# Sammendrag

Denne bacheloroppgaven består i å designe, implementere og evaluere en eventbasert arkitektur for HelseCERT for å knytte alle deres distribuerte systemer sammen gjennom ett sentralt grensesnitt. Disse systemene er i dag tungvinte å operere fordi det kreves manuelle interaksjoner mot hvert enkelt system. Eksempler på slike systemer kan være brukerkataloger, mailservere eller databaser. Vårt system gjør det mulig for en ansatt i HelseCERT å interagere med flere av disse systemene gjennom én enkelt handling. Løsningen vår er realisert gjennom blant annet meldingskøer, bruk av logging database og flere egenkonstruerte komponenter skrevet i Python. Vi har benyttet oss av en iterativ utviklingsprosess hvor vi kontinuerlig har forbedret systemet vårt med ny funksjonalitet. Evalueringen av systemet vårt har også vært en sentral del av rapporten. Der har vi vurdert fordeler og ulemper med en eventbasert tilnærming til den gitte problemstillingen. Arbeidet med denne bacheloroppgaven har ført fram til et produkt som er en klar forbedring sammenlignet med dagens situasjon i HelseCERT. Implementeringen av vårt produkt vil altså gjøre arbeidshverdagen til de ansatte enklere, tryggere og mer effektiv.

# Abstract

This bachelor thesis consists of designing, implementing and evaluating an event-driven architecture for HelseCERT to connect all their distributed systems together through one central interface. These systems are currently cumbersome to operate because it requires manual interactions with each system. Examples of such systems can be user identity management systems, mail servers or databases. Our system makes it possible for an employee of HelseCERT to interact with several of these systems through one single action. Our solution is realized through, among other things, message queues, the use of a logging database and several self-designed components written in Python. We have used an iterative development process where we have continuously improved our system with new functionality. The evaluation of our system has also been a key part of the report. The evaluation has assessed the advantages and disadvantages of an event-driven approach to the given problem. The work with this bachelor thesis has led to a product that is a clear improvement compared to the current situation at HelseCERT. The implementation of our product will thus make the everyday work of the employees easier, safer and more efficient.

# Forord

Denne bacheloroppgaven er gjennomført for Institutt for datateknologi og informatikk ved Norges teknisk-naturvitenskapelige universitet (NTNU) i Trondheim på oppdrag fra vår oppgavestiller HelseCERT. Oppgaven er avsluttende i forbindelse med vårt studie, Digital infrastruktur og cybersikkerhet, våren 2022.

Vi hadde tidlig et ønske om å skrive en oppgave som omhandlet utvikling og systemarkitektur, fortrinnsvis hos en sikkerhetsorientert bedrift. Dette bunner i en felles interesse blant studentene innen disse temaene. Da vi fikk denne oppgaven fra HelseCERT ble valget om bacheloroppgave derfor relativt enkelt for oss.

Vi ønsker å takke de ansatte i HelseCERT, og spesielt vår kontaktperson Lars Kulseng, for mye god hjelp og innspill underveis. Vi har blitt inkludert i deres arbeidsmiljø og fått masse ny kunnskap gjennom samtaler og diskusjoner på arbeidsplassen deres.

Vi vil også takke vår kullkoordinator Joakim Klemets med hjelp til utforming av sluttrapporten og andre gode tips i forbindelse med denne.

# Innholdsfortegnelse

# Figurer

# Akronymer og forkortelser

API - Application programming interface
HTTP - HyperText Transfer Protocol
HTML - HyperText Markup Language
UI - User Interface
GUI - Graphical User Interface
CLI - Command Line Interface
VM - Virtual machine
YAML - YAML Ain’t Markup Language
JSON - JavaScript Object Notation
SQL - Structured Query Language
PEP - Python Enhancement Proposal
CSV - Comma Separated Values
UUID - Universally Unique Identifier
RPC - Remote Procedure Call
UML – Unified Modeling Language
NTNU - Norges Teknisk-Naturvitenskapelige Universitet

# 1. Introduksjon

## 1.1 Problemstilling

Mange bedrifter er avhengige av flere selvstendige og eksterne systemer i sin daglige drift. Det være seg kundedatabaser, bestillingssystem, bookingsystem og lignende. I enkelte tilfeller vil man trenge å interagere med mer enn ett system for å utføre en handling. Denne utfordringen oppstår nok spesielt ofte i tilfeller der oppgavene ikke direkte involverer kunder, ettersom terskelen kanskje blir høyere for å legge ressurser i automatisering internt. Et eksempel på en handling som involverer flere adskilte systemer kan være at en nyansatt i en bedrift vil trenge å få opprettet en bruker flere steder, få tildelt en e-post-bruker, bli lagt til i riktige mailinglister og få de korrekte tilgangene internt.

HelseCERT ønsker å automatisere og integrere sine systemer gjennom et system som gjør det lettere og tryggere for eksisterende, og kanskje spesielt nye, ansatte å bruke systemene sine på en mer effektiv måte. Dette innebærer å gå over til en løsning der kundedata kan leses, opprettes, redigeres og slettes fra ett sentralt grensesnitt. Det skal også kunne utføres visse handlinger som blir trigget av at én eller flere andre handlinger allerede har blitt gjennomført. Dette er eksempler på operasjoner som krever at enkelte ting skjer i en bestemt rekkefølge. En løsning som takler slike avhengigheter vil derfor bli en sentral del av systemet.

HelseCERT ønsker at et slikt system skal realiseres gjennom et eventbasert integrasjonssystem hvor forskjellige tjenester lytter på eventer gjennom en form for meldingskø-programvare. HelseCERT ønsker et system som er fleksibelt og utvidbart i form av at en enkelt skal kunne knytte til flere eksterne systemer i fremtiden. Dette systemet skal gjøre at sluttbrukere bare trenger å forholde seg til ett sentralt system, mens de i realiteten utfører endringer på opptil flere forskjellige bakenforliggende, eksterne systemer. Med andre ord ønsker HelseCERT et system med høy grad av interoperabilitet. Med det menes det at systemet enkelt og sømløst kan interagere med mange forskjellige, uavhengige systemer.

I tillegg til dette ønsker HelseCERT at vi drøfter og konkluderer rundt integreringen og nytten av et eventbasert system. Oppgaven består altså i å designe, implementere, og evaluere en eventbasert tilnærming til et slikt system.

## 1.2 Bakgrunn

Denne bacheloroppgaven er utgitt av HelseCERT. HelseCERT ble opprettet i 2011 og er helse- og omsorgssektorens nasjonale senter for cybersikkerhet. HelseCERT er en del av Norsk Helsenett SF, hvor deres oppgave er å øke helsesektorens evne til å oppdage, forebygge og håndtere dataangrep. De jobber med monitorering og rådgivning i forbindelse med cybersikkerhet hos en rekke forskjellige kunder og partnere tilknyttet helse-Norge som kommuner, sykehus og andre helseforetak. [1]

Per i dag er mye av deres informasjon og kundedata spredt ut i flere forskjellige interne systemer, og er derfor krevende å jobbe med. En enkel endring hos en kunde, slik som å for eksempel legge til en ny bruker, vil kunne kreve flere manuelle operasjoner mot forskjellige systemer for de ansatte i HelseCERT. Disse manuelle operasjonene er ofte tungvinte og krever ofte dyp forståelse av de bakenforliggende systemene for å kunne utføres. Dette medfører at sjansen for brukerfeil er stor, og det går også med mye unødvendig tid til dette. I tillegg øker kompleksiteten i takt med antall systemer en har å jobbe med ettersom en manuelt må interagere med hvert enkelt. Denne problematikken er illustrert i Figur 1, som viser frustrasjonen til en person som er nødt til å forholde seg til mange forskjellige systemer.

*Figur 1: Problematikken med å forholde seg til mange forskjellige systemer. [2]*

## 1.3 Avgrensing av oppgaven

For å avgrense oppgaven har vi sammen med HelseCERT kommet frem til deler som vi kommer til å prioritere og legge mer vekt på, og andre deler vi vil nedprioritere i prosjektet. Disse begrensningene bunner i hva som ikke er et behov for HelseCERT, slik at man i størst grad har kunnet jobbe mer med det som er av størst verdi for både dem og oss. Vi har gjennom møter og samtaler med HelseCERT valgt å avgrense oppgaven til å ha mest fokus på selve arkitekturen i det eventbaserte systemet.

Vi har følgelig nedprioritert å bruke masse tid på et avansert GUI, hovedsakelig av to grunner. Grunn nummer én er at et GUI i seg selv ikke har noe direkte å gjøre med tematikken i oppgaven vår. Vår oppgave dreier seg først og fremst om et eventbasert integrasjonssystem, og GUI-et i denne oppgaven skal kun vise og sende data mot et

API, og tar ikke del i selve systemarkitekturen som sådan. Grunn nummer to er at HelseCERT allerede har et internt GUI med alle deres eksisterende løsninger, som da også vår løsning skal implementeres inn i. Dette følger et eget rammeverk, struktur og stilark som gjør at store deler av vårt GUI uansett måtte blitt skrevet om av HelseCERT for å kunne bli integrert inn.

Videre har vi brukt Microsoft Azure som utviklingsplattform etter ønske av HelseCERT. Her har vi satt opp flere VM-er på samme virtuelle nettverk. Dette er kun av utviklingshensyn da HelseCERT til vanlig hoster sine tjenester on-premises. Dette er infrastruktur vi ikke har hatt tilgang til fordi dette er i produksjon og skal ikke direkte arbeides med av oss. Følgelig er vår bakenforliggende infrastruktur i skyen irrelevant og ikke en del av denne oppgaven.

## 1.4 Målgruppe for rapporten

I sammenheng med avsluttende bacheloroppgave for avgang av bachelor i Digital infrastruktur og cybersikkerhet vil denne rapporten hovedsakelig ha HelseCERT og NTNU som målgruppe. Men vi håper også at andre aktører som selv vurderer å benytte seg av et lignende system kan dra mye nytte av vår fordypning i eventbaserte integrasjonssystemer, og vi vil derfor også anse dem som en del av vår målgruppe.

## 1.5 Arbeid og rollefordeling

Prosjektgruppen består av Petter Lauvrak, Jacob Theisen og Wilhelm Bjoland, som alle studerer sammen på siste året av bacheloren i Digital infrastruktur og cybersikkerhet ved NTNU i Trondheim. Alle deltakere på prosjektet har god kjennskap til hverandre gjennom studiet, og har vært på flere gruppeprosjekter tidligere gjennom studiet. I tillegg bor alle deltakerne sammen, dette har gjort det enkelt å jobbe sammen som gruppe og holde god flyt i kommunikasjon gjennom hele prosjektperioden.

I prosjektplanen ble det definert en rollefordeling for prosjektet. Denne rollefordelingen fungerer som hovedansvarsområder for de større delene av prosjektet. Ansvarsområdene som ble definert var UI og API, mellomvaren (unntatt API), backend og sluttrapport, hvor Petter er ansvarlig for UI og API, Jacob for mellomvaren og Wilhelm for backend og sluttrapporten. Gjennom hele arbeidsprosessen har det vært stort fokus på å jobbe sammen på alle disse hovedområdene slik at alle får en dyp forståelse av hele systemet.

Oppdragsgiver for denne bacheloroppgaven er HelseCERT, sikkerhetsavdelingen hos Norsk helsenett, og er representert av Lars Kulseng og Gunnar A. Johansen. Veiledere for prosjektgruppen er Helge Hafting og Joakim Klemets som begge er ansatt ved fakultet for informasjonsteknologi og elektronikk ved NTNU Trondheim.

Gjennom arbeidet med bacheloren har vi fått faste plasser på kontoret til HelseCERT som har gjort at vi har hatt god kommunikasjon og jevnlige møter med våre veiledere hos bedriften gjennom hele semesteret.

## 1.6 Rapportstruktur

Rapporten starter med kapittel 1. Introduksjon, som beskriver problemstillingen vår og bakgrunnen for oppgaven. Vi avgrenser så oppgaven og tydeliggjør hvilke temaer som ikke er relevante i denne sammenhengen. Videre følger målgruppen for rapporten og hvilken bakgrunn vi i prosjektgruppen har for å gjennomføre denne bacheloren. Sist kommer forklaring av strukturen i rapporten med rapportstruktur.

Videre vil vi i kapittel 2. Teori presentere den teknologien som blir brukt i systemet vårt. Her introduserer vi de mest sentrale teknologiene vi har brukt gjennom bacheloroppgaven. Vi vil også skrive mer generelt om hva et eventbasert system går ut på. Vi introduserer også relevante fagord og -begreper som vi bruker for å beskrive og omtale deler og komponenter i systemet.

Kapittel 3. Metode, handler om hvordan vi har jobbet med denne oppgaven i løpet av dette semesteret. Her går vi over hvordan vi brukte starten av semesteret på å opparbeide oss et kunnskapsgrunnlag som skulle hjelpe oss med å ta riktige og gode avgjørelser underveis i utviklingen. Videre forteller vi om hvordan vi har brukt en iterativ tilnærming som metode under utviklingsprosessen. Deretter går vi inn på hvilke støtteverktøy vi har brukt i løpet av prosjektet.

I kapittel 4. Resultat, forteller vi om den ferdige systemarkitekturen. Vi skriver her både om hvordan de forskjellige komponentene i systemet interagerer med hverandre og mer i detalj om hvordan den enkelte komponenten opererer. Vi går også gjennom noen brukerscenarioer og noen av kjerneegenskapene til systemet. Funksjonelle og ikke funksjonelle krav til systemet er også definert her.

Videre i kapittel 5. Diskusjon, vil vi diskutere og drøfte hva vi har lært og funnet ut av i løpet av utviklingen av dette systemet. Dette vil også inkludere drøfting rundt sentrale veivalg under utviklingen, hvorfor vi valgte teknologien vi valgte og hvordan systemet kan bli brukt av andre aktører.

Vi avslutter så med å konkludere rapporten i kapittel 6. Konklusjon.

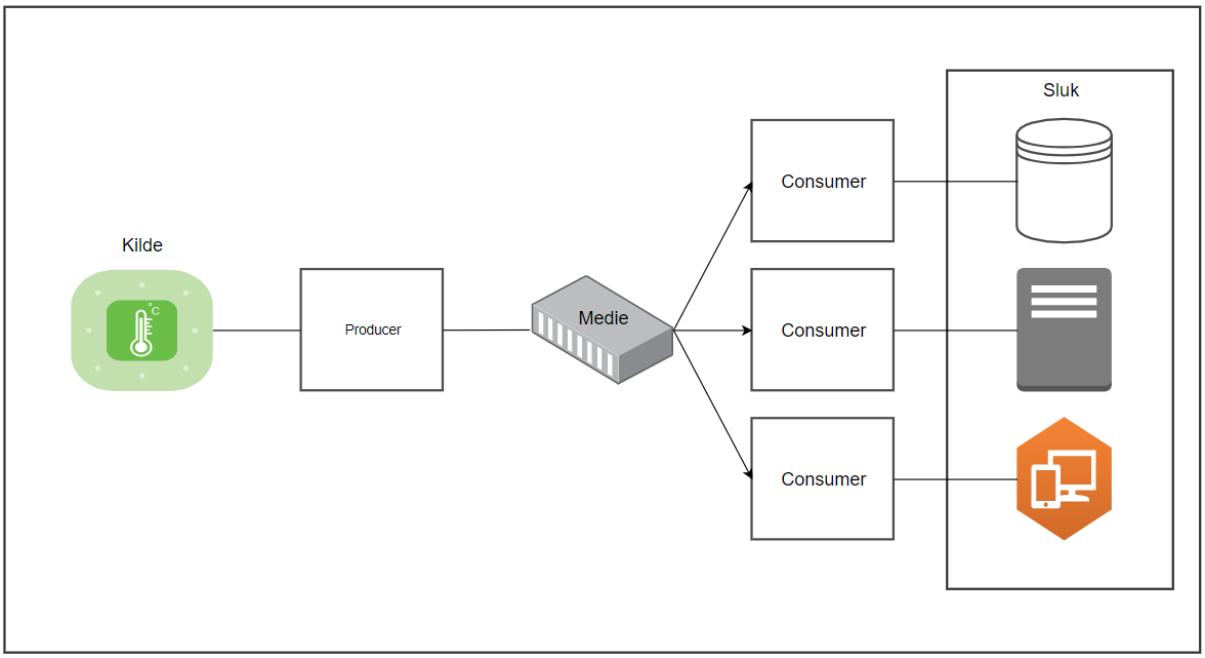
Rapporten vil også ha tre vedlegg. Disse er Brukermanual, Prosjekthåndbok og en ZIP-fil med all koden tilhørende denne bacheloroppgaven.

