forskjellige kilder. Et bilde kan strengt tatt ikke startes direkte, men en konteiner startes fra et bilde. [23].



Figur 2.1: Oppbygning av et konteinerbilde

2.6 Konteiner



Figur 2.2: Flere konteiner-applikasjoner på samme infrastruktur

En konteiner er et virtualisert kjøremiljø som lar brukere kjøre applikasjoner isolert fra det underliggende operativsystemet. Ettersom miljøet inne i en konteiner er standardisert, er det enkelt å la hvilket som helst konteinerbilde kjøre i en konteiner. Dette muliggjør enkel deling av konteinerbilder på tvers av personer, organisasjoner og kjøremiljøer. Fordi hver konteiner er isolert fra andre konteinere, og dermed ikke kan forstyrre

andre konteinere eller maskinen som kjører den, gjør det kjøring av bilder sikkert. En annen fordel med konteinere er at virtualiseringen foregår i applikasjonslaget, i motsetning til virtuelle maskiner som virtualiserer i maskinvarelaget. Dette betyr at flere konteinere kan dele samme maskin og kjerne, og virtualisere operativsystemet for å kjøre isolerte prosesser. Dermed blir konteinere små og sparer ressursbruk. Da er det i mange tilfeller mulig kjøre samme kode på en mindre instans enn ved bruk av virtuelle maskiner. Hovedforskjellen mellom et konteinerbilde og en konteiner er at konteinerbilder kan eksistere uten en konteiner, mens en konteiner behøver et konteinerbilde for å eksistere [24], [25].

2.7 Konteiner-orkestrering

Konteiner-orkestrering automatiserer utplassering, nettverksoppsett, skalering, tilgjengelighet og livssyklus-håndtering av konteinere. Forskjellige verktøy innenfor konteiner-orkestrering løser dette på litt forskjellige måter, men i bunn og grunn så er det tre steg i konteiner-orkestrering: Først definerer man konfigurasjonen til en klynge. En klynge er en samling av konteinerbilder og andre tjenester benyttes av konteiner-orkestreringsverktøyet. Deretter håndterer verktøyet utplasseringen av konteinere ved å sette opp nødvendig ressurser for å håndtere de forskjellige konteinerne. Til slutt håndterer verktøyet livssyklusen til konteinerne basert på bildet som tilhører hver konteiner [26].

2.8 Kubernetes

Kubernetes er et ledende verktøy innenfor konteiner-orkestrering og tilgjengelig som åpen kildekode. Det håndterer automatisk utplassering, skalering og håndtering av konteineriserte applikasjoner. Ved bruk av Kubernetes definerer man oppsettet sitt ved bruk av yaml-filer. I yaml-filene defineres alt fra hvordan én app skal kjøre, til hvordan kommunikasjon mellom apper skal foregå, hvordan appene skal få snakke med omverdenen og hvordan skalering skal håndteres.Tilstanden er også immutabel som betyr at ved hver oppdatering så opprettes en ny versjon av tjenesten. Hovedkonsepter innenfor Kubernetes er klynger, noder, pods, deployments, services og ingresser:

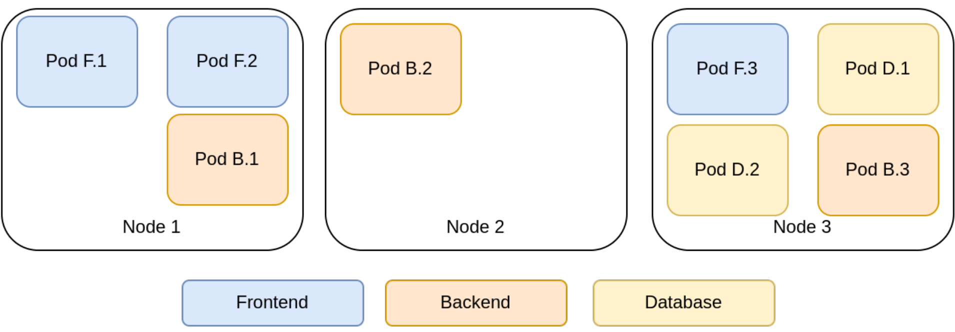
Klynge er et sett med noder som kjører konteineriserte applikasjoner. En klynge kan tenkes på som hetten til hele Kubernetes applikasjonen [27].

Node er en virtuell eller fysisk maskin som inneholder pods. Kubernetes kjører arbeidsmengden ved å plassere konteinere i pods, hver node er har et kontrollplan som innholder de forskjellige tjenestene som trengs for å kjøre en pod. Masternoden kontrollerer tilstanden til hele klyngen. Arbeidsnoder kjører applikasjoner som handteres av masternoden [28].

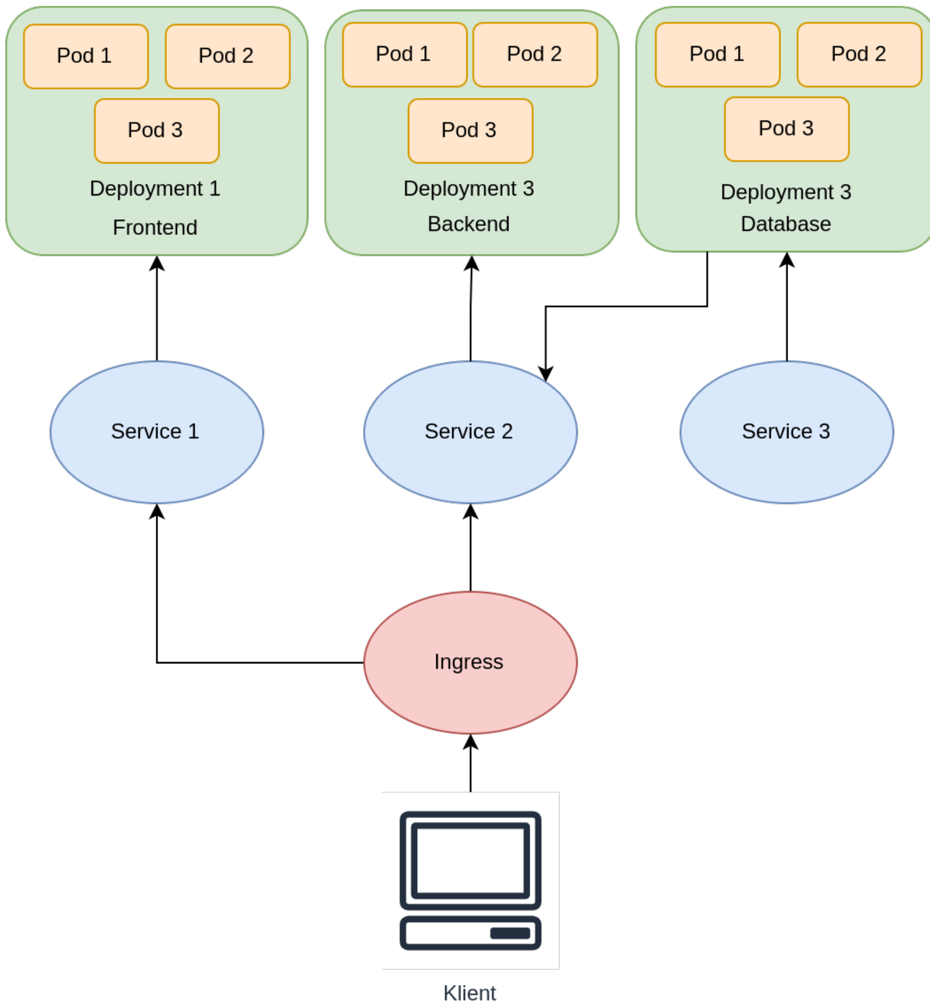
Pod består typisk av en konteiner og spesifikasjon for hvordan denne skal kjøres. Det er også mulig å kjøre flere tett sammenkoblede konteinere i samme pod som da deler lagring og nettverksressurser. Innholdet i en pod er samlokalisert, samplanlagt og i samme kontekst. Pods er ment til å ha kortvarige liv fordi en pod er immutabel. Når det gjøres en endring i oppsettet av en pod vil en ny pod lages [29], [30].

Deployment inneholder en samling av samme type pod og en definert ønsket tilstand for denne samlingen. Sammen med dette defineres også et ønsket antall pods som skal eksistere i samlingen til en hver tid. En deployment definerer også hvordan vi skal håndtere oppdateringer når nye versjoner av konteinerbilder rulles ut. Deployment håndterer antall pods automatisk. Om 3 pods kræsjer vil en deployment starte 3 nye pods i klyngen [30], [31].

Service tilbyr et stabilt endepunkt til kommunikasjon med en eller flere pods, uavhengig av oppdateringer av de underliggende podsene. Det finnes flere forskjellige typer services, en type er for eksempel last-balanserer som kan delegere trafikk til forskjellige pods [30], [32].



Figur 2.3: Eksempel på fordeling av pods i noder



Figur 2.4: Eksempel på en Kubernetes-klynge

2.9 Container as a Service (CaaS)

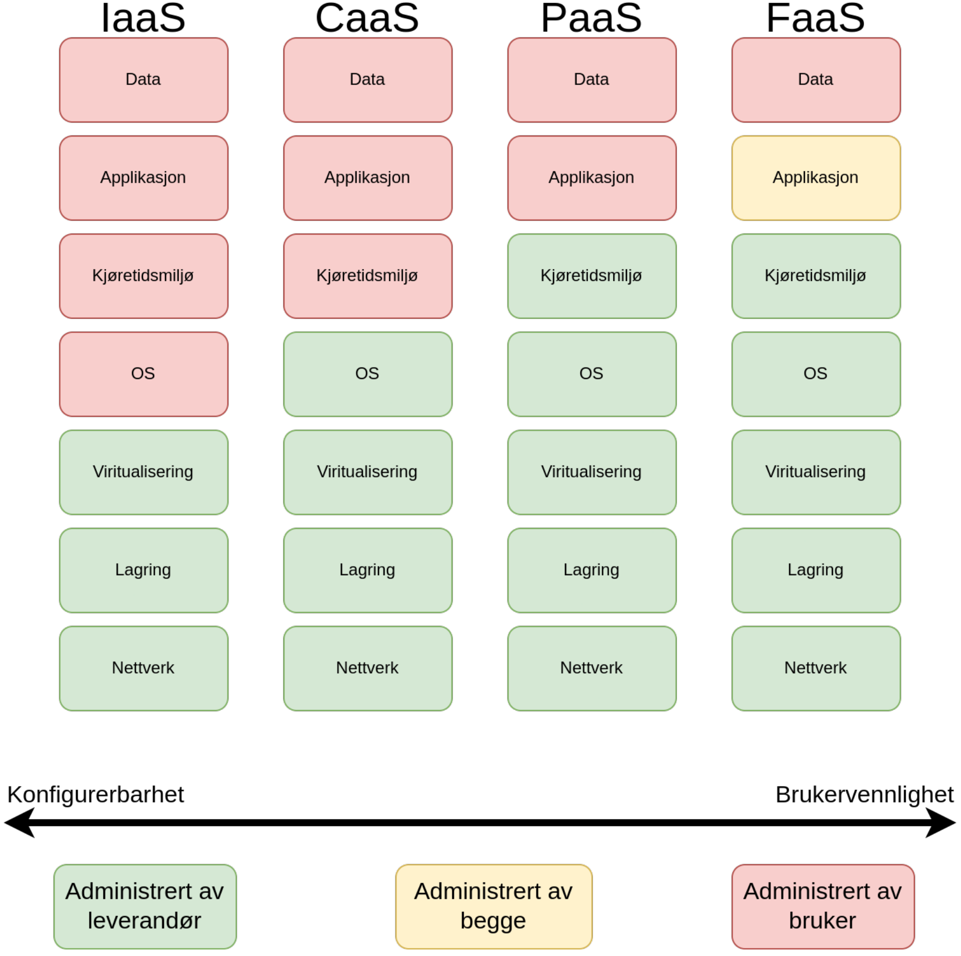
Container as a Service (CaaS) er en skyløsning som fokuserer på å kjøre konteinere. Med CaaS behøver ikke bruker å ta hånd om den underliggende infrastrukturen som applikasjonen kjører på. Det finnes flere forskjellige modeller innen CaaS. Det er mulig å kjøre en applikasjon kontinuerlig, eller kun ved spesifikke hendelser som for eksempel en HTTP-forespørsel. Det er vanlig at pris defineres ut fra hvor lenge applikasjonen har kjørt. Ettersom CaaS-løsninger benytter konteinere er det opp til brukeren hva slags språk eller kjøremiljø applikasjonen tar i bruk. Enkelte CaaS-løsninger er også bare en abstraksjon av et konteiner-orkestrerings-verktøy med mindre kompleks konfigurasjon [33].

2.10 Platform as a Service (PaaS)

Platform as a Service (PaaS) er en skyløsning der PaaS-leverandøren tilbyr en fullstendig skyplattform med maskinvare, programvare og infrastruktur. Ved bruk av PaaS blir servere, nettverk, lagring, operativsystem, software, databaser og utviklingsverktøy både vedlikeholdt og styrt av leverandøren. Med en slik modell er det vanlig å enten betale for en gitt kvote, eller bruke en «pay-as-you-go»-modell der brukeren kun betaler for ressursene som benyttes. Uavhengig av betalingsmetode lar PaaS-modellen brukere bygge, teste, publisere og oppdatere applikasjonene sine raskere og mindre kostbart enn hvis man selv skulle håndtert fysisk infrastruktur. PaaS-løsninger tilbyr vanligvis ulike kjøremiljøer slik at bruker selv kan velge hvilket som skal brukes. For eksempel kjøremiljøer for Node.js, Python, Java eller andre programmeringsspråk, samt kjøring av konteinerbilder i konteinere [34].

2.11 Function as a Service (FaaS)

Function as a Service (FaaS) blir ofte omtalt sammen med serverløs databehandling ettersom det er en undertype av sistnevnte. Det som skiller FaaS fra andre serverløse skytjenester er at den kun kjører som respons til hendelser. Når en bruker FaaS skriver en dermed funksjoner med definerte triggere som forteller når funksjonen skal kjøres. Slike triggere kan for eksempel være HTTP-kall eller endringer i en database. Når funksjonen ikke kjører så betaler en heller ikke noe. Funksjonene skalerer som regel automatisk. Mange mener at FaaS gir brukere en raskere vei til markedet, ettersom det tar kortere tid fra du starter å skrive kode til den er klar for å motta trafikk. Dette fordi det tar lenger tid å skrive et API selv enn kun å lage funksjoner [35].



Figur 2.5: IaaS vs CaaS vs PaaS vs FaaS

2.12 Infrastructure as Code (IaC)

Infrastructure as Code (IaC) er håndtering av infrastruktur som nettverk, VMer, last-fordelere og lignende gjennom kode. Dermed kan infrastruktur bli en del av versjonskontroll på samme måte som kildekode. En IaC-modell vil generere samme miljø hver gang den kjøres, på samme måte som kode produserer samme produkt hver gang den kompileres. Fordelen med å skrive infrastruktur som kode er dermed reproduserbarhet, enkel mulighet til å finne tidligere konfigurasjoner og mindre sannsynlighet for manuelle feil. Alternativet til IaC er gjerne bruk av kommandolinjeverktøy eller grafiske brukergrensesnitt på skyleverandørens nettside. Der kan det i større grad skje manuelle feil og det er vanskeligere og mer tidkrevende å reprodusere infrastruktur. Det er to forskjellige måter å ta i bruk IaC: deklarativt og imperativt [36]:

2.12.1 Deklarativ IaC

Med en deklarativ framgangsmåte definerer brukeren modellen og alle ressurser som systemet trenger statisk. Modellen blir så satt opp av et IaC-verktøy. Nåværende tilstand lagres på et eget lagringssted. Når brukeren gjør endringer på modellen sammenlignes den nye modellen og eksisterende tilstand for å finne ut hvilke endringer som skal gjennomføres. Ettersom nåværende tilstand lagres er det veldig lett å ta ned all eksisterende opprettet infrastruktur om ønskelig [37].

2.12.2 Imperativ IaC

Med en imperativ framgangsmåte definerer brukeren de spesifikke kommandoene som er nødvendig for å oppnå den riktig oppsett av systemet. Disse kommandoene må da bli utført i riktig rekkefølge. Dette kan for eksempel skrives som et script med nesten hvilket som helst programmeringsspråk. Med en imperativ IaC må man selv finne ut hvordan oppdateringer skal utføres på systemet [37].

Kapittel 3

Metode

Dette kapittelet beskriver hvordan vi har gått frem ved sammenligning av skyløsningene. Dette innebærer hvilke teknologier som er valgt ved utvikling av applikasjonen, hvordan applikasjonen er brukt til testing, valg av verktøy for å bruke skyløsningene, hvilke parametre vi har sett på for å sammenligne skyløsningene og hvilke parametre som er brukt for hver type skyløsning.¨

3.1 Applikasjon

Vi hadde i dette prosjektet bruk for en web-applikasjon til utprøvingen og analysen av de forskjellige skytjenestene. For at resultatene skulle være relevante måtte web-applikasjonen gjenspeile hvordan en reell applikasjon ser ut og fungerer. Bruk av en database, potensiell tidskrevende filtering av data, autentisering og noe funksjonalitet ble ansett som viktig for at web-applikasjonen skal kunne regnes som reell. Det var også viktig at applikasjonen ble utviklet slik at skytjenestene ble benyttet på den måten de er laget for å brukes. FaaS er for eksempel bygd for å kjøre funksjoner og ikke hele API-er, og reelle funksjoner har dermed vært viktig å lage for å teste FaaS.

3.2 Valg av teknologi

3.2.1 Quarkus

Quarkus [38] er et Java/Kotlin-rammeverk for å lage API-er. Quarkus har samlet flere hundre mye benyttede Java-biblioteker og gjort det enkelt å bruke disse i applikasjonen. Rammeverket har åpen kildekode, er i konstant utvikling og har et stort felleskap som er med på å utvikle og dis-

kutere hvordan rammeverket skal utvikles videre. Quarkus ble laget for å løse flere problemer med gamle Java applikasjoner [39], blant annet lange oppstartstider og stort forbruk av minne. Som et moderne rammeverk har Quarkus fokus på blant annet enkel kjøring i skyløsninger. Rammeverket har en såkalt «konteiner først»-mentalitet [40]. Kantega anbefalte i tillegg rammeverket og det fremstod dermed totalt sett som et godt valg.

3.2.2 NextJS

NextJS [41] er et React-rammeverk for å lage nettsider. Det tilbyr funksjonalitet som blant annet Server-side Rendering (SSR) og Static Site Generation (SSG). SSR innebærer at HTML genereres på serveren før den sendes til brukeren. Dette fører til at innlasting av sider som bruker data fra eksterne kilder gjerne oppleves som raskere. I tillegg unngår en at innhold flyttes rundt på siden etter innlasting fordi mer data lastes inn. Dette kan føre til en bedre brukeropplevelse. Med SSG genereres HTML-en ved bygging. Dette er nyttig om innholdet som vises frem er statisk eller sjeldent endres slik at det kan hentes inn ved bygging og lagres i bygg-filene. Om hele applikasjonen kan bygges statisk er ikke kjøring på server nødvendig. NextJS optimaliserer dette automatisk slik at dynamisk innhold kan bruke SSR, mens statisk innhold bruker SSG. Kantega oppfordret oss til å velge dette rammeverket. Delvis fordi de skulle arbeide videre med applikasjonen etter prosjektet og ønsket å kjenne til teknologiene som brukes, men også fordi de hadde gode erfaringer med NextJS.

3.2.3 PostgreSQL

PostgreSQL [42] er et gratis relasjonsdatabase-system med åpen kildekode. Det ligner på MySQL, som vi har brukt gjennom studiet og har gode erfaringer med. Quarkus har god støtte for PostgreSQL, noe som gjør utvikling enklere. I Stack Overflow sin utviklerundersøkelse for 2021 var PostgreSQL databasen flest ønsket å ta i bruk [43], noe vi så på som en fordel ettersom populære teknologi ofte er trygge valg.

3.2.4 Docker

Docker [44] er en plattform for konteinerisering med åpen kildekode. Ved bruk av Docker definerer man en «Dockerfile» som er oppskriften på hvordan konteinerbildet skal se ut. Det er mulig å konteinerisere applikasjoner uten å bruke Docker, men vi har valgt Docker fordi den er mest brukt og enkel å bruke. Docker er fordelaktig under utvikling ettersom kildekode kjøres likt uavhengig av utviklingsmiljø. I tillegg forenkler det testing og bruk av skyløsninger ettersom det er enkelt å lage konteinerbilder.

Docker Compose

Docker Compose [45] er et verktøy som lar deg definere en applikasjon med flere konteinere. Dette gjøres med *yaml*-filer der hver konteiner er en tjeneste med hver sin konfigurasjon. Konteinerne i de ulike tjenestene kan enkelt kommunisere med hverandre internt uten at tjenestene må være eksponert mot omverdenen.

3.2.5 Pulumi

Pulumi [46] er et IaC verktøy som lar brukere skrive kode i mange forskjellige programmeringspråk for å definere den ønskede infrastrukturen. Pulumi lar deg skrive oppsettet av infrastrukturen som en helt vanlig applikasjon, med løkker, funksjoner, kondisjoner, klasser og mye mer. I dette prosjektet ble Pulumi valgt for å gjøre det enkelt å sette opp de forskjellige skyløsningene som skal testes. Pulumi gjør det også enkelt å endre konfigurasjonen av skytjenesten som skal testes.

3.2.6 Eksterne tjenester

Applikasjonen har tatt i bruk to skybaserte eksterne tjenester, med mål om å gjøre utviklingen av web-applikasjonen lettere og raskere. Auth0 [47] er en autentiseringsplattform som forenkler brukerautentisering slik at vår applikasjon ikke må håndtere sikkerhet rundt oppretting og lagring av brukerpassord og autentisering. Alternativer som Firebase Authentication og AWS Cognito ble også vurdert, men ikke valgt.

Algolia [48] er en indeksbasert søkemotor. Den gjør det enkelt å implementere søk og filtrering i store datamengder. Dette skjer ved at applikasjonen sender data til Algolia som blir indeksert. Deretter kan Algolia ta hånd om hele søket. Dette sparer mye tid under utvikling, samtidig som det gir et søk med god kvalitet.

Bruken av disse eksterne tjenestene var ikke særlig viktig for utprøvningen og sammenligningen av skyplattformene, men heller nyttige hjelpemidler i arbeidet med å lage en god applikasjon. De eksterne tjenestene som er benyttet er mye brukt i mange applikasjoner. De er dermed godt utprøvd og det er dokumentert at de gir nyttig funksjonalitet til applikasjoner som en dermed ikke behøver å lage selv.

3.3 Valg av utviklingmetode

3.3.1 Versjonskontroll

Vi valgte å ta i bruk GitHub som versjonskontrollverktøy. Dette ble hovedsaklig valgt fordi oppdragsgiver ønsket dette. I tillegg hadde vi mye erfaring med GitHub og kunne dermed raskt sette opp blant annet CI/CD med GitHub Actions. Pull requests har også blitt benyttet for kvalitetssikring av ny kode. Andre mye benyttede alternativer som GitLab, Bitbucket og Azure DevOps Repos ble også sett på, men ikke vurdert som mer hensiktsmessige enn GitHub for dette prosjektet på grunn av argumentene nevnt over.

3.3.2 Kanban

Det ble i dette prosjektet valgt å gå for en arbeidsmetodikk basert på Kanban [49]. Kunden applikasjonen har blitt utviklet for er en oppstartsbedrift. Dermed hadde de ikke hadde en fast definisjon på hvordan produktet skulle bli og hva det skulle inneholde. Kanban ble vurdert som hensiktsmessig fordi prioriteringer fortløpende kunne tilpasses under prosjektet, basert på kundens ønsker. Dette var mulig fordi et av hovedkonseptene i Kanban er omprioritering av oppgaver og kontinuerlig arbeid. Kanban er forøvrig også en metodikk som fungerer godt for mindre grupper. På bakgrunn av dette ble Kanban vurdert som et godt valg.

3.3.3 Kanban-brett

Ettersom Kanban ble benyttet var det viktig å ta i bruk et Kanban-brett fordi dette er et av hovedprinsippene innenfor Kanban. Vi valgte GitHub sitt *issue board*. Det gjør det mulig å lett knytte utviklingen i form av pull requests sammen med *issues* som lot oss ha en god oversikt over hva som hadde blitt gjort, og hva som måtte gjøres. Det ble også satt en WIP-grenser på *In progress* kolonnen og *In review* kolonnen til fire. Dette betyr at det kun kan være fire issues i den raden samtidig, før noe må flyttes for å frigjøre plass. Dette gjør at hvert team medlem i teorien bare kan jobbe på to issues om gangen og må stoppe opp å hjelpe andre issues i mål hvis det ikke er mer plass i kolonnene.

3.4 Forskningsmetoder

«Forskningsmetoder er framgangsmåter som benyttes i vitenskapelig forskning» [50]. Systematiske framgangsmåter brukes «for å etablere pålitelig kunnskap og holdbare teorier» [50]. Metodene som benyttes avhenger av

hva som skal undersøkes og hvilke data som finnes. Her skilles det mellom kvalitative og kvantitative metoder:

3.4.1 Kvalitativ metode

Kvalitativ metode er en metode der man undersøker gjennom å tolke og forstå. Heller enn å undersøke et stort antall forekomster ved bruk av statistikk (som ved kvantitativ metode) vil kvalitativ metode typisk bli brukt på noen få forekomster [51], [52].

3.4.2 Kvantitativ metode

Kvantitativ metode er en metode der man undersøker et stort antall data. Dette kan for eksempel være tall eller noe som kan uttrykkes som en mengde. Gjentagende eksperimenter kan for eksempel være et kvantitativt eksperiment [53].

3.5 Parametre for sammenligning

All sammenligning ble gjort ved å rulle ut backend-løsningen av applikasjonen til skyen. Det betyr at alle tester og analyse har disse forutsetningene. Som beskrevet i kapittel 1.2.1 var pris, anvendbarhet, robusthet og resiliens, skalering og leverandørinnlåsing valgt ut som parametre i sammenligningen.

3.5.1 Pris

Det er ikke enkelt å sammenligne pris på tvers av ulike type skyløsninger, som for eksempel PaaS og Kubernetes. Skyløsningene inneholder forskjellig funksjonalitet og gir dermed bruker forskjellig verdi basert på bruksbehov. I tillegg tilbys det for hver løsning forskjellige størrelser på instansene som brukes. Det innebærer gjerne at bruker kan velge både CPU-kraft og minnestørrelse. Dette påvirker også sluttprisen. Det er derimot mulig å sammenligne pris for samme skyløsning som tilbys av forskjellige leverandører ettersom de gir samme verdi for bruker. Vi har fremskaffet kvantitative data for pris-forskjeller mellom instanser av sammenlignbare størrelser, for eksempel en instans med én CPU og 4 GB minne.

3.5.2 Anvendbarhet

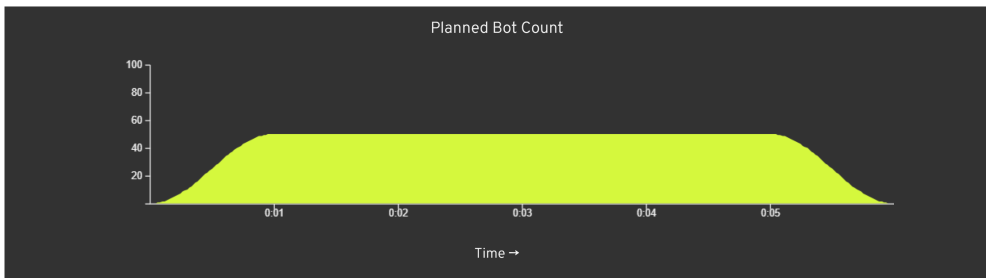
Forskjellige applikasjoner har ulike behov ved kjøring. Dermed er det viktig at skyløsningen som velges er anvendbar nok til å støtte behovene til

applikasjonen. Dette innebærer hvilke typer applikasjoner som er mulig å kjøre i skyløsningen og på hvilken plattform. Et mulig ønske kan være å kjøre en Java-applikasjon uten å selv pakke den inn i en konteiner. For konteinere kan et relevant spørsmål være om det er mulig å kjøre flere sammen og la dem kommunisere internt. Noen ønsker å kjøre et program i 15 minutter hver dag, men ikke betale for datakraft resten av døgnet.

