

Hovedopgave

hos Efio ApS

Andreas Zoega Vesterborg Vikke
cph-av105@cphbusiness.dk

Martin Eli Frederiksen
cph-mf237@cphbusiness.dk

Asger Hermind Sørensen
cph-as466@cphbusiness.dk

Vejleder: Palle Bech



Hovedopgave for Datamatiker 5. semester

Datamatiker
Cphbusiness Lyngby
Danmark
11. januar 2021

Forord

Denne rapport er udarbejdet af undertegnede i forbindelse med afslutning på vores uddannelsesforløb som datamatikere fra CPHbusiness (Lyngby) og på baggrund af praktik- og hovedopgaveforløb hos virksomheden Efio. Rapporten omhandler udviklingen af applikationen Sterling og dokumentation af hovedopgaveperioden løbende fra d. 1/11-20 til d. 11/1-21 samt udviklingsperioden af applikationen, der strækker sig fra d. 9/11-20 til d. 14/12-20. Formålet med udviklingen af Sterling kommer som følge af en problematik i virksomheden, der omhandler medarbejdernes manglende arbejdstidsregistrering. Applikationen er udviklet med øje for simplicitet, brugervenlighed og med en ambition om, at den vil blive brugbar for Efio eller andre virksomheder i fremtiden.

I denne rapport er enkelte elementer udviklet i praktikperioden, men har haft så stor betydning for hovedopgavens udvikling, at de bliver præsenteret som en del af det udarbejdede materiale fra hovedopgaveperioden. For at dokumentere applikationens stadier, har vi kort redegjort for, hvordan applikationen virkede efter den fuldendte praktikperiode.

Indhold

1. Indledning	3
2. Problemformulering	4
3. Praktikperioden	5
4. Forretningsmodel	10
4.1. Efio	10
4.2. Kundesegment	10
4.3. Slack	12
4.4. Forventet- og opnået forretningsværdi	12
Produkt	14
5. Kravspecifikation	14
5.1. Scope	14
5.2. Funktionelle krav	14
5.3. Ikke-funktionelle krav	16
5.4. Dokumentation af kravene	17
5.5. Vurdering	19
6. Valg af teknologier	20
6.1. Slack	20
6.2. AWS	20
6.3 Python 3.8	25
6.4 Vurdering	26
7. Valg af arkitektur	27
7.1 Backend	27
7.2 Trelags-arkitektur	29
7.3 Microservice	30
7.4 CI/CD-pipelinee	31
7.4 Vurdering	32
8. Design	33
8.1 Brugergrænseflade	33
8.2 Databasedesign	36
8.3 Patterns	38
8.4 Vurdering	38
8. Implementering	39
9.1 Mappestruktur	39
9.2 Pylint	40
9.3 Dependency Injection Factory for Boto3SNS	42

9.4 Time/Description input box decoder	43
9.5 Delete Message in Slack	45
9.6 Create Work Time Hint	47
9.6 Vurdering	48
10. Test	49
10.1 Software Kvalitet	49
10.2 Kodestandarder	49
10.3 Test Driven Development	50
10.4 Software Testing	51
10.5 Vurdering	52
11. Evaluering af produktet	54
Proces	56
12. Projekt set-up	56
13. Valg af udviklingsmetode	57
13.1 Udviklingsmetoder	57
13.2 Vurdering	60
13.3 Radar chart	61
13.4 Verificering	62
14. Planlægning	63
14.1 Sprint overblik	63
14.2 Vurdering	65
15. Dokumentation af processen	66
15.1 Scrum	66
15.2 Definition of ready	67
15.3 Branch strategi	68
15.4 Definition of done	70
15.5 Brugerfeedback	70
15.6 Extreme Programming	71
16. Refleksion over processen	72
16.1 Scrumroller	72
16.2 Retrospektive møder	72
16.3 Sprint review	72
16.4 Definition of ready	72
16.5 Branch strategi	73
16.6 Definition of done	73
16.7 Brugerfeedback	73
16.8 Extreme Programmering	73
17. Evaluering af processen	75
18. Konklusion	76
19. Litteraturliste	77
20. Bilag	79
20.1 Bilag 1	79
20.2 Bilag 2	82

20.3 Bilag 3	84
--------------	-------	----

1. Indledning

I Efio var det en udfordring for konsulenterne at huske, at registrere deres arbejdstid. Vi har derfor i samarbejde med virksomheden udviklet applikationen *Sterling*, som er en open source-tidsregistreringsapplikation til slack. Udviklingen af denne applikation strakte sig over en periode fra d. 17/08-20 til d. 23/10-20 samt fra d. 09/11-20 til d. 11/12-20 for hhv. vores praktik- og hovedopgaveperiode hos Efio.