# 2. Teori

I dette kapittelet beskriver vi teorien bak eventbasert systemarkitektur. I tillegg beskriver vi to sentrale teknologier i sammenheng med denne oppgaven, Kafka og API.

## 2.1 Eventbasert systemarkitektur

Eventbasert systemarkitektur er en designtilnærming som tillater ett eller flere systemer å reagere på hendelser fra andre kilder. En hendelse representeres i form av en datapakke og kan være alt fra en tilstandsendring i en verdi, til opprettelse av en bruker. Denne datapakken inneholder da en eller annen form for informasjon om hendelsen. Vanlige konsepter i et eventbasert system er kilder, producere, consumere og sluk. Kilder er der hendelsen oppstår. Produceren er komponenten som fanger hendelsen og publiserer til et medie. Consumeren er komponenten som leser hendelsen fra mediet og avleverer dataen til sluket. I Figur 2 ser man illustrasjon av de forskjellige konseptene.




*Figur 2: Et eventbasert integrasjonssystem*

En av kjerneegenskapene til et eventbasert system er hvordan et mediet kan ha flere consumere, som vist på Figur 2 ovenfor. Dette gir muligheten til at én enkelt hendelse kan trigge mange nye hendelser ved at flere consumere plukker opp samme hendelse fra mediet og avlever denne til flere forskjellige sluk. Disse slukene kan være helt selvstendige og frakoblede tjenester som ikke er kjent med andre komponenter. [3]

Videre trenger heller ikke noen av de andre komponentene å være klar over andre komponenter enn dem de selv har direkte kontakt med. Dette betyr blant annet at forskjellige consumere også kan operere helt uavhengig av hverandre. Denne tankegangen skiller seg fra den mer tradisjonelle RPC-tankegangen som innebærer en direkte kommunikasjon mellom tjenester hvor både avsendere og mottakere er kjent med hverandre.

Mediet i et eventbasert system er ofte en form for meldingskø. En meldingskø er en type asynkron tjeneste-til-tjeneste-kommunikasjon. Dette betyr at meldinger som blir lagret på meldingskøen blir lest og prosessert på et vilkårlig tidspunkt i etterkant. En melding har en pakke med nyttelast som kan ha forskjellig metadata. En melding kan for eksempel være en hendelse med informasjon om en ny bruker som skal registreres. Et eventbasert system kan ha mange meldingskøer. Disse meldingskøene blir da naturlige grupperinger av kommunikasjonen mellom tjenestene. Med dette menes det at forskjellige typer hendelser gjerne blir sendt til forskjellige meldingskøer og dermed prosessert på forskjellige måter av tjenestene i sluket.

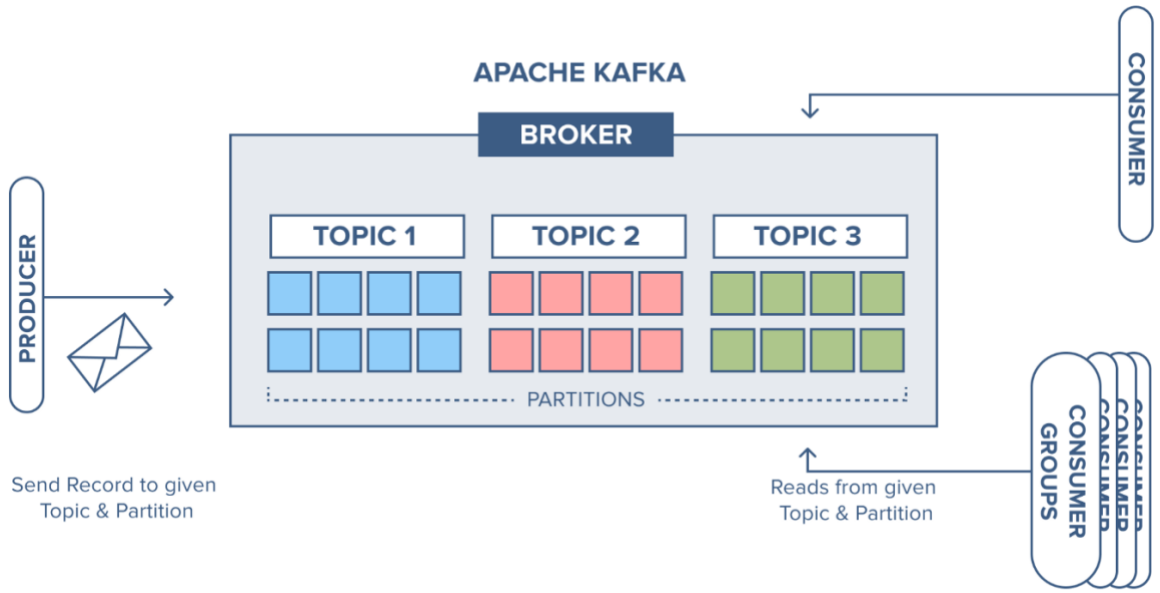
Et eksempel på et scenario i et eventbasert system, illustrert gjennom Figur 2 ovenfor, kan være ny værdata som blir plukket opp av en sensor. Denne dataen blir plukket opp av produceren. Produceren publiserer deretter værdataen til mediet. På mediet lytter flere consumere som konsumerer den samme værdataen uavhengig av hverandre. Disse consumerne sender så værdataen videre til forskjellige sluk hvor de blir prosessert.

Det finnes mange forskjellige leverandører som leverer litt forskjellige varianter av slike eventbaserte systemer. Forskjeller mellom disse variantene kan være hvordan systemet prioriterer hendelser og størrelsen på data som kan sendes i en melding. To eksempler på slike tjenester er Apache Kafka og RabbitMQ.

## 2.2 Apache Kafka

Apache Kafka (heretter bare Kafka) er en distribuert event-strømningsplattform vedlikeholdt av Apache Software Foundation. Kafka er open-source og blir brukt av mange av verdens største firmaer. [4]

Kafka er mye brukt på grunn av sine mange gode kjerneegenskaper. Dette innebærer at Kafka har mulighet for å takle svært stor gjennomstrømning av data. Kafka er også enkelt skalerbart med oppsett for klynger og partisjoner, noe som også kan sikre meget god oppetid. Videre har Kafka den unike egenskapen at all dataen på en meldingskø blir lagret også etter at den er konsumert av en consumer, og man har derfor en effektiv logg over data som er blir behandlet. En annen viktig egenskap hos Kafka er at det har ingen grenser for antall meldingskøer. [5]




*Figur 3: Elementene i Apache Kafka [6]*

Figur 3 over illustrerer de forskjellige elementene i Kafka. En Kafka Broker er representert som den grå boksen i midten. Det er tjenesten som vedlikeholder klyngen av meldingskøer og partisjoner, og det er disse man kobler seg til for å poste og abonnere på meldingskøer. Boksene av samme farge illustrerer en meldingskø i Kafka, og kalles et *Topic.* En melding i et topic kalles et *Event*. Et topic kan deles inn i partisjoner. En partisjon i topicet har en andel av alle eventene i dette topicet. Partisjoner gjør det mulig å fordele meldinger innad i samme topic. Dette er blant annet en fordel når det kommer til skalerbarhet og redundante løsninger.

Alle events får en posisjon i meldingskøen representert i form av et *Offset.* Offsetet representerer et heltall som indikerer avstanden fra meldingkøens første event. Offsetet kan brukes i etterkant til å hente spesifikke events fra meldingskøen. Kafka har også støtte for *consumer groups*. Dette gjør at flere consumere kan lytte på samme topic uten at et event blir prosessert mer enn én gang.

## 2.3 API

API er en forkortelse for *Application programming interface*, og er programvare som tillater at systemer kan kommunisere med hverandre. Det fungerer som et bindeledd mellom systemer, og gjør at et system kan ta imot og svare på forespørsler fra andre systemer. Det finnes mange typer API. En type API er et web-API. Denne typen API nås gjennom HTTP-forespørsler over et nettverk. [7] HTTP er en protokoll for informasjonsutveksling over Internett. [8]

Man kan sammenligne et API med måten en servitør fungerer i en restaurant. Servitøren tar imot en bestilling på en lignende måte som et API tar imot en forespørsel. Servitøren leverer så bestillingen til kjøkkenet, som behandler bestillingen. På lignende måte vil et API tar imot en forespørsel og levere det til et system som behandler forespørselen. Deretter vil servitøren levere tilbake produktet av bestillingen på samme måte som et

system returnerer en respons gjennom API. Systemer som trenger å sende og motta data mellom tjenester bruker ofte API til å holde kommunikasjon mellom seg. [9]

En måte å implementere et API i et system er å bruke rammeverket Flask. Flask er et av de mest populære rammeverkene for å implementere nettverks-API-er med Python. Flask er veldig populært fordi det er en svært rask og enkel prosess å komme i gang med, samtidig som den har egenskapene til å skalere opp til mer komplekse applikasjoner. Flask er utviklet og blir vedlikeholdt av The Pallets Projects som er en dugnadsbasert gruppe med formål om å vedlikeholde flere open-source prosjekter. [10]

# 3. Metode

Dette kapittelet tar for seg informasjon om hvilke metode som er brukt i utviklingsprosessen og hvordan vi har arbeidet for å komme frem til resultatet i oppgaven.

## 3.1 Sentrale verktøy i utviklingsprosessen

Gjennom arbeidsprosessen har vi brukt ulike verktøy for å støtte opp rundt arbeidet med bacheloroppgaven. Dette er verktøy som ikke har noe å gjøre med resultatet av oppgaven direkte, men har vært nyttige for oss i arbeidsprosessen gjennom semesteret.

Microsoft teams er en digital kommunikasjons- og samarbeidsplattform som tilbyr gruppechatter og videokonferanse. [11] Vi har brukt denne plattformen som vår kommunikasjonskanal mellom oss, veileder hos helseCERT og veileder hos NTNU. Teams er blitt brukt gjennom hele semesteret for å holde jevn kontakt, få svar på konkrete spørsmål og for å avtale møter. Plattformen har også blitt brukt til å avholde videomøter når vi ikke har hatt mulighet til å møtes fysisk.

Google Disk er en skytjeneste for lagring og deling av filer. Vi har brukt denne tjenesten til felles lagring av dokumenter, møteinnkallinger, referater, rapporter og annen dokumentasjon som er relevant for bacheloroppgaven. På grunn av store mengder dokumentasjon har det vært nødvendig men en slik tjeneste for å samle alt på ett felles sted hvor alle har tilgang. Google Disk tilbyr også mulighet til å samarbeide i sanntid gjennom google docs, noe som har vært veldig nyttig. Vi har ikke brukt denne plattformen til lagring av skript eller kode-filer under utvikling av systemet. [12]

Gitlab er et utviklingsverktøy for programvareutvikling som tilbyr versjonskontroll gjennom *git*, muligheter for å se kode-historikk, lagring og deling av filer. Git er et versjonskontrollsystem som håndterer versjon av kode og distribuerer endringer til de man samarbeider med. Bruk av et slikt versjonskontrollsystem har vært nødvendig når vi samarbeider på samme kode, og på grunn av tidligere erfaring med bruk av Gitlab var det naturlig å velge denne plattformen. [13]

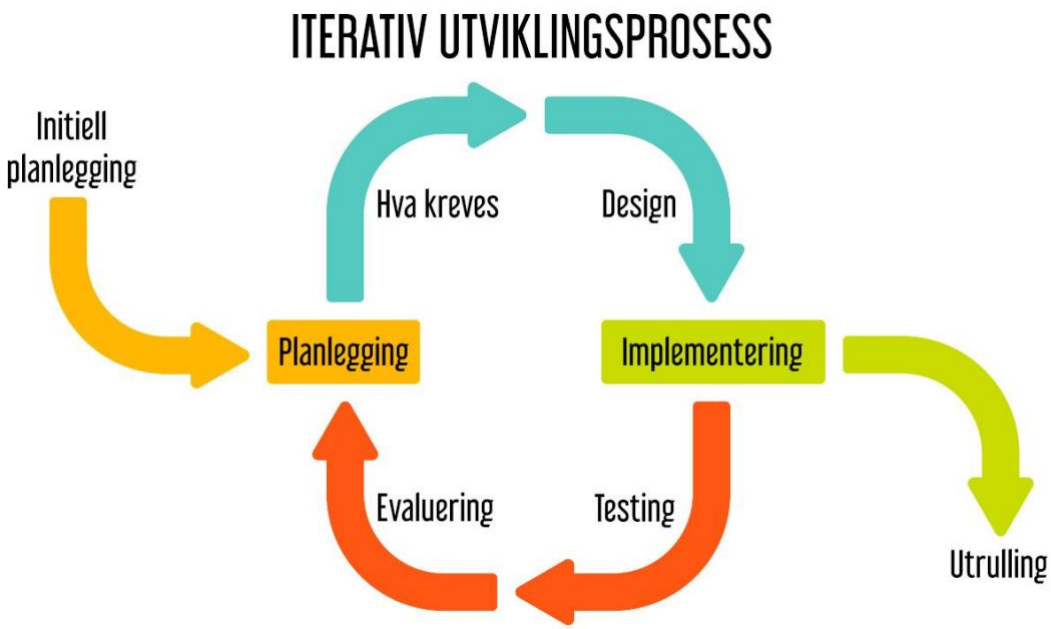
Microsoft Azure er en skyplattform som tilbyr administrering av skybasert infrastruktur, gjøre beregninger og utføre databehandling. Vi kjører infrastrukturen vår på VM-er på denne plattformen. Vi valgte å bruke Microsoft Azure fordi vi har tidligere erfaring med bruk av plattformen og fordi bruk av VM-er i skyen gir oss gode forutsetninger for utvikling og testing. Vi fikk tilgang til HelseCERTs egen Azure plattform og det var denne vi tok i bruk under utviklingen av prosjektet. [14]

Visual Studio Code er et koderedigeringsprogram vi har valgt å bruke til utvikling av systemet. Visual Studio Code er verdens mest populære koderedigeringsprogram ifølge en undersøkelse gjennomført av Stack Overflow. [15] Visual Studio Code er populært på grunn av de mange utvidelsene som er tilgjengelig i programmet. Dette kan være alt fra [16] til utvidelser som støtter nye kodespråk.

PEP8 er den offisielle kodestilen for koding i Python. PEP8 inneholder et sett med retningslinjer og konvensjoner for hvordan koden skal følge samme struktur gjennom hele prosjektet. Dette innebærer blant annet hvordan man navngir funksjoner, klasser og variabler, samt hvordan man strukturerer innrykk i linjer og linjeskift. [17] Vi har benyttet oss av PEP8 i denne bacheloroppgaven, og det har være med på å gjøre koden vår gjennomført og lettleselig.

## 3.2 Iterativ utvikling som utviklingsmetodikk

Noe av det første vi gjorde i semesteret før vi startet arbeidet med utvikling av systemet var å bli enige om en utviklingsmetodikk. Vi ble enige om å gå for en iterativ utviklingsprosess. Dette en en metode innen programvareutvikling som går ut på å gradvis utvikle systemet i løpet av flere sykluser. [18]




*Figur 4: Iterativ utviklingsprosess*

Som vist i Figur 4 over ser man iterativ utvikling illustrert. Figuren viser at hele prosessen starter med Initiell planlegging. I denne fasen ble vi enige om hvilke hovedkomponenter systemet skulle inneholde. Deretter lagde vi en helt enkel, men fungerende løsning som knyttet sammen disse komponentene gjennom et "Hello world"-eksempel. Denne fremgangsmåten kalles *vertical slicing*, og skiller seg fra *horizontal slicing* hvor man gjør ferdig én og én komponent om gangen. [19] Fordelen med vertical slicing er at man får et slags skjelett av arkitekturen som det egner seg godt å jobbe videre med i syklusene i iterativ utvikling. [20]

Etter den initielle planleggingen startet vi på første syklus. I en syklus starter vi med å planlegge hva slags funksjonalitet vi ønsker å legge til og ser på hva som kreves for å oppnå ønsket resultat for syklusen. Etter dette designer vi arkitekturen med den ønskede funksjonaliteten og blir enige om hvordan vi implementerer den. Deretter

implementerer vi løsningen i systemet og tester opp mot krav til kvalitet og funksjonalitet som vi hadde definert før utviklingsprosessen helt i starten av semesteret. De funksjonelle og ikke-funksjonelle kravene kommer vi til å beskrive i kapittel 4.2 Introduksjon og krav til produktet. Til slutt tar vi en evaluering av den nye versjonen før vi planlegger endringer som må med i neste syklus. En syklus varte som regel fra en til to uker, dette var avhengig av hvor omfattende endringene var og hvor godt vi kjente teknologien som var brukt.