Applikasjoner som skal eksponeres for en bruker ønsker gjerne å kunne koble til egne domener og bruke SSL-sertifikater for å sikre siden med HTTPS. Interne applikasjoner som allmennheten ikke skal ha tilgang til bør nok heller ligge i et VPC. Mange applikasjoner har stor nytte av å avdekke nye feil før hele brukermassen ser den, i disse tilfellene kan mulighet til å dele trafikk mellom versjoner være nyttig. Anvendbarhet handler oppsummert i stor grad om hvilke bruksområder skytjenestene dekker og støtter, samt funksjonalitet som tilbys. For å undersøke anvendbarhet har vi gjennomført kvalitative undersøkelser av skyleverandørenes dokumentasjon.

3.5.3 Ytelse og cold start

For å teste ytelsen til løsningene har vi brukt tjenesten Loadster [54] til å gjennomføre et kvantitativt eksperiment. Loadster tilbyr last-tester av API-er med resultater som inneholder mange forskjellige metrikker, blant annet gjennomsnittlig responstid. Testen foregikk ved en naturlig oppskalering fra 0 til 50 roboter i løpet av ett minutt, 50 roboter i 4 minutter og en naturlig nedskalering i ett minutt. De tilgjengelige robotene sendte forespørsler kontinuerlig til forhåndsdefinerte endepunkter i applikasjonen.



Figur 3.1: Strategi for last-testing. X-akse viser tid i timer, y-akse viser antall bot's

TEST FINISHED

DURATION	CONCURRENT BOTS	ITERATIONS	HITS	ERRORS
0:05:19	590 / 590	22551	112755	0
AVG RESPONSE TIME	**P50 RESPONSE TIME**	**P90 RESPONSE TIME**	**DOWNLOADED**	**UPLOADED**
0.74 s	0.12 s	3.63 s	277.1 MB	15.5 MB

Figur 3.2: Eksempel på deler av resultat fra Loadster

Cold start skjer når en applikasjon starter opp for første gang i en instans.

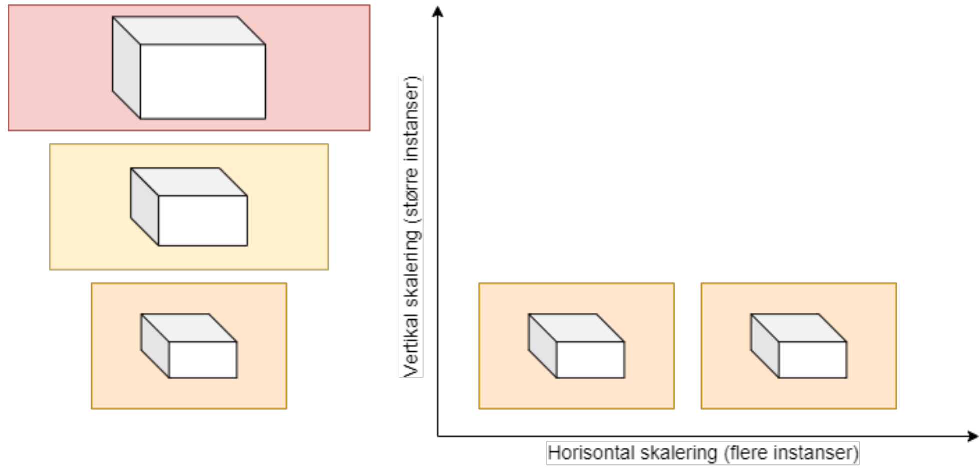
Dette skjer i praksis når instanser opprettes og applikasjonen da skal startes i den nye instansen. Dette er viktig for applikasjoner i skyen som kan skalere til null, ettersom forespørsler da vil føre til en cold start. I disse tilfellene er det viktig at tiden en cold start tar er så lav som mulig slik at brukeren ikke opplever applikasjonen som treg. For å teste hvor lang tid en cold start tar har det blitt brukt et script som kaller et HTTP-endepunkt i applikasjonen og måler tiden det tar før en respons mottas. Dette scriptet har blitt kjørt flere ganger, der instansen ble stoppet mellom hver kjøring, for å finne en gjennomsnittlig responstid.

3.5.4 Skalering

Om trafikken til en applikasjon varierer, er et naturlig ønske å automatisk kunne justere den tilgjengelige kapasiteten basert på trafikkmengden. I disse tilfellene ønsker en å kunne sette opp autoskalering. Når en skal skalere opp er det to måter dette kan løses på. En kan enten gjøre instansen større ved å gi den mer minne og prosesseringskraft, dette kalles vertikal skalering. Den andre metoden er å opprette flere instanser og dele trafikken mellom dem. Det kalles horisontal skalering.

Hver metode har sine styrker og svakheter. Vertikal skalering kan bare gjøres opp til en viss størrelse, mens horisontal som regel kan skaleres til så mange en ønsker. På den andre siden må en tenke gjennom og kanskje endre kode for å støtte horisontal skalering, fordi det blir kjørende flere instanser samtidig. Det kan være enklere å forholde seg til én instans. En ulempe med vertikal skalering er at en får et enkelt feilpunkt, om instansen feiler er det ingen annen instans som kan ta imot trafikk, noe som unngås med horisontal skalering. Horisontal skalering kan på lik linje med vertikal skalering ikke umiddelbart senke responstid som øker ved økning i trafikk, fordi nye instanser må gjennom en cold start før de kan ta imot trafikk.

Vi har brukt kvalitativ evaluering av dokumentasjon, samt manuell utprøving for å se sett på muligheter for skalering både vertikalt og horisontalt. Vi har også undersøkt om en kan bestemme øvre og nedre grense for autoskalering. Øvre grense er nyttig for å ikke skalere langt forbi budsjett, mens en nedre grense er praktisk for å unngå for mange cold starts. I tillegg har vi sjekket i hvilken grad brukeren selv kan bestemme kriteriene for når applikasjonen skal skalere, for eksempel basert på utnyttelsesgraden til CPU-en.



Figur 3.3: Vertikal og horisontal skalering

3.5.5 Leverandørinnlåsing

Når en velger en skytjeneste er det viktig å vurdere om det kan oppstå problemer hvis skytjenesten i fremtiden legges ned eller endres. Hvis man velger en skytjeneste som binder deg til en spesifikk skyleverandør, risikerer en å måtte godta endringer uten et reelt mulighetsrom for å bytte leverandør. Hvis det derimot velges en skytjeneste som er basert på åpen kildekode og/eller åpne standarder er det stor sannsynlighet for at det også finnes andre tilbydere av samme tjeneste. Da er det mulig å enkelt bytte tilbyder om ønskelig. Det er likevel ikke nødvendigvis en ulempe at en skytjeneste kun er tilgjengelig hos én leverandør. Man bør bare være klar over det. Det kan og være klokt å gjennomføre risikovurdering rundt hvorvidt eventuelle endringer kan tolereres og om en tror tjenesten vil eksistere i lang tid.

Vi har sett på hvorvidt skytjenester binder brukeren til en plattform eller ikke. For å undersøke dette har vi tatt i bruk resultatene fra både anvendbarhet og skalering, samt dokumentasjon og egen utprøvning av tjenestene i en kvalitativ analyse.

3.6 Analysemetode

Her kommer en oversikt over hvilke typer skyløsninger vi har sett på og hvordan de har blitt analysert. Dette innebærer hvilke parametre vi har sett på for skyløsningen og hvilke spesifikke skytjenester Azure, AWS og GCP tilbyr for hver av dem.

3.6.1 FaaS

Pris var et naturlig parameter ettersom funksjoner har samme oppgave uavhengig av plattform, og er uten store forskjeller i funksjonalitet. Innen anvendbarhet har vi sett på hva som faktisk er mulig å kjøre i en FaaS. For eksempel kun funksjoner skrevet spesifikt for plattformen eller mer generelle funksjoner og konteinere.

Vi har også sett på hvilke mulige triggere som støttes og hvor lenge funksjonene kan kjøre etter at de blir startet. Vi har ikke sett på ytelse ettersom FaaS i utgangspunktet er laget for å skalere etter bruk. Dermed skal ytelsen i utgangspunktet ikke endres ved trafikkøkninger, fordi det opprettes nye instanser. Skalerbarhet har på bakgrunn av dette vært et viktig punkt. Både for å se hvilke instans-størrelser som er tilgjengelig, men også om en kan kontrollere skaleringen med hensyn til både minimum og maksimum antall instanser. Vi har også kjørt tester som måler hvor lang tid en cold start tar.

I tillegg har vi sett på forskjellene mellom plattformene for å gjøre en vurdering på hvor mye arbeid det er å flytte en funksjon fra en plattform til en annen.

Tabell 3.1: FaaS-løsninger som har blitt sammenlignet

Leverandør	Produkt
Azure	Functions
AWS	Lambda
Cloud	Functions

3.6.2 PaaS

Prisforskjeller mellom de ulike leverandørene har naturligvis blitt sammenlignet. Her har vi undersøkt produksjonsinstanser i tre forskjellige sammenlignbare størrelser. I tilfellene der en plattform tilbyr kjøring på både Linux og Windows har det blitt sett på Linux. Vedrørende anvendbarhet har vi sett på hvor fri brukeren er til å kjøre applikasjoner, hvilke kjøretidsmiljøer som er støttet og hvorvidt det er mulig å kjøre konteinere.

Vi har studert mulige forskjeller i ytelse mellom kjøring av kildekode sammenlignet med kjøring av konteinere for plattformene som støtter begge deler. Dette ble testet med last-tester på samme instans-størrelser på alle tre plattformer. For skalering har vi vurdert mulighetene for skalering, både vertikalt og horisontalt, samt om løsningen støtter automatisk skalering og konfigureringsmulighetene rundt dette.

Vi har også undersøkt om kjøring av kildekode krever et spesifikt oppsett som ikke er direkte overførbart til andre plattformer.

Tabell 3.2: PaaS-løsninger som har blitt sammenlignet

Leverandør	**Produkt**
Azure	App Service
AWS	Elastic Beanstalk
GCP	App Engine

3.6.3 CaaS

For CaaS har vi sammenlignet pris mellom leverandørene på tre forskjellige bruksområder der det tilbys: kontinuerlig kjøring av instans, samt aktiv og inaktiv instans. En inaktiv instans er gjerne en instans som kun har tilgang til minne, men ikke CPU, for å redusere tiden en cold start tar. For å analysere anvendbarheten til skytjenestene har vi vurdert mulighetene til å velge hvilken tilstand konteineren skal ha mellom forespørsler, for eksempel å skalere ned til null.

Vi har også testet om det er mulig å kjøre flere konteinere sammen slik at de enkelt kan kommunisere uten å bruke virtuelle nettverk, samt mulighet til å knytte en applikasjon til vedvarende lagring. Tilknytning til egne domener og SSL-sertifikat har også blitt utforsket. I tillegg har vi undersøkt mulighetene for skalering. Det innebærer om det er mulig å skalere vertikalt og horisontalt, og om det isåfall er mulig å sette opp automatisk skalering basert på metrikker som antall forespørsler og CPU-bruk.

Vi har kommet frem til at å se på om løsningene kan skape problemer i forhold til leverandørinnlåsing ikke er relevant. Dette fordi konteinere i seg selv er en åpen standard som alle løsningene må støtte for å kjøre dem.

Tabell 3.3: CaaS-løsninger som har blitt sammenlignet

Leverandør	**Produkt**
Azure	Container Instances
Azure	Container Apps
AWS	App Runner
AWS	Fargate
GCP	Cloud Run

3.6.4 Konteiner-orkestrering

For konteiner-orkestrering har vi hovedsakelig vurdert muligheten for om det er verdt det å bruke ekstra tid og energi på å lære seg for uerfarne, sammenlignet med mindre fleksible men enklere løsninger slik som CaaS.

Kubernetes er i praksis løsningen innen konteiner-orkestrering som benyttes pr. i dag. Vi har sammenlignet pris mellom leverandørene på tre forskjellige instans-størrelser for kjøring av et Kubernetes-oppsett.

Kubernetes [55] tilbyr last-balansering, automatisk horisontal skalering, automatisk restart ved hendelser, orkestrering av lagring og flere andre nyttige løsninger. Vi har sett på hvordan dette oppleves å sette opp og konfigurere basert på at vi ikke har noe særlig erfaring med Kubernetes fra før. Dette har vi gjort ved å kjøre web-applikasjonens backend og database i en Kubernetes-klynge. På den måten har vi testet både eksponering av applikasjonen, samt internt kommunikasjon i klyngen.

Tabell 3.4: Kubernetes-løsninger

Leverandør	**Produkt**
Azure	Kubernetes Service (AKS)
AWS	Elastic Kubernetes Service (EKS)
GCP	Google Kubernetes Engine (GKE)

3.6.5 Virtuell maskin

Virtuell maskin (VM) var den første skytjenesten som ble tilbudt og ligger i stor grad til grunn for mange av dagens skytjenester. IBM [56] har definert DevOps-support, testing av nye operativsystemer, undersøkelse av skadevare, kjøring av programvare som trenger et annet type OS og å surfe trygt på nettet som bruksområdene til en VM. Google [57] og Microsoft [58] definerer også bruksområdene relativt likt som IBM. De ovennevnte bruksområdene er ikke relevant for vår web-applikasjon, og derfor valgte vi å se bort i fra VM i vår analyse. Gjennom samtaler med vår oppdragsgiver Kantega, har vi også kommet frem til at direkte bruk av VM-er i dag ikke lenger regnes som et moderne alternativ for kjøring av web-applikasjoner. Dette er også en grunn til å utelukke VM-er fra sammenligningen.

Kapittel 4

Resultater

I dette kapittelet skal resultatene fra analysen rundt skyløsninger presenteres. Alle resultatene presentert i dette kapittelet skal bidra til å besvare problemstillingen presentert i kapittel 1. Hvert resultat er basert på metodebeskrivelsen fra kapittel 3 og er delt inn i resultater for hver type skyløsning.

4.1 Ingeniørfaglige resultater

Her skal vi studere resultatet fra utviklingsdelen av prosjektet hvor vi har utviklet en web applikasjon. Vi vil også undersøke hvordan denne applikasjonen har bidratt til innhentingen av de vitenskapelige resultatene. Selve produktet er underlagt en taushetserklæring og vi kan ikke dele selve forretningsidéen. Derfor har vi i bilder av web-applikasjonen byttet ut enkelte ord, skjult enkelte elementer, samt kun avbildet selve funksjonaliteten og ikke nødvendigvis hele siden for å unngå å vise frem forretningsidéen.

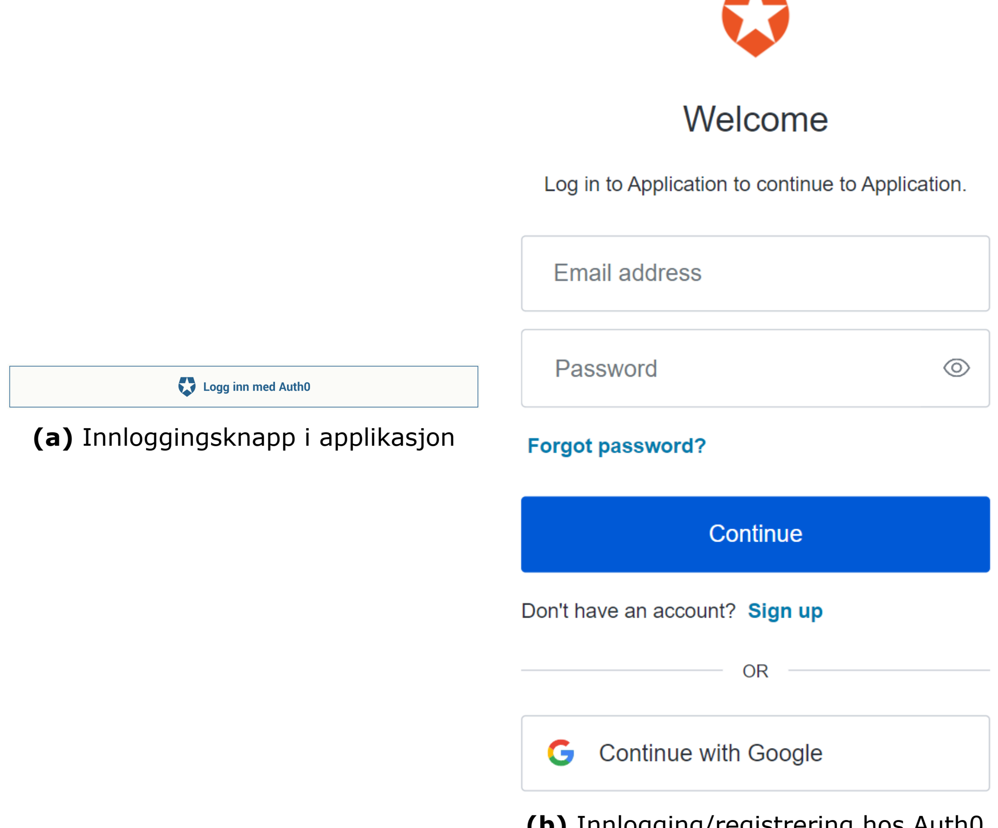
4.1.1 Applikasjon

Det var, som forklart i metodekapitlet, vesentlig å bruke en reell web-applikasjon som grunnlag i testene våres. Som et resultat av dette har applikasjonen blitt utviklet slik at det ville være lett å utføre de forskjellige testene på applikasjonen. Vi har hatt ukentlige møter med oppdragsgiver der vi har vist frem status på applikasjonen og mottatt ønsker for videre utvikling. Dette har sørget for at applikasjonen er representativ for et reelt bruksbehov, noe oppdragsgivers bruk av applikasjonen til verifisering av markedsinteresse også har vist. APIet tar i bruk en PostgreSQL-database for å lagre data og har mange endepunkter. Applikasjonen inneholder brukerautentisering med Auth0 (3.2.6) og alle endepunkter er beskyttet ved hjelp av dette. Det er også tatt i bruk andre former for tilgangsstyring,

blant annet for å sikre at brukere kun skal ha tilgang til å se og redigere egen data. Det har også blitt implementert en hel del funksjonalitet som gir applikasjonen bruksverdi for oppdragsgiver.

Brukerhåndtering og sikkerhet

Vi har som nevnt tatt i bruk Auth0 som verktøy for brukerautentisering. I applikasjonen er dette implementert ved at nettsiden sender bruker til Auth0 sitt nettsted der bruker kan logge inn eller opprette en ny bruker, vist i figur 4.1a og 4.1b. Brukeren blir sendt tilbake til vår applikasjon med en tilgangsnøkkel, «access token» på engelsk, som ved dekryptering inneholder en unik id. Ved nyregistrering blir bruker sendt til en egen side der den må oppgi eget navn, hvorpå vi lagrer brukeren i vår egen database med brukeren sin unike id som primærnøkkel. Ved senere forespørsler til APIet vårt som inneholder brukerens tilgangsnøkkel i HTTP-forespørselen kan vi da verifisere at nøkkelen fremdeles er gyldig, samt at brukeren faktisk har tilgang til det som etterspørres. Nye endepunkter er som standard helt stengt og må åpnes manuelt med logikk for tilgangsstyring slik at brukere kan benytte det. Dette øker sikkerheten i applikasjonen.



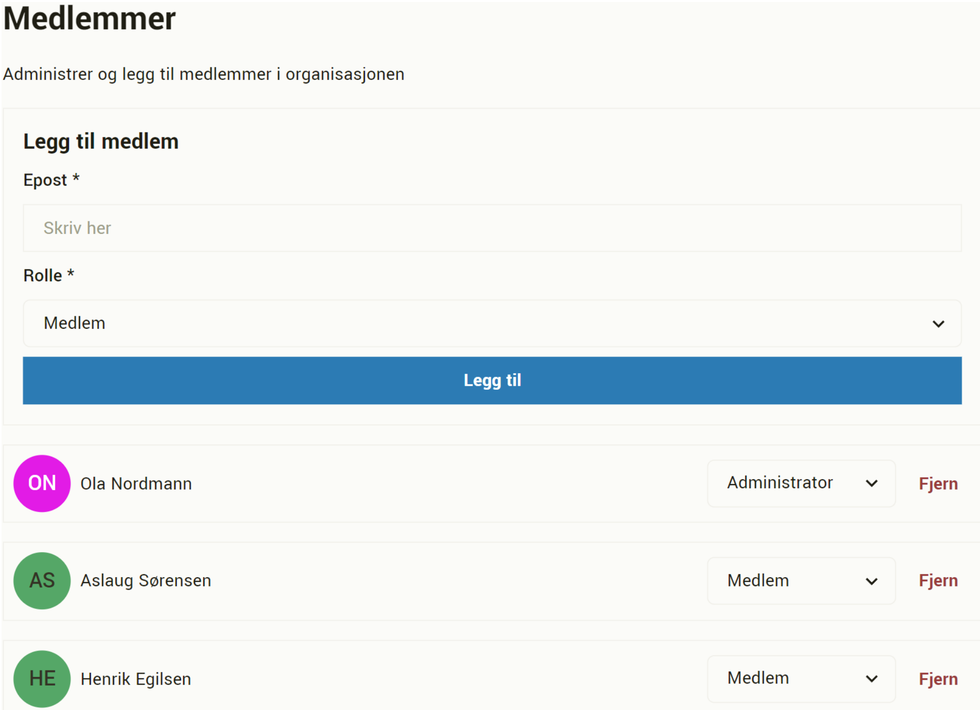
(a) Innloggingsknapp i applikasjon

(b) Innlogging/registrering hos Auth0

Figur 4.1: Flyt for innlogging og registrering med Auth0

Rollesystem

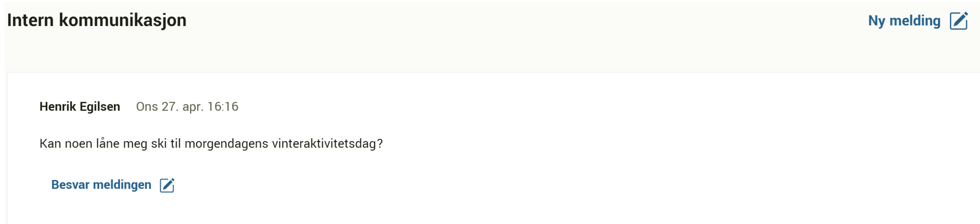
Applikasjonen inneholder organisasjoner. Organisasjoner er grupper med brukere som deler tilgang til funksjonalitet. Hver bruker kan være medlem av fra null til mange organisasjoner. Hver tilknytning har definert en rolle brukeren har i organisasjonen, enten som administrator eller som medlem. Dette styrer hva slags tilganger en bruker har og hva slags innhold som brukeren ser. Administratorer kan legge til nye medlemmer og bestemme deres roller gjennom et eget administrasjonspanel som vist i figur 4.2.



Figur 4.2: Administrering av brukere og deres roller

Intern kommunikasjonskanal

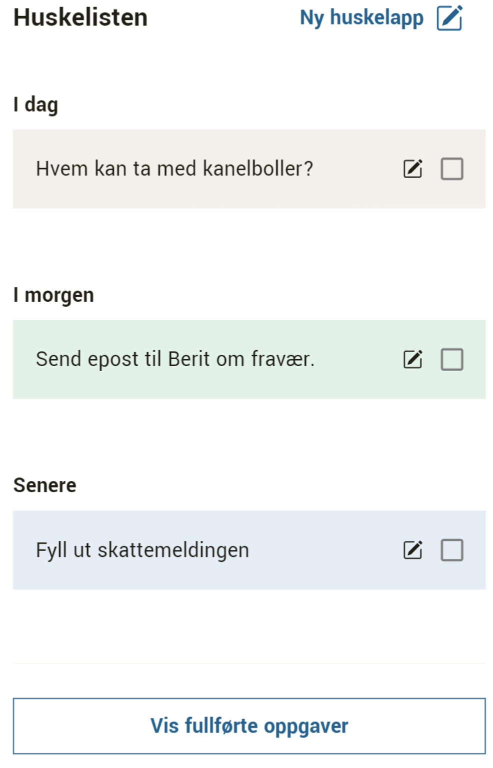
Innad i hver organisasjon er det mulig å kommunisere med andre medlemmer i organisasjonen. I kommunikasjonskanalen kan brukere legge ut innlegg eller oppgaver som andre brukere kan påta seg. Det var opprinnelig planlagt å la andre brukere svare på innlegg, men i samarbeid med oppdragsgiver ble det bestemt at vi heller skulle prioritere annen funksjonalitet. Knapp for å besvare er likevel opprettet etter oppdragsgivers ønske, for å vise fremtidig funksjonalitet.



Figur 4.3: Intern kommunikasjonskanal

Huskeliste

Brukere kan ta på seg oppgaver som legges ut i kommunikasjonskanalen. Disse oppgavene vil da automatisk legges til i brukerens egen huskeliste. Her kan brukeren også legge til egne oppgaver. Oppgavene er kategorisert etter dato for når de må utføres, i dag, i morgen og senere. Oppgavene kan markeres som utført, og om oppgaven stammer fra kommunikasjonskanalen vil også innlegget bli markert som utført. Brukere kan også se tidligere utførte oppgaver.



Figur 4.4: Personlig huskeliste

Telefonkatalog

Hver organisasjon har sin egen telefonkatalog. Organisasjonen kan kategorisere telefonnumrene i egenopprettede kategorier for å enklere kunne finne spesifikke numre. I telefonkatalogen har vi tatt i bruk søkemotoren Algolia for å lage et raskt indeksert søk. Hver organisasjon har sin egen indeks for å holde data adskilt. Indeksene oppdateres ved hjelp av serverløse funksjoner som indekserer telefonnumre som blir lagt til, oppdatert og slettet.

Figur 4.5: Telefonkatalog

Eksterne systemer

De fleste brukere benytter flere forskjellige systemer i løpet av en hverdag. Planen var dermed opprinnelig å integrere et eller flere eksterne systemer inn i vår applikasjon gjennom API-er. Dette kom vi frem til sammen med oppdragsgiver at ble for tidkrevende og komplisert i løpet av dette prosjektet. Vi har istedenfor implementert en menylinje med linker til eksterne systemer, samt visning av eksterne systemer inne i vår applikasjon ved hjelp av HTML-elementet *iframe*, som vist i figur 4.6.

Figur 4.6: Menylinje for eksterne systemer

Ikke-funksjonelt

Vi har brukt Docker til å konteinerisere applikasjonen slik at det har blitt enkelt å teste den i skyløsninger. Alle endepunkt er testet med integrasjonstester for å fange opp potensielle feil ved endringer. Vi har brukt CI i Github Actions for å kjøre de nevnte testene ved hver pull request. CD skjer ved oppdatering av main-branchen, hvor APIet bygges til et konteinerbilde som sendes til Docker Hub. Der kalles en webhook som forteller

skyleverandøren at en ny versjon av konteinerbilde kan hentes og kjøres. For nettsiden har vi brukt Netlify [59] som automatisk følger med på main-branchen og ruller ut på nytt ved endringer.

4.2 Vitenskapelige resultater

4.2.1 FaaS

Anvendbarhet

Tabell 4.1: Kjøretidsmiljøer FaaS

	Azure Functions	**GCP Cloud Functions**	**AWS Lambda**
C#	✓	✓	✓
F#	✓		
Go		✓	✓
Java	✓	✓	✓
Javascript	✓	✓	✓
Konteiner	✓		✓
Powershell	✓		✓
PHP		✓	
Python	✓	✓	✓
Ruby		✓	✓
Typescript	✓		

Overordnet presenteres FaaS som en rask måte å utplassere kode til produksjon på, mot at man har mindre kontroll over oppsettet rundt selve koden sin. Hos alle tre leverandørene er det mulig å sette opp HTTP-triggere som gjør at funksjonen kjører når en HTTP-forespørsel sendes til funksjonen. De tilbyr også handelses-baserte triggere. Det innebærer at funksjonen kjører basert på en hendelse som funksjonen lytter til. Et eksempel på dette kan være at en annen tjeneste utfører en oppgave, og dette sender en melding til funksjonen om at den skal da kjøre eller at funksjonen lytter på en Kø som den utfører oppgaver fra. Den siste mulige triggeren er lagrings-basert kjøring. Dette ligner på hendelsesbasert kjøring, men funksjonen lytter heller på spesifikke endringer i et lagringssystem, og ikke meldinger om hendelser. Skytjenesteleverandøren er også i alle tilfellene

ansvarlig for infrastrukturen som settes opp, dermed er det også en begrensning for hvor mye datakraft som kan tilgjengeliggjøres til kjøring av funksjoner.