## 3.3 Arbeid og utviklingsprosess

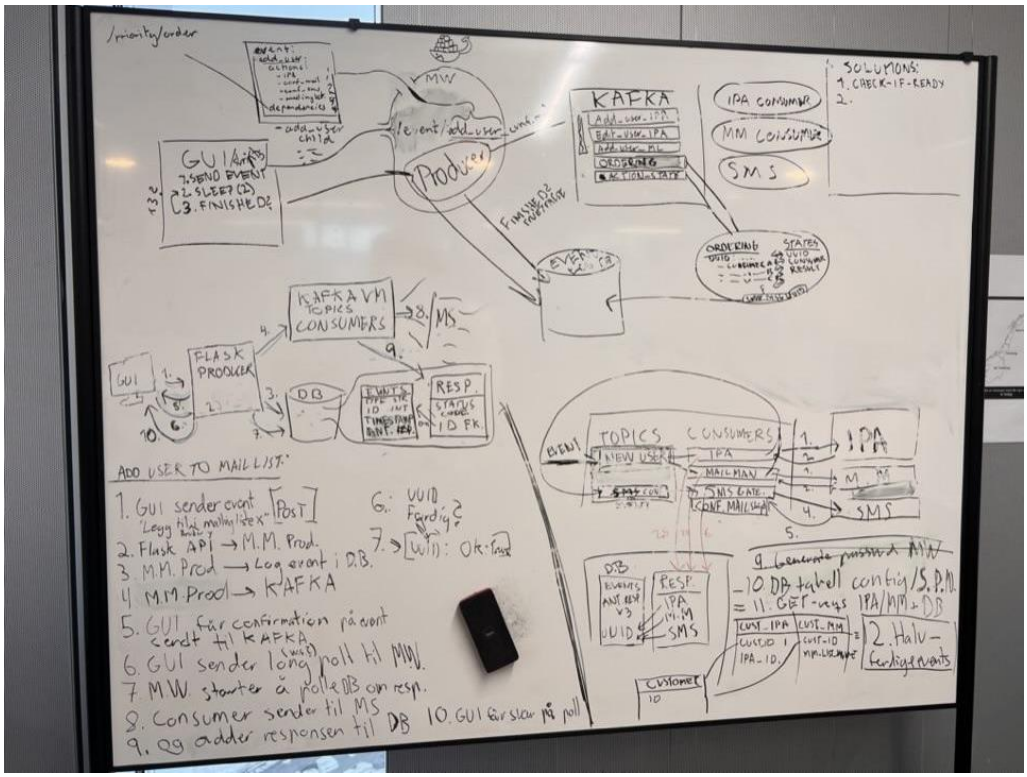
Ved oppstarten av prosjektet lagde vi en plan for semesteret om hvordan vi skulle komme frem til et godt resultat. Planen gikk ut på å ha en oppstartsfase hvor vi skulle lese oss opp på teori før vi startet med utviklingen av produktet som skulle leveres. Denne oppstartsfasen av prosjektet gikk ut på å bygge kompetanse og forståelse av teori knyttet til oppgaven. Deler av teknologien for å utvikle systemet hadde vi lite eller ingen kunnskap om fra før av. For eksempel er meldingskøer og eventbasert arkitektur noe vi ikke har vært borti før, og vi brukte derfor mye tid på å sette oss godt inn i den nye teorien for å kunne ta gode avgjørelser underveis i prosjektet. Disse valgene drøfter og forklarer vi i 5.2 Tekniske løsninger på systemkrav.

Etter oppstartsfasen med bygging av kompetanse kom vi i gang med planlegging av systemet vi skulle utvikle. Vi startet med å planlegge en fleksibel grunnstruktur på arkitekturen som vi tok utgangspunkt i ved utviklingen, altså vertical slicing. Som nevnt tidligere i 3.2 Iterativ utvikling som utviklingsmetodikk, valgte vi å ha en iterativ utviklingsmetodikk for å utvikle systemet gjennom sykluser. I starten trengte vi lengre tid til å designe og planlegge strukturen da mye av teknologien var ny og uvant. En syklus lå som regel på rundt to uker, men ble kortere når det nærmet seg påske. Da hadde vi god forståelse av den nye teknologien og brukte kortere tid på å integrere større endringer. Utfordringer og viktige valg som oppstod underveis håndterte vi med rask omstilling på grunn av effektiv jobbing i sykluser og en fleksibel grunnstruktur.

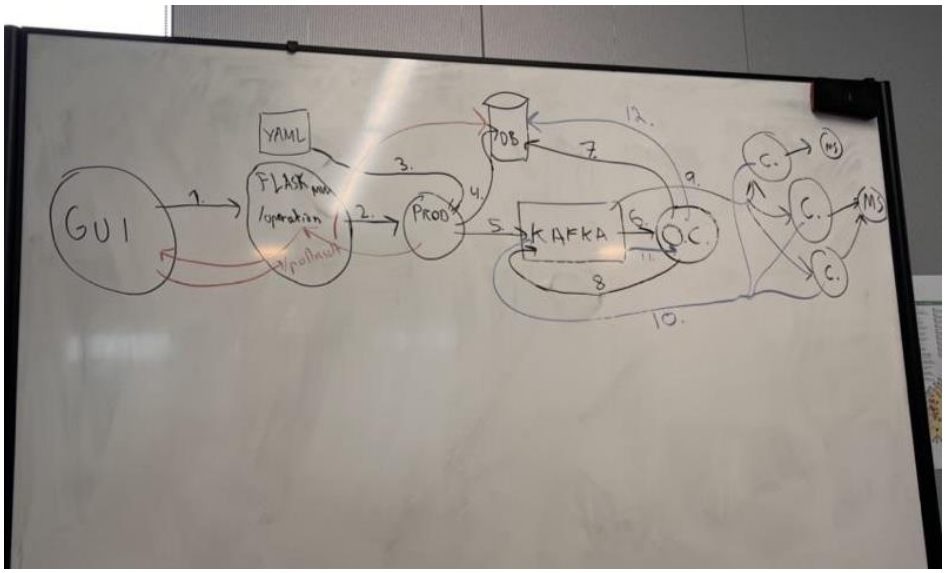
Gjennom semesteret har vi hatt jevne møter med vår veileder i HelseCERT. Møtene har gått ut på å gi oppdateringer på prosessen og for å fortelle om utfordringer og videre planer. På møtene fikk vi hjelp og tilbakemeldinger til å forbedre systemet. Dette har hjulpet oss med å vite hva de ønsker å prioritere slik at vi kunne ha fokus på hva som var mest viktig med systemet. I møtene med HelseCERT ble det utvekslet mange interessante synspunkter rundt oppgaven som ga dypere forståelse av problemstillingen og inspirerte oss til tenke nytt på ulike problemstillinger. Slike utfordringer og valg kommer vi til å beskrive nærmere i 5.2 Tekniske løsninger på systemkrav.

Under hele semesteret har vi fått tildelt faste plasser på kontoret til HelseCERT i Trondheim. På kontoret har vi hatt god tilgang til whiteboard, noe som har hjulpet oss godt gjennom semesteret. Whiteboard ble brukt flittig som et verktøy for å visualisere arkitekturen i systemet og har hjulpet oss med å samarbeide og forstå hverandres tanker bedre. Whiteboardet ble brukt til å lage oversikter over systemet, to-do lister på hva vi måtte gjøre og andre tegninger for å visualisere funksjonaliteter i systemet. Dette har vi satt veldig pris på da det har hjulpet oss med å visualisere tanker og ideer for

hverandre og hjulpet oss med å holde oversikt over prosessen. Under har vi lagt til to eksempler, bilde 5 og bilde 6, på hvordan vi har brukt whiteboardet til planlegging og å holde oversikt over arbeidet.



*Figur 5: Whiteboard 1*



*Figur 6: Whiteboard 2*

# 4. Resultat

I kapittel 4 skal vi gjøre rede for og beskrive systemet og arkitekturen vi har utviklet i denne bacheloroppgaven.

## 4.1 Begreper

I kapittel 4.1 beskriver vi sentrale begreper som er nødvendig å definere for å entydig kunne beskrive og omtale resultatene våre videre i dette kapittelet.

### 4.1.1 Ypsilon

Vi har valgt å kalle det eventbaserte systemet vi har utviklet for Ypsilon. Ypsilon er navnet på den greske bokstaven Y. Sett nedenfra og opp er Y en strek som splitter seg i to. Dette kan minne om logikken i et eventbasert system hvor én melding kan føre til flere nye handlinger. I tillegg er Ypsilon navnet på grupperommet gruppen vår har sittet på i deler av semesteret og jobbet fra. Teknisk beskrivelse av hva som er en del av Ypsilon og hva som ikke er det, kommer naturligvis senere i dette kapittelet.

### 4.1.2 Aksjon

En aksjon (fra eng. action) er i dette systemet definert som én bestemt handling/forespørsel mot én bestemt tjeneste. En aksjon kan for eksempel være å registrere en bruker i en brukerdatabase. Hver aksjon har fått definert én eller flere parametere som er nødvendige for å kunne gjennomføre aksjonen. Et parameter er en variabel. En parameter her betyr altså verdier som tilhører en aksjon, eksempelvis brukernavn, mailadresse og telefonnummer. Noen aksjoner har også én eller flere valgfrie parametere. Alle disse er definert gjennom en konfigurasjonsfil. I denne sammenheng kan en si at en aksjon er en atomisk hendelse, ettersom den enten blir helt utført eller ikke utført i det hele tatt.

### 4.1.3 Operasjon

En operasjon er i vårt system definert som et sett med aksjoner (beskrevet i 4.1.2 Aksjon) som skal skje når man sender en operasjon inn i systemet vårt. Disse aksjonene i operasjonen kan være delt opp i så mange steg man ønsker. Disse stegene utføres sekvensielt, mens alle aksjonene i et steg kan utføres samtidig.

En operasjon inneholder en nyttelast med samtlige parametere som er nødvendige for å kunne gjennomføre alle aksjonene i den. En operasjon og stegene i den er definert gjennom en konfigurasjonsfil. En aksjon kan i enkelte tilfeller generere nye parametere som legges til som et nytt parameter i operasjonen og som andre aksjoner kan bruke i et senere steg i operasjonen. Et eksempel på en operasjon kan være å registrere en bruker i flere interne systemer hos en bedrift. Dersom man vil legge til en bekreftelsesmelding i slutten av denne operasjonen, vil dette bli et nytt steg og dermed danne en stegvis operasjon. Det er fordi man er avhengig av at alle aksjoner i brukerregistrering skal være ferdige før man sender bekreftelsesmeldingen.

### 4.1.4 Eksterne systemer

Med "eksterne systemer" menes de systemene som et eventbasert system sender forespørsler mot. Disse systemene kan være store og komplekse eller små og simple. Slike systemer kan for eksempel være en database, brukersystemer eller mailservere. Alle handlinger som kan bli utført på et slikt system vil være knyttet til en aksjon. Flere av aksjonene i vårt system kan derfor være knyttet til det samme eksterne systemet. Alle de eksterne systemene er selvstendige og helt uavhengige fra de andre systemene. De er også helt autoritative og dataen i disse systemene kan endres fra flere steder enn kun vårt system.

## 4.2 Introduksjon og krav til produktet

I utviklingen av Ypsilon har vi satt opp noen funksjonelle og ikke-funksjonelle systemkrav som vi har tatt utgangspunkt i gjennom hele utviklingen og ferdigstilling av arkitekturen. Dette er systemkrav vi har kommet frem til ved å se på funksjonalitet og kvaliteter som er nødvendig for å utvikle et system som besvarer problemstillingen. Disse har hjulpet oss underveis i utviklingen ved å konkretisere hvilke krav problemstillingen innebærer. Systemkravene sikrer altså at vi kan levere et system av forventet kvalitet og funksjonalitet.

### 4.2.1 Funksjonelle krav

Funksjonelle krav sier noe om hva systemet vårt er ment å gjøre. Dette innebærer bestemt atferd og funksjonalitet vi ser på som nødvendig for å løse oppgaver som vi har definert i problemstillingen. [21]

Ypsilon skal håndtere operasjoner gjennom en eventbasert tilnærming. Med dette menes det at en operasjon skal kunne påvirke ett eller flere eksterne systemer basert på innholdet i den innkommende operasjonen. Systemet må ha to brukergrensesnitt, et GUI skrevet i TypeScript med rammeverket React og et CLI som tar imot bruker-inputs. Ved kjøring av operasjoner fra et grensesnitt skal det komme fortløpende tilbakemelding til brukeren på om aksjoner i operasjonen er blitt gjennomført og hvorvidt de var vellykket eller ei. Dersom en aksjon ikke er vellykket skal Ypsilon tillate bruker å prøve på nytt med alle aksjonene som ikke ble gjennomført.

Systemet skal håndtere at noen spesifikke aksjoner må være utført før andre aksjoner kan forsøkes gjennomført. Noen aksjoner kan ikke angres, eksempelvis utsending av e-post og SMS, og krever at andre aksjoner har kjørt og er suksessfulle først. Andre ganger kan det være avhengigheter mellom eksterne systemer som også gjør at en operasjon krever en form for rekkefølge. Det kan være at et system genererer et parameter som et annet system senere skal bruke. Dette må altså kunne skje i samme operasjon. I tillegg er systemet nødt til å gjøre det mulig å modifisere og opprette operasjoner på en enkel måte uten å gjøre omfattende endringer på systemet.

### 4.2.2 Ikke-funksjonelle krav

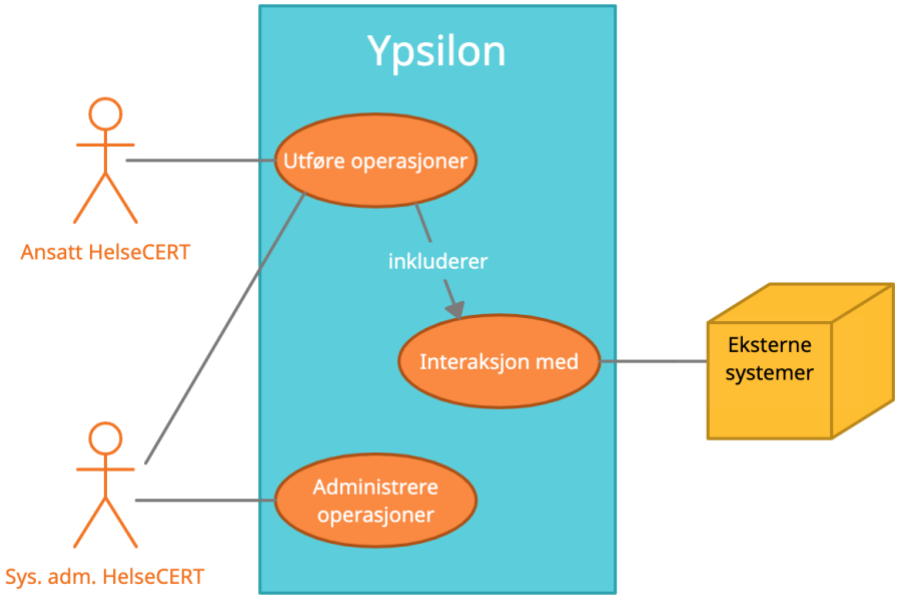
Ikke-funksjonelle krav sier noe om kriterier og kvaliteter til systemets bruk og drift. Det sier altså noe om hvordan et system er ment å være når det kommer til kvalitative egenskaper. [21]

Et sentralt ikke-funksjonelt krav er at arkitekturen skal ha teknisk interoperabilitet i henhold til samspill mellom vårt system og de eksterne systemene. Interoperabilitet defineres av EU som "den evne IT-systemer med tilhørende forretningsprosesser har til å utveksle data og dele informasjon og kunnskap". [22] I vår forstand går det ut på å ha et system som administrerer utveksling av data og deling av informasjon.