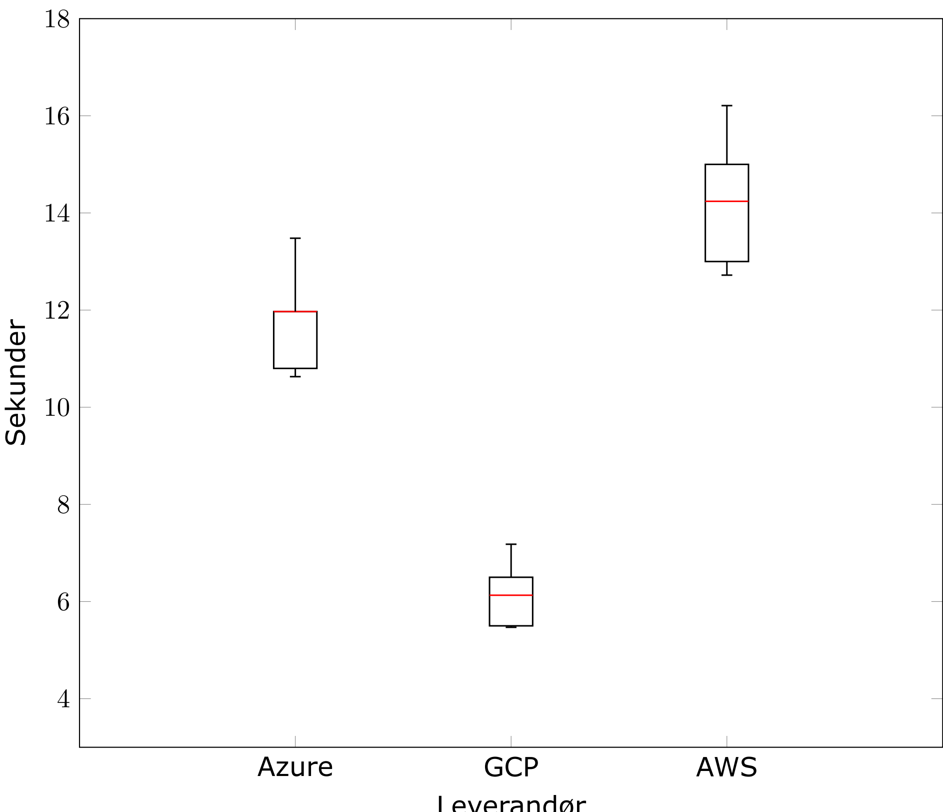
GCP Cloud Functions tilbyr funksjoner basert på HTTP-triggere og hendelsesbaserte triggere. De tilbyr to forskjellige typer hendelsesbaserte triggere der Node.js, Go, Python og Java støtter bruk av triggere fra andre tjenester i GCP sitt økosystem som Cloud Storage. PHP, C# og Ruby støtter hendelser basert på CloudEvents som er et standardisert format for sending av hendelser [60]. Når man skal rulle ut funksjonen, må dette skje gjennom opplasting av kildekoden, gjerne som en ZIP-fil. Cloud Functions bygger koden til et konteinerbilde som kjøres. Selv om Cloud Functions kjører konteinere, er det likevel ikke mulig å bygge konteineren selv og gi denne til Cloud Functions. Cloud Functions har en kjøretidsgrense på maks 9 minutter i Cloud Functions v1. Cloud Functions v2 kan kjøre i opptil 60 minutter etter HTTP-triggere og 10 minutter etter hendelsesbaserte triggere. GCP Cloud Functions har også sitt eget kommandolinjeverktøy for å utplassere og testing av funksjoner. I motsetning til de to andre leverandørene bruker dette kommandolinjeverktøyet flagg, og ikke konfigurasjonsfiler [61].

AWS Lambda sine triggere er hovedsaklig triggere som lytter på andre AWS produkter samt enkelte tredjepartsprodukter som i samarbeid med AWS kan brukes gjennom Amazon EventBridge. Lambda kan også brukes til å lage et API noe enklere enn i Cloud Function og Azure Functions. Det kan gjøres ved å legge funksjoner bak Amazon API Gateway som sender trafikk til riktig funksjon. I tillegg til at Lambda har støtte for en hel del ulike programmeringsspråk, tilbyr det også å kjøre binærfiler og konteinerbilder. Det er også mulig å laste opp en ZIP-fil, binærfil eller filer fra en S3-instans gjennom AWS sin nettside. Det er en maksimumsstørrelse på 250 MB ved opplastning av ikke-komprimerte filer og 50 MB ved opplastning av komprimerte filer pr. funksjon. AWS Lambda har også en begrensning på 15 minutter kjøretid. AWS Lambda tilbyr sitt eget kommandolinjeverktøy kalt Sam. Sam utplasserer funksjoner ved hjelp av konfigurasjonsfiler i yaml. Disse filene definerer hvilke triggere som skal kjøre hvilke funksjoner og hvordan funksjonene skal kjøres. Sam lar det også teste funksjonen din lokalt [62].

Azure Functions har i likhet med de andre leverandørene best støtte for integrasjoner for å lytte til hendelser fra andre tjenester hos samme leverandør. Azure noe er mer fleksibel i forhold til hvordan funksjonene kan utplasseres. De tilbyr flere operativsystemer, samt mulighet til å kjøre konteinere som funksjoner. Azure Functions muliggjør også å ta i bruk programmeringsspråk som ikke er direkte støttet av Azure. Dette gjøres ved at du selv implementerer en fil som starter en web-server med funksjonskoden som tar i mot forespørselen og lager en respons. Dette kan dermed gjøre det mulig å bruke GO ved å kompilere web-serveren i GO til en *.exe*-fil. Hos Azure er kjøretidsgrensen på funksjonene basert på hvilken plan man har valgt. På en Consumption plan er maksgrensen 10 minutter. På både Premium og Dedicated planer er det ikke noen grense. For HTTP-trigger funksjoner er kjøretids grensen 230 sekunder uansett.

Dette er fordi Azure sin last-blanserer har en inaktivitets-timeout på 230 sekunder. Azure Functions tilbyr også et eget kommandolinjeverktøy for Functions kalt Func. Func benytter JSON-filer som definerer hvor funksjonene ligger og hvordan de kjøres. Azure tilbyr også et egen Maven-plugin som kan rulle ut funksjoner skrevet i Java [63].

Skalering



Figur 4.7: Cold starts hos de ulike leverandørene

Figur 4.7 viser en ganske betydelig forskjell mellom de forskjellige leverandørene sine cold starts. Testene er gjennomført ved å utplassere en funksjon hos den gitte leverandøren. Deretter er det sendt en forespørsel til funksjonen og notert hvor lang tid det tok før responsen kom tilbake. Hos alle leverandørene er det valgt den minste instans-størrelsen som er stor nok til at vår funksjon kan kjøres hos den gitte leverandøren. Vi ser her at GCP har raskest cold start.

Hos GCP er muligheten for skalering presentert i formen av et maksimum og minimum antall instanser som skal kunne startes. Standardverdi for minimum er null, og det er ikke satt en standardgrense for maks. GCP

anbefaler å sette minimum antall til minst én for å redusere antall cold starts. GCP lar deg også endre hvor mye ram som instansen skal ha tilgang til [64].

I Azure definerer du en «Hosting plan». Denne definerer hvordan funksjonen skal skalere. Det finnes tre forskjellige hosting planer, Consumption plan, Premium plan og Dedicated plan. Alle planene lar deg velge hvor mange instanser du vil at funksjonen din skal kunne lage, men de forskjellige planene har ulike grenser for maksimum antall instanser. Alle planene lar deg også slå av «Scale out» som sørger for at funksjonen autoskalerer etter behov. Consumption plan er en «pay-as-you-go» modell som autoskalerer om en ønsker og en betaler kun for det som brukes. I en Premium plan får du sterkere instanser og dermed også gjerne raskere responstider. Du får også tilgang til mer avanserte tjenester som blant annet virtuelle nettverk. På en Dedicated plan bestemmer du i forveien hvor mye du skal bruke og betaler for dette [63].

På AWS får du tilgang til å endre hvor minne instansen din skal få tilgang til og CPU-størrelse bestemmes basert på hvor mye minne som er valgt. Det er også mulighet for å ta i bruk AWS Compute Optimizer som velger resusser for deg automatisk basert på hva den tror funksjonen trenger når den ser hvordan funksjonen blir benyttet. Lamda skalerer automatisk funksjonenen opp til et visst antall instanser, standardgrensen er 1 000 instanser. Man kan også sette opp en pause-tid som bestemmer hvor lenge funksjonen skal kunne kjøre før den automatisk blir stoppet. Det kan være nyttig for å unngå at funksjoner blir kjørende lenge at alle nye forespørsler fører til at det blir startet nye instanser [65].

Pris

Alle tre leverandørene tar betalt fra en kombinasjon av priser basert på både kjøretid og antall kjøringer. Kjøretid måles i GB/s (antall sekunder med 1 GB minne tilgjengeliggjort) og GHz/s (antall sekunder med 1GHz CPU tilgjengeliggjort). Pris kan også variere noe basert på hvilken lokasjon man velger å plassere funksjonene. Tallene som er presentert her er for lokasjoner som ligger enten i eller så nært Norge som mulig.

Tabell 4.2: Pris FaaS-løsninger

	Azure Functions	**GCP Cloud Functions**	**AWS Lambda**
Pr. GB/s	0,0139 øre	0,0022 øre	0,015 øre
Pr. GHz/s	0 øre	0,0087 øre	0 øre
Pr. million kjøringer	1,733 kr	3,49 kr	1,74 kr

Hos Azure gis de første 400 000 GB/s og 1 million kjøringene gratis hver måned. Hvis du velger premium-planen får du også muligheten til å konfigurere CPU-kraft og minnebruk, men her tilbys det ikke en gratiskvo-

te [66]. GCP tilbyr de første to millioner kjøringer gratis hver måned. Man kan selv bestemme mengde minnebruk, som også bestemmer hvilken CPU-kraft funksjonen får [67]. AWS tar betalt kun basert på antall GB/s og antall kjøringer. I enkelt regioner har AWS også lansert mulighet til å kjøre funksjonene på arm-arkitekturen der prisen er 0,012 øre pr. GB/s. Prisen pr. million kjøringer er uavhengig av arkitektur [68].

Leverandørinnlåsing

Som vi kan i resultatene fra Anvendbarhet er de forskjellige FaaS-tjenestene relativt låst til egen leverandør. Det innebærer at triggere i stor grad er basert på bruk av andre tjenester hos samme leverandør. Det betyr at bytte av leverandør krever oppdatering av triggere og dermed også kode. I tillegg til dette støtter tjenestene forskjellige språk, og GCP Cloud Functions støtter heller ikke konteinere. Dette i seg selv kan gjøre det vanskelig å bytte leverandør.

Det finnes riktignok løsninger for å hindre at man blir veldig innlåst. Quarkus tilbyr et FaaS-bibliotek, Funqy [69], som gjør funksjoner plattformuavhengig. Ulempen med slike biblioteker er riktignok at den fulle funksjonaliteten til tjenestene ikke blir tilgjengelig, ettersom det må være generelt for å støtte alle.

4.2.2 PaaS

Anvendbarhet

Tabell 4.3: Kjøretidsmiljøer PaaS

	Azure App Service	**AWS Elastic Beanstalk**	**Google App Engine Standard**	**Google App Engine Fleksibel**
Docker konteiner	✓	✓		✓
Docker Compose	✓	✓		
Go		✓	✓	✓
Java	✓	✓	✓	✓
.NET	✓	✓		✓
Node.js	✓	✓	✓	✓
PHP	✓	✓	✓	✓
Python	✓	✓	✓	✓
Ruby	✓	✓	✓	✓

Azure App Service støtter kjøring i både Linux og Windows for alle kjøretidsmiljø bortsett fra PHP, Python og Ruby [70]. Hvis man bruker Docker Compose må denne filen legges inn på nettsiden. Port kan også konfigureres for alle applikasjoner [71].

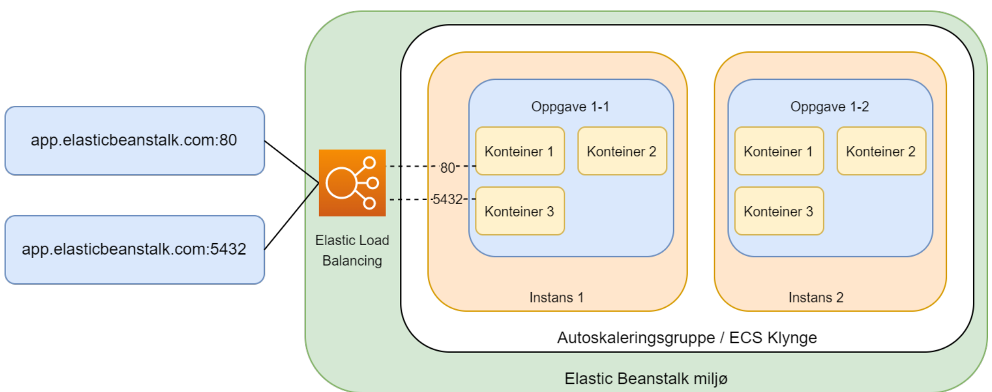
App Service tilbyr mulighet til å kjøre en egen MySQL-database på samme instans uten ekstra kostnad ved kjøring i Windows. Denne er riktignok kun anbefalt til utvikling og testing ettersom den ikke kan skaleres [72]. For monitorering tilbys det ulike alternativer [73]. En får også tilgang til mer detaljerte logger ved å besøke «Advanced tools» der en kan laste ned en zip-fil som inneholder logger for hver prosess. Hvis en bruker for eksempel Docker Compose, får en da én fil pr. konteiner.

Når det gjelder nettverking og domener, blir det tildelt et standard domene med SSL-sertifikat. Man kan koble til et eget domene med en DNS-oppføring. Det er også mulig å kjøpe domener og la Azure administrere og fornye dem automatisk gjennom App Service. SSL-sertifikat til egne domener kan også enkelt settes opp med et eget sertifikat eller ved å la Azure håndtere det [74].

Kontinuerlig utrulling av kode kan løses ved å knytte tjenesten til et Git-repository hos GitHub, Azure Repos eller Bitbucket, eventuelt rulle ut manuelt ved bruk av kommandolinjeverktøyet deres. Om en bruker konteine-

re, kan man også sette opp automatisk oppdatering ved push til et Azure konteiner-register eller bruke en webhook for å oppdatere fra et eksternt register [75]. En oppstarts-kommando kan også settes slik at det er enkelt å kjøre applikasjonen akkurat som en ønsker for alle kjøretidsmiljøer. App Service tilbyr «Deployment slots» som muliggjør kjøring av forskjellige versjoner av koden på samme instans. Det gjør det mulig å rulle ut en ny versjon i en egen slot der en kan validere at alt fungerer før all produksjonstrafikk flytter over til den nye versjonen. Det er også mulig å gradvis flytte over trafikk ved å velge hvor stor prosentandel som skal føres til hvilke slots gjennom trafikk-fordeling. Dette kan også kombineres med «Auto swap» slik at nye versjoner blir varmet opp i en egen slot før de tar over produksjon, og dermed unngå nedetid [76].

AWS Elastic Beanstalk lar brukere kjøre applikasjonen på flere forskjellige måter. Enten kildekode fra utvalgte kjøretidsmiljøer som sett i tabell 4.3, enkeltstående konteinere eller flere konteinere sammen ved hjelp av enten Docker Compose eller deres egen Elastic Container Service (ECS) [77]. Man kan selv definere hvilken port som skal brukes, som standard brukes det port 5000 [78].



Figur 4.8: AWS Elastic Beanstalk på Elastic Container Service

En står relativt fritt til å konfigurere det en ønsker. Miljøvariabler kan settes og endres på nettsiden. Man kan velge hvilken lagringsenhet som skal benyttes og sette opp CloudWatch for å monitorere logger. Logger tilbys også som nedlastning av samtlige loggoppføringer til en zip-fil eller siste 100 linjer samlet i én fil [79]. Det er også mulig å sette opp en database fra administrasjonspanelet og koble denne til applikasjonen. Dokumentasjonen anbefaler riktignok å bruke en database som er ekstern og ikke direkte tilknyttet Elastic Beanstalk-applikasjonen. En database som først opprettes fra Elastic Beanstalk kan imidlertid enkelt gjøres ekstern i ettertid [80].

Når applikasjonen opprettes tildeles det et domene basert på region og applikasjonens navn. Dette domene er ikke sikret med et SSL-sertifikat. for å brukeet eget domene må dette tilknyttes ved å bruke en annen tjeneste, Amazon Route 53, det kan ikke håndteres fra samme administrasjonspanel. Etter å ha gjort dette kan HTTPS også konfigureres [81].

Utrulling av ny kode kan gjøres på forskjellige måter. En kan laste opp filer manuelt på nettsiden, bruke kommandolinjeverktøyet til AWS og bruke AWS CodePipline for å hente kode fra GitHub eller S3-lagring. For å styre hvordan applikasjonen skal bygges og kjøres brukes det konfigurasjonsfiler. Ved bruk av konteinere benyttes det en *Dockerrun.aws.json*-fil som inneholder all konfigurasjon. Med andre kjøretidsmiljøer brukes det *Buildfile* for å fortelle hvordan koden skal bygges og *Procfile* for å si hvordan koden skal kjøres. Man kan styre hvordan flytting av trafikk skal foregå ved hjelp av forskjellige alternativer [82]:

- *Alt på en gang* - Fører til nedetid.
- *Rullerende* - Instanser byttes ut en etter en slik at forskjellige instanser kan kjøre med forskjellige versjoner.
- *Rullerende med nye instanser* - Nye instanser lages og de gamle fjernes når de nye er klare.
- *Immutabel utrulling* - Samme som rullerende med nye instanser men det gjøres i en egen autoskaleringsgruppe slik at det kan kjøres egne helsesjekker og avbryte hvis de ikke er vellykket.
- *Trafikk-splitting* - Helt nye instanser startes og trafikk føres delvis over for å sjekke at alt fungerer med helse-sjekker før all trafikk flyttes.

Google App Engine består av to forskjellige miljøer en kan velge mellom; standard og fleksibelt. I dokumentasjonen [83] forklares det når en bør velge hvilket miljø. Oppsummert passer standardmiljøet når applikasjonen har store svingninger i trafikk og en ikke trenger tilgang til spesifikk konfigurering. Det fleksible miljøet passer best for applikasjoner som mottar en jevn trafikk og man ønsker mer tilgang til de underliggende ressursene eller vil bruke konteinere. En grunn til at det fleksible miljøet passer best for jevn trafikk er at det ikke kan skalere like kjapt som standardmiljøet.

Når man skal sette opp applikasjonen, bruker man en *app.yaml*-fil for dette. Det er i praksis ingen konfigurasjonsmuligheter på nettsiden. Alt må legges inn i *app.yaml*-filen som styrer bygging, miljø, instans-størrelser, skalering og miljøvariabler. Figur 4.9 viser eksempler på konfigurasjonsfiler for både standard og fleksibelt miljø [84].

Kodeliste (4.1) Standard miljø

```
service: jvm
entrypoint: java -jar runner.jar
runtime: java11
instance_class: F4
automatic_scaling:
  target_cpu_utilization: 0.6
  min_instances: 5
  max_instances: 15
  min_pending_latency: 30ms
  max_pending_latency: automatic
  max_concurrent_requests: 50
env_variables:
  DATABASE_HOST: "db.cloud.com"
```

Kodeliste (4.2) Fleksibelt miljø

```
service: container
runtime: custom
env: flex
resources:
  cpu: 2
  memory_gb: 8
  disk_size_gb: 10
automatic_scaling:
  min_num_instances: 1
  max_num_instances: 15
  cool_down_period_sec: 180
  cpu_utilization:
    target_utilization: 0.6
env_variables:
  DATABASE_HOST: "db.cloud.com"
```

Figur 4.9: Eksempler på konfigurasjon av Google App Engine med app.yaml

App Engine tilbyr også muligheten til å sette opp planlagte jobber med Cron. Dette kan enkelt settes opp ved å lage en «cron.yaml»-fil som inneholder en Cron-plan som forteller når jobben skal kjøres og en url i applikasjonen som skal kalles. For å unngå at disse endepunktene kan bli kalt av hvem som helst legger Cron-tjenesten til «X-Appengine-Cron: true» som en HTTP-header. Hvis en klient sender denne headeren, vil den bli fjernet fra forespørselen av App Engine [85].

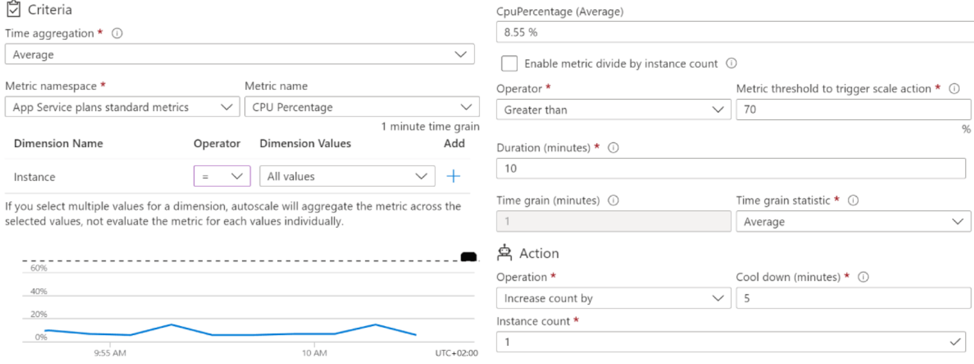
Utrulling av kode for Java-applikasjoner dokumenteres med bruk av enten en Maven-pakke eller GCP sitt kommandolinjeverktøy [86]. I det fleksible miljøet for konteinere kan man også bruke «–image-url» og peke til et konteinerbilde med kommandolinjeverktøyet. Dette krever at bildet ligger i Google sitt Container Registry. Eksterne registre som for eksempel Docker Hub er ikke støttet [87]. En kan bruke løsninger gjennom for eksempel GitHub for kontinuerlig utrulling, men dette er ikke noe som er dokumentert. Når ny kode blir lastet opp, flyttes trafikken gradvis over til den nye versjonen, etter at helse-sjekker har sjekket at den nye versjonen er klar. Dette kan skrus av om ønskelig slik at trafikk sendes til ny versjon umiddelbart, men dette kan føre til noe nedetid [88]. Det fleksible miljøet støtter ikke gradvis flytting av trafikk.

Nye applikasjoner får automatisk tildelt et domene som også er sikret med SSL-sertifikat av Google. Man kan enkelt tilknytte applikasjonen til sitt eget domene gjennom administrasjonspanelet på nettsiden [89]. Disse får også automatisk et SSL-sertifikat, men man kan også velge å bruke et eget sertifikat om man ønsker det [90].

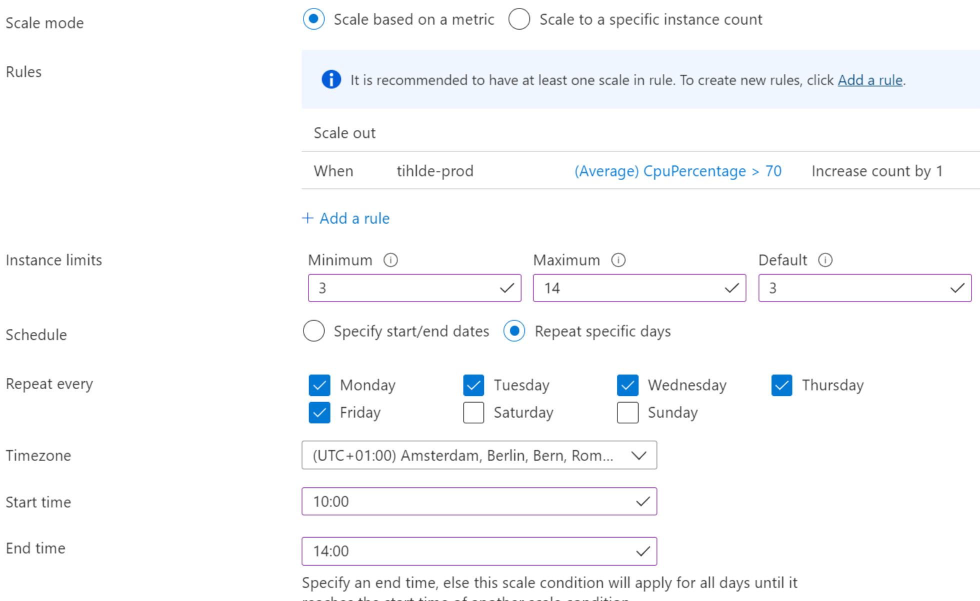
Skalering

Azure App Service støtter at brukere selv kan velge instans-størrelse og det er mulig å skalere horisontalt til opptil 30 instanser. Horisontal auto-

skalering kan settes opp på planer som er ment for produksjon. Som en kan se i figur 4.10 og figur 4.11 kan autoskaleringen styres ved hjelp av et stort antall metrikker og tidspunkter som kan kombineres etter eget ønske [91].

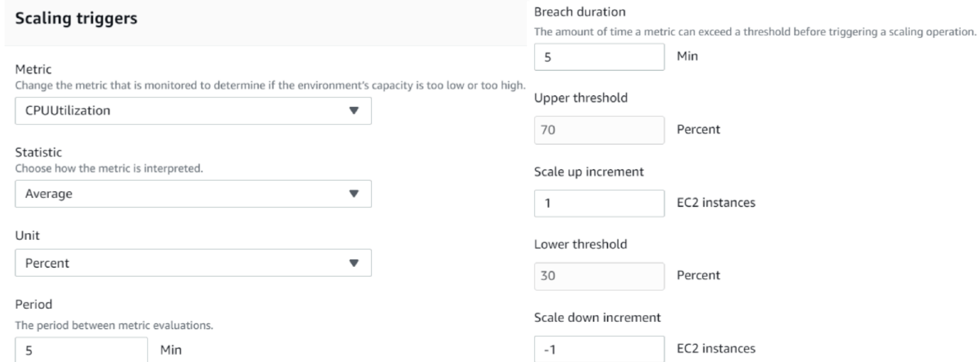


Figur 4.10: Azure App Service - triggere for autoskalering

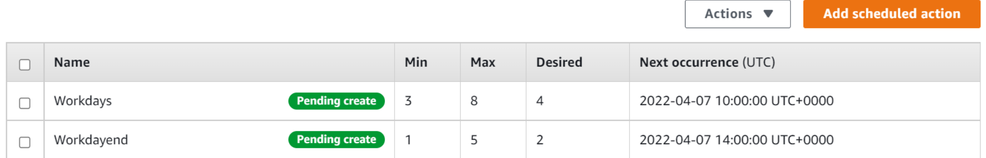


Figur 4.11: Azure App Service - skalering basert på tid

Kapasiteten i AWS Elastic Beanstalk kan justeres ved å skru på automatisk skalering horisontalt [92]. Da kan en velge hvilke instans-typer som skal brukes, minimum og maksimum (opptil 10 000) antall instanser og velge hvilke triggere som skal brukes, se figur 4.12. Det er også mulig å skalere basert på tidspunkter, se figur 4.13.



Figur 4.12: AWS Elastic Beanstalk - triggere for autoskalering



Figur 4.13: AWS Elastic Beanstalk - skalering basert på tid

For Google App Engine viser figur 4.9 hvordan autoskalering kan settes opp der. Konfigurasjonen er noe forskjellig mellom miljøene, men de kan oppnå stort sett det samme. Man kan kun sette opp minimum og maksimum antall instanser, samt hvilken verdi som bestemmer om skalering skal gå opp eller ned. Skalering basert på tidspunkt støttes ikke. I standard-miljøet fører også valg av instanstype til begrensninger på hvilken konfigurasjon som er støttet [93].

Ytelse - Kildekode vs Konteiner

Tabell 4.4: Gjennomsnittlig responstid for PaaS-løsninger med differanse mellom opplasting av kildekode i forhold til konteiner

	Azure App Service	**AWS Elastic Beanstalk**	**Google App Engine**
Kildekode	134 ms	113 ms	134 ms
Konteiner	144 ms	115 ms	127 ms
Differanse	10 ms	2 ms	-7 ms

Last-testene som har blitt kjørt tyder på at det ikke er noe særlig forskjell mellom å kjøre appen ved å laste opp kildekode i forhold til en kontainer.

En forskjell på 2 til 10 ms regner vi som innenfor usikkerheten i eksperimentet. Alle resultatene kan sees i [94]–[99].

Pris

Det er definert tre forskjellige instans-størrelser som vi har regnet ut et estimat på månedlig pris for. Liten instans har 1 CPU-kjerne og 4 GB minne. Medium instans har 2 CPU-kjerner og 8 GB minne. Stor instans har 4 CPU-kjerner og 16 GB minne. Alle de tre leverandørene tilbyr en stor mengde forskjellige valg som påvirker prisene. I Azure velger man en plan som definerer instansstørrelser, i AWS VM-instanser og i GCP nøyaktig mengde CPU og minne. I Azure ble det valgt premium-planene ettersom disse var de eneste som oppfylte kravene til ratio mellom CPU og minne. Standard-planen er noe billigere. I AWS ble det valgt VM-instanser som kjører på Arm-prosessorer. Disse er billigere enn Intel-prosessorer men krever at applikasjonen støtter denne. Pris er regnet ut for kontinuerlig kjøring i en hel måned.

Tabell 4.5: Estimert pris pr. måned for PaaS-løsninger [100]–[102]

	Azure App Service	**AWS Elastic Beanstalk**	**Google App Engine**
Liten	760 kr (P1V2)	396 kr (a1.large)	607 kr
Medium	1 081 kr (P1V3)	792 kr (a1.xlarge)	1 131 kr
Stor	2 162 kr (P2V3)	1 583 kr (a1.2xlarge)	2 262 kr

Leverandørinnlåsing

Som vi kan se i resultatene fra Anvendbarhet blir en i utgangspunktet ikke særlig låst til leverandøren uansett hvilken av PaaS-tjenestene som velges. Konfigurasjon må naturligvis endres slik at Azure App Service får satt oppstartskommando, AWS Elastic Beanstalk har tilgang til *Buildfile*, *Procfile* eller *Dockerrun.aws.json* og GCP App Engine har en *app-yaml*-fil. Bortsett fra dette kan koden som skal kjøres være uendret. Det eneste unntaket er hvis noe av ekstra-funksjonaliteten tas i bruk, for eksempel GCP App Engine sin løsning for CRON-jobber. I slike tilfeller må det selvfølgelig gjøres endringer også i koden.

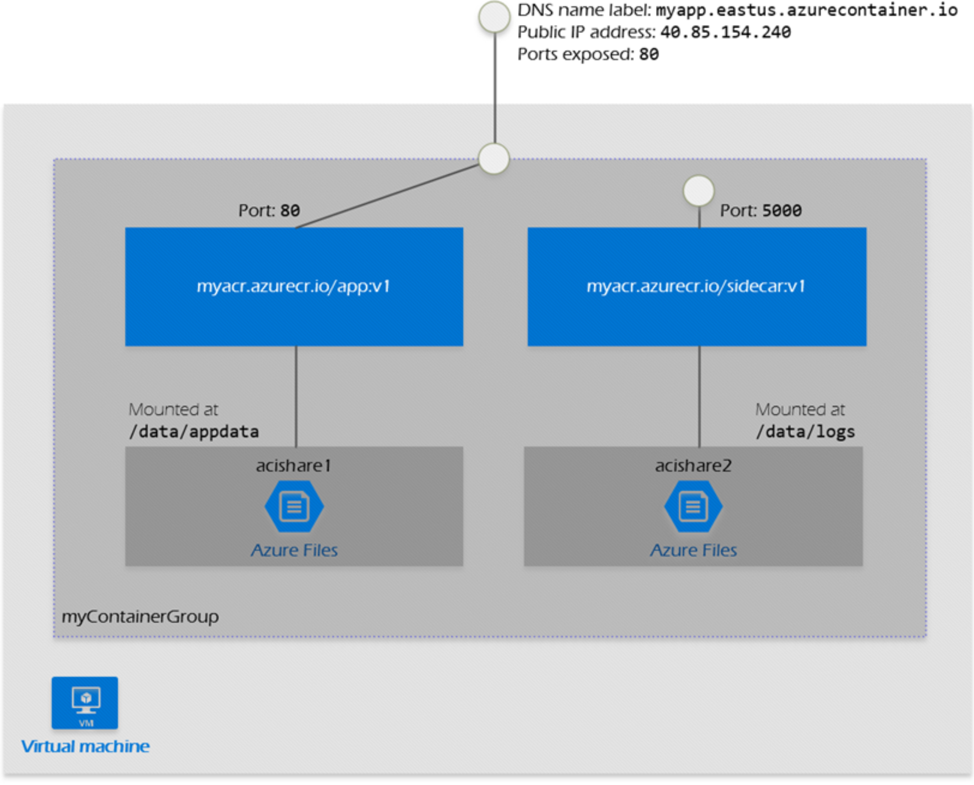
4.2.3 CaaS

Anvendbarhet

Azure Container Instances tilbyr ikke særlig mye utover å kjøre en gitt konteiner. Når du lager instansen velger du hvilken konteiner som skal

kjøres, enten fra et offentlig eller privat konteiner-register, operativsystem og størrelse. Det er mulig å legge til opptil fire NVIDIA Tesla GPUer [103]. For nettverking kan man velge mellom offentlig, privat og ingen tilgang. Om den er offentlig kan man navngi deler av DNS-adressen selv, samt velge hvilke porter som skal eksponeres. Det er ikke mulig å legge til et eget domene selv. Private kan legges i et virtuelt nettverk og subnett. Om det kun behøves tilgang til kommandolinjen, er ingen nettverk et godt alternativ. Det er også mulig å styre retningslinjer for omstart, samt å legge inn miljøvariabler og en oppstartskommando [104].

Container Groups [105] gjør det mulig å kjøre flere konteinere sammen. Figur 4.14 viser hvordan det gir mulighet til å legge flere konteinere sammen bak ett endepunkt. Dette kan utnyttes til å kjøre for eksempel NGINX slik at en kan bruke et eget SSL-sertifikat, noe som ikke tilbys på andre måter. Det er også mulig å knytte instanser til vedvarende lagring med Azure File Share [106]. Når en instans er opprettet, er det ikke mulig å endre den, men må heller ta ned instansen og lage en ny. Dette gjelder riktignok ikke kontainere som ligger i en Container Group. Der er det mulig å oppdatere konteiner-versjon, DNS-navn og miljøvariabler [107]. Last-balanserer, autoskalering og kontinuerlig utrulling støttes ikke.



Figur 4.14: Azure Container Group med flere Container Instances, hentet fra [105]

Azure Container Apps er foreløpig kun lansert som en forhåndsvisning. Løsningen er bygget på toppen av Azure Kubernetes Service med begrenset tilgang til Kubernetes API-er [108]. Alle Container Apps må ligge i et miljø. Et miljø kan ha flere apper. Disse appene blir plassert i samme virtuelle nettverk og skriver logger til samme Log Analytics arbeidsområde [109]. Hver Container App kan inneholde en eller flere konteinere som kjører og skalerer sammen, samt ha direkte kommunikasjon innad. En Container App tilsvarer en Pod i Kubernetes [110]. Det er også relativt enkelt ta i bruk en mikrotjeneste-arkitektur. Det kan løses ved å ha flere Container Apps i samme miljø og bruke Dapr for kommunikasjon. Man får automatisk tildelt et domene av Azure som er sikret med SSL-sertifikat. Det er også mulig å tilknytte egne domener. Pr. i dag får man ikke automatisk SSL-sertifikat for disse, men det er mulig å legge til egne sertifikater [111].

Container Apps lar brukerne skalere basert på blant annet lengden i en Azure Storage Queue. Dette kan benyttes til å kjøre oppgaver i bakgrunnen ved at applikasjonen kun kjører når det er nye oppgaver og deretter skalerer tilbake til null [112].

Utrulling av ny kode er også relativt enkelt og kan løses med blant annet kommandolinjeverktøy, nettsiden eller CI-verktøy som Github Actions [113]. Når man skal oppdatere til en ny versjon skjer dette ved å lage en ny revisjon. Endringer av konteinere, Dapr-innstillinger eller skalering skaper også en ny revisjon [114]. Når den nye revisjonen har startet kan man selv velge om man skal flytte all trafikk over til den, eller for eksempel bruke trafikk-splitting for å sende bare et eget bestemt antall prosent av trafikken dit. Det er enkelt å justere hvordan trafikk skal fordeles mellom ulike revisjoner [115].

AWS Fargate består av klynger som inneholder en eller flere tjenester (service) eller oppgaver (task) [116]. Tjenestene eller oppgavene kjører en revisjon som inneholder en definisjon av hva som skal kjøres. En definisjon kan inneholde én eller flere konteinere som skal kjøres sammen. I tillegg kan det bestemmes hvilke porter som skal eksponeres og hvor mye CPU-kraft og minne som skal tilgjengeliggjøres. CPU og minne kan fordeles på konteinerne om det er flere som kjører sammen. Miljøvariabler for hver konteiner kan også legges til [117]. Tjenester er beregnet på applikasjoner som skal kjøre lenge og kan stoppes og restartes, for eksempel web-applikasjoner [118]. Oppgaver er beregnet for kjøring av konteinere som kun skal fullføre og avsluttes.

Når man lager en tjeneste eller oppgave bestemmer man enkelt hvordan det skal eksponeres mot nettet med en last-balanserer. Denne kan senere brukes for å legge til et eget domene med Amazon Route 53. Det er også mulig å legge Fargate i et VPC og subnett i VPCen for å finjustere i enda større grad hvordan nettverkingen skal foregå [119]. Når man skal rulle ut ny kode gjøres det ved å lage en ny revisjon av konteineren og deretter oppdatere tjenesten til å benytte denne nye revisjonen. Det kan velges mellom å bytte ut eksisterende revisjon umiddelbart eller å gradvis flytte trafikk over til ny revisjon i bestemte prosentvise intervaller [120].

AWS App Runner er et annet AWS-alternativ for å kjøre konteinere og er enda mer simpelt enn AWS Fargate [121]. Det eneste som er nødvendig for å kjøre applikasjonen er å velge et GitHub-repository som applikasjonen skal bygges fra eller oppgi et Elastic Container Registry (ECR) der et konteinerbilde ligger [122]. Det er ikke mulig å bruke konteinerbilder som ligger i andre registre som Docker Hub. I tillegg kan man spesifisere CPU-størrelse og mengde minne, samt miljøvariabler, port som skal eksponeres og en eventuell oppstartskommando [123]. Det er også mulig å bestemme hvorvidt applikasjonen skal få sende forespørsler til hvor som helst eller kun til andre endepunkter i et VPC. Du får automatisk en last-balanserer som ikke kan konfigureres, et tilfeldig domene som er sikret med SSL-sertifikat og autoskalering. Du kan også tilknytte ditt eget domene.

Utrulling av ny kode kan gjøres på hovedsaklig to måter [124]. Man kan fortelle selv at den skal oppdateres, eller man kan sette opp automatisk oppdatering. Hvis man bruker et offentlig tilgjengelig ECR-bilde er kun manuell oppdatering tilgjengelig. Hvis man velger å la App Runner bygge koden for deg støtter den for øyeblikket kun Java, Node.js og Python. Når en ny utrulling gjennomføres er det ikke mulig å bestemme hvordan trafikk skal flyttes over til ny kode. App Runner flytter noe av trafikken over og kontrollerer med en helse-sjekk at det går fint før den flytter resten over. Det er ikke mulig å dele trafikk mellom ulike versjoner manuelt.

GCP Cloud Run er GCP sitt CaaS-alternativ og skal være en serverløs variant av Kubernetes [125]. Her kan man velge hvilken konteiner som skal kjøres fra et konteiner-register hos GCP eller kode fra enten GitHub, Bitbucket eller Google Source Reposity som skal bygges til en konteiner og kjøres [126]. Hver instans kan kun kjøre én konteiner [127]. Deretter velger man hvor mye CPU og minne som ønskes pr. instans og maks antall forespørsler som hver instans skal kunne håndtere samtidig. Man kan også velge hvorvidt applikasjonen skal ha tilgang til CPU hele tiden eller kun når den mottar forespørsler [128]. Dette påvirker prisen som sett i 4.2.3. I tillegg kan man styre minimum og maksimum antall instanser, ved å sette minimum på minst én kan for eksempel cold starts reduseres.

Miljøvariabler og hemmeligheter, hvilken port trafikk skal rutes til, oppstartskommando og argumenter til konteiner, medlemskap til et VPC og tilgang til å koble til en instans av Cloud SQL kan også konfigureres. Når det gjelder nettverking, kan det velges hvilken trafikk som skal ha tilgang til instansene, enten all trafikk, trafikk fra Cloud Load Balancing og intern trafikk eller kun intern trafikk [129]. Det er enkelt å knytte applikasjonen til et eget domene fra Cloud Run sitt administrasjonspanel ved hjelp av et par DNS-oppføringer. Et SSL-sertifikat blir automatisk tildelt, det ikke bruke et eget sertifikat. Det er også mulig å bruke en last-balanserer for å tilknytte et domene, samt Firebase Hosting [130]. Når en skal rulle ut ny kode kan nye revisjoner lages automatisk, eller en kan lage nye revisjoner selv. I en ny revisjon kan også parametre som styrer størrelse på instansene og minimum og maksimum antall instanser endres. Man kan styre hvordan trafikken skal flyttes over til den nye revisjonen. Enten kan alt flyttes umiddelbart, eller den kan flyttes gradvis manuelt ved hjelp av trafikk-splitting. Ettersom det lages nye revisjoner ved nye utrullinger er

det også enkelt å rulle tilbake til eldre revisjoner om noe går galt ved å flytte trafikken til ønsket revisjon [131].

Skalering

Azure Container Instances gir ingen muligheter til å skalere en allerede opprettet instans. Det er mulig å skalere horisontalt manuelt ved opprette flere instanser horisontalt [132]. Azure Container Apps gir brukere mulighet til å bruke KEDA [133] for å styre skalering. Dette gir et stort handlingsrom til å bestemme akkurat den skaleringen som er best for enhver applikasjon. Det er også mulig å skalere til null instanser [134]. AWS Fargate lar brukere velge en eller flere metrikker til å bestemme om antall instanser skal skaleres opp eller ned mellom et øvre og nedre antall instanser. Det er mulig å skalere ned til null instanser [135]. AWS App Runner tilbyr kun autoskalering med minimum og maksimum antall instanser med bestemmelse av antall samtidige forespørsler for å definere når det må opprettes ny instanser. Det er ikke mulig å skalere til null instanser [136]. GCP Cloud Run tilbyr ikke manuell justering av skalering utover maks antall samtidige forespørsler til en instans. Man kan derimot skalere ned til null om ønskelig og også bestemme en øvre grense for antall instanser [137].

Pris

Tabell 4.6: Estimert pris pr. måned for CaaS-løsninger for instans med 1 CPU-kjerne og 4 GB minne [138]–[142]

Tjeneste	Pris
Azure Container Instances	370 kr
Azure Container Apps - Aktiv	820 kr
Azure Container Apps - Inaktiv	342 kr
AWS Fargate	375 kr
AWS App Runner - Aktiv	591 kr
AWS App Runner - Inaktiv	180 kr
GCP Cloud Run - Alltid aktiv	602 kr
GCP Cloud Run - Aktiv	787 kr
GCP Cloud Run - Inaktiv	289 kr

Aktiv innebærer at instansen bruker CPUen kontinuerlig hele måneden mens *Inaktiv* betyr at instansen ikke mottar forespørsler i løpet av hele måneden og ikke har tilgang til CPU-kraft. *Alltid aktiv* hos GCP Cloud Run

innebærer at det på forhånd bestemmes at CPU alltid skal være tilgjengelig og instansen dermed vil være aktiv hele tiden. Tabell 4.6 viser at det en del variasjoner i pris for CaaS-løsninger. Hovedforskjellene oppstår for løsningene som tilbyr forskjellig prising for instanser som varierer mellom å være aktiv og inaktiv. Azure Container Instances og AWS Fargate tilbyr ikke muligheten til å skalere til null og har dermed en fast pris pr. måned så lenge en ikke skalerer opp. Med pris på 370 kr og 375 kr pr. måned kommer de helt likt ut.

Azure Container Apps, AWS App Runner og GCP Cloud Run tilbyr muligheten til å automatisk skalere opp og ned basert på trafikk. For å unngå at cold start tar for lang tid skaleres det ned til inaktiv tilstand der minne opprettholdes men CPU fjernes. Det vi kan lese fra prisene for disse løsningene er at dersom applikasjonen ikke må kjøre konstant men heller kan kjøre når det kommer forespørsler kan disse løsningene være enda billigere enn de to førstnevnte. Om de derimot kjører hele tiden blir det raskt dyrere. AWS er billigst med App Runner foran GCP Cloud Run og dyrest er Azure Container Apps. GCP Cloud Run tilbyr også en plan der instansen alltid er aktiv som kan sammenlignes med Azure Container Instances og AWS Fargate, men prisen her er signifikant høyere. Det bør påpekes at automatisk oppdatering av kode hos AWS App Runner koster $1 pr. måned, samt at bygging av kode også koster noe.

4.2.4 Kubernetes

Anvendbarhet

Det er mulig å sette opp en Kubernetes-klynge over flere maskiner. En node i en klynge kan ligge i Norge, mens en annen node ligger i for eksempel Storbritannia. Disse kan fremdeles kommunisere og forbli i samme økosystem ved hjelp av master-noden og Kubernetes-APIet [28].

Siden en Pod er en eller flere konteinere kan man ta i bruk eksterne konteinerbilder. En kan for eksempel benytte et PostgreSQL konteinerbilde og la Kubernetes håndtere applikasjonens database. Bruken av konteinerbilder i en Pod kan ligne på Docker Compose ettersom flere konteinerbilder kan kjøres sammen med en egenbestemt konfigurasjon [29].

Det er mulig å kjøre klynger lokalt under utvikling. Dette muliggjør enkel testing og eventuell feilretting før klyngen rulles ut i skyen. Ettersom det lokale miljøet er helt likt det som kjøres i skyen, brukes akkurat de samme kommandoene for å håndtere Kubernetes. Dette gjør at skyplattformen i teorien ikke har noe å si for utplasseringen av applikasjonen. Å kjøre Kubernetes i en VM er for eksempel også fullt mulig [143].

Om det oppstår behov for feilretting eller inspeksjon av logger, kommer Kubernetes med et eget logg-system som forteller status til de forskjellige tjenestene. Man kan også få direkte tilgang til loggene som kommer fra Podene. Feilmeldinger kan noen ganger være litt kryptiske fordi det brukes begreper eller informeres om hendelser som det ikke er direkte enkelt å

forstå [143].

Det er mulig å ta i bruk miljøvariabler og hemmeligheter i klyngene. Dette gjøres ved å lage en egen secret-tjeneste. Denne tjenesten håndterer hemmeligheter og miljøvariabler. Disse kan deretter brukes i yaml-filene for å definere konfigurasjon. For å lage secret-tjenesten, kan en enten lage en yaml-fil med hemmelighetene eller konvertere en .env-fil til tjenesten ved hjelp av kommandolinjeverktøyet [144].

Utrulling av kode til en klynge er relativt rett frem. Det er mulig å merke konteinerbildet som en Pod bruker med nyest-markelappen (lastest). Dette fører til at når klyngen oppdateres vil det nyeste bildet hentes ut og tas i bruk. Det er flere forskjellige måter man kan oppdatere og rulle ut den nye versjonen av klyngen, for eksempel med kommandolinjeverktøyet og kommandoen *rollout restart*.

Innad i en klynge har man en intern DNS-tjeneste som ruter trafikk til riktig destinasjon. En Pod sitt DNS-navn er for eksempel bygd opp fra navnerommet til Poden, pluss klygen sitt domene. Det er mulig å definere statiske IP-adresser som kan eksponeres og benyttes til å definere et eget domene som gir tilgang til en av applikasjonene i klyngen. Dette kan gjøres med en last-blanserer som når den blir utplassert får tildelt en ekstern IP-adresse. Denne IPen kan man deretter definere at alltid skal benyttes gjennom yaml-filer [143].

Hvis en ønsker å beskytte applikasjonen med et SSL-sertifikat, tilbyr Kubernetes et eget API for å kontrollere dette for tjenestene [145].

Skalering

Kubernetes håndterer selv skalering gjennom bruken av Deployments. Først defineres antall Pod-instanser en Deployment skal opprettholde til enhver tid med feltet *spec.replicas*.

Deretter kan man for eksempel definere en *Horisontal Pod Autoskalering*-tjeneste som definerer skalering når for eksempel CPU-bruken treffer et visst nivå ved å spinne opp flere Poder. Det er også mulig å definere *Vertikal Pod Autoskalering* som kan justere ressursbruk til en bestemt Pod ved en hendelse. Den tredje og siste typen skalering er *Klynge Autoskalerer*. Denne endrer antall noder i en klynge for å håndtere mer eller mindre belastning [31], [146].

Man bør unngå å bruke horisontal autoskalering sammen med vertikal skalering på CPU og minne. Det er en del begrensninger for vertikal autoskalering ved at det fortsatt er eksperimentelt og ikke kan reagere på alle hendelser og metrikker. Vertikal autoskalering restarter også Poden, noe som fører til at konteinerne i den restartes. Autoskalering av klynger fungerer ikke på alle typer Kubernetes plattformer. Lokale PersistentVolumes kan heller ikke benyttes sammen med klynge autoskalering. Dette betyr at det ikke er mulig å bruke klynge-autoskalering på tjenester som lagrer til disk [31], [146].

Leverandørinnlåsing

Siden Kubernetes er et åpen kildekode-verktøy vil oppsettet av en klynge være så og si likt på alle plattformer. Hovedkonseptene vil være det samme, som Pods, Deployments og Services. Det som skiller plattformene fra hverandre er hvis du skal interagere med flere datamaskiner eller å ta i bruk leverandør-spesifikke produkter. Om nodene for eksempel skal spres utover flere maskiner, er dette likt hos alle, men hvordan klyngen gis tilgang til de forskjellige maskinene vil variere mellom leverandører. Både Azure, AWS og GCP er partnere med Kubernetes [147]. Dette har ført til at det finnes en hel del løsninger for å knytte leverandør-spesifikke skytjenester mot Kubernetes.

Pris

Prisen på Kubernetes-skytjenester varierer basert på hvor lenge klyngen kjører og størrelsen på instansen. AWS og GCP tar i tillegg en håndteringsavgift på 0,94 kr pr. time. Vi har estimert pris på klynger som kjører kontinuerlig i en hel måned. Vi har også sett på tre forskjellige størrelser på maskinen som kjører klyngen. Liten har 2 kjerner og 8 GB minne. Medium har 4 kjerner og 16 GB minne. Stor har 8 kjerner og 32 GB minne. Disse prisene er uten rabatter. Hos alle leverandørene er det mulig å få rabatter hvis en låser seg til å bruk tjenesten i *x* antall år. Bruk av andre skytjenester hos leverandøren fra klyngen koster naturligvis også ekstra [148]–[150].

Tabell 4.7: Estimert pris pr. måned for Kubernetes-løsninger uten rabatt

	Azure Kubernetes Service	**Amazon Elastic Kubernetes Service**	**Google Kubernetes Engine**
Liten	690,38 kr	961,23 kr	468,55 kr
Medium	1 383,16 kr	1660,93 kr	848,54 kr
Stor	2 775,25 kr	2 594,17 kr	1,697,08 kr
Håndteringsavgift pr. time	0 kr kr	0.94 kr	0.94 kr

4.3 Administrative resultater

I dette delkapittelet blir resultater rundt prosess under utvikling presentert.

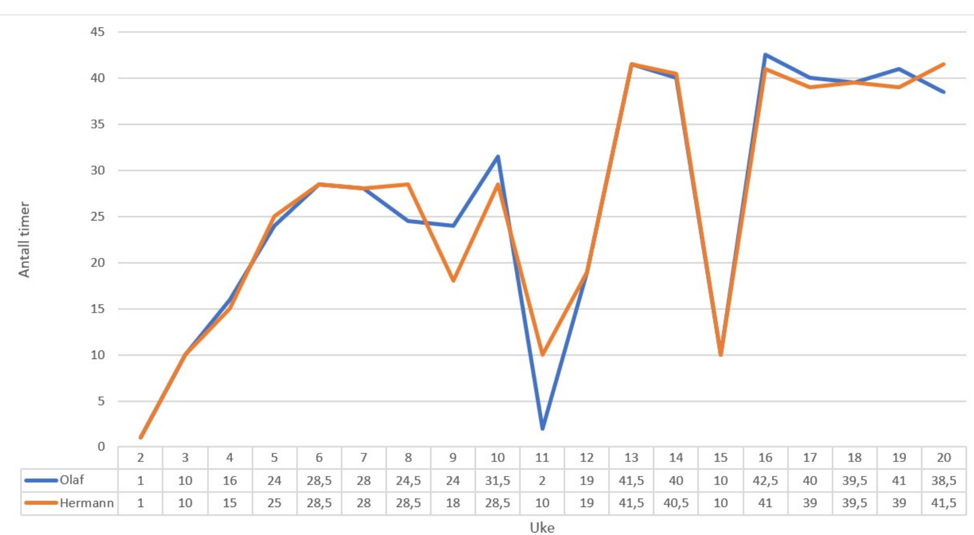
4.3.1 Milepæler

Dette er milepælene i prosjektet som ble bestemt i forprosjektplanen:

- 9. februar - Starte på hovedrapport
- 11. februar - Ferdig visjonsdokument
- 11. februar - Prosjektstruktur og foreløpig databasemodell
- 28. mars - Lage poster og holde muntlig presentasjon
- 20. april - Ferdig med testing av skyløsninger
- 1. mai - Ferdig med å kode
- 13. mai - Ferdig kravdokumentasjon og systemdokumentasjon
- 20. mai - Innlevering bachelor
- 27. mai - Muntlig presentasjon bachelor

Alle oppførte milepæler ble fullført innen tidsfrist som spesifisert over.

4.3.2 Timeforbruk



	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Olaf	1	10	16	24	28,5	28	24,5	24	31,5	2	19	41,5	40	10	42,5	40	39,5	41	38,5
Hermann	1	10	15	25	28,5	28	28,5	18	28,5	10	19	41,5	40,5	10	41	39	39,5	39	41,5

Figur 4.15: Timeforbruk fordelt på person og uke gjennom prosjektet

Som en kan se i figur 4.15 har antall timer arbeidet hver uke variert. Vi hadde i uke 12 en eksamen i et annet emne. Forberedelser og noe sykdom førte dermed til få arbeidstimer i uke 11 og deler av uke 12. I tillegg lå timeantallet på mellom 25 og 30 timer pr. person fra uke 5 til 10 for å ha tid til det andre emnet. Etter nevnte eksamen økte timeantallet til rundt 40 timer pr. uke i snitt, med unntak av påsken i uke 15. Ettersom det ble arbeidet relativt mye i starten av prosjektet har vi ikke hatt behov å jobbe ekstra mye mot slutten, men heller jobbe tilsvarende ordinære arbeidsuker.

En mer detaljert oversikt over timeforbruk pr. person kan sees i Vedlegg D Prosjekthåndbok.

4.3.3 Versjonskontroll

Vi har tatt i bruk *feature branching*. Denne måten å organisere branches går ut på at hver ny endring er på sin egen branch som definerer hva slags endring som blir gjort. Dette ble gjort slik at issues på Kanban-brettet kunne knyttes til en branch. Det gjorde det lettere å holde styr på hva man jobbet med utenom å måtte gå gjennom koden eller sjekke Github. Når man ble ferdig med en feature opprettet man en pull request. Her kunne teammedlemmer komme med endringforslag slik at koden ble holdt lesbar og ryddig. Dette gjorde at vi kunne opprettholde en god standard på koden gjennom utviklingen. For at en PR skulle bli sendt gjennom, måtte en annen på teamet godkjenne at koden som skulle legges til så bra ut.

Kapittel 5

Diskusjon

I dette kapittelet skal vi analysere resultatene fra kapittel 4. Resultatene fra kapittel 4 vil bli brukt til å diskutere bruksområdene til de forskjellige skytjenestene samt hvilke sterke og svake sider de forskjellige skytjenestene har. Diskusjonen her vil presentert pr. skyløsning slik som i kapittel 4. Kapittelet innholder også en diskusjon rundt de ingeniørfaglige og administrative resultatene fra utviklingsprosessen.

5.1 Ingeniørfaglige resultater

5.1.1 Sluttprodukt

Applikasjonen var til stor hjelp under analysen. Den skapte et godt grunnlag for rettferdig sammenligning der alle skytjenestene ble vurderte på samme grunnlag. I tillegg ble de vurderte på en realistisk måte ettersom applikasjonen var reell. Applikasjonen har dermed gitt et godt grunnlag for å finne styrker og svakheter ved de forskjellige skytjenestene. Som produkt har applikasjonen mottatt positive tilbakemeldinger fra produkteier. Den har gitt dem en mer håndfast fremstilling av deres visjon og vært nyttig i samtaler med potensielle investorer.