Ypsilon skal være fleksibelt og utvidbart. Det vi si at systemet skal være skalerbart i form av antall eksterne systemer tilkoblet. Det skal være en enkel prosess å legge til nye eksterne systemer uten å trenge å gjøre omfattende endringer. Dette skal i tillegg gjelde for operasjoner. Operasjoner knyttet til de eksterne systemene skal være modifiserbare og det skal være mulig å legge til flere operasjoner.

Systemet skal også være robust i form av at det skal ha nok kapasitet til å håndtere HelseCERT sin bruk. Et annet krav fra HelseCERT er at systemet må være selvutviklet og open-source. Dette er på grunn av retningslinjer for kontroll og oversikt over programvare de bruker. I tillegg skal systemet være brukervennlig og enkelt å vedlikeholde.

## 4.3 Brukerscenario

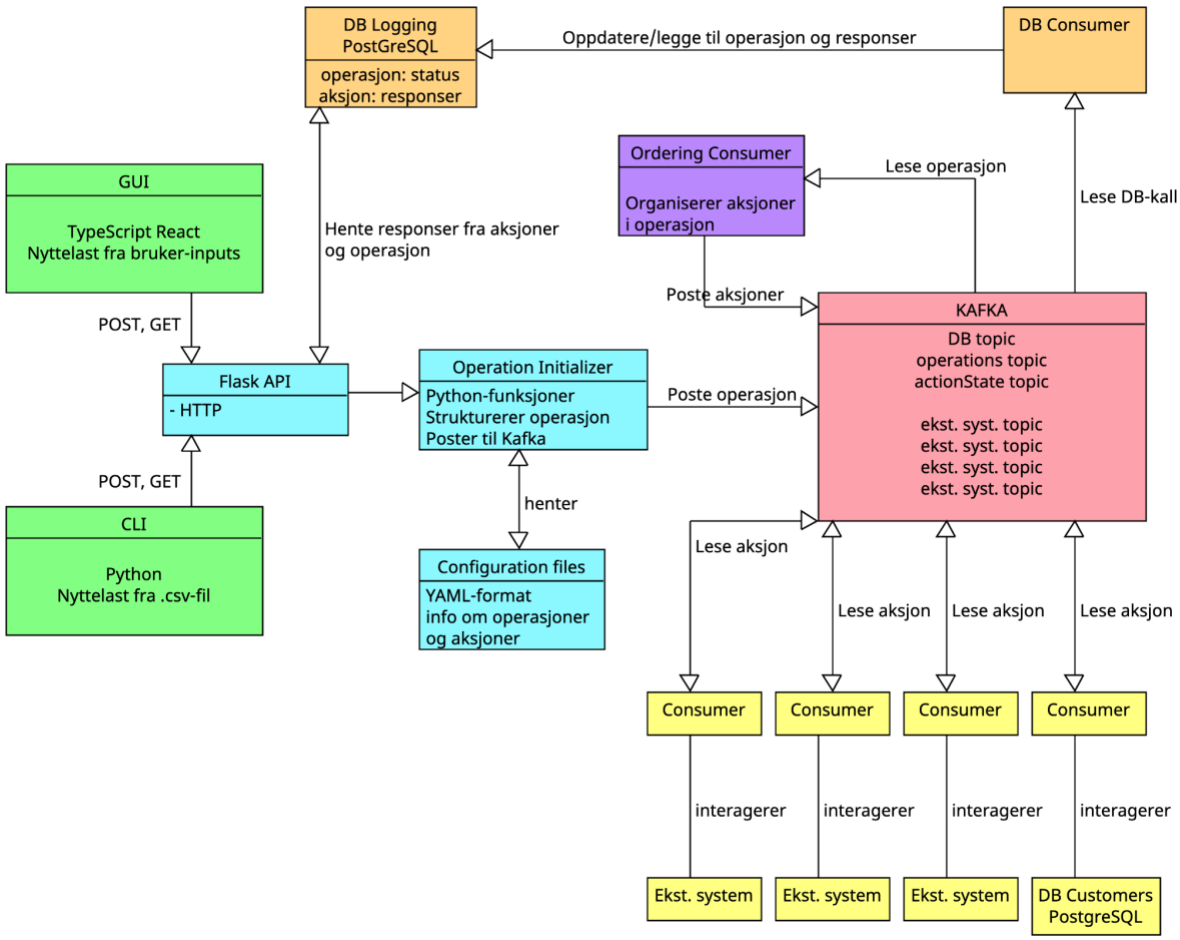



*Figur 7: Bruksmønster for Ypsilon*

Figur 7 ovenfor viser et typisk bruksmønster, eller *use case*, for systemet vårt. Aktørene vi har tatt med her er "Ansatt HelseCERT" og "Sys. adm. HelseCERT". Førstnevnte er den

mest sentrale aktøren her, og det er verdt å merke seg at begge aktørene i HelseCERT naturligvis kan være samme person. En HelseCERT-ansatt vil ønske å gjennomføre en operasjon mot de eksterne systemene. Videre ser man at å utføre en operasjon inkluderer interaksjon med ett eller flere eksterne systemer. En systemadministrator i HelseCERT kan også administrere operasjoner.

## 4.4 Systemarkitektur




*Figur 8: UML av systemarkitektur*

Figur 8 ovenfor er en illustrasjon av alle komponentene i Ypsilon, samt komponenter som interagerer med Ypsilon. I figuren kan man se at komponentene er delt opp i seks forskjellige farger. Disse representerer de seks subsystemene som vi har funnet det naturlig å dele hele systemet opp i, og beskrives hver for seg i detalj i kapittel 4.5 Beskrivelse av sentrale komponenter. I 4.6 Standard prosessflyt med eksempel knyttes alle komponentene sammen gjennom et eksempel på en standard prosess i Ypsilon. I 4.7 og 4.8 beskrives de to andre prosessene i Ypsilon, henholdsvis feilhåndtering og GET-requests mot eksterne systemer. I kapittel 4.9 Utvidelse mot nye eksterne systemer beskriver vi hvordan man legger til nye tjenester i Ypsilon.

Brukergrensesnitt representeres av en grønn farge og innebærer GUI og CLI. Dette er komponentene som en vanlig sluttbruker av systemet vårt interagerer direkte med.

Flask API-et samt Operation Initializer og Configuration files representeres av turkis. Disse er gruppert sammen fordi de sammen representerer det første logiske steget i Ypsilon som en operasjon går gjennom. Som man ser ut ifra pilene i Figur 8 er Kafka-tjenesten representeres av rød farge. Kafka tjenesten kjører i midten av Ypsilon og fungerer som systemets message broker.

Python-programmet kalt *Ordering Consumer* er representert i en lilla farge. Programmet har som oppgave å sørge for rekkefølgen på alle aksjonene i en operasjon blir opprettholdt. Det er denne komponenten som faktisk sender aksjoner til forskjellige topics i Kafka.

Logging databasen i Ypsilon samt dens egne consumer er representert gjennom oransje farge. Alle responser fra aksjoner lagres her via databasens egne consumer som ligger mellom Kafka og denne. Brukergrensesnittet spør kontinuerlig denne databasen for oppdateringer på pågående operasjoner.

De eksterne systemene samt deres consumerne er respresentert i gult. Disse consumerne som fungerer som bindeleddet mellom Ypsilon og de eksterne systemene, som kan være alt fra store brukerkatalog-tjenester til små programmer. Felles for dem er allikevel at de er et selvstendig system med en eller annen form for API som kan motta forespørsler utenfra.

## 4.5 Beskrivelse av sentrale komponenter

I kapittel 4.5 vil vi beskrive funksjonaliteten til de sentrale komponentene i Ypsilon og hvordan disse interagerer med de andre komponentene i systemet.

### 4.5.1 Brukergrensesnitt

Vi har to forskjellige brukergrensesnitt; et GUI og et CLI. GUI-et er laget i TypeScript med rammeverket React. GUI lar brukeren velge operasjon som skal utføres. Brukeren må så fylle inn de nødvendige parametrene for at denne jobben skal kunne utføres. CLI-et er laget i Python. Her fyller brukeren ut en CSV-fil med de nødvendige parametrene som operasjonen trenger. CSV står for "Comma-separated values" og brukes for å liste verdier som enkelt kan leses av et program. Brukeren velger så flagget tilsvarende operasjonen som skal utføres når skriptet kjøres.

Når brukeren kjører skriptet fra CLI og dermed initierer operasjonen, sendes en POST-forespørsel til Ypsilon. Videre skal brukergrensesnittene fremvise responsene fra operasjon og aksjonene i sanntid. Slik får brukeren fortløpende vite når de forskjellige aksjonene i en operasjon resulterer i suksess eller når og hvor eventuelle feil har oppstått i prosessen.

```
#In this CSVfile lines starting with '#' is ignored
#Seperate arguments with ',' (commas)

username,first_name,last_name,email,phone,customer,maillist_name
OlaN,Ola,Nordmann,OlaN@gmail.com,46969469,Trondheim Kommune,TrdKomV
```

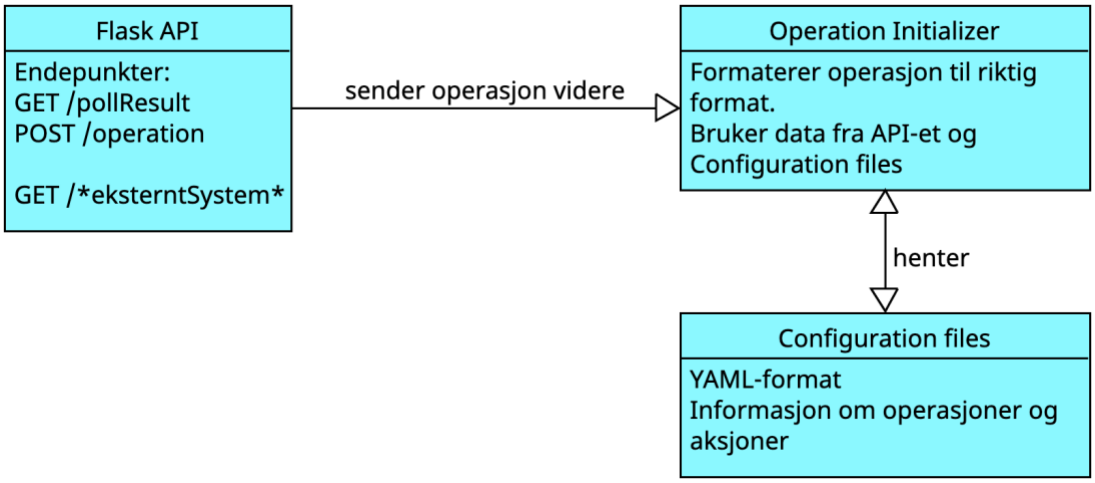
*Figur 9: Skjermutklipp CSV-fil*

```
[azureuser@GUI-and-scripts scripts]$ python3 genericScript.py -p ./generic.csv -o add-user

               Action                Status                                    Text
-------------------------------------------------------------------------------------------------------------------------------

 -------------------------------------------------------------------------------------------------------------------------------
       UUID :916f0a12-4fdb-4dd5-8cde-90d41d5f4587, username: OlaN

  Add_new_user_to_mailinglists_mailman      0            The email OlaN@gmail.com was added to the mailing list TrdKomV
       Add_customer_relation_db             0               User OlaN is now related to customer Trondheim Kommune
           Add_new_user_ipa                 0                    User Ola Nordmann successfully added to IPA
      Send_credentials_new_user             0      Email with ZIP attachment and SMS with password sent successfully to OlaN

       Operation finished working

       Proccess done!
[azureuser@GUI-and-scripts scripts]$
```

*Figur 10: Skjermutklipp CLI*

I Bilde 9 og 10 over ser vi et eksempel på hvordan en typisk operasjon vises i CLI. Her fyller brukeren ut parameterne som trengs for den ønskede operasjonen i CSV-filen. Den første hvite linjen nøklene til parameterne, mens den neste linjen representerer de korresponderende verdiene til disse nøklene. Ut ifra eksempelet i Bilde 9 vil det altså si at "username" er "OlaN", "first_name" er "Ola" og så videre. Når CSV-filen er fyllt ut kjører brukeren skriptet "genericScript.py", her spesifiserer man også hvilken operasjon som man ønsker å gjennomføre. I dette tilfellet ser man at dette er "add-user". Brukeren får deretter oppdatering på statusen til de forskjellige aksjonene i operasjonen i sanntid.

### 4.5.2 API og initiering av operasjoner




*Figur 11: Detaljert utklipp fra figur 8*

Tre sentrale komponenter i Ypsilon er Flask API-et, *Operation Initializer* og *Configuration files*. API-et er komponenten som tar imot alle forespørsler fra brukergrensesnittene. Operation Initializer er et sett med Python-funksjoner som strukturer og formaterer en operasjon for deretter å sende denne videre til Kafka. Configuration Files er YAML-filer som inneholder data om operasjon og aksjoner. YAML er en type dataserialiseringspråk som brukes for å strukturere og organisere dataverdier på en human-readable måte. Dataen i disse YAML-filene er informasjon om hvilke aksjoner som inngår i hver operasjon, og eventuelle steg disse er delt opp. I tillegg ligger det informasjon om hvilke parametere som kreves for å gjennomføre hver aksjon.

Når en POST-forespørsel blir sendt til endepunktet "/operation" i API-et vil det sende denne dataen videre til funksjonen Operation Initializer. Denne dataen inneholder et sett med parametere, men også en attributt for hvilken operasjon som skal utføres. Komponenten Operation Initializer vil hente aksjonene tilhørende denne operasjonen fra en YAML-konfigurasjonsfil med denne informasjonen. Dette er informasjon om struktureringen på aksjonene som skal utføres. Operation Initializer genererer også en UUID for den aktuelle operasjonen. UUID står for *Universally Unique Identifier* og er et 128-bits nummer som brukes som ID for den aktuelle operasjonen. Deretter postes metadata om operasjonen til databasen via Kafka slik at den får en oppføring i loggen. Avslutningsvis blir hele operasjonen postet til Kafka i topicet "Operations".

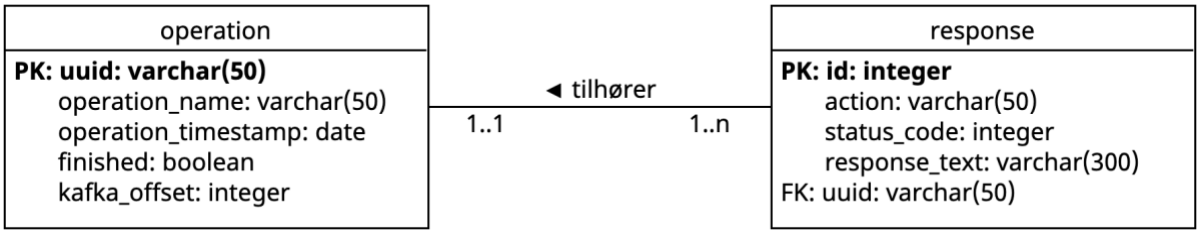
Operation Initializer har også en egen klasse kalt RebuildAndRetry som håndterer operasjoner som har feilet og deretter blir forsøkt gjennomført på nytt. Hele denne prosessen blir beskrevet i detalj i kapittel 4.7 Feilhåndtering.

### 4.5.3 Kafka

Kafka er en eventbasert programvare som er en sentral del i Ypsilon. Kafka fungerer som en meldingskø-tjeneste i systemet vårt. I kafka har vi har ett topic for hver mulige aksjon som kan utføres av en ekstern tjeneste. I tillegg til disse topicene har vi ett for databasekall, samt to som brukes av Ordering Consumer. Når en operasjon blir lagret i databasen vil også operasjonens offset, eller plassering i topicet, bli lagret. Årsaken til at vi gjør dette kommer vi mer tilbake til i 4.7 Feilhåndtering. Her kommer Kafkas egenskap med lagring av gamle meldinger til nytte.

### 4.5.4 Logging database

I Ypsilon benytter vi en logging database laget i PostgreSQL. Databasen brukes for å logge informasjon om operasjoner og aksjoner, deriblant responsen fra de eksterne systemene. Rader i databasen blir lagt til og endret (INSERT, UPDATE) med kall gjennom en egen consumer i Kafka, mens lese-kall (SELECT) gjøres direkte fra Flask API-et.