Selv om løsningen oppfyller kravene fra produkteier, finnes det fremdeles enkelte potensielle svakheter i løsningen. Sammen med produkteier ble det bestemt at det skulle fokuseres på visning på store skjermer, ettersom hovedmålgruppen hovedsaklig bruker dette. Visningen av applikasjonen på mobilenheter fungerer dermed ikke optimalt. Applikasjonen har også tatt i bruk det PostgreSQL-spesifikke liste-feltet. Det har vært veldig praktisk og forenklet utviklingen, men betyr også at det ikke er mulig å enkelt migrere til en annen databasetype på et senere tidspunkt. Brukerautentisering er avlastet til Auth0 og søk til Algolia. Dette har gitt fordeler som vi er fornøyde med og mener har gitt et bedre produkt enn vi hadde hatt tid til å lage selv i samme periode. Det betyr likevel at applikasjonen i større

grad blir avhengig av at tredjepartstjenester fungerer som de skal.

De nevnte svakhetene har etter vår mening ikke påvirket svaret på problemstillingen fordi de ikke har påvirket hvorvidt applikasjonen er reell eller ikke. Det er i all hovedsak punkter som kan ha noe å si for videre utvikling. Valgene ble som nevnt besluttet i samarbeid med produkteier. De har som nevnt heller ikke påvirket produkteiers muligheter til å bruke applikasjonen i for eksempel samtaler med potensielle investorer.

5.1.2 Teknologier

NextJS viste seg å være et et godt valg for dette prosjektet. Det var enkelt å ta i bruk eksterne komponent- og styling-biblioteker, slik at utviklingen gikk raskt og utseende ble konsistent gjennom hele applikasjonen. Struktureringen av sider ga stor fleksibilitet til å lage en god URL-struktur. Dette kunne i tillegg enkelt kombineres med NextAuth.js [151] slik at sidene ble beskyttet bak autentisering. Det oppstod enkelt utfordringer underveis med å finne beste løsning for data-innlasting på tvers av server og klient på grunn av det store mulighetsrommet, men dette ble etterhvert løst av gode biblioteker. Quarkus fungerte fint i backend fordi det hadde integrasjoner for blant annet databasetilkobling og brukerautentisering. Konteinerisering av applikasjonen ble også løst av Quarkus automatisk. Dette gjorde testing av skytjenestene betydelig enklere. PostgreSQL ga også en god brukeropplevelse ved at det kunne kjøres både lokalt og i skyen. Da vi tok i bruk liste-kolonnetypen som for eksempel MySQL ikke tilbyr, skapte det noen utfordringer ettersom det ikke fantes veldig mye dokumentasjon på det sammen med Quarkus. Da dette ble løst, skapte det en merkbar positiv fordel.

Det ble opprinnelig planlagt å bruke Pulumi til å raskt og enkelt sette opp skyløsningene som skulle testes. Det viste seg å være mer utfordrende enn først tenkt. Vi opplevde at mye av tiden gikk til søking etter eksempler på hvordan det skulle gjøres, samt at veldig mange søketreff viste til det mer populære alternativet Terraform. For å unngå at mye tid skulle gå tapt til knoting ble det besluttet å heller sette opp skytjenestene ved hjelp av nettsidene eller kommandolinjeverktøyet til skyleverandørene. Dette økte hastigheten på testingen, samtidig som muligheten til å reprodusere testmiljøene raskt og enkelt dessverre forsvant. Det ble likevel konkludert med at dette ikke hadde noe påvirkning på sluttresultatet ettersom løsningene i seg selv uansett er like.

5.1.3 Resultatmål

I prosjektets arbeidskontrakt, Vedlegg D Prosjekthåndbok, ble det beskrevet flere resultatmål. Det første målet var at «Oppdragsgivers kunde får en web-applikasjon som muliggjør demonstrasjoner for kunde», med underpunkt at «Applikasjonen skal inneholde som et minimum brukersystem med innlogging, roller og tilgangsstyring. Den skal også inneholde

en «feature» som kunde ønsker». De ingeniørfaglige resultatene viser at applikasjonen oppfylte underpunktet. Oppdragsgivers kunde har brukt applikasjonen til demonstrasjoner, dermed er også resultatmålet oppfylt.

Det andre resultatmålet var «Det fastsettes tydelige og objektive fordeler og ulemper ved forskjellige skyløsninger hos Azure, AWS og GCP» med underpunktet «Her skal det også skrives om hva de forskjellig løsningene er og gjør». De vitenskapelige resultatene i kapittel 4 inneholder en grundig oversikt over de ulike tjenestene som tilbys blant hver skyløsning hos de tre leverandørene. I tillegg forklares grunnprinsippene ved de ulike skyløsningene i kapittel 2.

5.2 Vitenskapelige resultater

5.2.1 FaaS

Funksjoner har flere bruksområder. De kan brukes som endepunkter i et REST API der hver funksjon er et eget endepunkt, men sidejobber er bruksområdet der en virkelig kan se nytten. Siden funksjoner i FaaS er lettvektige og ikke kjører når de ikke brukes, passer de nemlig godt til å utføre oppgaver som kun kjører etter spesifikke hendelser. Den store mengden forskjellige triggere hos de forskjellige leverandørene viser også hvordan dette er bruksområdet som fremheves.

Styrker ved å ta i bruk funksjoner er at det er en rask måte å få funksjonalitet til produksjon, siden man i praksis kun skriver koden som skal utføres når funksjonen kjører. Funksjoner gjør det også lett å sette opp oppgaver som skal utføres ved spesifikke hendelser, noe som kan være mer vrient med et fullverdig API-rammeverk. Ettersom man normalt sett kun betaler for tiden funksjonen kjører, er også funksjoner en god løsning hvis man har et API som ikke blir brukt hele tiden. Om endepunktene er selvstendige og mottar forskjellig trafikkmengde, kan funksjoner fungere som en simplifisert måte å sette opp et API der funksjonalitet kan skalere hver for seg. Skyplattform-leverandørene legger også opp til at det skal være lett å interagere med andre produkter innenfor deres miljø. Hvis en allerede benytter andre produktet hos en skyleverandør, kan funksjoner enkelt brukes til å utvide eksisterende funksjonalitet ved å for eksempel reagere på hendelser.

Svakheter ved å ta i bruk funksjoner er at en gjerne låser seg til en skyleverandør. Hver leverandør håndterer funksjoner på forskjellige måter. Med Quarkus, som vi har benyttet i vår applikasjon, behøver hver leverandør hver sin avhengighet for å ta i bruk funksjoner. Hver avhengighet har i tillegg forskjellige krav til prosjektstruktur og forskjellige inngangspunkt i koden. FaaS fører dermed til mer leverandørinnlåsing enn andre skyløsninger. En annen svakhet ved funksjoner er cold start. Det fører til at funksjoner har varierenede responstid og man kan ikke stole på at de alltid utfører jobben like raskt hver gang. Det kan delvis unngås ved å alltid kjøre

minst én instans, men det fører igjen til høyere utgifter. En annen svakhet i enkelte tilfeller er at mye av kontollen over de underliggende ressursene abstrakteres vekk. Selv om det er mulig å teste lokalt er ikke dette alltid enkelt hvis funksjonen lytter på spesielle triggere. Hvis en funksjon for eksempel lytter på en kø eller en database, må man også sette opp dette lokalt for å teste om alt fungerer som det skal.

Analysen viser en del forskjeller mellom de tre leverandørene. Funksjoner som ikke brukes ofte nok til at cold starts unngås bør kjøre på GCP Cloud Functions for å få raskest cold starts. Alternativt kan minimum antall instanser settes til minst én. Azure er mest fleksibel når det gjelder utplassering av funksjonen og hvilke kjøretidsmiljøer som er støttet. Prisene hos alle tre leverandører er relativt like. Den mest markante forskjellen er at GCP også tar betalt pr. GHz/s. GCP er også betydelig dyrere pr. million kjøringer, men gir til gjengjeld en gratis kvote på to millioner kjøringer som er høyere enn de andre leverandørene.

GCP Cloud Functions har flest restriksjoner i forhold til utplassering og kjøring av kode. Hos GCP er det ikke mulig å kjøre et konteinerbilde som en funksjon. Alle tre plattformene tilbyr sitt eget CLI som kan brukes for å laste opp kode og teste funksjoner lokalt. Azure Functions tilbyr den lengste mulige kjøretiden, men for HTTP-triggere er begrensningen i utgangspunktet 230 sekunder.

5.2.2 PaaS

PaaS-løsninger tar som tidligere nevnt sikte på å gjøre det så enkelt som mulig for brukere å kjøre sin applikasjon i skyen. Det innebærer at den skal ta hånd om det meste av konfigurasjon, slik at tid heller kan brukes til utvikling av applikasjonen. Ulike applikasjoner har ulike behov og en PaaS-løsning må dekke så mange som mulig av disse for at flest mulig skal kunne benytte seg av den. Både Azure App Service, AWS Elastic Beanstalk og Google App Engine Fleksibel støtter det aller meste av kjøretidsmiljøer og konteinere og kan dermed sees på ganske likt når det gjelder hva som kan kjøres. Som det blir beskrevet i kapittel 4.2.2, har også Google App Engine Standard noen styrker men kommer likevel totalt sett dårligst ut. Ettersom forskjellen i ytelsen mellom kjøring av kildekode rett i kjøretidsmiljø i forhold til konteiner er ubetydelig, er det viktigste at en velger en løsning som støtter miljøet koden skal kjøres i.

App Service er den eneste løsningen som støtter å automatisk sette opp en Github Action for kontinuerlig utrulling. Det samme er mulig å sette opp for både Elastic Beanstalk og App Engine, men ikke automatisk. En annen fordel med App Service er hvordan man konfigurerer koden som skal kjøres ettersom man der kun trenger å oppgi en en oppstartskommando eller konteinerbildet. Hos de to andre løsningene må dette legges inn ved hjelp av filer som konfigurerer dette. App Engine begrenser i tillegg konteinerbilder til kun dem som ligger i et Google-register.

Når det gjelder domener og SSL-sertifikater, kommer App Service og App

Engine likt ut ettersom begge skytjenestene legger opp til at en enkelt kan koble til egne domener og i tillegg gir SSL-sertifikater til både egne domener og domene som gis av tjenesten. Elastic Beanstalk tilbyr det samme gjennom å bruke en annen AWS-tjeneste, men det blir dermed også noe mer kronglete. I tillegg er det en svakhet ved Elastic Beanstalk at ikke SSL-sertifikat tilbys med standarddomene ettersom det kan være nyttig i flere sammenhenger, for eksempel i test-miljøer der en ikke nødvendigvis trenger et eget domene.

Et annet viktig punkt er at skyløsningen skal skalere godt når trafikkmengden øker. Alle de tre tjenestene støtter autoskalering, men det er en viss variasjon i hvorvidt brukere selv kan bestemme nøyaktig hvordan skaleringen skal foregå. Der App Service tilbyr finkornet spesifisering av skalering basert på både tid og flere metrikker kombinert, tilbyr Elastic Beanstalk en noe mer begrenset skaleringskonfigurasjon. App Engine skårer dårligst med kun mulighet til å spesifisere øvre og nedre grense for skalering med kun CPU-utnyttelse som metrikk.

PaaS-løsningene tilbyr også diverse annen nyttig funksjonalitet i tillegg, men disse oppleves ikke like nyttige som de kan se ut ettersom de gjerne binder deg til plattformen. Et eksempel på dette er GCP App Engine sine Cron-jobber som krever plattform-spesifikt oppsett for å kjøre, samt for å sikre at ikke hvem som helst kan kjøre funksjonene. I tillegg passer flere av disse funksjonalitetene bedre som egne tjenester, noe de gjerne finnes som også. For eksempel MySQL-databasen som Azure App Service tilbyr, men som ikke kan skalere og en dermed heller bør ta i bruk Azure Database for MySQL.

Prismessig kan en se fra tabell 4.5 at Elastic Beanstalk kommer billigst ut i sammenligningen mot App Service og App Engine. At prisforskjellene er såpass store, særlig fra AWS til Azure og GCP var overraskende. Selv om noe av grunnen er at AWS bruker Arm-prosessorer, skulle en tro at Azure og GCP ville kjempe hardere for å fremstå som konkurransedyktige på pris.

5.2.3 CaaS

Resultatene for anvendbarhet i kapittel 4.2.3 viser at det er relativt stor variasjon mellom hva løsningene tilbyr. Azure Container Instances mangel på fleksibel vedrørende oppsett av egne domener og SSL-sertifikat, oppdatering av eksisterende applikasjoner og skalering tyder på at løsningen er rettet mot andre bruksområder enn eksponering mot sluttbrukere eller kjøring over lang tid. Fordelene Container Instances gir er at det er veldig enkelt å kjøre opp en ny konteiner og la den kjøre en oppgave. Om oppgaven trenger GPU-kraft, er dette enkelt å legge til. Kjøring av flere konteinere sammen er også enkelt å sette opp med Container Groups. Dermed er Container Instances en løsning som retter seg mot de som har behov for å raskt og enkelt kjøre en eller flere konteinere i skyen. Tunge oppgaver som krever mye datakraft ved behov kan løses med Container Instances. Om en behøver vedvarende lagring mellom kjøringer, kan dette

løses med Azure File Share.

Azure Container Apps er sammenlignet med Azure Container Instances en løsning som i mye større grad kan rettes mot bruk over tid og til sluttbrukere. Muligheten til å tilknytte egne domener viser dette. Det er også lagt opp til å enkelt kunne rulle ut nye versjoner og flytte trafikk frem og tilbake mellom versjoner. Skaleringen er veldig fleksibel ettersom den er basert på KEDA [133] og gjør det enkelt å skalere ved forskjeller i trafikk. For applikasjoner som benytter en mikrotjeneste-arkitektur er muligheten til å bruke Dapr også nyttig.

AWS Fargate er relativt fleksibel som støtter både kjøring av kortvarige oppgaver og applikasjoner som skal over tid. På denne måten dekkes til dels både bruksområdet til Azure Container Instances og Azure Container Apps gjennom én løsning. I tillegg støttes det å kjøre flere konteinere sammen og styre nettverking akkurat som en vil. Et mulig minus er at det ikke er mulig å skalere ned til null, noe som kan være nyttig om applikasjonen kun må være aktiv i perioder. Skaleringen kan tilpasses, men er ikke like fleksibel som den Azure Container Apps kan vise til.

Med AWS App Runner får man et enda enklere alternativ for konteinerkjøring enn AWS Fargate. Dette kommer samtidig med enda flere restriksjoner. Det er kun mulig å kjøre én konteiner. For å kjøre flere konteinere, må dette skje i hver sine App Runner-applikasjoner. Kommunikasjon kan dermed ikke skje over *localhost* slik som i AWS Fargate. Det er mulig å skalere til null instanser, men skaleringen kan kun styres basert på antall samtidige forespørsler. Om man vil stille med et ferdigbygd konteinerbilde, må dette ligge i et Elastic Container Registry. For nettverking får man automatisk SSL-sertifikat og kan tilknytte egne domener rett i administrasjonspanelet. Det er også mulig å legge applikasjonen i et VPC.

GCP Cloud Run ligner en del på App Runner, ved at det tilbyr skalering til null instanser, med kun antall samtidige forespørsler som metrikk. Det er kun mulig å kjøre en konteiner. Domener får automatisk SSL-sertifikater og egne domener kan enkelt tilknyttes. Konteinerbilder må ligge i et kontiner-register hos GCP. Cloud Run er likevel noe mer fleksibel ved at man kan velge at applikasjonen skal ha CPU-tilgang hele tiden mot en lavere pris. Man kan styre hvordan trafikk skal fordeles mellom revisjoner og styre hvordan trafikk skal rutes til applikasjonen gjennom lastbalanserer. Prisoversikten i kapittel 4.2.3 viser at hva applikasjon vil koste ikke kun avhenger av hvilke løsning som velges, men også hvor mye applikasjonen vil være aktiv. Det er dermed viktig å tenke gjennom hvordan trafikken til applikasjonen er, og om den i praksis mottar trafikk jevnlig nok til at applikasjonen vil kjøre kontinuerlig.

For å kjøre en oppgave som kun behøver datakraft i en kort periode, er Azure Container Instances og AWS Fargate de to åpenbare kandidatene, og på pris kommer de også relativt jevnt ut. For applikasjonen som behøver kun én konteiner og skal kjøre over tid, er både AWS Fargate, AWS App Runner og GCP Cloud Run gode kandidater. Om appen ikke mottar trafikk kontinuerlig, er App Runner og Cloud Run billigst der Cloud Run gir mest fleksibilitet. Med kontinuerlig eller nesten kontinuerlig trafikk kom-

mer Fargate billigst ut. Azure Container Apps kunne også blitt benyttet, men fremstår som enda bedre på tilfeller der flere konteinere skal kjøres sammen, i tillegg til at det er dyrere enn de andre alternativene. Sammenlignet med AWS Fargate for kjøre flere konteinere kommer Container Apps likevel greit ut. Ved å kun se på prising fremstår Container Apps betydelig dyrere, men tilbyr etter vår mening også ekstra funksjonalitet som i flere tilfeller kan være verdt det.

5.2.4 Konteiner-orkestrering

Å ta i bruk et konteiner-orkestrerings verktøy som Kubernetes krever at en er villig til å lære seg ny teknologi. Det er en høyere læringskurve ved bruk av Kubernetes enn en CaaS- eller PaaS-løsning. Til gjengjeld tilbyr Kubernetes veldig god kontroll over applikasjonen, ettersom det er bruker sitt ansvar å sette opp funksjonalitet. For et stort system med flere avhengigheter på tvers av applikasjoner, kan Kubernetes gjøre det lettere å la disse snakke sammen.

Man betaler for maskinene som skal kjøre klyngen. I teorien blir det nesten det samme som om man skulle kjøpe en VM å og kjøre innholdet i klyngen selv. I tillegg tar både AWS og GCP betalt for håndteringen av Kubernetes-klyngen, med en pris på 0,94 kroner pr. time. Dermed skapes det hos disse leverandørene en inngangspris, men ved kjøring av store klynger blir denne prisen ikke særlig stor i forhold til prisen for maskinene.

Det finnes mye dokumentasjon som gjør det enklere å lære Kubernetes. Oppsett av en simpel applikasjon i Kubernetes er lite komplisert ettersom det finnes mange eksempler. Dermed blir læringskurven vesentlig slakkere enn den ellers hadde vært. Det er også mulig å kjøre en Kubernetes-klynge lokalt. Dette forenkler testing og gir mulighet til å kjøre et miljø lokalt som er helt likt miljøet i skyen.

Overordnet står Kubernetes sterkt ved behov for en fleksibel og oversiktlig tjeneste som kan styre og håndtere applikasjonen. Som presentert i kapittel 4.2.4 og kapittel 4.2.4 får man tilgang til alt CaaS og PaaS tilbyr, men med enda mer kontroll og muligheter for tilpasning. Det faktum at Kubernetes er en åpen standard som tilbys på lik linje hos alle tjenestene er en annen stor fordel. En slipper å låse seg til en spesifikk skyleverandør og kan relativt enkelt flytte applikasjonen. Måten Kubernetes tilbyr administrering av alle tjenester som er nødvendig for applikasjonen, slik som blant annet last-balansering og nettverking, betyr også at man ikke trenger å forhold seg til andre leverandørspesifikke skytjenester.

De svake sidene ved Kubernetes er knyttet til læringskurven og kompleksiteten. For å ta i bruk funksjonaliteten som tilbys er det mye å sette seg inn i og forstå, i motsetning til andre løsninger som løser mye automatisk. Løsninger kan ikke bare settes opp på en-to-tre, men må konfigureres først. Dermed blir det fort et kost-nytte spørsmål om tidsbruken og læringen er verdt det om en ikke har erfaring med Kubernetes fra før. Det gjelder naturligvis kun så lenge andre løsninger som CaaS eller PaaS kan

gjøre samme jobben.

Totalt sett egner Kubernetes seg best om stor fleksibilitet og kontroll er nødvendig. Hvis ikke, er det stor sannsynlighet for at andre løsninger kan gjøre en like god jobb.

5.3 Administrative resultater

5.3.1 Milepæler

Milepælene som ble satt i forprosjektplanen har gitt en oss nyttig oversikt over tidsfrister gjennom hele prosjektet. Det har sikret at vi har prioriert viktige oppgaver, samt ferdigstilt arbeid som blir igangsatt. I tillegg fant vi såpass stor nytte i frister at vi satte opp ekstra frister gjennom prosjektet. Teori og metode-kapitlet ble gitt en intern tidsfrist til 2. april, slik at tiden frem til 20. april kunne gå til innhenting av resultater basert på bestemt metodikk. Etter at resultatene var innhentet, ble det satt en ny frist for et førsteutkast av diskusjon og konklusjon 4. mai, slik at vi i god tid før innlevering kunne få tilbakemeldinger fra oppdragsgiver og eventuelt revidere.

Selv om milepælene har blitt fulgt, har det blitt gjort et par unntak for å kunne gjøre enkelte revideringer basert på endrede behov i prosjektet. Siden produkteier fra start ikke hadde en konkret plan for produktet, ble visjonsdokumentet revidert flere ganger etter fristen. Det samme gjelder naturligvis også databasemodellen.

5.3.2 Timeforbruk

Gjennom prosjektet har antall timer arbeidet pr. uke variert noe. De første ti ukene hadde vi et annet fag som tok litt tid. Vi er likevel fornøyd med å ha jobbet såpass mye disse ukene at vi i prosjektets siste ni uker ikke gikk særlig over 40 timer pr. uke. Det var også mulig med redusert arbeid i påskeferien. Det kan også påpekes at vi mener løsningen med å bruke mye tid på utvikling i starten av prosjektet og deretter vri arbeidet over mot forskning og rapport, var smart. En god og relativt fullverdig applikasjon i testing av skyløsningene var viktig for kvaliteten på resultatene.

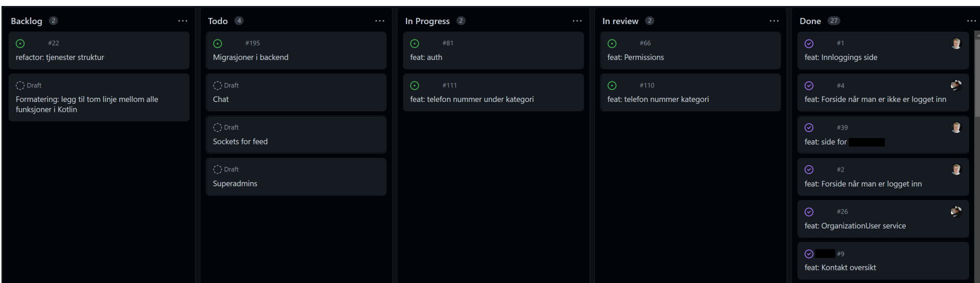
5.3.3 Arbeidsmetodikk

Kanban

Som en del av utviklingsprosessen ble Kanban benyttet. Kanban-brettet har gitt en god oversikt over hva som er gjort, og hva som må gjøres.

Ved hjelp av WIP-grenser holdt vi også lettere styr på antall oppgaver som ble gjort samtidig og passet på at arbeid ikke hopet seg opp uten at det ble fullført. Dette gjenspeiles i prosjektstatus pr. uke, Vedlegg D Prosjekthåndbok, der status ble satt til grønt for fremdrift hver uke.

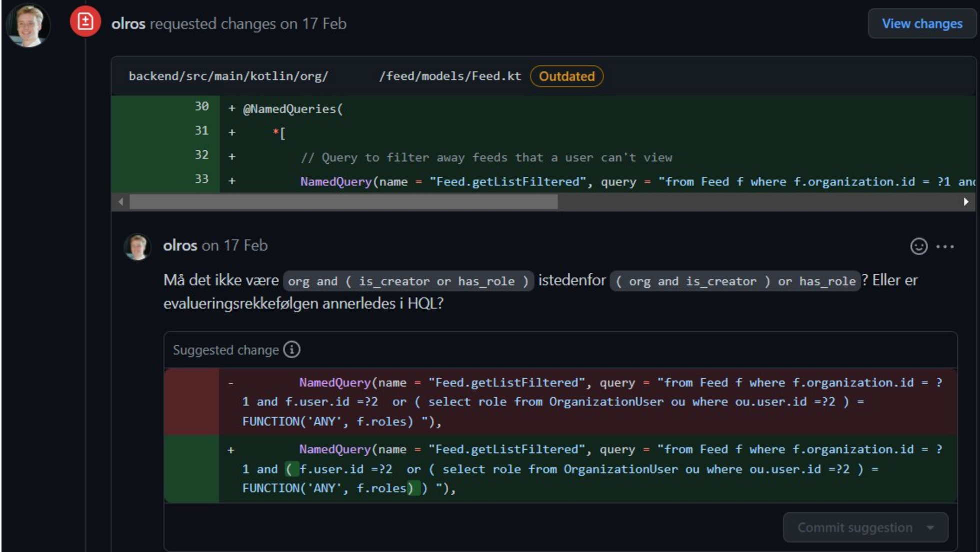
Vi skulle gjerne ønske at GitHub i sitt issue board kunne lagt inn WIP-grense som en innstilling slik at man ikke selv må huske hva WIP-grensen er. Da vi fant ut at GitHub ikke støttet å legge til en WIP-grense, ble det vurdert å bruke andre verktøy slik som for eksempel Trello eller Jir. Fordelene ved å bruke GitHub til alt ble likevel vurdert som store nok til at vi selv heller kunne håndtere WIP-grensen.



Figur 5.1: Skjermbilde fra Kanban-brettet under utviklingen av applikasjonen

Versjonskontroll

Bruken av versjonkontroll fungerte bra. Det var enkelt å holde oversikt over hva den andre på teamet gjorde og holde kvaliteten på arbeidet høy. Pull requests ble benyttet aktivt slik at ny kode ble kvalitetssikret av den andre, samt at CI med tester ble kjørt. GitHub sitt system for å kommentere kode i en pull request, som sett i figur 5.2, gjorde det enkelt å diskutere løsninger, stille spørsmål eller komme med endringsforslag vedrørende spesifikk kode. Totalt sett har dette gjort koden mer ryddig, lesbar og effektiv.



Figur 5.2: Eksempel på kommentar i pull request fra GitHub

Det ble satt opp to miljøer i skyen basert på hovedbranchen slik at vi og produkteier hadde hvert vårt miljø å teste i. På den måten ble det unngått at testing kunne påvirke produkteier. Dette fungerte fint ettersom produktet ikke ble brukt av sluttbrukere. Når sluttbrukere skal ta i bruk løsningen på et senere tidspunkt, ville vi endret oppsettet noe, slik at testmiljøet alltid speiler det nyeste i hovedbranchen, mens produksjonsmiljøet speiler den nyeste versjonen som er valgt ut som helt stabil.

5.3.4 Gruppearbeid

Gjennom prosjektet har vi brukt en flat struktur ved avgjørelser, begge to måtte være enige. Bruken av pull request er et eksempel på dette der den andre måtte gi godkjenning før koden kunne merges inn i hovedbranchen. Vi har heldigvis ikke støtt på noen særlig utfordringer eller uenigheter underveis. Gjennom arbeidet har det vært konsensus om at arbeidstid skal være fra 8 til 16 hver dag, med mulighet for fleksibilitet rundt dette om det skulle være nødvendig underveis. Arbeid åtte timer sammen daglig hos oppdragsgiver har fungert bra. Det har også hindret eventuelle misforståelser, ettersom vi har hatt mulighet til kontinuerlig kommunikasjon og diskusjon av løsninger. Gjennom utviklingen av applikasjonen oppstod det en relativt naturlig deling av ansvar, der hver av teammedlemmene enten jobbet med frontend- eller backend-delen av applikasjonen.

5.4 Feilkilder

Det er naturligvis flere punkter ved gjennomføringen av dette prosjektet som kan skape spørsmål rundt validiteten.

Vi er ikke er skyeksperter og har for eksempel ingen sertifiseringer. I tillegg finnes det store mengder dokumentasjon for hver eneste skytjeneste, slik at det ikke er usannsynlig at vi har oversett enkelte detaljer.

Sammenligningen av priser mellom skytjenestene er en annen feilkilde. Som det har blitt nevnt er det for flere av tjenestene mange forskjellige pris-løsninger som kan velges, med både forskjellige valg for CPU-kraft, minnestørrelse og annen funksjonalitet som følger med. Dermed er det i enkelte tilfeller ikke mulig å sammenligne på 100% likt grunnlag. I tillegg kan det spørres om de instans-størrelsene som er valgt ut er representative. Vi har brukt instans-størrelser med mellom 1 og 8 CPU-er, og fra 1 til 32 GB minne. Hvis de relative prisforskjellene mellom skytjenestene endrer seg for instans-størrelser som for eksempel er ti ganger større i veldig trafikkerte løsninger, vil ikke prisestimatene i denne oppgaven være korrekte. Dette vil likevel ikke påvirke konklusjonen ettersom pris ikke har blitt i evalueringen av styrker og svakheter på tvers av skyløsningene.

I figur 4.7 ser vi store variasjoner mellom skyleverandørene. Selv om resultatene er basert på gjennomsnitt av flere tester, kan forskjeller ha kommet av dårlig testing eller konfigurasjon hos de forskjellige plattformene. Unøyaktighet i resultatinnhentingen kan også ha oppstått. Eventuelle feil her kan være kritisk om GCP Cloud Functions velges på grunn av den laveste cold start-tiden. Det har derimot ikke veldig mye å si for konklusjonen rundt styrker og svakheter for FaaS fordi det der er brukt mange flere argumenter i betraktning.

Det eksisterer og usikkerhet rundt innhenting av forskjell i responstid for PaaS-tjenester, figur 4.4. Det konkluderes med at ettersom ytelsesforskjellen mellom konteinere og ulike kjøretidsmiljøer er ubetydelig er PaaS er godt valg om en ikke vil konteinerisere applikasjonen selv. Om det likevel skulle vise seg at det er markant forskjell i retning enten konteinere eller motsatt vei vil dette påvirke hvordan en bør benytte PaaS-løsninger.

Kapittel 6

Konklusjon og videre arbeid

Målet med denne oppgaven var å identifisere «Hvilke egenskaper, styrker og svakheter finnes ved de forskjellige alternativene for å kjøre webapplikasjoner hos de store skyplattformene?» Konklusjonen presenteres pr. skyløsning. I tillegg drøftes ideer til videre arbeid og samfunnspåvirkning.

6.1 Konklusjon

FaaS gir en god utviklinghastighet og gjerne minimalt med kode og oppsett før sluttbrukere kan ta den i bruk. Funksjoner er også et utmerket valg hvis man har oppgaver som skal kjøres etter spesifikke hendelser eller har funksjonalitet som kan leve selvstendig uten å trenge alt et helt API tilbyr.

Fleksibilitet er en svakhet ved FaaS. Cold starts kan bli et problem, og det er også begrensinger på hvordan koden skal kjøres. Funksjoner gir heller ikke kontroll over den underliggende infrastrukturen. Selv om det er mulig å teste de fleste funksjoner lokalt, er det ikke alltid mulig å reprodusere produksjonsmiljøet 100%.

PaaS lar utviklere fokusere på utvikling av løsningen, mens skyleverandøren håndterer utplasseringen av den. De fleste kjøretidsmiljøer er tilgjengelige hos alle leverandørene, og man kan velge mellom å la leverandøren bygge for deg eller selv levere kjøreklar kode. PaaS-løsningene gir også enkel tilgang til andre skytjenester som blant annet databaser og oppsett av Cron-jobber.

Muligheten til å ta i bruk flere tjenester gjennom samme PaaS-løsning kan likevel skape utfordringer fordi tjenestene ligger sammen med applikasjonen og ikke kan skalere separat. De kan også føre til innlåsing hos leverandøren. I tillegg har man gjerne ikke særlig stor kontroll på infra-

strukturen som ligger til grunn, for eksempel last-balanserer, om en behøver det. CaaS-tjenester fremstår totalt sett som bedre alternativer med mindre man ikke vil konteinerisere applikasjonen selv.

CaaS gjør det enkelt å kjøre konteinere. Her finnes det største utvalget av forskjellige skytjenester som hver for seg løser forskjellige problemer. Det finnes løsninger for å kjøre enkeltoppgaver, web-applikasjoner og applikasjoner med en mikrotjeneste-arkitektur. Her får en også gjerne større tilgang til underliggende infrastruktur enn med PaaS. Flere av tjenestene tilbyr også å la applikasjonen skalere til null, noe som kan senke kostnader.

På grunn av forskjellene mellom tjenestene varierer også svakhetene. Et par av tjenestene har begrensninger på hvor konteinerbildet kan hentes fra, samt at noen av dem også gir en begrenset mulighet til å kontrollere skalering av applikasjonen. Det er vanskelig å sette opp en løsning med like mye kontroll som med Kubernetes.

Kubernetes tilbyr god kontroll over oppsettet av applikasjonen. Du får konfigurasjon av hele løsningen på ett sted. Dette gjør det enkelt å sette opp større applikasjoner som benytter flere tjenester sammen, samt kontrollere hvordan kommunikasjon mellom dem skal foregå. Kubernetes tilbyr også avansert skalering og konfigurasjon for hvordan applikasjonen skal håndtere forskjellige hendelser.

En svakhet ved Kubernetes er at det er et omfattende verktøy med mye funksjonalitet som kan føles overveldende. Man får ikke hjelp av skyleverandøren til oppsett av applikasjonen og konfigurasjonsfilene. Kubernetes er for eksempel ikke nødvendig for applikasjoner med kun én konteiner, men kommer til nytte når flere skal kjøres sammen.

6.2 Videre arbeid

Applikasjonen som vi har utført testene med er pr. i dag ikke optimalisert for mindre skjermer. Mobile enheter brukes ofte til visning av applikasjoner. Et naturlig videre arbeid er dermed å styrke opplevelsen av applikasjonen på mobilenheter.

Vi har ikke foretatt noen grundige undersøkelser av løsninger for sikkerhet i og rundt skyløsninger i denne analysen. Sikkerhet er et veldig viktig aspekt ved alle applikasjoner og bør også vurderes skikkelig ved oppsett i skyen. Å analysere på hvor godt og enkelt skyløsningene tilbyr oppsett av sikkerhet kan være nyttig å se nærmere på.

Flere av leverandørene tilbyr løsninger for å kjøre applikasjoner både hos leverandøren og hos kunde samtidig og koblet sammen. Dette er et interessant konsept som kan løse flere utfordringer ved migrering til skyen. Utfordringer her som blant annet sikkerhet, konfigurasjon og fleksibilitet er et spennende tema det hadde vært nyttig med mer informasjon om. Spesielt ettersom det potensielt kan gi en lettere overgang til skyen.

Som forklart i kapittel 5.1.2 var ikke vårt forsøk på å bruke Pulumi for IaC vellykket. Noe av grunnen til dette var at de fleste søkeresultatene var for en av deres konkurrenter, Terraform. Et interessant videre arbeid kan være forskjeller mellom disse og hva som er mest optimalt av Deklarativ IaC og Imperativ IaC, der for eksempel Pulumi og Terraform bruker hver sin teknikk.

6.3 Samfunnspåvirkning

Analysen i denne oppgaven kan gi en pekepinn på hvordan man kan bruke skytjenester i et systemutviklingsprosjekt. Formålet var å se på styrker og svakheter ved skytjenester som finnes hos de tre store skyleverandørene. Dette kan hjelpe en leser av oppgaven med å velge en passende skytjeneste for sin applikasjon. Selv om skyen ikke lenger kan regnes som et nytt konsept, er det likevel et overveldende utvalg skytjenester som er tilgjengelig. Denne rapporten har sammenlignet flere av disse til bruk for web-applikasjoner, slik at det kan bli enklerer å gjøre et godt valg.

Bruk av skyløsninger fører til opptil 90% lavere CO_2-utslipp, sammenlignet med datasentre hos hver enkelt bedrift [152], [153]. En enklere vei til bruk av skyløsninger kan dermed bidra til å redusere utslipp fra IT-bransjen. Dette er mulig fordi det er mulig å utnytte de tilgjengelige ressursene i mye større grad. I tillegg er det mer energi-effektivt å kjøle ned store datasentre enn servere hos enkeltstående bedrifter.

Skyløsninger skaper likevel enkelte nye problemstillinger. GDPR og kontroll på hvor data lagres er en av disse [154]. Særlig viktig er det hvor persondata lagres, ettersom GDPR-lovgivningen i all hovedsak begrenser muligheten til å overføre personopplysninger ut av EU/EØS [155]. Andre lover eller ønsker kan i tillegg gjøre det nødvendig å lagre data i et spesifikt land. Blant de tre store leverandørene som er sammenlignet i denne oppgaven er det for eksempel kun Azure som tilbyr datasenter i Norge.

Referanser

[1] Wikipedia contributors. «Cron — Wikipedia, The Free Encyclopedia.» (2022), adresse: https://en.wikipedia.org/w/index.php?title=Cron&oldid=1084487115 (sjekket 29.04.2022).

[2] Dapr. «Dapr - Distributed Application Runtime.» (), adresse: https://dapr.io/ (sjekket 29.04.2022).

[3] Wikipedia contributors. «Command-line interface — Wikipedia, The Free Encyclopedia.» (2022), adresse: https://en.wikipedia.org/w/index.php?title=Command-line_interface&oldid=1083631560 (sjekket 29.04.2022).

[4] GitHub Docs. «About pull requests.» (), adresse: https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests (sjekket 29.04.2022).

[5] Datatilsynet. «What is cloud computing?» (23. jun. 2018), adresse: https://www.datatilsynet.no/personvern-pa-ulike-omrader/internett-og-apper/skytjenester/ (sjekket 04.02.2022).

[6] Kaspersky. «Hva er et SSL-sertifikat – definisjon og forklaring.» (), adresse: https://www.kaspersky.no/resource-center/definitions/what-is-a-ssl-certificate (sjekket 11.03.2022).

[7] Cloudflare. «What is a virtual private cloud (VPC)?» (), adresse: https://www.cloudflare.com/learning/cloud/what-is-a-virtual-private-cloud/ (sjekket 29.04.2022).

[8] T. Gramstad. «Åpen kildekode.» (14. apr. 2021), adresse: https://snl.no/%C3%A5pen_kildekode (sjekket 29.04.2022).

[9] M. Varian, «VM and the VM Community: Past, Present, and Future,» s. 17, aug. 1997. adresse: http://www.leeandmelindavarian.com/Melinda/25paper.pdf.

[10] H. Bothner-By, H. Dvergsdal og O. Nordal. «ARPANET.» (), adresse: https://snl.no/ARPANET (sjekket 11.03.2022).

[11] T. Berners-Lee. «World Wide Web project.» (6. aug. 1991), adresse: https://www.w3.org/People/Berners-Lee/1991/08/art-6484.txt.

[12] COMPAQ COMPUTER CORPORATION. «Internet Solutions Division Strategy for Cloud Computing.» (14. nov. 1996), adresse: https://s3.amazonaws.com/files.technologyreview.com/p/pub/legacy/compaq_cst_1996_0.pdf.

[13] Wikipedia. «Timeline of Amazon Web Services.» (), adresse: https://en.wikipedia.org/wiki/Timeline_of_Amazon_Web_Services (sjekket 11.03.2022).

[14] R. Miller. «How AWS came to be.» (2. jul. 2016), adresse: https://techcrunch.com/2016/07/02/andy-jassys-brief-history-of-the-genesis-of-aws/ (sjekket 11.03.2022).

[15] A. Avram. «Docker: Automated and Consistent Software Deployments.» (), adresse: https://www.infoq.com/news/2013/03/Docker/ (sjekket 27.03.2013).

[16] S. Garfinkel og H. Abelson, *Architects of the Information Society: 35 Years of the Laboratory for Computer Science at MIT*, ser. Architects of the Information Society: Thiry-five Years of the Laboratory for Computer Science at MIT. MIT Press, 1999, s. 1, ISBN: 9780262071963. adresse: https://books.google.no/books?id=Fc7dkLGLKrcC.

[17] OpsRamp, «Five Trends Reveal The Emergence of Cloud-First Enterprises,» s. 5, 2020. adresse: https://www.opsramp.com/wp-content/uploads/2017/11/Report-Cloud-First-Enterprises.pdf.

[18] IBM. «What is cloud computing?» (18. aug. 2020), adresse: https://www.ibm.com/cloud/learn/cloud-computing (sjekket 04.02.2022).

[19] IBM. «What is serverless?» (28. aug. 2021), adresse: https://www.ibm.com/cloud/learn/serverless (sjekket 04.03.2022).

[20] The Editors of Encyclopaedia Britannica. «time-sharing.» (), adresse: https://www.britannica.com/technology/time-sharing (sjekket 03.05.2022).

[21] IBM. «What is IaaS (Infrastructure-as-a-Service)?» (12. jul. 2021), adresse: https://www.ibm.com/cloud/learn/iaas (sjekket 04.02.2022).

[22] RedHat. «What is a virtual machine (VM)?» (9. sep. 2019), adresse: https://www.redhat.com/en/topics/virtualization/what-is-a-virtual-machine (sjekket 04.03.2022).

[23] J. Flade. «Docker — What it is, How Images are structured.» (5. jul. 2020), adresse: https://ragin.medium.com/docker-what-it-is-how-images-are-structured-docker-vs-vm-and-some-tips-part-1-d9686303590f (sjekket 25.03.2022).

[24] IBM. «What is containerization?» (23. jun. 2021), adresse: https://www.ibm.com/cloud/learn/containerization (sjekket 04.02.2022).

[25] Docker. «What is a container?» (), adresse: https://www.docker.com/resources/what-container/ (sjekket 25.03.2022).

[26] IBM. «What is container orchestration?» (27. mai 2019), adresse: https://www.ibm.com/cloud/learn/kubernetes (sjekket 08.03.2022).

[27] VMWare. «What is a Kubernetes cluster?» (), adresse: https://www.vmware.com/topics/glossary/content/kubernetes-cluster.html (sjekket 26.04.2022).

[28] Kubernetes. «Nodes.» (), adresse: https://kubernetes.io/docs/concepts/architecture/nodes/ (sjekket 26.04.2022).

[29] Kubernetes. «Pods.» (), adresse: https://kubernetes.io/docs/concepts/workloads/pods/ (sjekket 26.04.2022).

[30] J. Jordan. «An introduction to Kubernetes.» (26. nov. 2019), adresse: https://www.britannica.com/technology/time-sharing (sjekket 03.05.2022).

[31] Kubernetes. «Deployment.» (), adresse: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/ (sjekket 26.04.2022).

[32] Kubernetes. «Service.» (), adresse: https://kubernetes.io/docs/concepts/services-networking/service/ (sjekket 26.04.2022).

[33] Atlassian. «What is containers as a service?» (), adresse: https://www.atlassian.com/microservices/cloud-computing/containers-as-a-service (sjekket 25.04.2022).

[34] IBM. «What is PaaS (Platform-as-a-Service)?» (14. jul. 2021), adresse: https://www.ibm.com/cloud/learn/paas (sjekket 04.02.2022).

[35] IBM. «What is FaaS (Function-as-a-Service)?» (30. jul. 2019), adresse: https://www.ibm.com/cloud/learn/faas (sjekket 04.03.2022).

[36] Microsoft Azure. «What is Infrastructure as Code?» (29. jun. 2019), adresse: https://docs.microsoft.com/en-us/devops/deliver/what-is-infrastructure-as-code (sjekket 04.03.2022).

[37] RedHat. «What is Infrastructure as Code (IaC)?» (1. des. 2020), adresse: https://www.redhat.com/en/topics/automation/what-is-infrastructure-as-code-iac (sjekket 04.03.2022).

[38] Quarkus. (), adresse: https://quarkus.io/ (sjekket 31.03.2022).

[39] Quarkus. «What is Quarkus?» (), adresse: https://quarkus.io/about/ (sjekket 03.05.2022).

[40] Quarkus. «Container First.» (), adresse: https://quarkus.io/container-first/ (sjekket 03.05.2022).

[41] Vercel. «What is Next.js?» (), adresse: https://nextjs.org/learn/foundations/about-nextjs/what-is-nextjs (sjekket 02.05.2022).

[42] PostgreSQL. «PostgreSQL: The World's Most Advanced Open Source Relational Database.» (), adresse: https://www.postgresql.org/ (sjekket 02.05.2022).

[43] Stack Overflow. «Developer Surver 2021 - Databases - Most wanted.» (2021), adresse: https://insights.stackoverflow.com/survey/2021#most-loved-dreaded-and-wanted-database-want (sjekket 14.05.2022).

[44] Docker. «Why Docker.» (), adresse: https://www.docker.com/why-docker/ (sjekket 02.05.2022).

[45] Docker. «Overview of Docker Compose.» (), adresse: https://docs.docker.com/compose/ (sjekket 02.05.2022).

[46] Pulumi. (), adresse: https://www.pulumi.com/ (sjekket 31.03.2022).

[47] Auth0. (), adresse: https://auth0.com/ (sjekket 31.03.2022).

[48] Algolia. (), adresse: https://www.algolia.com/ (sjekket 31.03.2022).

[49] Atlassian. «What is kanban?» (), adresse: https://www.atlassian.com/agile/kanban (sjekket 08.03.2022).

[50] S. Grønmo. «Forskningsmetode.» (10. mai 2021), adresse: https://snl.no/forskningsmetode_-_samfunnsvitenskap (sjekket 16.05.2022).

[51] S. Grønmo. «Kvalitativ metode.» (3. nov. 2020), adresse: https://snl.no/kvalitativ_metode (sjekket 16.05.2022).

[52] Wikipedia. «Kvalitativ metode.» (2021), adresse: https://no.wikipedia.org/w/index.php?title=Kvalitativ_metode&oldid=21603012 (sjekket 16.05.2022).

[53] S. Grønmo. «Kvantitativ metode.» (7. nov. 2021), adresse: https://snl.no/kvantitativ_metode (sjekket 16.05.2022).

[54] Loadster. «Loadster: Load & Stress Testing for High-Performance Websites.» (), adresse: https://loadster.app/ (sjekket 03.05.2022).

[55] Kubernetes. «What is Kubernetes?» (), adresse: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ (sjekket 08.03.2022).

[56] IBM. «What is a virtual machine (VM)?» (20. jun. 2019), adresse: https://www.ibm.com/cloud/learn/virtual-machines (sjekket 30.03.2022).

[57] Google Cloud Platform. «What is Compute Engine? Use cases, security, pricing and more.» (25. mai 2021), adresse: https://cloud.google.com/blog/topics/developers-practitioners/what-compute-engine-use-cases-security-pricing-and-more (sjekket 30.03.2022).

[58] Microsoft Azure. «What is a virtual machine (VM)?» (), adresse: https://azure.microsoft.com/en-us/overview/what-is-a-virtual-machine/#overview (sjekket 30.03.2022).

[59] Netlify. (), adresse: https://www.netlify.com/ (sjekket 28.04.2022).

[60] Google. «Calling Cloud Functions.» (), adresse: https://cloud.google.com/functions/docs/calling (sjekket 03.05.2022).

[61] Google. «Concepts.» (), adresse: https://cloud.google.com/functions/docs/concepts (sjekket 02.05.2022).

[62] Amazon. «What is AWS Lambda?» (), adresse: https://docs.aws.amazon.com/lambda/latest/dg/welcome.html (sjekket 02.05.2022).

[63] Azure. «Introduction to Azure Functions.» (), adresse: https://docs.microsoft.com/en-us/azure/azure-functions/functions-overview (sjekket 02.05.2022).

[64] Google. «Configuring Cloud Functions.» (), adresse: https://cloud.google.com/functions/docs/configuring (sjekket 03.05.2022).

[65] Amazon. «Lambda function scaling.» (), adresse: https://docs.aws.amazon.com/lambda/latest/dg/invocation-scaling.html (sjekket 03.05.2022).

[66] Microsoft Azure. «Priser på Azure Functions.» (), adresse: https://azure.microsoft.com/nb-no/pricing/details/functions/ (sjekket 21.04.2022).

[67] Google Cloud Platform. «Cloud Functions pricing.» (), adresse: https://cloud.google.com/functions/pricing (sjekket 21.04.2022).

[68] Amazon. «AWS Lambda Pricing.» (), adresse: https://aws.amazon.com/lambda/pricing/ (sjekket 21.04.2022).

[69] Quarkus. «FUNQY.» (), adresse: https://quarkus.io/guides/funqy (sjekket 11.05.2022).

[70] Microsoft Azure. «App Service overview.» (13. apr. 2022), adresse: https://docs.microsoft.com/en-us/azure/app-service/overview (sjekket 13.05.2022).

[71] Microsoft Azure. «Configure an App Service app.» (26. apr. 2022), adresse: https://docs.microsoft.com/en-us/azure/app-service/configure-common (sjekket 13.05.2022).

[72] Microsoft Azure. «Announcing MySQL in-app for Web Apps (Windows).» (18. aug. 2016), adresse: https://azure.github.io/AppService/2016/08/18/Announcing-MySQL-in-app-for-Web-Apps-(Windows).html (sjekket 13.05.2022).

[73] Microsoft Azure. «Monitoring App Service.» (31. mar. 2022), adresse: https://docs.microsoft.com/en-us/azure/app-service/monitor-app-service (sjekket 13.05.2022).

[74] Microsoft Azure. «Secure a custom DNS name with a TLS/SSL binding in Azure App Service.» (27. apr. 2022), adresse: https://docs.microsoft.com/en-us/azure/app-service/configure-ssl-bindings (sjekket 13.05.2022).

[75] Microsoft Azure. «Continuous deployment with custom containers in Azure App Service.» (24. mar. 2022), adresse: https://docs.microsoft.com/en-us/azure/app-service/deploy-ci-cd-custom-container (sjekket 13.05.2022).

[76] Microsoft Azure. «Set up staging environments in Azure App Service.» (29. apr. 2022), adresse: https://docs.microsoft.com/en-us/azure/app-service/deploy-staging-slots (sjekket 13.05.2022).

[77] AWS. «Using the Amazon ECS platform branch.» (), adresse: https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/create_deploy_docker_ecs.html (sjekket 20.04.2022).

[78] AWS. «Configuring the reverse proxy.» (), adresse: https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/java-se-nginx.html (sjekket 13.05.2022).

[79] AWS. «Environment configuration using the Elastic Beanstalk console.» (), adresse: https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/environments-cfg-console.html (sjekket 13.05.2022).

[80] AWS. «Adding a database to your Elastic Beanstalk environment.» (), adresse: https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/using-features.managing.db.html (sjekket 13.05.2022).

[81] AWS. «Your Elastic Beanstalk environment's Domain name.» (), adresse: https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/customdomains.html (sjekket 13.05.2022).

[82] AWS. «Deployment policies and settings.» (), adresse: https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/using-features.rolling-version-deploy.html (sjekket 13.05.2022).

[83] Google Cloud Platform. «Choose an App Engine environment.» (7. apr. 2022), adresse: https://cloud.google.com/appengine/docs/the-appengine-environments (sjekket 07.04.2022).

[84] Google Cloud Platform. «app.yaml Configuration File.» (12. mai 2022), adresse: https://cloud.google.com/appengine/docs/standard/java-gen2/config/appref (sjekket 13.05.2022).

[85] Google Cloud Platform. «Scheduling Jobs with cron.yaml.» (12. mai 2022), adresse: https://cloud.google.com/appengine/docs/standard/java-gen2/scheduling-jobs-with-cron-yaml (sjekket 13.05.2022).

[86] Google Cloud Platform. «Testing and Deploying your Application.» (12. mai 2022), adresse: https://cloud.google.com/appengine/docs/standard/java-gen2/testing-and-deploying-your-app (sjekket 13.05.2022).

[87] Google Cloud Platform. «Testing and Deploying your Application.» (12. mai 2022), adresse: https://cloud.google.com/appengine/docs/flexible/custom-runtimes/testing-and-deploying-your-app (sjekket 13.05.2022).

[88] Google Cloud Platform. «Migrating Traffic.» (12. mai 2022), adresse: https://cloud.google.com/appengine/docs/standard/java-gen2/migrating-traffic (sjekket 13.05.2022).

[89] Google Cloud Platform. «Mapping Custom Domains.» (12. mai 2022), adresse: https://cloud.google.com/appengine/docs/standard/java-gen2/mapping-custom-domains (sjekket 13.05.2022).

[90] Google Cloud Platform. «Securing Custom Domains with SSL.» (12. mai 2022), adresse: https://cloud.google.com/appengine/docs/standard/java-gen2/securing-custom-domains-with-ssl (sjekket 13.05.2022).

[91] Microsoft Azure. «Get started with Autoscale in Azure.» (23. apr. 2022), adresse: https://docs.microsoft.com/en-us/azure/azure-monitor/autoscale/autoscale-get-started (sjekket 13.05.2022).

[92] AWS. «Auto Scaling group for your Elastic Beanstalk environment.» (), adresse: https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/using-features.managing.as.html (sjekket 13.05.2022).

[93] Google Cloud Platform. «How Instances are Managed.» (12. mai 2022), adresse: https://cloud.google.com/appengine/docs/standard/python/how-instances-are-managed (sjekket 13.05.2022).

[94] Loadster. «Test report - Azure App Service JAR-file.» (), adresse: https://loadster.app/dashboard/reports/c9yEtGupzjheUys8 (sjekket 04.04.2022).

[95] Loadster. «Test report - Azure App Service container.» (), adresse: https://loadster.app/dashboard/reports/MtGDagXyRRw4b9bY (sjekket 04.04.2022).

[96] Loadster. «Test report - AWS Elastic Beanstalk JAR-file.» (), adresse: https://loadster.app/dashboard/reports/l7mvOiduD1rZ77zv (sjekket 04.04.2022).

[97] Loadster. «Test report - AWS Elastic Beanstalk container.» (), adresse: https://loadster.app/dashboard/reports/YwvDsUrtcJipZfY3 (sjekket 04.04.2022).

[98] Loadster. «Test report - Google App Engine JAR-file.» (), adresse: https://loadster.app/dashboard/reports/stBTftWaLIiONpRr (sjekket 05.04.2022).

[99] Loadster. «Test report - Google App Engine container.» (), adresse: https://loadster.app/dashboard/reports/NmoxaYHmIruJfgJY (sjekket 05.04.2022).

[100] Microsoft Azure. «Priser på Azure App Service.» (), adresse: https://azure.microsoft.com/en-us/pricing/details/app-service/linux/ (sjekket 03.05.2022).

[101] AWS. «AWS Elastic Beanstalk Pricing.» (), adresse: https://aws.amazon.com/elasticbeanstalk/pricing/ (sjekket 03.05.2022).

[102] Google Cloud Platform. «App Engine pricing.» (), adresse: https://cloud.google.com/appengine/pricing (sjekket 03.05.2022).

[103] Microsoft Azure. «Deploy container instances that use GPU resources.» (9. feb. 2022), adresse: https://docs.microsoft.com/en-us/azure/container-instances/container-instances-gpu (sjekket 29.04.2022).

[104] Microsoft Azure. «YAML reference: Azure Container Instances.» (15. des. 2021), adresse: https://docs.microsoft.com/en-us/azure/container-instances/container-instances-reference-yaml#ipaddress-object (sjekket 29.04.2022).

[105] Microsoft Azure. «Container groups in Azure Container Instances.» (21. apr. 2021), adresse: https://docs.microsoft.com/en-us/azure/container-instances/container-instances-container-groups (sjekket 21.04.2022).

[106] Microsoft Azure. «Mount an Azure file share in Azure Container Instances.» (21. apr. 2021), adresse: https://docs.microsoft.com/en-us/azure/container-instances/container-instances-volume-azure-files (sjekket 29.04.2022).

[107] Microsoft Azure. «Update containers in Azure Container Instances.» (15. des. 2021), adresse: https://docs.microsoft.com/en-us/azure/container-instances/container-instances-update (sjekket 29.04.2022).

[108] Microsoft Azure. «Azure Container Apps.» (), adresse: https://azure.microsoft.com/en-us/services/container-apps/#features (sjekket 29.04.2022).

[109] Microsoft Azure. «Azure Container Apps Preview environments.» (20. feb. 2022), adresse: https://docs.microsoft.com/en-us/azure/container-apps/environment (sjekket 29.04.2022).

[110] Microsoft Azure. «Azure Container Apps Preview environments.» (1. mar. 2022), adresse: https://docs.microsoft.com/en-us/azure/container-apps/containers (sjekket 29.04.2022).

[111] Microsoft Azure. «Custom domain names and certificates in Azure Container Apps.» (14. mai 2022), adresse: https://docs.microsoft.com/en-us/azure/container-apps/custom-domains-certificates (sjekket 14.05.2022).