*Figur 12: UML logging database*

Loggen består av to tabeller, som vist i Figur 12. Den første er en tabell for alle operasjonene og heter "operation". Hver operasjon har en UUID som unik identifikator som responsene i den andre tabellen "response" bruker som fremmednøkkel. En fremmednøkkel er en referanse fra et element i en tabell til et annet element i en annen tabell. Hver aksjon i response-tabellen vil bli oppdatert fortløpende med responser når aksjonen er ferdig utført. Når alle aksjonene i en operasjon er ferdig vil dette bli modifisert i databasen ved at "finished" blir satt til true. Hver operasjon har en "operation_timestamp" som blir generert automatisk når en ny operasjon blir satt inn i databasen.

### 4.5.5 Ordering Consumer

Ordering Consumer (heretter omtalt som "OC") er et program skrevet i Python. OC-en er programmet som sørger for at hver aksjon i en operasjon blir postet til Kafka i riktig rekkefølge. Operasjoner skal som kjent kunne ha flere steg med aksjoner som skal kjøre kun når alle aksjonene i det foregående steget er ferdig utført. Dette sørger OC for. OC kjører både som en Kafka producer og en consumer. Den er en consumer fordi den mottar hele operasjoner gjennom Kafka fra Operation Initializer. Den er naturligvis også en producer fordi den poster aksjoner inn til sine respektive topics i Kafka når de klare til å bli utført.

OC er også consumer av topicet "ActionState" i Kafka. Gjennom "ActionState" tar den imot responsen fra aksjoner når disse er ferdig utførte. OC-en vil oppdatere responsene og tilstandene til aksjonene fortløpende i databasen. Med tilstand menes det hvorvidt en aksjon er underveis, ventende eller ferdig utført. Når alle aksjonene i et steg er ferdig vil OC-en initiere neste steg i operasjonen, eller oppdatere parameteren "finished" til "TRUE" for hele operasjonen dersom det ikke er flere steg. Dersom én eller flere aksjoner ikke er utført suksessfullt vil operasjonen avbrytes før den går videre til neste steg. Da vil operasjonen også bli få oppdatert parameteren "finished" til "TRUE". Dermed vil det være opp til brukergrensesnittet å tolke resultatene fra aksjonene for å finne ut om hele operasjonen ble gjennomført suksessfullt eller om noe feil skjedde underveis.

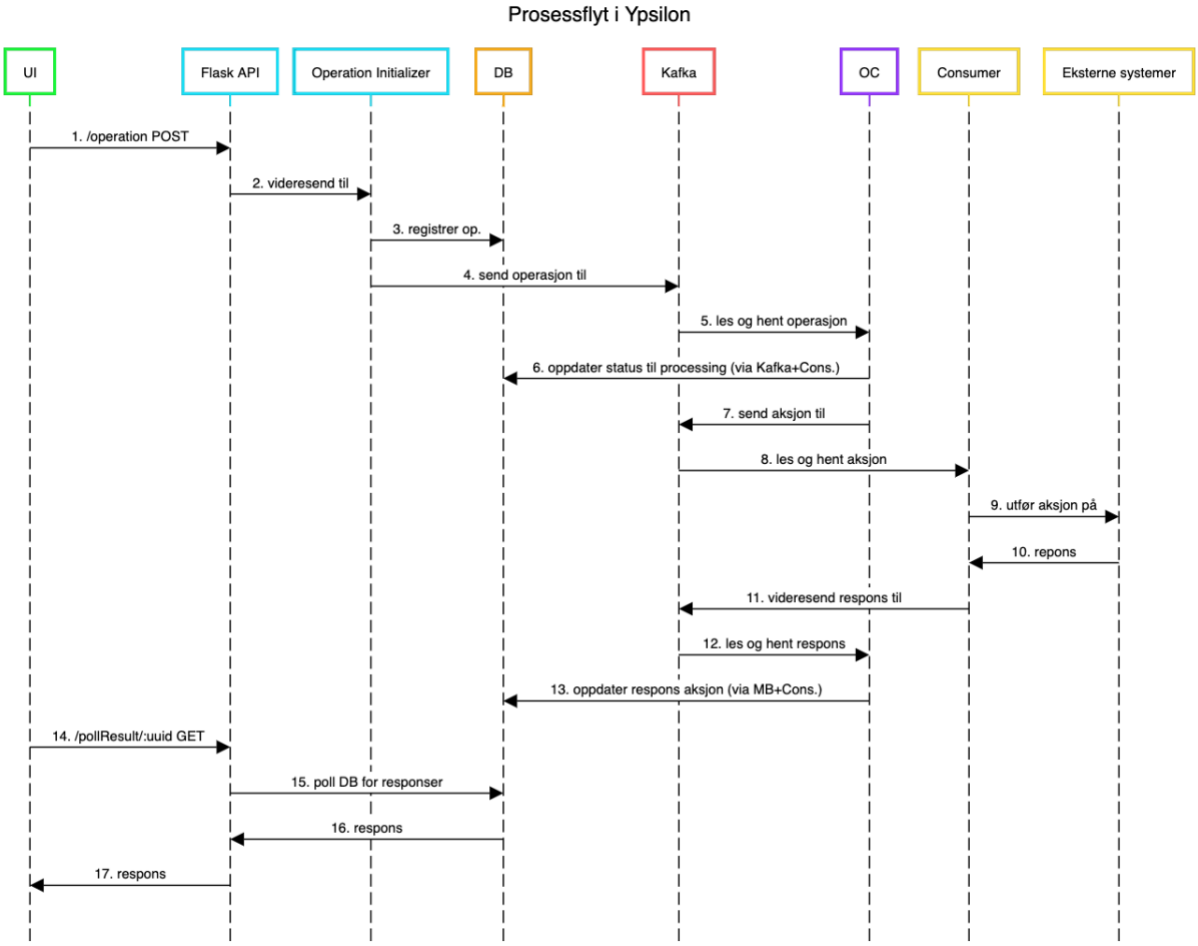
### 4.5.6 Consumere for eksterne systemer

Med consumere for eksterne systemer menes de Kafka-consumerne som har som oppgave å være et bindeledd mellom Kafka og de eksterne systemene. Disse consumerne er Python-funksjoner som utfører helt konkrete oppgaver mot de eksterne systemene. Consumeren leser én og én melding fra meldingskøen og utfører en aksjon mot et eksternt system basert på informasjonen og nyttelasten i meldingen. Når consumeren får en respons fra det eksterne systemet på aksjonen som er utført vil

consumeren fungere som en Kafka-producer og legge responsen inn i "ActionState"-topicet som OC-en deretter vil lese.

Én av consumerne i systemet vårt snakker med det eksterne systemet "DB Customers". Dette er en HelseCERT-spesifikk database vi har laget som en del av oppgaven vår. Den fungerer som en autoritativ database med en intern oversikt over kundene som HelseCERT arbeider med. Dette er av den hensikt at alle ansatte skal ha en fullstendig oversikt på ett og samme sted over alle deres kunder. Dette har de ikke hatt noen god løsning på tidligere.

## 4.6 Standard prosessflyt med eksempel

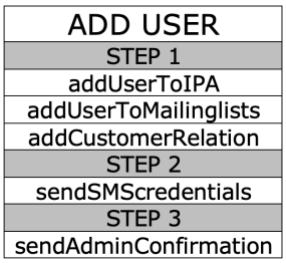



*Figur 13: Sekvensdiagram standard prosessflyt*

Figur 13. viser hele prosessflyten når en operasjon blir sendt fra brukergrensesnittet (UI i Figur 13), gjennom Ypsilon, ut til eksterne systemer og responsene derfra hele veien tilbake til brukergrensesnittet. Denne figuren viser hvordan den helhetlige kommunikasjonen mellom de forskjellige komponentene fungerer, og i hvilken rekkefølge det hele skjer.

For å beskrive hele prosessflyten steg for steg eksemplifiserer vi det gjennom en "ADD USER"-operasjon. Denne operasjonen inneholder følgende aksjoner og steg som vist på

Figur 14 under. Denne operasjonen er en typisk operasjon de ansatte i HelseCERT vil utføre, og krever interaksjon med flere eksterne systemer.

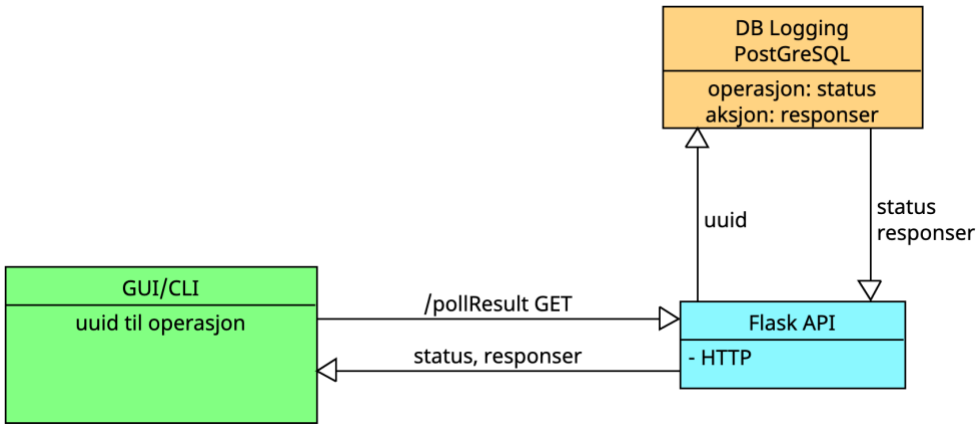



*Figur 14: Oppsett av en operasjon*

Som man ser fra Figur 14 krever denne operasjonen fem aksjoner fordelt på tre forskjellige steg. I dette tilfellet betyr det altså at aksjonene "addUserToIPA", "addUserToMailinglists" og "addCustomerRelation" må være suksessfullt gjennomført for at OC-en skal kunne fortsette med aksjonen i steg 2.

Tallene i parentes videre i dette kapittelet viser til de nummererte pilene i Figur 13. Hele prosessen starter når operasjonen blir sendt som en POST-forespørsel fra brukergrensesnittet til Flask API-et (1). API-et sender operasjonen videre til Operation Initializer (2). Operation Initializer genererer en UUID for operasjonen, strukturerer operasjonen og poster all informasjon om operasjonen til Logging DB (3). Deretter postes operasjonen videre til Kafka (4) i topicet "operations" hvor den videre blir lest av OC (5). OC-en vil nå oppdatere status på de tre aksjonene i det første steget til "processing" i Logging DB (6) og deretter poste aksjonene til sine respektive topics i Kafka (7). Consumerne for disse topicene konsumerer meldingene (8) og utfører aksjonene mot sine eksterne systemer (9). For eksempel vil consumeren for "addUserToIPA" sende et RPC-kall til det eksterne brukerkatalog-systemet FreeIPA om at det skal opprette en bruker med de parameterne som følger med operasjonen. Dette vil blant annet innebære e-post, telefonnummer og navn.

De eksterne systemene svarer etter hvert consumeren sin med en respons på hvordan aksjonene gikk (10). Denne responsen blir så sendt videre til OC-en via Kafka (11 og 12). OC-en vil da oppdatere Logging databasen med denne responsen. Videre vil OC oppdatere sin interne oversikt med status over operasjonen og sjekke om alle stegene i operasjonen er ferdige. Dersom dette er tilfelle vil den poste det neste steget inn i Kafka (pil 7 igjen). Hvis én eller flere aksjoner i et steg fortsatt arbeider vil OC bare vente til flere responser kommer inn. Hele denne prosessen (6-13) fortsetter helt til hele operasjonen er ferdig utført eller til noe har gått galt. I dette eksempelet blir alt suksessfullt utført. Da oppdaterer OC-en parameteren "finished" til "TRUE" i Logging DB.




*Figur 15: UML Polling*

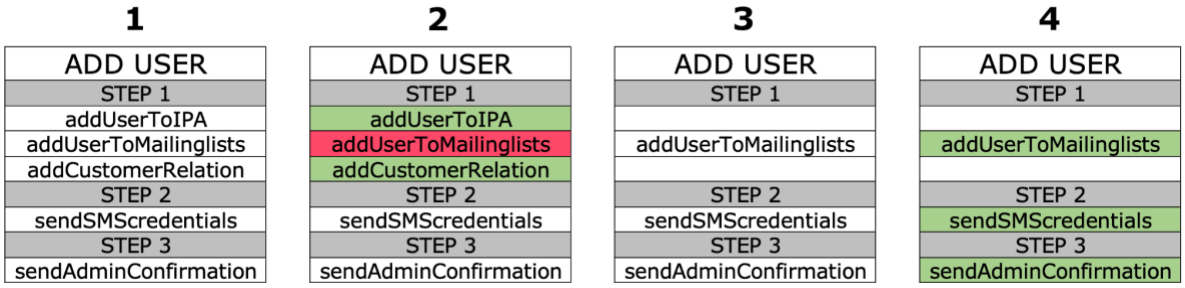
Etter at en UUID for operasjonen er generert vil denne returneres tilbake til brukergrensesnittet. Umiddelbart etter dette vil brukergrensesnittet starte å polle API-et for responser (14). Med polling menes her kontinuerlige GET-forespørsler som blir sendt til API-et. Pollingen er også illustrert i Figur 15 ovenfor. I hver forespørsel blir UUID-en til operasjonen sendt med. API-et forespør så alle responser til operasjonen med denne UUID-en i Logging databasen (15), får respons (16) og svarer brukergrensesnittet med alle responsene (17). Da er det opp til brukergrensesnittet å tolke disse responsene og videre fremvise sluttbrukeren hva status på operasjonen er.

Det er verdt å merke seg at pilen som går fra "OC" til "DB" (pil 6 og 13 i Figur 13) er en liten logisk forenkling, slik beskrivelsen på pilen i figuren også tilsier. OC gjør ikke databasekallene direkte til DB, men sender disse til Kafka slik at en dedikert database-consumer kan lese dette og gjøre de faktiske kallene mot databasen. Som oftest hadde det fungert helt fint å gjøre kallene direkte fra OC til logging databasen. Men i noen tilfeller er databasen nede eller overbelastet, og da vil ikke dette bli noen direkte flaskehals for OC dersom disse kallene går gjennom Kafka. Da vil OC kunne fungere videre siden den bare sender kallene til et topic i Kafka. Der vil kallene bli liggende som eventer i topicet selv om databasen er nede, og vil kunne bli prosessert gjennom på nytt når databasen etter hvert er oppe og kjører igjen.