[112] Microsoft Azure. «Tutorial: Deploy a background processing application with Azure Container Apps.» (29. mar. 2022), adresse: https://docs.microsoft.com/en-us/azure/container-apps/background-processing (sjekket 29.04.2022).

[113] Microsoft Azure. «Publish revisions with GitHub Actions in Azure Container Apps.» (30. des. 2021), adresse: https://docs.microsoft.com/en-us/azure/container-apps/github-actions-cli (sjekket 29.04.2022).

[114] Microsoft Azure. «Revisions in Azure Container Apps.» (24. feb. 2022), adresse: https://docs.microsoft.com/en-us/azure/container-apps/revisions (sjekket 29.04.2022).

[115] Microsoft Azure. «Application lifecycle management in Azure Container Apps.» (20. feb. 2022), adresse: https://docs.microsoft.com/en-us/azure/container-apps/application-lifecycle-management (sjekket 29.04.2022).

[116] AWS. «What is AWS Fargate?» (), adresse: https://docs.aws.amazon.com/AmazonECS/latest/userguide/what-is-fargate.html (sjekket 13.05.2022).

[117] AWS. «Amazon ECS task definitions.» (), adresse: https://docs.aws.amazon.com/AmazonECS/latest/userguide/task_definitions.html (sjekket 13.05.2022).

[118] AWS. «Amazon ECS services.» (), adresse: https://docs.aws.amazon.com/AmazonECS/latest/userguide/ecs_services.html (sjekket 13.05.2022).

[119] AWS. «Service definition parameters.» (), adresse: https://docs.aws.amazon.com/AmazonECS/latest/userguide/service_definition_parameters.html (sjekket 13.05.2022).

[120] AWS. «Amazon ECS Deployment types.» (), adresse: https://docs.aws.amazon.com/AmazonECS/latest/userguide/deployment-types.html (sjekket 13.05.2022).

[121] AWS. «What is AWS App Runner?» (), adresse: https://docs.aws.amazon.com/apprunner/latest/dg/what-is-apprunner.html (sjekket 13.05.2022).

[122] AWS. «App Runner architecture and concepts.» (), adresse: https://docs.aws.amazon.com/apprunner/latest/dg/architecture.html (sjekket 13.05.2022).

[123] AWS. «Managing your App Runner service.» (), adresse: https://docs.aws.amazon.com/apprunner/latest/dg/manage.html (sjekket 13.05.2022).

[124] AWS. «Deploying a new application version to App Runner.» (), adresse: https://docs.aws.amazon.com/apprunner/latest/dg/manage-deploy.html (sjekket 13.05.2022).

[125] Google Cloud Platform. «What is Cloud Run.» (13. mai 2022), adresse: https://cloud.google.com/run/docs/overview/what-is-cloud-run (sjekket 13.05.2022).

[126] Google Cloud Platform. «Deploying container images.» (12. mai 2022), adresse: https://cloud.google.com/run/docs/deploying (sjekket 13.05.2022).

[127] Google Cloud Platform. «Configure containers.» (12. mai 2022), adresse: https://cloud.google.com/run/docs/configuring/containers (sjekket 13.05.2022).

[128] Google Cloud Platform. «CPU allocation (services).» (12. mai 2022), adresse: https://cloud.google.com/run/docs/configuring/cpu-allocation (sjekket 13.05.2022).

[129] Google Cloud Platform. «Connecting to a VPC network.» (12. mai 2022), adresse: https://cloud.google.com/run/docs/configuring/connecting-vpc (sjekket 13.05.2022).

[130] Google Cloud Platform. «Mapping custom domains.» (12. mai 2022), adresse: https://cloud.google.com/run/docs/mapping-custom-domains (sjekket 13.05.2022).

[131] Google Cloud Platform. «Rollbacks, gradual rollouts, and traffic migration.» (12. mai 2022), adresse: https://cloud.google.com/run/docs/rollouts-rollbacks-traffic-migration (sjekket 13.05.2022).

[132] Microsoft Azure. «What is Azure Container Instances?» (22. okt. 2021), adresse: https://docs.microsoft.com/en-us/azure/container-instances/container-instances-overview (sjekket 29.04.2022).

[133] KEDA. «KEDA | Kubernetes Event-driven Autoscaling.» (), adresse: https://keda.sh/ (sjekket 26.04.2022).

[134] Microsoft Azure. «Set scaling rules in Azure Container Apps.» (12. mai 2022), adresse: https://docs.microsoft.com/en-us/azure/container-apps/scale-app (sjekket 29.04.2022).

[135] AWS. «Service auto scaling.» (), adresse: https://docs.aws.amazon.com/AmazonECS/latest/userguide/service-auto-scaling.html (sjekket 13.05.2022).

[136] AWS. «Managing App Runner automatic scaling.» (), adresse: https://docs.aws.amazon.com/apprunner/latest/dg/manage-autoscaling.html (sjekket 13.05.2022).

[137] Google Cloud Platform. «About container instance autoscaling.» (12. mai 2022), adresse: https://cloud.google.com/run/docs/about-instance-autoscaling (sjekket 13.05.2022).

[138] Microsoft Azure. «Priser på Azure Container Instances.» (), adresse: https://azure.microsoft.com/nb-no/pricing/details/container-instances/ (sjekket 21.04.2022).

[139] Microsoft Azure. «Priser på Azure Container Apps.» (), adresse: https://azure.microsoft.com/nb-no/pricing/details/container-apps/ (sjekket 21.04.2022).

[140] AWS. «AWS App Runner Pricing.» (), adresse: https://aws.amazon.com/apprunner/pricing/ (sjekket 21.04.2022).

[141] AWS. «AWS Fargate Pricing.» (), adresse: https://aws.amazon.com/fargate/pricing/ (sjekket 21.04.2022).

[142] Google Cloud Platform. «Cloud Run pricing.» (), adresse: https://cloud.google.com/run/pricing (sjekket 21.04.2022).

[143] Kubernetes. «Monitoring, Logging, and Debugging.» (), adresse: https://kubernetes.io/docs/tasks/debug/ (sjekket 03.05.2022).

[144] Kubernetes. «Secrets.» (), adresse: https://kubernetes.io/docs/concepts/configuration/secret/ (sjekket 03.05.2022).

[145] Kubernetes. «Manage TLS Certificates in a Cluster.» (), adresse: https://kubernetes.io/docs/tasks/tls/managing-tls-in-a-cluster/ (sjekket 03.05.2022).

[146] Kubernetes. «Horizontal Pod Autoscaling.» (), adresse: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/ (sjekket 03.05.2022).

[147] Kubernetes. «Partners.» (), adresse: https://kubernetes.io/partners/ (sjekket 03.05.2022).

[148] Microsoft Azure. «Priser på Azure Kubernetes Service (AKS).» (), adresse: https://azure.microsoft.com/nb-no/pricing/details/kubernetes-service/ (sjekket 21.04.2022).

[149] Amazon. «Amazon EKS pricing.» (), adresse: https://aws.amazon.com/eks/pricing/ (sjekket 21.04.2022).

[150] Google Cloud. «Google Kubernetes Engine pricing.» (), adresse: https://cloud.google.com/kubernetes-engine/pricing (sjekket 21.04.2022).

[151] NextAuth.js. (), adresse: https://next-auth.js.org/ (sjekket 09.05.2022).

[152] AWS Public Sector Blog. «Why moving to the cloud should be part of your sustainability strategy.» (25. okt. 2021), adresse: https://aws.amazon.com/blogs/publicsector/why-moving-cloud-part-of-sustainability-strategy/ (sjekket 19.05.2022).

[153] Tenacity. «The Sustainability of Public Cloud vs. On Prem Data Centers.» (21. apr. 2022), adresse: https://www.tenacitycloud.com/blog/the-sustainability-of-public-cloud-vs-on-prem-data-centers (sjekket 19.05.2022).

[154] T. Judin. «Privacy Shield-avtalen mellom USA og EU/EØS er opphevet.» (16. jul. 2020), adresse: https://www.datatilsynet.no/aktuelt/aktuelle-nyheter-2020/privacy-shield-avtalen-mellom-usa-og-eueos-er-opphevet/ (sjekket 09.05.2022).

[155] Datatilsynet. «Overføring av personopplysninger ut av EØS.» (4. nov. 2021), adresse: https://www.datatilsynet.no/rettigheter-og-plikter/virksomhetenes-plikter/overforing-av-personopplysninger-ut-av-eos/ (sjekket 19.05.2022).

Vedlegg A

Visjonsdokument

60

Bacheloroppgave for Kantega

Visjonsdokument

Versjon 1.0

Revisjonshistorie

Dato	Versjon	Beskrivelse	Forfatter
27/01/22	0.1	Små start på prosjekt beskrivelse	Hermann Elton, Olaf Rosendahl
01/02/22	0.2	Jobbet med sammendrag rundt produkt	Hermann Elton
02/02/22	0.3	Arbeidet med beskrivelse av brukere og interessenter	Hermann Elton, Olaf Rosendahl
04/02/22	0.4	Språkvask, lagt til kilder.	Olaf Rosendahl
16/02/22	0.5	Revidert for å ikke forteller for mye om start-up og holde mer rød tråd rundt forskningsdelen	Olaf Rosendahl
04/03/22	1.0	Utdypet mer rundt hvordan forskningsdelen skal gjøres og innebære	Olaf Rosendahl

Innholdsfortegnelse

1. Innledning 4

 1.1 Referanser 4

2. Sammendrag problem og produkt 4

 2.1 Problemsammendrag 5

 2.2 Produktsammendrag 5

3. Overordnet beskrivelse av interessenter og brukere 6

 3.1 Oppsummering interessenter 6

 3.2 Oppsummmering brukere 7

 3.3 Brukermiljøet 7

 3.4 Sammendrag av brukernes behov 7

 3.5 Alternativer til vårt produkt 9

4. Produktoversikt 10

 4.1 Produktets rolle i brukermiljøet 10

 4.2 Forutsetninger og avhengigheter 10

5. Produktets funksjonelle egenskaper 10

6. Ikke-funksjonelle egenskaper og andre krav 11

1. Innledning

Dette dokumentet beskriver kravene til systemet som skal utvikles for kunde av oppdragsgiver Kantega. Systemet utvikles med to formål. Formål én er en prototype som kan bli benyttet til brukertesting og som et «proof of concept» for kunde av Kantega. Kunden er en start-up som ikke ønsker at deres idé skal offentliggjøres før dette prosjektet er ferdigstilt. Vi kommer dermed ikke til å gå inn på nøyaktig hva deres totale, ferdigstilte produkt vil være. Kunden av Kantega vil heretter bli omtalt som start-up'en. Vi kommer heller ikke til å utvikle hele produktet i denne oppgaven, det ville vært for mye jobb. Istedenfor vil det fokuseres på grunnleggende funksjonalitet som må utvikles for å muliggjøre brukertesting slik at start-up'en kan sjekke om ideen har et marked. Formål to henger tett sammen med prosjektets problembeskrivelse. I tillegg til systemutviklingen i dette prosjektet skal det også gjennomføres en komparativ analyse av forskjellige løsninger for hosting av web-applikasjoner. Systemet som utvikles vil benyttes til denne testingen. Det vil være produktet som skal hostes på de forskjellige plattformene slik at det blir en rettferdig sammenligning. Dette dokumentet vil også beskrive hvordan sammenligningen skal skje.

2. Sammendrag problem og produkt

De tre store skyplattformene (Google Cloud Platform, Microsoft Azure og AWS) tilbyr en stor mengde produkter og løsninger for å rulle ut og kjøre programvare. For utviklere og arkitekter er det utfordrende å orientere seg i dette store landskapet av muligheter. Hovedmålet med dette prosjektet er å gjøre en komparativ analyse av ulike alternativ for å kjøre programvare på de store skyplattformene. I analysen vil vi se på ulike faktorer som kostnad, fleksibilitet, skalerbarhet, resilens, robusthet, mulighet til å flytte mellom leverandører og DX («developer experience»). Det skal defineres klart hvordan løsningene kan sammenlignes på like vilkår. Som et mål i seg selv, skal vi utvikle en prototype på en web-applikasjon for start-up'en. Denne applikasjonen vil bli benyttet for sammenligningen av skyløsninger.

2.1 Problemsammendrag

Problem med	Google Cloud Platform (GCP), Microsoft Azure og AWS tilbyr en stor mengde produkter og løsninger for å rulle ut og kjøre programvare. For utviklere og arkitekter er det utfordrende å orientere seg i dette store landskapet av muligheter.
berører	Utviklere, programvarearkitekter og andre som ønsker å kjøre programvare i skyen.
som resultatet av dette	Er det vanskelig å velge riktig løsning for sin programvare. Dette kan føre til ulemper som blant annet dårlig opplevelse, samt dyrere og mindre fleksibel løsning enn det som kunne vært mulig.
en vellykket løsning vil	Gi en oversikt over fordeler og ulemper ved de forskjellige skyløsningene som finnes hos GCP, Azure og AWS for kjøring av en web-applikasjon. Det er og ønskelig å kunne peke på hvilke løsninger som er mest optimale for forskjellige behov.

2.2 Produktsammendrag

Systemet som utvikles skal inneholde grunnleggende funksjonalitet som er nødvendig for at brukere skal kunne benytte den. I tillegg skal det utvikles et par løsninger som gir brukere lyst til å bruke den. Dette utvikles for å skape en representativ web-applikasjon for uttesting av skyløsninger. Det er et viktig poeng at web-applikasjonen som benyttes for testing inneholder funksjonalitet som mange andre web-applikasjoner og har. Dette er for å sikre at vi ser på en reel problemstilling og ikke for eksempel bruker en for enkel applikasjon som det i praksis ikke er etterspørsel for å hoste i skyen.

For	start-up’en
som	ønsker en prototype for å teste ut brukernes behov
er vår prototype	en applikasjon
som	muliggjør slik testing
ettersom	den inneholder grunnleggende funksjoner slik som brukerhåndtering med autentisering, roller og organisasjonstilhørighet
samt	et par ekstra funksjoner som vil gi bruker ønske om å benytte applikasjonen.

3. Overordnet beskrivelse av interessenter og brukere

3.1 Oppsummering interessenter

Navn	Utdypende beskrivelse	Rolle under utviklingen
Kantega	Kantega er et konsulentselskap og er leid inn av deres kunde for å utviklet deres produkt ide.	Kantega fungerer som prosjektleder og produkteier sammen med start-up’en. De vil også bidra underveis med designer som lager skisser i Figma, samt veiledning rundt teknologi og valg av teknologi.
Start-up	Start-up’en er selskapet som systemet utvikles for. De har forretningsideen og visjonen om hvordan systemet skal se ut og brukes.	Start-up’en fungerer som produkteier og vil på ukentlige møter kunne gi tilbakemeldinger på det som utvikles slik at det blir som de ønsker. De kommer

		også til å gjennomføre brukertester.

3.2 Oppsummering brukere

Navn	Utdypende beskrivelse	Rolle under utviklingen	Representert av
Kunde av start-up	Start-up'en vil på et tidspunkt (ikke i løpet av dette prosjektet) selge produktet sitt til kunder. Disse kundene er hovedmålgruppen og web-appen skal forenkle hverdagen deres	Det vil utføres brukertester på de potensielle kundene av start-up'en	Start-up har kontakt med flere interessenter og vil benytte dem for brukertesting

3.3 Brukermiljøet

Systemet skal være enkelt å bruke. Målet er at nye brukere ikke skal trenge opplæring fordi nettsiden er så intuitiv at en skjønner hvordan det brukes umiddelbart. De mulige kundene av start-up'en har ikke nødvendigvis samme utstyr overalt. Det er derfor viktig at web-appen fungerer i de mest benyttede nettleserne. Ifølge «Statcounter GlobalStats» har Chrome, Safari, Firefox og Edge en total markedsandel på 95.3% i Norge [1]. Vi vil som minimum støtte alle disse. Nettsiden skal også være responsiv slik at skjermstørrelse og eventuell berøringsskjerm ikke har påvirkning på bruksmulighetene til brukerne. Mobil, nettbrett og pc skal dermed være støttet på lik linje.

3.4 Sammendrag av brukernes behov

Behov	Prioritet	Påvirker	Dagens løsning	Foreslått løsning

Brukerhåndtering	Høy	Innlogging	N/A	Bruke tredjeparts løsning for autentisering kombinert med lagring av brukere i eget system.
Rollesystem	Høy	Oppgaver og innlogging	N/A	Rollesystem hvor hver bruker kan ha en eller flere roller basert på deres stilling i organisasjonen. Disse kan også brukes til å bestemme tilganger.
Intern kommunikasjons-kanal	Høy	Oppgaver	Flere systemer og kommunkasjons-måter	Intern kommunkasjonsfeed for kommunikasjon innad i organisasjon. Meldinger skal kunne sendes til spesifikke grupper innad i organisasjonen.
Huskeliste	Middels	Oppgaver	Hver bruker har egen oversikt	Oversikt over oppgaver som brukere må huske å gjøre. Ved å knytte dette sammen med kommunikasjon kan oppgaver legges ut i kommunkasjonskanalen med forespørsel om noen kan påta seg oppgaven.

Egenutviklet telefonkatalog	Middels	Oppgaver	Hver bruker har egen oversikt	Telefonkatalog, som kan inneholde både lokale egenbestemte numre per organisasjon, samt telefonnumre lagt til av start-up
Bruke eksternt system	Lav	Oppgaver	Direkte bruk med separat innlogging for hvert system	Integrasjon av eksternt system slik at det kan brukes rett fra nettsiden gjennom f.eks. et API
Finne eksterne system	Høy	Oppgaver	Hver bruker har egen oversikt	Hver organisasjon kan ha sin egen tilpassede liste over lenker til eksterne systemer

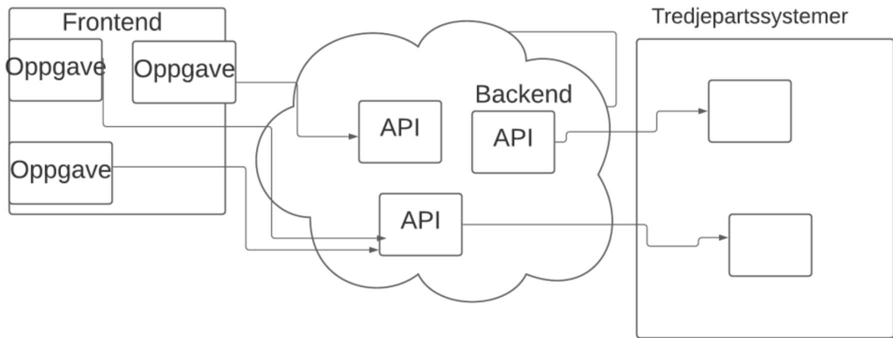
3.5 Alternativer til vårt produkt

Start-up'en ønsker å lage et produkt som samler mange tjenester som deres mulige kunder i dag har spredt ut over mange forskjellige systemer. På den måten ønsker de å forenkle hverdagen til brukerne og unngå at de må ha mange faner i nettleseren oppe samtidig og mange bokmerker for å holde oversikt. Start-up har gjennomført grundig utforskning og er selv fra bransjen de skal selge til og det er per dags dato ingen direkte alternativer til produktet som skal utvikles.

Gjennom våre undersøkelser har vi ikke funnet en sammenligning som likner på den vi skal gjennomføre. Det finnes generelle sammenligninger av skyleverandører som inkluderer alle løsningene som tilbys. Disse er ofte korte med kun et avsnitt per leverandør eller en tabell med hvilke løsninger som tilbys. [2] er et eksempel på en slik sammenligning. Vi mener ikke at slike sammenligninger ikke har noe verdi, men ettersom de forsøker å dekke hele tilbudet til leverandørene mister en også innsikten i spesifikke tjenester man er interessert i. Når jeg skal hoste en web-applikasjon er det for eksempel lite interessant å vite at samme leverandør er gode på maskinlæring.

4. Produktoversikt

4.1 Produktets rolle i brukermiljøet



Bruker av systemet vil kun forholde seg til en nettside (frontend), som forklart i 3.3. Denne nettsiden vil ta seg av kommunikasjon med et backend-system som igjen håndterer kommunikasjon med eventuelle tredjepartssystemer. Et slikt tredjepartssystem kan for eksempel være en autentiseringsløsning.

4.2 Forutsetninger og avhengigheter

I dette prosjektet har vi ingen viktige forutsetninger eller avhengigheter. Vi har blitt stilt relativt fritt til hva vi ønsker å utvikle og hva vi ser på som relevant og nødvendig for å lage en god applikasjon for testing av skyløsninger. Vi kommer til å jobbe sammen med en designer hos Kantega som vil jobbe med den visuelle utformingen av applikasjonen. I denne fasen av produktutviklingen er allikevel mye på konseptstadiet slik at skissene i starten vil være relativt grove. Dermed vil vi bli stilt ganske fritt til å selv designe deler av applikasjonen.

5. Produktets funksjonelle egenskaper

Beskrivelse
Funksjonalitet for å logge inn som bruker. Det skal være mulig å logge inn med brukernavn og passord, samt redigere informasjon knyttet til en bruker.
Innlogging med sosiale tjenester. Ikke alle brukere ønsker å lage nye kontoer på hver side som brukes, men vil heller bruke sin Google-, Facebook-, Microsoft-konto overalt. Det skal være mulig å logge inn med minst en slik innloggingstilbyder.
Funksjonalitet for å opprette nye organisasjoner. Superbrukere, gjerne ansatt i start-up, må kunne opprette nye organisasjoner for nye kunder. De må også kunne administrere organisasjonen etterpå.

Funksjonalitet for å administrere brukere i en organisasjon. En administrator må kunne legge til og fjerne brukere, samt endre roller på eksisterende brukere.
Funksjonalitet for å velge brukere som har flere rettigheter enn andre brukere i en administrasjon. Disse brukerne skal kunne se administrasjonsfunksjonalitet.
For å illustrere en tjeneste som er relevant for systemet, skal det implementeres en telefon-katalog som inneholder telefonnummer som hver organisasjon har behov for i forskjellige situasjoner. Denne telefonkatalogen skal ha funksjonalitet for å legge til et telefonnummer i sin lokale liste, samt se numrene som er lagt inn av både organisasjonen og start-up'en selv.
Brukere skal kunne kommunisere innad i en organisasjon. Dette skal skje gjennom en «feed» på forsiden til en organisasjon. Her skal brukere kunne legge til innlegg og svare på andre sine innlegg.
Brukere skal kunne lage oppgaver med dato for å lettere kunne huske hva som må gjøres. Disse skal kunne lages fra innlegg i «feeden» som ber om at noen tar på seg en oppgave.
Oppgavene til en bruker skal kunne flyttes mellom dager slik at man selv kan forskyve når en ønsker at en oppgave skal være fullført om ønskelig.
Oppgaver i «feeden» skal kunne markeres som påtatt slik at andre kan se at noen vil gjøre den.
Oppgaver i «feeden» skal kunne markeres som fullført slik at andre kan se at den er gjennomført.
Systemet skal være lett å sette opp for brukertester, osv.

6. Ikke-funksjonelle egenskaper og andre krav

6.1 Web-applikasjon

- Informasjons utveksling, foregår gjennom REST-tjenester.
- Løsningen være i samsvar med WCAG 2.1 [3] og ha en god brukskvalitet, den skal være tilpasset målgruppen, lett og intuitiv å bruke.
- Løsningen skal ha god sikkerhet. Krav om å sjekke opp mot OWASP punkt 1 og 3 [4].
- Det skal tas i bruk byggesystemer for å gjøre det lett å kjøre systemet. Docker vil bli benyttet for å sikre likt miljø uavhengig av hvor koden kjøres og utvikles.
- Continous Integration – CI, for å sørge for at all kode kjører som det skal ved endringer. Det skal skrives integrasjon og unit-tester som vil kjøres ved hver pull-request.
- Continous Deployment – CD, for å sørge for at alle brukere bruker en oppdatert versjon.

- En plattformuavhengig nettleser med støtte for en nyere HTML-standard skal brukes som klient mot applikasjonen. Minste kravet er at det er mulig å ta i bruk Chromium (Chrome), Firefox, og Safari.
- Klient (web-applikasjon) skrives i Typescript med Next.js [5] som rammeverk.
- Server/mikrotjenester skrives i Kotlin med Quarkus [6] som rammeverk.

6.2 Forskning og analyse

- Under testing av skyløsninger vil det i all hovedsak benyttes Pulumi [7]. Pulumi er et verktøy som kan brukes til å sette opp instanser i skyen ved hjelp av kode. På den måten kan vi være sikker på at når vi setter opp samme instans flere ganger, så skjer det på akkurat samme måte hver gang.
- Skyleverandørene som skal sammenlignes er Google Cloud Platform (GCP), Microsoft Azure og Amazon Web Services (AWS).
- Hos hver skyleverandør vil vi se på deres løsninger innen de fem kategoriene: Virtuelle maskiner (VM), Kubernetes, Serverless containere, Platform as a Service (PaaS) og Function as a Service (FaaS). En oversikt over løsningene hos hver leverandør er laget i tabellen nedenfor. Oversikten er utarbeidet ved hjelp av leverandørenes egne oversikter over produkter [8] [9] [10].

Hva	Azure	AWS	Google Cloud
VM	Azure Virtual Machines	Amazon Elastic Compute Cloud (EC2)	Compute Engine
Kubernetes	Azure Kubernetes Service (AKS)	Elastic Kubernetes Service (EKS), Elastic Container Service (ECS)	Google Kubernetes Service (GKS), Knative
Serverless containers	Azure Container Instances	AWS Fargate, AWS Lambda, AWS App Runner	Cloud Run
Platform as a Service	Azure App Service, Web App for Containers	AWS Elastic Beanstalk	App Engine
Function as a Service	Azure Functions	AWS Lambda	Cloud Functions

- I sammenligningen skal vi se på flere parametere. Kostnad, fleksibilitet, skalerbarhet, resilens, robusthet, mulighet til å flytte mellom leverandører og DX («developer experience») er hovedpunktene her. I hovedrapporten vil vi utdype nærmere hvordan hver kategori skal sammenlignes slik at det blir en rettferdig sammenligning.
- Konseptene som brukes skal forklares. Det innebærer sky, VM, Kubernetes, Containere, PaaS, FaaS, Iaas, IaC og eventuelle andre relevante begreper.

7. Referanser

[1] B. M. S. Norway, «Statcounter GlobalStats,» [Internett]. Available: https://gs.statcounter.com/browser-market-share/all/norway. [Funnet 16 02 2022].

[2] E. Kaleynik, «Hackernoon,» 20 06 2021. [Internett]. Available: https://hackernoon.com/top-cloud-platform-comparison-2021-edition-t7n35xd. [Funnet 04 03 2022].

[3] World Wide Web Consortium (W3C), «Web Content Accessibility Guidelines (WCAG) 2.1,» 05 06 2018. [Internett]. Available: https://www.w3.org/TR/WCAG21/. [Funnet 04 03 2022].

[4] OWASP, «OWASP,» [Internett]. Available: https://owasp.org/www-project-top-ten/#. [Funnet 04 03 2022].

[5] Next.js, «Next.js,» [Internett]. Available: https://nextjs.org/. [Funnet 04 03 2022].

[6] Quarkus, «Quarkus,» [Internett]. Available: https://quarkus.io/. [Funnet 04 03 2022].

[7] Pulumi, «Pulumi,» [Internett]. Available: https://www.pulumi.com/. [Funnet 04 03 2022].

[8] Google Cloud, «Compare AWS and Azure services to Google Cloud,» 16 02 2022. [Internett]. Available: https://cloud.google.com/free/docs/aws-azure-gcp-service-comparison. [Funnet 04 03 2022].

[9] Microsoft Azure, «AWS to Azure services comparison,» 24 01 2022. [Internett]. Available: https://docs.microsoft.com/en-us/azure/architecture/aws-professional/services. [Funnet 04 03 2022].

[10] Microsoft Azure, «Google Cloud to Azure services comparison,» 04 01 2022. [Internett]. Available: https://docs.microsoft.com/en-us/azure/architecture/gcp-professional/services. [Funnet 04 03 2022].