## 4.7 Feilhåndtering

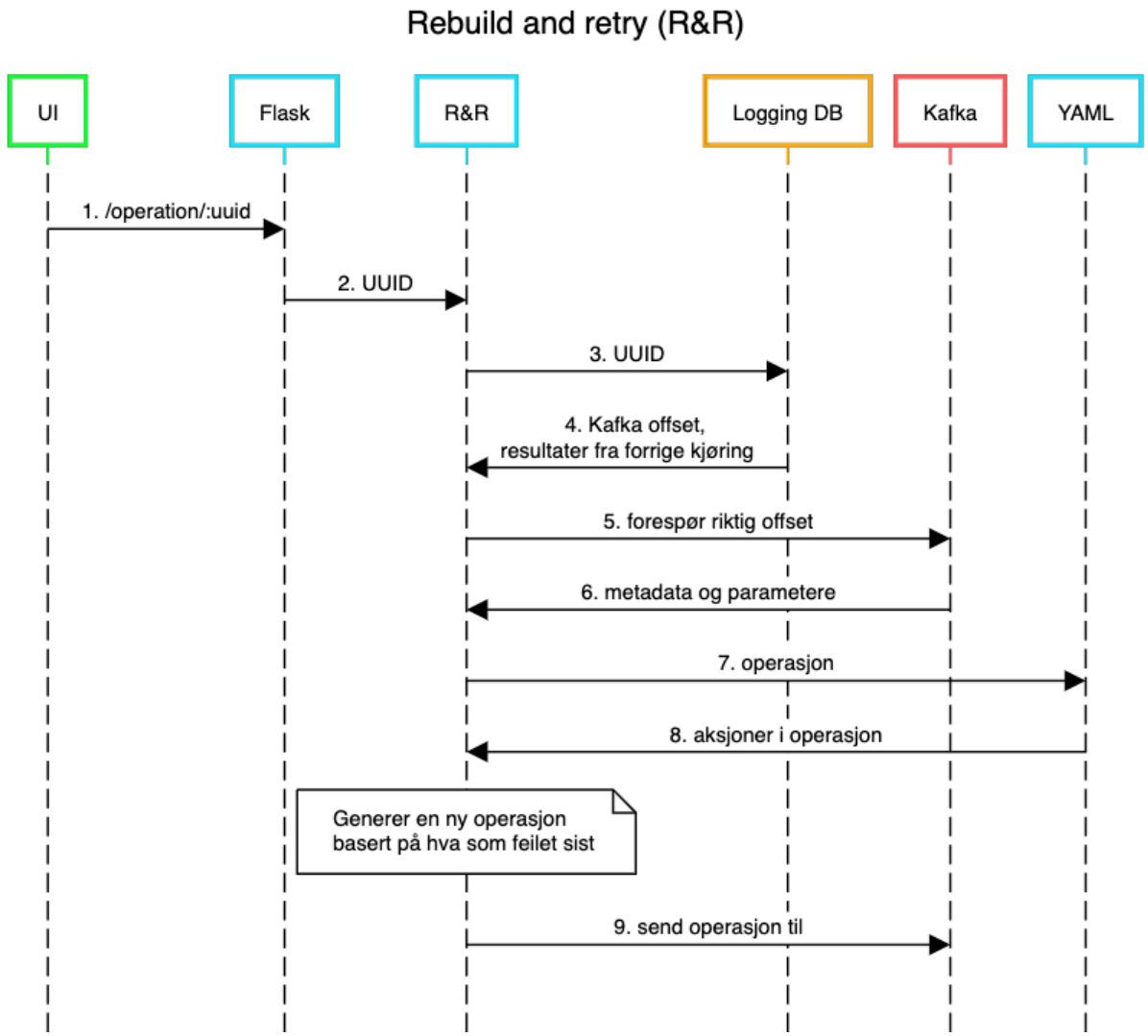
I noen tilfeller vil en stegvis operasjon bare bli delvis ferdig. Dette kan skje når én eller flere av aksjonene i et steg mislykkes. Da vil brukeren sitte igjen med en *delvis* gjennomført operasjon. Når dette skjer får brukeren mulighet til prøve å kjøre de resterende aksjonene i operasjonen på nytt. Denne prosessen har vi valgt å kalle "Rebuild and Retry" (heretter forkortet R&R), og beskrives videre i dette delkapittelet.

Vi eksemplifiserer dette gjennom den samme operasjonen som i 4.6 Standard prosessflyt med eksempel, "ADD USER".




*Figur 16: Feilhåndtering av en operasjon*

Tabell 1 i Figur 16 viser den opprinnelige "ADD USER"-operasjonen før den har blitt kjørt gjennom systemet. Når denne operasjonen blir sendt til Ypsilon fra brukergrensesnittet følger den naturligvis hele prosessflyten beskrevet i 4.6 Standard prosessflyt med eksempel. Men tenk deg at det i dette eksempelet går noe galt i gjennomføringen av aksjonen "addUserToMailinglists". De to andre aksjonene i steg 1 gjennomføres uten feil. Som man kan se fra tabell 2 i Figur 16 vil dette nå være status på operasjonen. Steg 2 og 3 i operasjonen har ikke blitt gjennomført fordi dette krever at alle aksjonene i de foregående stegene er gjennomført suksessfullt. Nå vil brukergrensesnittet gi brukeren mulighet til å prøve å gjennomføre operasjonen på nytt.




*Figur 17: Sekvensdiagram Feilhåndtering*

Når brukeren velger å forsøke å gjennomføre operasjonen på nytt, starter R&R-prosessen. Alle stegene i denne prosessen er illustrert ovenfor i Figur 17, og tallene i parentes videre i dette kapittelet refererer til pilene i denne figuren. Det første som skjer, er at brukergrensesnittet sender en forespørsel til API-et på endepunktet "/operation" (1). Forskjellen i et slikt R&R-tilfelle kontra en standard operasjon, er at brukergrensesnittet sender med UUID-en som ble generert første gangen operasjonen ble forsøkt utført. Når Flask API-et mottar en UUID på endepunktet "/operation", skjønner den at dette er en eksisterende operasjon. Da sender den UUID-en videre til en klasse i Operation Initializer som heter RebuildAndRetry (2).

R&R gjennomfører da flere kall mot andre deler av Ypsilon for å sette sammen en ny operasjon med alle aksjoner som ikke ble gjennomført første gangen. Som man kan se ut i fra tabell 2 i Figur 16 gjelder dette nå "addUserToMailingslists" (som feilet), samt "sendSMScredentials" (ikke utført) og "sendAdminConfirmation" (ikke utført). R&R omgjør med andre ord operasjonen fra tabell 2 til tabell 3 i Figur 16. Først forespør R&R logging databasen (3) for å hente tidligere resultater på aksjonene i operasjonen, samt operasjonens offset i Kafka. Dette offsetet bruker den deretter til å hente ut metadata og

parametere til den aktuelle operasjonen fra Kafka (5 og 6). Denne informasjonen ligger som kjent fortsatt i Kafka som en tidligere melding fra første gangen operasjonen ble forsøkt utført. Deretter henter den alle aksjonene tilhørende denne operasjonen fra YAML-konfigurasjonsfilen for aksjoner (7).

Nå har R&R all informasjonen den trenger for å bygge opp en ny operasjon med alle aksjonene som enten feilet eller ikke fikk sjansen til å bli gjennomført ved forrige forsøk. Når den nye operasjonen er bygget opp, sender R&R den videre til Kafka (9). Pil 9 i Figur 17 tilsvarer pil 4 i Figur 13. Herfra oppfører den nye retry-operasjonen seg akkurat som en ordinær operasjon. Den resterende delen av en R&R-prosess er altså identisk med en standard operasjon.

## 4.8 GET-requests mot eksterne systemer

Noen ganger har man som bruker behov for å hente ut informasjon fra et eksternt system og få dette fremvist i brukergrensesnittet. Det kan eksempelvis være en GET-request om å hente alle eksisterende brukere i et eksternt system. Disse requestene går ikke gjennom Kafka, men direkte fra Flask API-et og til det aktuelle eksterne systemet. Hovedgrunnen til dette er at Kafka ikke er spesielt godt egnet for denne typen GET-requests ettersom Kafka er et meldingskø-system hvor informasjonsflyten bare går i en retning fra klienten og til de eksterne systemene.

## 4.9 Utvidelse mot nye eksterne systemer

Et sentralt funksjonelt krav i denne oppgaven er at administratorer av Ypsilon enkelt skal kunne utvide mot nye eksterne systemer. Hvert eksterne system knyttes som kjent sammen med Ypsilon gjennom consumere som lytter på bestemte topics i Kafka. Denne må lages for hvert nye eksterne system som skal integreres. Consumer-kodene inneholder hovedsakelig tre elementer. 1. En consumer-modul som henter meldinger fra valgte topics i Kafka. 2. Funksjon som sender forespørsler mot sitt eksterne system med data fra Kafka. 3. En producer-modul som poster responsen fra det eksterne systemet inn igjen i Kafka. Både punkt 1 og 3 er like for alle consumere, så dermed er det bare punkt 2 som må kodes forskjellig fra system til system.

Topicet som en consumer lytter på, som da samtidig er navnet på aksjonen den utfører, må defineres i YAML-konfigurasjonsfilene. Her må parametere som trengs for å gjennomføre aksjonen defineres. Da vil denne aksjonen kunne bli lagt til i hvilken som helst ønskelig operasjon.

Kort oppsummert vil det kun være to nødvendige modifikasjoner for å legge til et nytt eksternt system: 1. Kode en ny consumer som snakker med det eksterne systemet. 2. Oppdatere konfigurasjonsfilene med den nye aksjonen som har blitt tilført systemet.

# 5. Diskusjon

I dette kapittelet vil vi diskutere arkitektur-valg, teknologier og bruksområder ved produktet vårt. På denne måten prøver vi å drøfte og svare på hvordan Ypsilon svarer på problemstillingen og HelseCERT sine behov. Dette gjør vi ved å først overordnet gå gjennom problemstillingen og de viktigste kravene i 5.1. Deretter går vi mer i detalj på implementeringen av løsningene på systemkravene i 5.2. Videre i kapittelet drøfter vi også rundt egne erfaringer, opplevelser og arbeidsprosessen i dette prosjektet.

## 5.1 Eventbasert tilnærming hos HelseCERT

En viktig motivasjon for HelseCERT med denne oppgaven har vært å finne ut mulighetene for et eventbasert system som en løsning på dagens utfordringer rundt de eksterne systemene. Som beskrevet i 1.1 Problemstilling, har HelseCERT i dag mange eksterne systemer som de ansatte manuelt må gjennomføre operasjoner mot. Dette er både tungvint, vanskelig og sårbart for menneskelige feil. Man må ofte bruke mange forskjellige egenutviklede skript og det kreves ofte dyp forståelse av de eksterne systemene for å kunne bruke disse. Problemstillingen har altså gått ut på å designe, implementere, og evaluere en eventbasert tilnærming som på best mulig måte løser disse utfordringene.

Et sentralt krav i problemstillingen er at Ypsilon skal knytte eksterne systemer sammen gjennom ett og samme brukergrensesnitt. Brukeren av systemet skal kunne få fortløpende tilbakemeldinger på hvordan aksjoner mot de eksterne systemene går. Det skal videre være enkelt å legge til og fjerne eksterne systemer, altså høy grad av interoperabilitet og utvidbarhet.

Et annet systemkrav, som har vist seg å gjøre denne oppgaven mer utfordrende fra en eventbasert synsvinkel, er at noen operasjoner må skje i en stegvis rekkefølge. Dette betyr altså at noen aksjoner i operasjonen må skje etter at andre er fullførte. En "out-of-the-box" eventbasert programvare, som Kafka i vårt tilfelle, har ikke all nødvendig funksjonalitet for å dekke dette systemkravet på en god måte. Det har dermed vært nødvendig for oss å utvikle flere komponenter rundt Kafka for å lage et system med all ønsket funksjonalitet. Dette kommer vi nærmere inn på i 5.3 Modifisering av en klassisk eventbasert tilnærming.

Vi mener Ypsilon definitivt har potensiale til å bli videreutviklet, tilpasset og tatt i bruk av vår oppgavestiller. Det som virker tydelig er at vårt system vil løse flere av problemene til HelseCERT og vil kunne effektivisere arbeidshverdagen betraktelig sammenlignet med dagens situasjon. Det er også sannsynlig at det finnes andre arkitekturer og løsninger som også vil kunne løse dagens problemer, men dette har ikke vært en del av vår problemstilling å finne ut av. Det kan også godt være at det må gjøres noen endringer på Ypsilon for at det skal fungere optimalt hos HelseCERT. Vi mener uansett at mange av tankene og løsningene vi har kommet frem til som svar på problemstillingen vil være verdifulle for HelseCERT.

## 5.2 Tekniske løsninger på systemkrav

Som beskrevet i 3.2 Iterativ utvikling som utviklingsmetodikk startet vi med å lage en *vertical slice* av Ypsilon som vi jobbet ut ifra og kontinuerlig forbedret. Det vil si at vi lagde et system med kun nødvendig funksjonalitet for å demonstrere den mest grunnleggende prosessflyten i et eventbasert system. Dette innebar at man kunne sende en melding fra et brukergrensesnitt til et API, videre til et topic i Kafka og ut til et eksternt system som lyttet på dette topicet.

Gjennom den iterative utviklingsprosessen testet vi denne tidlige versjonen for funksjonalitet opp mot de funksjonelle og ikke-funksjonelle kravene i kapittel 4.2, og fant da ut at denne versjonen av Ypsilon hadde flere mangler og utfordringer. Disse utfordringene gikk på at systemet var for lite fleksibelt, for lite responsivt og at vi ikke hadde tilrettelegging for stegvise operasjoner. Videre i dette delkapittelet beskriver vi løsningene vi kom frem til for hvert av disse problemene. Disse implementeringene er de mest signifikante endringene vi utarbeidet gjennom arbeidet med Ypsilon. Hele denne prosessen endte til slutt opp i et endelig system som er detaljert beskrevet i kapittel 4. Resultat.

### 5.2.1 Respons tilbake til bruker

For det første hadde første versjon av Ypsilon ingen løsning for å gi respons tilbake til brukeren om hvordan aksjonene i operasjonen gikk. Consumerne leste en melding i et topic, og utførte videre aksjonen mot sitt eksterne system uten å sende noen respons noe sted. Brukeren visste altså ikke hvordan en operasjon hadde gått uten å manuelt gå inn og sjekke i de eksterne systemene. For å løse dette problemet fant vi ut av at consumerne på en eller annen måte måtte lagre responsene fra de eksterne systemene et sted. Dette førte fram til utviklingen av et logging-system i en logging database. Brukergrensesnittet kan dermed hente responser fortløpende fra denne databasen.

### 5.2.2 Rekkefølge på aksjoner

Et annet viktig funksjonelt krav i problemstillingen som manglet i førsteutkastet var at aksjoner i en operasjon skal kunne skje i en bestemt rekkefølge. Alle consumerne ville opprinnelig lese av en operasjon så fort en ny melding kom inn og umiddelbart utføre en aksjon mot sitt eksterne system. Dette skjedde så fort consumeren var ledig og de tok ikke hensyn til avhengigheter mellom hverandre. Generelt sett i en eventbasert systemarkitektur er dette helt vanlig. Det kan eksempelvis være værdata fra en målestasjon som kommer inn kontinuerlig i et topic i Kafka. Da kan det være mange eksterne systemer og mikrotjenester som behandler denne værdataen helt uavhengig av hverandre.

I HelseCERT sine arbeidsoppgaver kreves det derimot noen ganger en stegvis tilnærming, og ikke bare en fri og uavhengig konsumering av meldinger hos consumerne. I "ADD USER"-eksempelet vil man trenge å sende ut en SMS med et engangspassord til kunden som nylig har fått opprettet en bruker. Dersom SMS-tjenesten lyttet på samme topic uavhengig av de andre tjenestene, ville man risikere tilfeller der en SMS blir sendt ut til tross for at andre systemer feilet med sin oppgave.

Dette vil være et uønsket scenario for HelseCERT ettersom en sluttbruker eksempelvis vil få påloggingsdetaljer eller annen informasjon som er ugyldige eller feilaktige.

Dette problemet løste vi gjennom en form for organiserings-komponent som orkestrerer og holder styr på hvilke aksjoner i en operasjon som er klare til enhver tid. Programmet som ble utviklet valgte vi å kalle Ordering Consumer. Grunnen til at denne komponenten var nødvendig var fordi et eventbasert system ikke har noen iboende egenskaper som sørger for at aksjoner blir gjennomført stegvis. Dette er noe man må implementere selv, og OC-komponenten ble vår løsning på dette.

### 5.2.3 Generisk løsning

Første versjon av Ypsilon var lite utvidbar og generisk. Uten en generisk konfigurasjonsfil med info om aksjoner og operasjoner krevdes mye kode-endringer både i brukergrensesnittet og i Ypsilon for å legge til nye eksterne systemer. Behovet for en standardisert beskrivelse av en operasjon og dens aksjoner førte fram til en løsning hvor en fil kunne fungere som en mal for de forskjellige operasjonene. En slik fil vil gjøre det lettere for en utvikler eller drifter av systemet å utføre endringer og legge til nye operasjoner. Implementeringen av en slik fil beskrives i kapittel 4.5.2 API og initiering av operasjoner.

## 5.3 Vår modifisering av en klassisk eventbasert tilnærming

Underveis i utviklingen av Ypsilon måtte vi gjøre noen implementasjoner som ikke er så vanlige i et eventbasert system. Blant annet implementerte vi en OC-komponent. Kort fortalt orkestrerer OC-komponenten stegvise operasjoner. For eksempel kunne ikke alle consumerne lenger lytte på de samme topicene, men *hver aksjon* fikk nå sitt eget topic i Kafka. Dette gjorde det mulig for Ypsilon å sende ut dedikerte aksjoner til kun de riktige consumerne. I en klassisk eventbasert arkitektur er det flere consumere som lytter på de samme, mer generelle meldingskøene. I en slik arkitektur er det ikke tilrettelagt for stegvise operasjoner. Man kan dermed påstå at løsningen vi endte opp med, på et rent teknisk plan, er litt uvanlig praksis i en eventbasert arkitektur.

Man kan si at Ypsilon er et eventbasert system med noen restriksjoner og kontrollmekanismer gjennom OC-komponenten som skiller det fra et klassisk eventbasert system. Disse implementasjonene er dog noe vi ser på som nødvendige for å kunne dekke det funksjonelle kravet om stegvise operasjoner. Hvis man hadde sett bort ifra dette funksjonelle kravet kunne vi potensielt ha fått mer ut av funksjonaliteten til meldingskø-programvaren, og det kunne blitt et mindre komplekst system.

Dersom man derimot ser mer på den overordnede logikken i Ypsilon og hvordan en bruker interagerer med det, kjenner man igjen trekkene ved et eventbasert system. Én handling fra brukergrensesnittet kan påvirke mange eksterne systemer, uten at disse er klar over hverandre. Heller ikke brukeren, som sender operasjonen, trenger å være klar over hvilke systemer han/hun interagerer med. Det at et system er eventbasert er et

nokså bredt begrep, og ingen systemer er helt like og mange vil ha sine egne modifikasjoner.

Vi mener at det funksjonelle kravet som OC-en løser, er med på å gjøre vår løsning mer unik og interessant også for andre bedrifter. Da vi begynte med oppgaven var vi ikke klar over at stegvise operasjoner normalt sett ikke var tilrettelagt for i et eventbasert system. Det er grunn til å tro at andre bedrifter heller har valgt andre løsninger for lignende problem tidligere, og at vår oppgave dermed kan være en spennende måte å løse denne utfordringen på. I alle fall mener vi at løsningen vår har løst denne utfordringen hos HelseCERT på en god måte.

## 5.4 Bruk utenfor HelseCERT

### 5.4.1 Generelt bruksområde

Det er naturlig å anta at problemet til HelseCERT, hvor man har mange krevende distribuerte systemer å operere, er et vanlig problem. Man kan se for seg at problemet kan oppstå når en virksomhet utvider hvilke tjenester de leverer til sine kunder. Da vil virksomheten få flere systemer å forholde seg til. HelseCERT sine interne systemer er et eksempel på dette. Kjerneegenskapen til Ypsilon er å kunne knytte distribuerte systemer sammen gjennom et modifisert eventbasert system. Denne egenskapen er derfor dagsaktuell for flere andre virksomheter. Ypsilon kan være løsningen på denne utfordringen, også for andre bedrifter. Det er fordi vi har laget et system som er fleksibelt og utvidbart når det kommer til å legge til nye eksterne systemer. Dermed vil det være en simpel prosess å implementere Ypsilon hos andre bedrifter og integrere dette med sine systemer.

Som et eksempel på et bruksområde kan man se på et fiktivt charterselskap Napollo. Her vil det være mulig for kundene å bestille turer med både fly og hotell til en reise. Napollo eier hverken hotell eller opererer fly selv, men de benytter seg av andre leverandører i sine bestillinger. Når en kunde skal bestille en reise med både fly og hotell vil Napollo måtte bestille flybilletten og hotellet fra andre selskap. Det er derfor helt essensielt for kunden at Napollo faktisk får bekreftet at disse bestillingene går gjennom hos de eksterne selskapene før kunden blir trukket pengene og får bekreftelsesmelding. Dette er et problem som Ypsilon vil kunne håndtere ved å gjennomføre en stegvis operasjon. Ypsilon kobles sammen med de eksterne systemene og vil kunne garantere at pengene til kundene kun trekkes dersom bestillingen i de eksterne systemene er godkjent. Hele bestillingen vil oppleves som én operasjon initiert av ett tastetrykk for brukeren.

### 5.4.2 Mindre egnede bruksområder

Infrastruktur med distribuerte systemer der aksjoner ikke er avhengige av hverandre vil være et mindre egnet bruksområde for Ypsilon. Det vil fortsatt fungere helt fint, men det vil være et system som er mer komplisert enn nødvendig. Eventbaserte systemer fungerer generelt sett bra hvis du har mange eksterne systemer, men om disse ikke er avhengige av hverandre trenger man ikke all funksjonaliteten Ypsilon tilbyr. Da kan det

være mer hensiktsmessig å gå for en vanlig eventbasert tilnærming da det vil være mindre komplekst og man vil kunne utnytte funksjonaliteten til meldingskøen bedre. I et rent eventbasert system vil det for eksempel være lettere å legge til en ny consumer for et nytt eksternt system og så iterere gjennom alle tidligere meldinger i denne køen. Det er fordi alle tidligere eventer vil ligge i samme meldingskø, og ikke i hver sin kø slik som i Ypsilon.

## 5.5 Videreutvikling av prosjektet

### 5.5.1 Forbedring av responser til brukergrensesnitt

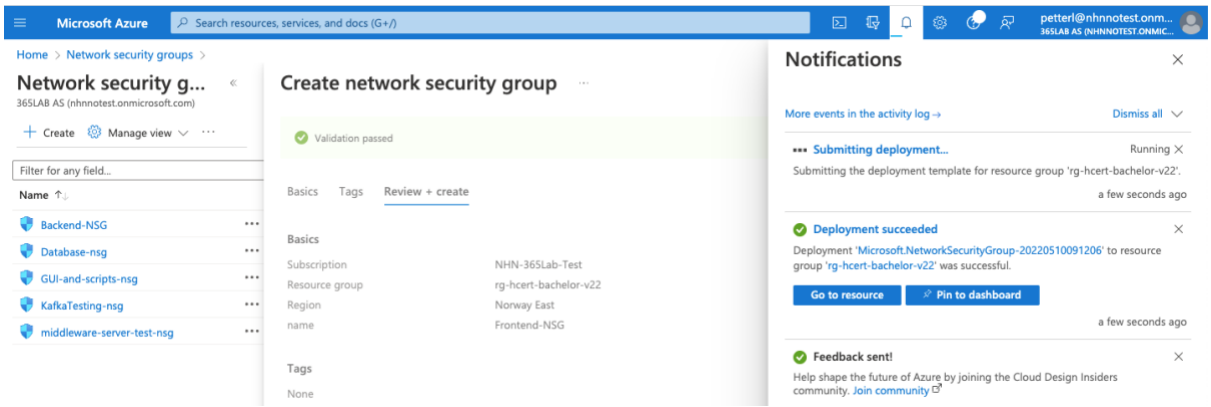
I delkapittel 4.6 Prosessflyten i systemet beskrev vi en form for kontinuerlig polling for å forklare hvordan brukergrensesnittet hentet oppdateringer på operasjoner. Kontinuerlig polling er i utgangspunktet ofte en lite effektiv måte å håndtere slike scenarioer på. Dette er fordi man hele tiden sender forespørsler til API-et og videre til databasen uavhengig om den har nye data eller ikke. Dette kan være problematisk for maskinvaren man bruker da det krever relativt stor ressursbruk både med tanke på nettverkskapasitet og prosessorkraft. En simpel løsning på dette er å sende færre forespørsler, for eksempel bare hvert tiende sekund, men dette skaper et nytt problem med en forsinkelse i "sanntids"-fremvisningen i brukergrensesnittet.

En vanlig løsning som ofte er brukt i webapplikasjoner er protokollen *Websocket*. Denne lar klienten opprette en full-dupleks forbindelse med serveren slik at serveren selv kan respondere til klienten når det har kommet ny informasjon i databasen, i stedet for scenarioet hvor klienten må sende kontinuerlige forespørsler. Dette er fullt mulig å gjennomføre i Flask også med for eksempel biblioteket Socket.IO. [23]

Grunnen til at vi ikke har valgt å ha fokus på å implementere denne teknologien er fordi Ypsilon mottar forespørsler fra ikke bare et web-grensesnitt, men også fra et Command Line Interface, CLI. Når man bruker Python-skript ønsker man å holde skriptene så simple som mulig slik at brukervennligheten er best mulig. Implementering av Websockets i skript krever ekstra konfigurering av eksterne moduler på maskinene skriptene kjøres fra. Å implementere Websockets for web-grensesnittet er enklere fordi alle standard nettlesere har støtte for Websocket. Derfor er Websocket noe man kunne prioritert på web-grensesnittet hvis ønskelig i fremtiden.

### 5.5.2 Forbedring av GUI

Som antydet i kapittel 1.3 Avgrensing av oppgaven har arbeid med GUI blitt nedprioritert. Hovedårsaken til dette er at dette ikke har vært av så stor interesse for arbeidsgiver HelseCERT og følgelig ikke vært en sentral del av vår problemstilling. Vi har likevel utarbeidet et GUI i TypeScript med rammeverket React som dekker kjerneoppgavene med et brukergrensesnitt til systemet vårt, nemlig sende operasjoner og motta fortløpende responser. Her er det selvsagt flere områder med rom for forbedringer. Designløsninger og generell grafisk utforming i GUI-et har nesten ikke blitt tildelt noe tid i vårt prosjekt, så her bør det legges ned en del videre arbeid.




*Figur 18: Skjermutklipp Azure Portal*

Azure Portal er et brukergrensesnitt vi mener har mange gode og brukervennlige løsninger som vi hadde tatt inspirasjon fra dersom vi skulle brukt mer tid på vårt eget GUI i fremtiden. Spesielt notifikasjonsløsningen i Figur 18 ovenfor ligner veldig på hvordan vi selv ville fremvist operasjoner og andre interaksjoner i vårt system. Denne løsningen fungerer godt sammen med asynkrone forespørsler, slik vi har i vårt system, ettersom man kan jobbe videre med andre oppgaver i GUI-et og samtidig enkelt få notifikasjoner og status på pågående operasjoner.

### 5.5.3 Automatisering av systemutvidelse

Ett av områdene som ble viktigst for oss å forbedre underveis i utviklingen var å lage systemet generisk og så enkelt som mulig å utvide med nye eksterne systemer. Dette var også et viktig ikke-funksjonelt krav i problemstillingen vår. I vårt endelige produkt er det fullt mulig å utvide med så mange eksterne systemer som man ønsker og har, men det krever naturligvis at man går teknisk inn i enkelte deler av koden. Man vil måtte legge til aksjonen og tilhørende parametere i YAML-konfigurasjonsfilene, og man må kode en ny consumer i Python som har kommunikasjon mot et eksternt system. Dette har vi beskrevet mer detaljert i kapittel 4.9 Utvidelse mot nye eksterne systemer.

Selve koden som sender forespørsler til eksterne systemer vil ofte variere og vil kreve noe manuell koding hver gang. Men prosessen med å legge til aksjonen inn i YAML-filen og opprette consumer-filen med de delene som er like for alle consumere og vil være mulig å automatisere. I bunn og grunn trengs det bare skripts som modifiserer eller oppretter en fil. En løsning kan være å utvikle en form for admin-GUI. Her kan man ha funksjonalitet for å opprette nye aksjoner mot nye eksterne systemer kun ved å plotte inn navn på aksjon, parametere og selve koden for å sende forespørselen mot det eksterne systemet. Ved å sende denne informasjonen til et endepunkt i mellomvaren vil man kunne ha en generisk løsning som fikser resten av implementeringen i systemet vårt. I et slikt admin-GUI vil det også naturligvis være mulig å implementere andre funksjoner som endrer andre deler av mellomvaren vår. Eksempelvis kan dette være å endre, opprette eller slette operasjoner som er definert i YAML-filen i mellomvaren.

### 5.5.4 Interne HelseCERT-brukere og forbedret sikkerhet

Innlogging og opprettelse av interne brukere for å bruke systemet vårt var utenfor omfanget til denne oppgaven. Så lenge man har tilgang til det interne subnettet som løsningen vår er ment å kjøre på, har man også tilgang til å nå GUI-et i en nettleser og API-et i mellomvaren gjennom HTTP-forespørsler. I GUI-et finnes det naturligvis muligheter for intern brukerinnlogging. Dette kan man gjøre ved å lage en innloggingsportal og opprette et sett med interne brukskontoer med passord som saltes, hashes og lagres på en lokal database. Meget sannsynlig er det nok dog at ansatte i HelseCERT allerede har en løsning for autentisering mot det eksisterende brukergrensesnittet de bruker i dag. Poenget er uansett at en eller annen form for autentisering i Ypsilon er et helt essensielt sikkerhetsgrep som må gjøres for at systemet skal kunne brukes i produksjon.

### 5.5.5 Mer sømløs og eventbasert logging

I Ypsilon er det sånn at alle database-kall mot logging databasen aktivt er kodet inn i systemet fra der kallet skjer. Det betyr at Operation Initializer og OC har kode som initierer logging til databasen. Kallene det er snakk om her er av typen INSERT eller UPDATE. Som beskrevet i 4.5.4 Logging database går disse kallene via et dedikert topic i Kafka og videre til en database-consumer som gjør selve kallet.

All prosessflyt i systemet vårt går uansett gjennom Kafka. Derfor er det nærliggende å tro at database-consumeren kunne tatt seg av mye av det som har med logging å gjøre uten en aktiv forespørsel fra Operation Initializer og OC. Det vil si at database-consumeren lytter på enkelte topics i Kafka, konsumerer eventer der og deretter oppdaterer databasen med informasjon i disse meldingene. Eksempelvis kunne database-consumeren lest at det kom inn en ny respons fra et eksternt system i "ActionState"-topicet og direkte oppdatert logging databasen med denne informasjonen.

Dette er en videreutvikling av Ypsilon som ideelt sett kunne forenklet noe av kompleksiteten til sentrale komponenter av systemet gjennom å utnytte litt mer av egenskapene et meldingskø-system har. Dette er noe som ville kunne være utfordrende å implementere i praksis, selv om logikken er ganske rett fram. I tillegg oppdaget vi denne forbedringen for sent i forhold til tiden vi hadde igjen til innlevering. Dermed har ikke denne løsningen faktisk blitt implementert. Vi nevner det likevel i rapporten som et forbedringspotensiale for fremtiden til HelseCERT eller andre brukere av Ypsilon.

## 5.6 Valg av event strømningsteknologi

Som forklart i kapittel 3.3 Arbeid og utviklingsprosess brukte vi mye tid i starten av bachelorprosjektet på å lese oss opp på relevant teori rundt eventbaserte systemer. Noe av det vi brukte mest tid på etter hvert som vi fikk satt oss inn i fagstoffet var undersøkelser rundt hvilken teknologi vi skulle bruke til orkestreringen av meldingskøer.

De to teknologiene som vi så på som mest relevante var Apache Kafka og RabbitMQ, dels på grunn av egenskapene til produktene, men også mye fordi disse er noen av de mest populære tjenestene på markedet og derfor har en stor og god mengde

dokumentasjon tilgjengelig. Å finne de store forskjellene på hvordan disse ville passe inn i vårt system var krevende og forskjellene var heller ikke alltid like åpenbare.

På et generelt grunnlag anbefales Apache kafka i de tilfellene hvor man skal ha høy gjennomstrømming av data i systemet. Kafka kan håndtere en gjennomstrømming på flere millioner meldinger i sekundet. I tillegg til dette har Kafka også egenskapen til å lagre meldingene som ligger i meldingskøen. Dette er kjerneegenskapene til Kafka, og hvis disse er viktig er Kafka det soleklart beste valget.

RabbitMQ baserer seg på en litt annen logikk enn Kafka og har derfor også noen andre kjerneegenskaper. RabbitMQ er mest brukt i systemer hvor man trenger mer avansert ruting fra Message Broker-en og ut til de forskjellige consumerne. En annen viktig egenskap som RabbitMQ har er at det ikke er noen begrensning på størrelsen på en melding som går gjennom systemet. [24]

Når vi vurderte Kafka og RabbitMQ opp mot hverandre var det lagringsegenskapene til Kafka og meldingsstørrelsen til RabbitMQ som vi anså som de mest relevante hovedegenskapene inn i vårt system. Den automatiske lagringen i Kafka var ønskelig for å kunne logge operasjoner i systemet. Den ubegrensede meldingsstørrelsen i RabbitMQ kunne være ønskelig hvis man kom i situasjoner hvor man for eksempel måtte overføre filer mellom de distribuerte systemene. Den avgjørende faktoren under dette valget av teknologi var at HelseCERT allerede hadde erfaring med bruk av Kafka. Da det sto relativt likt mellom egenskapene til Kafka og RabbitMQ ble erfaringen til HelseCERT grunnen til at vi valgte Kafka som meldingskø i vårt system.