Vedlegg B

Kravdokumentasjon

Gruppe 60

Bacheloroppgave for Kantega
Kravdokumentasjon

Versjon 1.0

Revisjonshistorie

Dato	Versjon	Beskrivelse	Forfatter
18/02/2022	0.1	Første utfylling av punkter	Hermann Elton, Olaf Rosendahl
04/03/2022	0.2	Videre revidering av punkter	Olaf Rosendahl
07/04/2022	0.3	La til flere punkter på stories og fylte ut underpunkter	Hermann Elton
05/05/2022	0.4	Første utkast ferdigstilt	Hermann Elton, Olaf Rosendahl
16/05/2022	1.0	Ferdigstilt og klar til innlevering	Hermann Elton, Olaf Rosendahl

Innholdsfortegnelse

1. Introduksjon 4
2. User Stories 4
3. Domenemodell 6
4. Prototyper 6
5. Referanser 6

1. Introduksjon

Dette dokumentet er skrevet i forbindelse med systemutviklingsprosjektet knyttet til vår bachelor-oppgave om skyløsninger. Dette dokumentet skal dokumentere hvorfor og hvordan systemet som ble utviklet er bygd opp, og kravene som ble satt for dette systemet. Kravene viser hvordan applikasjonen er et reelt produkt og dermed relevant for testing av skyløsningene.

2. User Stories

Som bruker ønsker jeg å kunne logge inn på web-applikasjonen, med brukernavn og passord. Sånn at brukeren kan ta i bruk applikasjonen som seg selv.

- Jeg kan logge inn med Google-konto eller opprette en bruker hos Auth0
- Jeg kan opprette en bruker på systemet hvis jeg ikke finnes fra før av ved å oppgi fornavn og etternavn
- Jeg kan ikke opprette en bruker uten å bruke Google- eller Auth0-konto

Som bruker ønsker jeg å kunne se at jeg er innlogget, slik at jeg kan åpne min profil

- Jeg kan logge ut herfra
- Jeg kan se mitt navn i profilen slik at jeg vet at det er jeg som er logget inn

Som administrator så ønsker jeg å legge inn telefonnumre for en organisasjon. Da kan alle i organisasjonen bruke samme katalog med telefonnumre.

- Jeg kan legge til telefonnummer i den valgte organisasjonen

Som bruker ønsker jeg å se en liste over nyttige telefonnumre, slik at jeg kan ringe dit jeg ønsker

- Jeg kan søke på telefonnumre
- Jeg kan filtrere telefonnumre basert på kategori

Som bruker ønsker jeg å opprette kategorier for telefonnumre til telefonlista, slik at jeg lettere kan kategorisere telefonnumre.

- Jeg kan opprette en kategori, ved å gi den et navn
- Jeg kan endre denne kategorien etter at jeg har laget den

Som bruker så ønsker jeg å legge til mine egne telefonnumre i telefonkatalogen

- Jeg kan opprette et telefonnummer ved å gi det et navn og et nummer, samt hvilken kategori det hører til
- Jeg kan ikke opprette et telefonnummer uten å knytte det til en kategori

Som administrator i en organisasjon vil jeg ha tilgang til å administrere organisasjonen min, slik at de nødvendige medlemmene er lagt til.

- Jeg kan legge til et medlem med eposten til det nye medlemmet
- Jeg kan velge hvilken rolle i organisasjonen medlemmene har
- Jeg kan fjerne brukere fra organisasjonen

Som bruker med medlemskap i flere organisasjoner ønsker jeg å velge hvilken organisasjon jeg vil se innholdet til, slik at jeg kun ser relevant innhold

- Jeg kan velge organisasjon ved innlogging
- Jeg kan raskt endre valgt organisasjonen gjennom profilen
- Jeg kan se en oversikt over alle organisasjonene jeg er medlem av

Som bruker ønsker jeg å kommunisere med resten av organisasjonen jeg jobber for gjennom en intern kommunikasjonskanal. Her skal jeg også kunne legge ut oppgaver som må gjøres.

- Jeg kan se alle innlegg, hvem som har lagt dem ut og om det er en oppgave
- Jeg kan legge ut nye innlegg, enten med eller uten oppgave
- Jeg kan spesifisere hvem i organisasjonen som skal se innlegget. Medlemmer, administratorer eller alle

Som bruker ønsker jeg en huskeliste med gjøremål med tidspunkt for når de bør være fullført.

- Jeg kan opprette gjøremål som blir satt til å skulle gjøres i dag
- Jeg kan flytte gjøremål til å skulle gjøres i dag, i morgen eller senere
- Jeg kan markere gjøremål som fullført
- Fullførte gjøremål med frist i fortiden skal skjules og vises under «Vis fullførte oppgaver»
- Når jeg oppretter et gjøremål, kan jeg velge bakgrunnsfarge for å skille dem fra hverandre
- Jeg kan endre teksten i et gjøremål

Som bruker så ønsker jeg å påta meg oppgaver i kommunikasjonskanalen. Denne skal så legges til i huskelisten.

- Jeg kan påta meg oppgaver som ikke allerede er tatt
- Jeg kan markere en påtatt oppgave som fullført
- Når jeg tar på meg en oppgave så blir gjøremålet lagt til i dag i huskelisten
- Hvis jeg merker et gjøremål fra kommunikasjonskanalen som fullført blir også innlegget merket som fullført

Som bruker ønsker jeg å ha en liste over lenker til eksterne systemer som gjør meg mer effektiv i hverdagen, ettersom jeg får lett tilgang til nyttige verktøy.

- Jeg kan se en liste over lenker til eksterne systemer
- Listen skal være forskjellig basert på hvilken rolle jeg har i organisasjonen

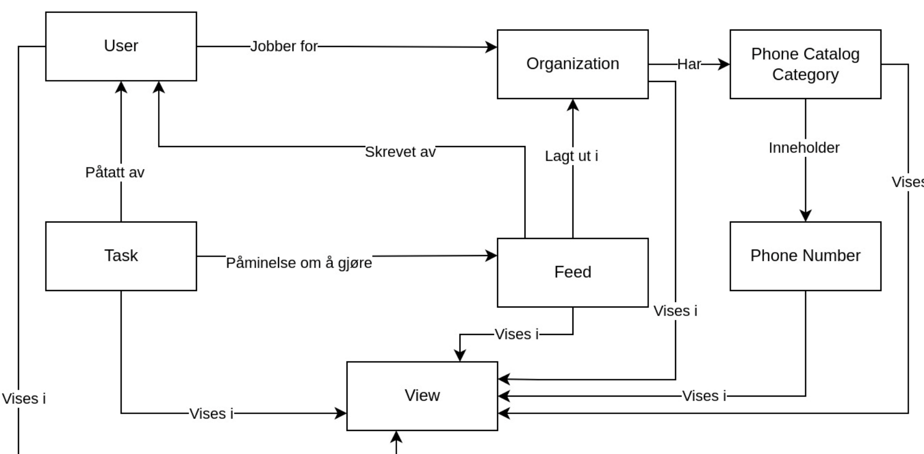
Som produkteier ønsker jeg at nettsiden viser kommende funksjonalitet, slik at jeg lettere kan vise fram produktet til eventuelle investorer.

- Jeg kan ses flere sider der innholdet kun er et bilde av skisse i Figma. Tydelig markert at det kun er en skisse
- Jeg kan se en boks som popper opp når jeg klikker på en knapp der funksjonaliteten ikke er implementert

Som svaksynt bruker ønsker jeg at nettsiden skal være universelt utformet slik at jeg kan bruke en skjermleser til å navigere rundt på siden.

- Nettsiden skal ta i bruk semantisk korrekte HTML-tagger som nav, main, ul, li, button, a, osv. slik at skjermleseren forstår hva som er på siden.
- Nettsiden skal bruke aria-attributter der innholdet eller ikke kan forstås automatisk. For eksempel aria-current på valgt menyelement og aria-label på ikon-knapper.
- Nettsiden skal ha gode kontraster slik at dem som kan se noe faktisk har muligheten til det.
- Nettsiden skal oppfylle kravene i WCAG 2.1 [1]
- Løsningen skal ha god sikkerhet. Krav om å sjekke opp mot OWASP punkt 1 og 3 [2]
- Ifølge «Statcounter GlobalStats» har Chrome, Safari, Firefox og Edge en total markedsandel på 95.3% i Norge [3]. Nettsiden skal som minimum støtte alle nettleserne.

3. Domenemodell



Figur 1: Domenemodell

4. Prototyper

Oppdragsgiver, Kantega, har en egen designer som jobber med prosjektet. Vi har dermed ikke utviklet prototyper eller wireframes selv ettersom vi har kunnet bruke det designeren lager.

5. Referanser

[1] World Wide Web Consortium (W3C), «Web Content Accessibility Guidelines (WCAG) 2.1,» 05 06 2018. [Internett]. Available: https://www.w3.org/TR/WCAG21/. [Funnet 04 03 2022].

[2] OWASP, «OWASP,» [Internett]. Available: https://owasp.org/www-project-top-ten/#. [Funnet 04 03 2022].

[3] B. M. S. Norway, «Statcounter GlobalStats,» [Internett]. Available: https://gs.statcounter.com/browser-market-share/all/norway. [Funnet 16 02 2022].

Vedlegg C

Systemdokumentasjon

Gruppe 60

Bacheloroppgave for Kantega
Systemdokumentasjon

Versjon 1.0

Revisjonshistorie

Dato	Versjon	Beskrivelse	Forfatter
07/04/2022	0.1	Skrevet en del på alle punkter, første utkast	Hermann Elton
05/05/2022	0.2	Oppdatert flere figurer og utdypet tekst. Lagt til mer om sikkerhet	Hermann Elton, Olaf Rosendahl
18/05/2022	1.0	Ferdigstilt og klar for levering	Hermann Elton, Olaf Rosendahl

Innholdsfortegnelse

1. Introduksjon 4

2. Arkitektur 4

Prosjektstruktur 5
2.1 Tjener 5
2.2 Klient 5

3. Klassediagram 6

4. Databasemodell 6

5. Sikkerhet 7

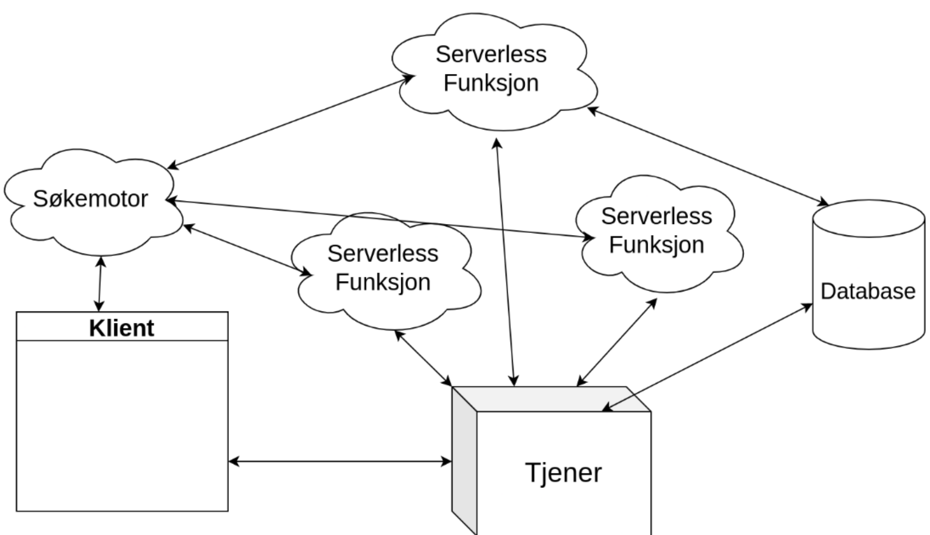
6. Installasjon og kjøring 8

7. Kontinuerlig integrasjon og testing 9

1. Introduksjon

Dette dokumentet er skrevet for å dokumentere systemutviklingsarbeidet i bacheloroppgaven til gruppe 60. Systemet er skrevet for en start-up som er kunde av vår oppdragsgiver Kantega. Enkelte deler av innholdet i dette dokumentet er anonymisert for å ikke publisere forretningsideen til start-up'en. Dette har ikke noe påvirkning på totalinnholdet ettersom den utviklede applikasjonen i dette prosjektet kun skal brukes som en reell applikasjon for testing av skyløsninger.

2. Arkitektur



Figur 1: Arkitektur-oversikt

Systemet inneholder en klient og en tjener. Tjener har hovedansvaret for å gi klienten informasjon om de forskjellige tjenestene på siden. Informasjonen er lagret i en relasjonsdatabase. Det er også tre serverløse funksjoner i applikasjonen som indekserer, oppdaterer og sletter innlegg i søkemotoren (Algolia). Indekseringsfunksjonen henter informasjon om relevant innhold i databasen og kopierer dette til Algolia. Klienten kan deretter snakke direkte med søkemotoren for å få et raskt søk med gode søkeresultater.

Figur 2: Laginndeling i tjener

Tjeneren er delt inn i tre lag: *view/controller*, *service* og *model*. Viewet tar imot forespørsler og delegerer dem til riktig endepunkt i applikasjonen. Endepunktet tar imot forespørselen og benytter seg av et eller flere servicer for å hente ut nødvendig informasjon fra databasen. Dette omgjøres til modeller som returneres til viewet og deretter til avsender av forespørselen.

Prosjektstruktur

2.1 Tjener

Tjenestene på tjeneren er delt inn i apper som ligger i hver sin mappe. Dette fører til at logikk som hører sammen også ligger sammen. Logikken rundt brukerhåndtering ligger for eksempel i *user*. Organisasjon har medlemmer (brukere), men ettersom medlemmer tilhører en organisasjon ligger medlemmer (*OrganizationUser*) i organisasjons-mappen.

exceptions
feed
function
organization
phonecatalog
user
utils

Figur 3: Tjenester i tjeneren

feed
dto
models
permissions
services
utils
views

Figur 4: Mappe-struktur i apper

Hver app er delt inn i mapper med lik struktur. *Dto* inneholder *data transfer objects* som bestemmer hvordan data som bruker mottar eller sender skal se ut. *Models*-mappen har klassene som modellerer database-entiteter/tabeller. I *permissions*-mappen logikken som styrer tilgangshåndteringen for innholdet i appen. Logikk for tilgangshåndtering i én app kan benyttes av andre apper også. *Services*-mappen inneholder logikk for å hente og endre informasjon i modellene. *Utils* inneholder annen funksjonalitet som er relevant for appen. Til slutt inneholder *views* alle endepunkter i appen. Disse endepunktene tar imot forespørsler.

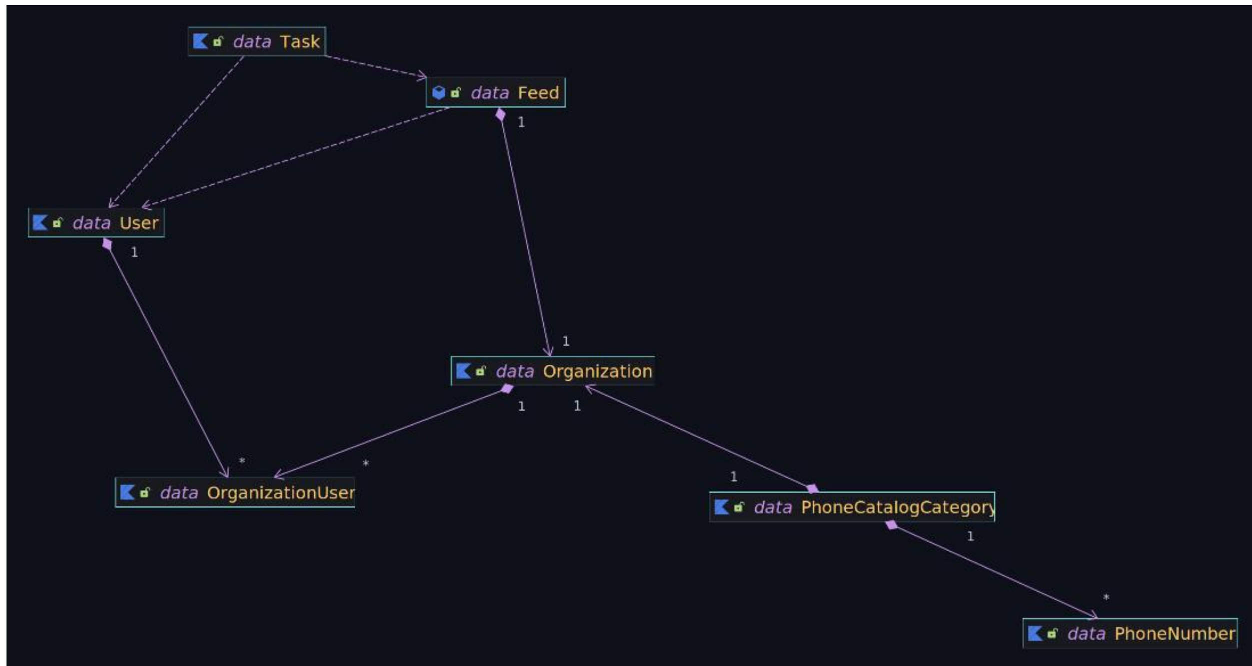
2.2 Klient

Api-mappen inneholder logikk for å snakke med tjeneren, dette innebærer én fil pr. tjeneste hos tjeneren med en liste over funksjoner som kan brukes til CRUD-operasjoner mot tjenesten. *Assets* inneholder filer som ikke er kode, men for eksempel bilder og ikoner. *Components* har gjenbrukbare og selvstendige komponenter som kan brukes overalt i koden. *Hooks*-mappen inneholder gjenbrukbare React-hooks. Det er både hooks som håndterer data fra tjeneren, samt andre nyttige hooks. *Pages* er en Next.js-spesifikk mappe som definerer URL'er i applikasjonen. Her kan en også definere data som skal hentes når en forespørsel prosesseres på serveren. Filene her importerer innholdet fra *screens*. Screens-mappen definerer selve sidene (hva som skal vises frem) til klienten. *Theme* inneholder kode som definerer utseende på siden og har globale variabler som gjør det enkelt å lage et gjennomgående utseende. Her finnes det verdier for farger og størrelser blant annet. Chakra-UI brukes som komponent-bibliotek og styling-løsning. *Types* inneholder Typescript-typer. Disse viser utseende på informasjonen som kommer fra tjeneren og samsvarer med dto-klassene i tjener-koden. Resten av filene er hjelpe-filer for slikt som konstanter, nyttige funksjoner og håndtering av React-kontekst på siden.

api
assets
components
hooks
pages
screens
theme
types
TS constant.ts
TS serverSideUtils.ts
TS urls.ts
TS useSession.ts
TS utils.ts

Figur 5: Filstruktur i klient

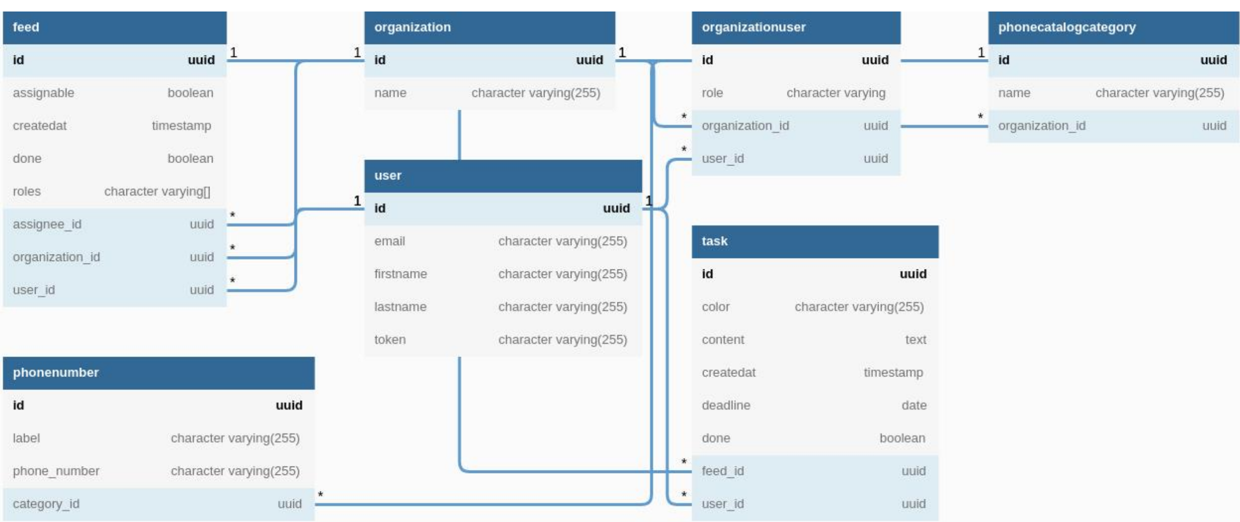
3. Klassediagram



Figur 6: Klassediagram

Dette er en oversikt av domeneklassene i systemet. For å ikke gjøre diagrammet for komplisert er kun database-entiteter inkludert. Hver klasse i dette diagrammet har sin egen service og et view. I tillegg eksisterer det klasser som håndterer tilgangsstyring. Dette kan leses mer om under sikkerhet. Det er også skrevet integrasjonstester til hver eneste klasse som sees i diagrammet.

4. Databasemodell



Figur 7: Databasemodell

Databasemodellen korresponderer med klassediagrammet, men med felter og fremmednøkler i tillegg.

5. Sikkerhet

Applikasjonen tar i bruk brukerautentiseringstjenesten Auth0. Dette er gjort for å slippe å håndtere brukerautentisering selv, og mulige problemer som kan oppstå ved å handtere slik logikk selv. Auth0 benytter JWT-tokens for å identifisere brukere. Dette token'et inneholder informasjon som e-post, fornavn, etternavn, og en unik ID. Det er denne unike ID-en som vi bruker for å koble sammen vår brukerinformasjon med Auth0 sin brukerinformasjon. Denne koblingen lar oss senere hente ut informasjon om brukeren i tilgangsstyringssjekker.

Når det kommer SQL-injections så bruker vi et bibliotek som håndterer databasespørringer for oss. Dette biblioteket fjerner muligheten for SQL-injections ved at brukerinput aldri blir lagt til i en spørring på en slik måte at det kan interagere med eller endre selve spørringen.

Next.js/React håndterer Cross-site Scripting (XSS) ved at all data som kommer fra tjener eller bruker vises frem som tekst-strenger og ikke som HTML-elementer. Dermed vil ikke mulig farlig brukerinput bli kjørt eller vist på andre måter enn som tekst.

Applikasjonen tar i bruk Kotlin-annotasjoner for tilgangshåndtering. Dette gjøres ved å opprette en annotasjon og et annotasjonsfilter. Disse to sjekker da tilgangen til en bruker for de forskjellige appene.

```
@NameBinding
@Retention(AnnotationRetention.RUNTIME)
annotation class UserOwnsFeedItem(val feedIdKey: String = "id")
```

Figur 8: Opprettelse av annotasjon

Her ser du en annotasjon. Her er *feedIdKey* navnet på url-variabelen som er ID-en til en oppføring i kommunikasjonskanalen (feed).

```
@RequestScoped
@Provider
@UserOwnsFeedItem
class UserOwnsFeedItemFilter : ContainerRequestFilter {

    @Inject
    private lateinit var feedService: FeedService

    @Context
    private lateinit var info: ResourceInfo

    @Context
    private lateinit var ctx: SecurityContext

    override fun filter(context: ContainerRequestContext) {
        val idKey = info.resourceMethod.getAnnotation(UserOwnsFeedItem::class.java).feedIdKey
        val feedId = context.uriInfo.pathParameters[idKey]?.get(0)
        val token = TokenUtils.getToken(ctx)
        if (feedId != null && feedService.checkIfUserOwnsFeed(token, UUID.fromString(feedId))) return
        throw ForbiddenException("User does not own this feed item")
    }
}
```

Figur 9: Annotasjonsfilter for en gitt annotasjon

Figur 9 viser en filter-klasse som sjekker om en bruker eier et FeedItem ved at man henter ut ID-en og sjekker om brukeren laget det ved hjelp av FeedService. Brukeren blir hentet ut gjennom den unike Auth0 ID-en som ligger i JWT-token'et som sendes med forespørselen.

```
@PUT
@Path("/{feed_id}")
@UserInOrg("org_id")
@UserOwnsFeedItem("feed_id")
fun put(feed_id: UUID, data: FeedUpdateDto): RestResponse<Any> {
```

Figur 10: Bruk av annotasjon for tilgangsstyring

En kan da bruke annotasjonen som vist i figur 10 og filteret vil automatisk kjøres før hver forespørsel i view'et.

6. Installasjon og kjøring

1. Clone prosjektet med Git eller last ned koden som zip-fil om du ikke behøver Git-integrasjon.
2. Klient
   a. yarn er nødvendig for å installere nødvendige pakker og kjøre applikasjonen.
   b. Kjør «*yarn*» i frontend-mappen for å installere avhengigheter og kjør deretter «*yarn dev*» for å kjøre koden i utviklingsmodus
   c. Kjør «*yarn build*» etterfulgt av «*yarn start*» for å kjøre i produksjonsmodus.
   d. «*docker-compose up frontend*» og «*make start-frontend*» kan også benyttes for å kjøre applikasjonen i produksjonsmodus
3. Tjener
   a. Maven eller Docker er nødvendig. For å kjøre med Maven må variablene i *quarkus.datasource.jdbc.url*-variabelen i *application.properties* endres.
   b. Med Maven: kjør «*mvn compile quarkus:dev*» for å installere avhengigheter og kjøre prosjektet i utviklingsmodus. For å kompilere til produksjon kjøres «*mvn -B clean -q package -DskipTests*». En *quarkus-run.jar*-fil blir da laget i *target*. Kjør denne filen med flaggene: «-*Dquarkus.http.host=0.0.0.0* -Dquarkus-profile=prod»
   c. Med Docker: «*docker-compose up backend db pgadmin*» eller «*make start-backend*». Da startes backend, en PostgreSQL-database og pgAdmin.
4. Hele applikasjonen med både klient og tjener kan kjøres med Docker. Du kan enten kjøre «*docker-compose up*» eller «*make start*».

7. Kontinuerlig integrasjon og testing

For tjenerdelen av applikasjonen er det skrevet integrasjonstester for alle view. Dette sikrer at endringer i koden ikke ødelegger eksisterende funksjonalitet. Her testes endepunkter, funksjonaliteten som endepunktene tilbyr og at data som sendes til eller fra endepunktet har korrekt format.

Det benyttes GitHub Actions for CI i både klient og tjener. I klienten sjekkes det om koden er formatert i henhold til de definerte reglene, samt at bygging av applikasjonen er vellykket. CI-konfigurasjonen kan sees i figur 12.

I tjener-koden kjøres integrasjonstestene og sjekk av kode-formatering. CI-konfigurasjon kan sees i figur 13

For å kjøre tester så trenger du enten Maven eller Docker. Ved bruk av Maven må en PostgreSQL-database legges til for bruk under testing. Du kan da kjøre «*mvn test*» for å kjøre testene. Med Docker kan du enten bruke «*docker-compose -f docker-compose.test.yml run --rm backend mvn test*» eller «*make test*».

```yaml
name: Check lint and build - Frontend
on:
  pull_request:
    paths:
      - 'frontend/**'
defaults:
  run:
    working-directory: frontend
jobs:
  build:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        node-version: [14.x]
    steps:
      - uses: actions/checkout@v2
      - name: Use Node.js ${{ matrix.node-version }}
        uses: actions/setup-node@v2
        with:
          node-version: ${{ matrix.node-version }}
      - name: Get yarn cache directory path
        id: yarn-cache-dir-path
        run: echo "::set-output name=dir::$(yarn cache dir)"
      - uses: actions/cache@v2
        id: yarn-cache
        with:
          path: ${{ steps.yarn-cache-dir-path.outputs.dir }}
          key: ${{ runner.os }}-yarn-${{ hashFiles('**/yarn.lock') }}
          restore-keys: |
            ${{ runner.os }}-yarn-
      - name: Install dependencies
        run: yarn
      - name: Lint
        run: yarn lint-check
      - name: Build
        run: yarn build
```

Figur 11: CI-konfigurasjon for klient

```yaml
name: Check lint and test - Backend
env:
  DOCKER_BUILDKIT: 1
  COMPOSE_DOCKER_CLI_BUILD: 1
on:
  pull_request:
    paths:
      - 'backend/**'
defaults:
  run:
    working-directory: backend
jobs:
  lint:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Set up JDK 11
        uses: actions/setup-java@v2
        with:
          java-version: '11'
          distribution: 'adopt'
      - uses: actions/cache@v1
        with:
          path: ~/.m2/repository
          key: ${{ runner.os }}-maven-${{ hashFiles('**/pom.xml') }}
          restore-keys: |
            ${{ runner.os }}-maven-
      - name: Lint with Maven
        run: mvn antrun:run@klint
  test:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout Code Repository
        uses: actions/checkout@v2
      - name: Run Tests
        run:  make test
```

Figur 12: CI-konfigurasjon for tjener

Vedlegg D

Prosjekthåndbok

NTNU
Kunnskap for en bedre verden