## 5.7 Alternative løsninger

Et nokså annerledes alternativ for HelseCERT kunne vært å ta i bruk en ferdig utviklet programvare som har noe av den samme funksjonaliteten som vi har utviklet gjennom Ypsilon. Microsoft Azure har en skybasert programvare som heter Azure Logic Apps. Microsofts egne dokumentasjon sier at "Azure Logic Apps is a cloud-based platform for creating and running automated workflows that integrate your apps, data, services, and systems.". [25] Dette høres veldig ut som noe som kunne løst mange av systemkravene i problemstillingen vår, og det kan det godt hende det kunne gjort. Det er imidlertid en problemstilling for en helt egen bacheloroppgave å utforske.

Det er derimot uansett noen betraktninger vi på forhånd kan gjøre oss om Azure Logic Apps i forbindelse med vår problemstilling. Den er ikke open source, noe som er et krav i HelseCERT med mindre man har kjøpt en lisens og opprettet en databehandleravtale hos den aktuelle tilbyderen. Bruken av skytjenester i HelseCERT har vært, og er, mye diskutert. Det er blant annet på grunn av GDPR ved lagring av kundedata, og per i dag kjører svært få av HelseCERTs tjenester i skyen. Azure Logic Apps er et omfattende system som også ville krevd opplæring og mye arbeid å etablere, sannsynligvis mer enn det som er formålstjenlig for problemene det ville løst i HelseCERT.

De ovennevnte utfordringene hos en tjeneste som Azure Logic Apps gjør at dette antakeligvis ikke er aktuelt for HelseCERT, i alle fall ikke på nåværende tidspunkt.

Apache Nifi er et noe lignende prosjekt utviklet av Apache Foundation. På deres egen hjemmeside beskrives det som følger: "Apache NiFi supports powerful and scalable directed graphs of data routing, transformation, and system mediation logic.". [26] Det er nærliggende å tenke at dette teknisk sett også kunne løst mye av det samme som Ypsilon gjør. I tillegg er dette open source og kan kjøres lokalt i HelseCERTs egen infrastruktur. Igjen blir spørsmålet om opplæring og implementering av dette systemet ville vært hensiktsmessig for bedriften. Det kan være, og er i alle fall en spennende hypotese som kunne vært diskutert blant systemutviklerne i HelseCERT.

## 5.8 Administrative resultater

### 5.8.1 Gruppearbeidet og arbeidsfordeling

I begynnelsen av semesteret definerte vi noen arbeidsfordelinger i visjonsdokumentet. Vi baserte denne arbeidsfordelingen på ulike områder hver av oss hadde et ekstra ansvar for. Dette ansvaret innebar å passe på tidsbruk, fremgang og eventuelle problemer, samt jevnlig oppdatere hverandre om dette. Vi ble enige om at Petter skulle være ansvarlig for UI og API, Jacob for den resterende mellomvaren og Wilhelm for backend og sluttrapporten. Denne fordelingen av ansvarsområder stemmer ikke nødvendigvis overens med hva vi faktisk jobbet med. Vi hadde hele tiden mye fokus på å inkludere hverandre i de forskjellige delene av utviklingen av systemet. Flere dager jobbet naturligvis alle på samme del av systemet, spesielt når store sentrale beslutninger måtte gjøres. Vi sitter igjen med en følelse av at dette har fungert veldig bra for oss. Ved at alle var med på å bidra på alle hovedområdene gjør at alle sitter igjen med en dyp forståelse av hele systemet, samt valgene som ble tatt underveis. Det er tydelig at hele gruppen har vært motiverte og engasjerte i arbeidet med oppgaven.

I tillegg til å ha mye erfaring med å jobbe sammen på tidligere gruppeprosjekt relatert til andre fag bor også hele gruppen sammen i kollektiv. Vi tror at dette har hatt en god fordel når det kommer til arbeidet med bacheloren. Vi kjenner hverandre godt og hverandres rutiner, i tillegg er det enkelt å holde en god flyt i kommunikasjon.

Fra starten av semesteret har vi hele tiden hatt gode arbeidsrutiner, og som regel sittet hver dag på kontorene til HelseCERT. Vi har jobbet sammen hver dag med unntak av ganger noen av oss har måtte jobbe remote. Dersom en har måtte jobbe remote har vi gjennomført videokonferanser og sørget for god kontakt. Dette har fungert sømløst.

### 5.8.2 Måloppnåelse

Gjennom egne opplevelser av sluttproduktet, så vel som tilbakemeldinger og signaler fra oppgavestiller HelseCERT, opplever vi å ha oppnådd en høy måloppnåelse på prosjektet. Dette inntrykket ble forsterket da vi tidlig i mai fikk muntlig tilbud om sommerjobb hos oppgavestiller hvor arbeidsoppgavene vil inkludere å implementere og videreutvikle Ypsilon inn i bedriftens infrastruktur. På det mer personlige plan føler vi at vi har lært masse om prosjektarbeid, det virkelige arbeidslivet og samarbeid generelt.

### 5.8.3 Utviklingsmetodikk og arbeidsprosess

Som beskrevet i kapittel 3. valgte vi å gå for en iterativ utviklingsmetodikk gjennom utviklingsprosessen. Dette er fordi vi tidlig i prosjektet følte at vi trengte en plan for hvordan vi sammen skulle jobbe på et så stort prosjekt. Da kom vi frem til en iterativ utvikling var hensiktsmessig. Hovedgrunnen til dette valget var at denne metoden gjør det enkelt å gjøre endringer underveis og man vurderer jevnlig produktet sitt. I en mer klassisk fossefallsmetode [27] må hele prosjektet være planlagt på forhånd og det er vanskeligere å gå tilbake for å gjøre endringer i ettertid.

Gjennom oppstarten at den iterative prosessen utviklet vi også, som nevnt i kapittel 3.2 Iterativ utvikling som utviklingsmetodikk, en vertical slice av systemet vårt som ble et utgangspunkt for videre utvikling av systemet. Dette fungerte godt fordi man gjennom hele prosessen beholdt en helhetlig oversikt over systemet, uten å ferdigstille én komponent om gangen.

Gruppen hadde ikke erfaring med iterativ utvikling som utviklingsmetodikk på en så strukturert og konsekvent måte. Vi er veldig fornøyde med hvordan metoden har fungert. I starten av utviklingsprosessen hadde vi lengre sykluser på rundt 2 uker, mens de varte rundt en uke mot slutten. Grunnen til at de ble kortere var både på grunn av økende kunnskap og effektivitet i arbeidet vårt, men også fordi endringene ble mindre omfattende og krevde rett og slett mindre tid.

Fordelen med å ta i bruk en iterativ utviklingsmetodikk har vært at vi effektivt har kunnet gjøre endringer på systemet ved å fokusere på mindre deler av utviklingen i hver syklus. Tidlig i utviklingen ble vi enige om hvilket funksjonaliteter som skulle utvikles og forsøksvis behandles gjennom én og samme syklus. Dette endret seg noe underveis, da det er vanskelig å forutsi nøyaktig hva som vil være mest utfordrende før man faktisk har startet å utvikle et produkt. Flere av syklusene sammenfaller godt med hvert av de store valgene beskrevet i kapittel 5.2 Tekniske løsninger på systemkrav. Dette viser at hver syklus førte fram til en konkret forbedring, og underbygger påstanden om at iterativ utvikling fungerte godt som arbeidsmetode for oss.

Gjennom hele semesteret har vi sittet mye på kontorene til HelseCERT. På kontoret har vi fått tilbudt faste plasser med god tilgang på egne grupperom og verktøy som personlige skjermer og whiteboard. Arbeidsforholdene våre har med andre ord vært optimale gjennom hele prosessen. Vi har hatt kontorplass bare noen titalls meter unna veilederen vår i HelseCERT, samt de øvrige ansatte på avdelingen. Ved å sitte sammen med resten av de ansatte på av avdelingen har vi fått en følelse av arbeidslivet. Gjennom dette har vi også plukket opp mye interessant kunnskap ved tilfeldige samtaler med ansatte og øvrig fagprat i fagmiljøet på kontoret.

# 6. Konklusjon

Gjennom denne bacheloroppgaven har vi gått fra en problemstilling, gjennom en teori- og planleggingsfase, videre til en utviklingsfase før vi til slutt sto med et egenutviklet system, Ypsilon. Vi er godt fornøyde med hele prosessen og alle de forskjellige aspektene ved arbeidslivet vi nå har fått bedre kjennskap til. Vi har utviklet et eventbasert integrasjonssystem for vår arbeidsgiver HelseCERT. Systemets hovedformål er å automatisere og integrere flere eksterne systemer sammen for å forenkle arbeidsflyten i arbeidshverdagen for de ansatte.

Som i de fleste utviklingsprosesser har flere mulige løsninger vært utforsket og drøftet i veien mot det endelige produktet. Det er som oftest "flere veier til Rom", og det er nok dermed også flere ulike måter denne problemstillingen kunne vært besvart på. Gjennom sluttproduktet vårt Ypsilon føler vi at vi har klart å dekke de forskjellige systemkravene på en god måte, uten å legge for mye eller lite vekt på noen av kravene.

I sluttfasen av bachelorprosjektet har vi sett på Ypsilon mer utenfra og prøvd å se det i en litt større kontekst. Dette for å best mulig kunne drøfte rundt produktet og hvorvidt dens funksjoner og egenskaper faktisk er verdifulle og nyttige for HelseCERT. Vi har også prøvd å se på hvordan deler av produktet vårt kan være nyttig også for andre bedrifter, og på den andre side i hvilke sammenhenger det er mindre egnet. Dette har gjort oss mer reflekterte rundt hva som er bra med Ypsilon, og hva som kanskje kunne vært gjort annerledes. I tillegg har vi utforsket videre på hvor produktet kan videreutvikles og forbedres ytterligere. I denne fasen har vi prøvd å være så objektive og selvkritiske som mulig, siden det tross alt er læringsprosessen som er det aller viktigste formålet med en bacheloroppgave. Hele denne refleksjonsdelen har vært svært lærerik for gruppen.

# 7. Referanser


[1] - HelseCERT [Internett]. Norsk Helsenett. [hentet 27. april 2022]. Tilgjengelig fra: https://www.nhn.no/Personvern-og-informasjonssikkerhet/helsecert

[2] - Making Sense of Chaos through Big Data [Internett]. Latize. [hentet 4. mai 2022]. Tilgjengelig fra: https://latize.com/making-sense-of-chaos-through-big-data/

[3] - Event-Driven Architecture [Internett]. Amazon Web Services. [hentet 20. april 2022]. Tilgjengelig fra: https://aws.amazon.com/event-driven-architecture/

[4] - Powered By [Internett]. Apache Kafka. [hentet 28. april 2022]. Tilgjengelig fra: https://kafka.apache.org/powered-by

[5] - What is Apache Kafka? [Internett]. Amazon Web Services. [hentet 18. mai 2022]. Tilgjengelig fra: https://aws.amazon.com/msk/what-is-kafka/

[6] - Johansson, L. Part 1: Apache Kafka for beginners - What is Apache Kafka? Cloud Karafka; 19. mars 2020. [hentet 24. januar 2022]. Tilgjengelig fra: https://www.cloudkarafka.com/blog/part1-kafka-for-beginners-what-is-apache-kafka.html

[7] - Gupta, L. HTTP Methods. Restful API Tutorial [Internett]. [oppdatert 11. desember 2021; hentet 10. mars 2022]. Tilgjengelig fra: https://restfulapi.net/http-methods/

[8] - An overview of HTTP [Internett]. [oppdatert 13. april 2022; hentet 25. april 2022]. Tilgjengelig fra: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview

[9] - What is an API? (Application Programming Interface) [Internett]. Mulesoft. [hentet 12. mars 2022]. Tilgjengelig fra: https://www.mulesoft.com/resources/api/what-is-an-api

[10] - Documentation page [Internett]. Flask [hentet 2. april 2022]. Tilgjengelig fra: https://flask.palletsprojects.com/en/2.1.x/

[11] - Microsoft Teams [Internett]. Microsoft. [hentet 2. mai 2022]. Tilgjengelig fra: https://www.microsoft.com/nb-no/microsoft-teams/group-chat-software

[12] - Google Drive [Internett]. Google. [hentet 2. mai 2022]. Tilgjengelig fra: https://www.google.com/drive/

[13] - Make your first Git commit [Internett]. GitLab. [hentet 2. mai 2022]. Tilgjengelig fra: https://docs.gitlab.com/ee/tutorials/make_your_first_git_commit.html

[14] - Hva er Azure? [Internett]. Microsoft. [hentet 2. mai 2022]. Tilgjengelig fra: https://azure.microsoft.com/nb-no/overview/what-is-

azure/?ef_id=CjwKCAjwve2TBhByEiwAaktM1BqeXYwz6NDS-XSdGvIjSyXo5pjOf6qEy46e4UhR08IDFLZWxNOb2RoC4o0QAvD_BwE%3AG%3As&OCID=AID2200230_SEM_CjwKCAjwve2TBhByEiwAaktM1BqeXYwz6NDS-XSdGvIjSyXo5pjOf6qEy46e4UhR08IDFLZWxNOb2RoC4o0QAvD_BwE%3AG%3As&gclid=CjwKCAjwve2TBhByEiwAaktM1BqeXYwz6NDS-XSdGvIjSyXo5pjOf6qEy46e4UhR08IDFLZWxNOb2RoC4o0QAvD_BwE

[15] - Integrated development environment. [Internett]. Stack Overflow. [hentet 4. mai 2022]. Tilgjengelig fra: https://insights.stackoverflow.com/survey/2021#section-most-popular-technologies-integrated-development-environment

[16] - Wikipedia. Lint (software) [Internett]. Wikipedia. [hentet 27. april 2022]. Tilgjengelig fra: https://en.wikipedia.org/wiki/Lint_(software)

[17] - PEP 8 – Style Guide for Python Code [Internett]. Python.org; 5. juli 2001. [oppdatert 11. mai 2022; hentet 25. april 2022]. Tilgjengelig fra: https://peps.python.org/pep-0008/

[18] - Francino, Y. Iterative Development [Internett]. TechTarget. [oppdatert november 2011; hentet 29. april 2022]. Tilgjengelig fra: https://www.techtarget.com/searchsoftwarequality/definition/iterative-development

[19] - Vertical slices [Internett]. DevIQ. [hentet 10. mai 2022]. Tilgjengelig fra: https://deviq.com/practices/vertical-slices

[20] - Woodfine, G. How to Implement Vertical Slice Architecture [Internett]. GaryWoodfine; 2. april 2022. [hentet 10. mai 2022]. Tilgjengelig fra: https://garywoodfine.com/implementing-vertical-slice-architecture/

[21] - Jansen, A. Kravdefinisjon og kravspesifikasjon [Internett]. NDLA. [oppdatert 16. november 2018; hentet 9. mai 2022]. Tilgjengelig fra: https://ndla.no/nb/subject:1:9d6d3241-014d-4a5f-b0bc-ae0f83d1cd71/topic:3:193104/topic:2:123578/resource:1:123591

[22] - Holte, HC. Bringedal, T. Høringssvar rapporten Felles IKT-arkitektur i offentlig sektor [Internett]. Oslo: Direktoratet for forvaltning og IKT (DIFI); 18. september 2008. [hentet 5. mai 2022]. Tilgjengelig fra: https://www.regjeringen.no/globalassets/upload/fad/vedlegg/ikt-politikk/faos/horing_faos_difi.pdf

[23] - Introduction [Internett]. Socket.IO. [oppdatert 5. mai 2022; hentet 16. mai 2022]. Tilgjengelig fra: https://socket.io/docs/v4/

[24] - RabbitMQ [Internett]. RabbitMQ. [hentet 24. januar 2022]. Tilgjengelig fra: https://www.rabbitmq.com

[25] - Microsoft. What is Azure Logic Apps? [Internett]. Microsoft; 9. februar 2022. [hentet 4. mai 2022]. Tilgjengelig fra: https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-overview

[26] - Apache NiFi [Internett]. Apache NiFi. [hentet 13. mai 2022]. Tilgjengelig fra: https://nifi.apache.org/

[27] - Holte Academy. Agil eller fossefall – hvilken metode skal vi velge? [Internett]. Holte Academy. [hentet 13. mai 2022]. Tilgjengelig fra: https://www.holteacademy.no/agil-eller-fossefall-hvilken-metode-skal-vi-velge/