Denne rapport er opbygget således, at vi starter med en præsentation af de givne problemstillinger i forbindelse med hovedopgaven efterfulgt af en kort redegørelse af elementerne, der blev udviklet i praktikperioden. Årsagen til vi vælger at redegøre kort for det udførte arbejde i praktikperioden, er for at tydeliggøre og skelne mellem det arbejde, der blev udført i hhv. praktik- og hovedopgaveperioden. Dernæst dykker vi ned i, hvilken betydning applikationen har haft for den pågældende virksomhed for at kunne forstå, hvorfor applikationen overhovedet blev udviklet i første omgang. Herefter præsenteres en gennemgang af produktet for at synliggøre de forskellige lag, der ligger i applikationen. I gennemgangen af produktet kommer vi ind på: *Kravspecifikation, valg af teknologier og arkitektur, design, implementering, test og evaluering*. Derefter gennemgår vi udviklingsprocessen af applikationen, hvor vi redegør og dokumenterer for hele processen, vi har været igennem som udviklingshold, for at skabe en forståelse for de valg, vi har truffet undervejs. Her kommer vi ind på: *Projekt set-up, valg af udviklingsmetode, planlægning, dokumentation af og refleksion over processen og evaluering*. Elementer med stor betydning for udviklingen af hovedopgaven, som reelt set er udviklet i praktikperioden, er flyttet til gennemgangen af udviklingsprocessen, ettersom vi ikke havde mulighed for at udvikle hovedopgaven uden disse elementer. Afslutningsvist vil vi konkludere på det samlede projekt.

Målgruppen for denne rapport er vores underviser og censor, hvilket vil sige at vi antager, at der er en bred forståelse af de præsenterede emner. Da applikationen er udviklet som open source, er den frit tilgængelig for alle og derfor har der decideret ikke været behov for at overdrage applikationen til Efio.

2. Problemformulering

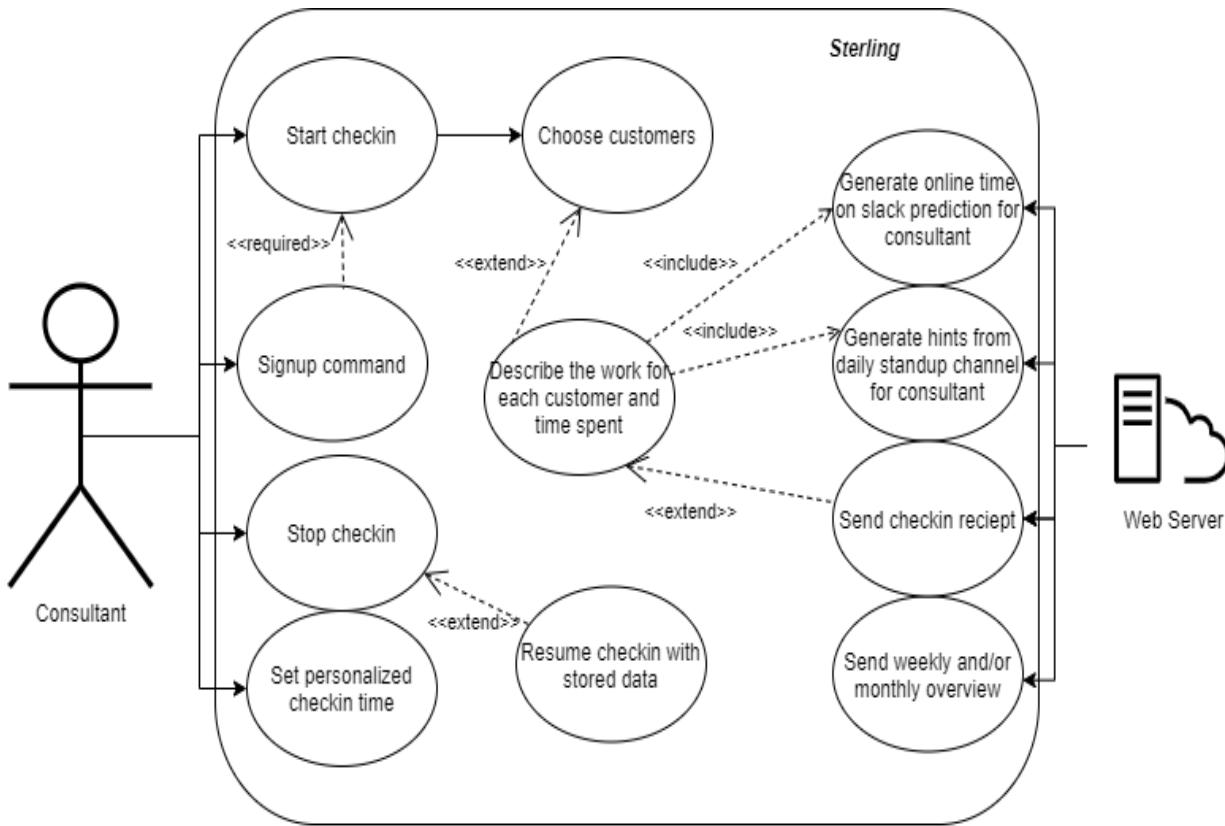
Vi vil gerne undersøge, om det er muligt for os som studerende med vores nuværende viden, erfaringer og kompetencer at løse følgende problemstillinger:

- Er det muligt for os at identificere og dokumentere alle krav?
- Hvordan sikrer vi at kravene hjælper os til at opfylde virksomhedens mål til systemet?
- Er det muligt på baggrund af de identificerede krav at planlægge og gennemføre en god systemudviklingsproces?
- Hvilke teknologier er bedst egnet til udvikling af produktet?
- Hvilken arkitektur er fordelagtig for produktet?
- Hvordan designer vi produktet med en tilstrækkelig brugervenlighed for de enkelte konsulenter?
- Hvordan tester vi produktet i et bredt nok omfang?

3. Praktikperioden

I dette afsnit vil vi redegøre for applikationen, vi udviklede i praktikperioden, samt dokumentere udviklingen i form af billeder. Vi vil dog undlade at redegøre for de valg, der blev truffet i praktikperioden, som havde stor betydning for vores hovedopgave. Disse valg vil i stedet blive redegjort for i et senere afsnit senere i rapporten.

Under praktikperioden var vores overordnede problem: "Hvordan kan vi hjælpe praktikvirksomhedens medarbejdere med at huske den daglige timeregistrering". Dette problem blev valgt af os som praktikgruppe ud fra fire givne problemstillinger, som vores praktikvirksomhed havde opstillet for os. For at danne et overblik over, hvordan applikationen så ud efter det afsluttede praktikforløb ses nedenfor et use case diagram over applikationen Sterling:



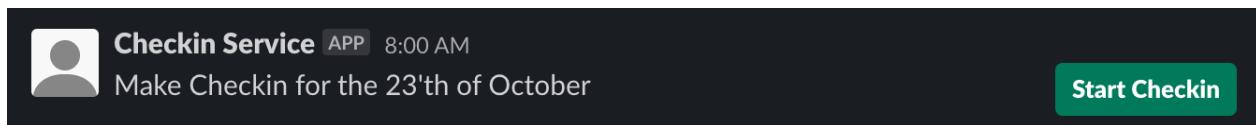
Figur 1: Use Case Diagram over Sterling applikationen fra praktikperioden.

Under praktikperioden blev vi præsenteret for nogle forudbestemte krav fra virksomhedens side. Disse krav blev efterlevet under praktikperioden og dannede samtidig rammen for hovedopgaveforløbet. Vi vil kort opridse de forudbestemte krav for at give en bedre forståelse af de valg, der er blevet truffet under udviklingen af både vores praktik- og hovedopgaveforløb:

- Systemet skal bygges som en integration til Slack.
- Systemet skal interagere med andre platforme såsom Close.
- Systemet skal udvikles vha. Amazon Web Service.
- Systemet skal køre igennem en fuld funktionel CI/CD-pipeline.
- Alle vigtige informationer bliver samlet først.
- Alt design skal godkendes af product owner.

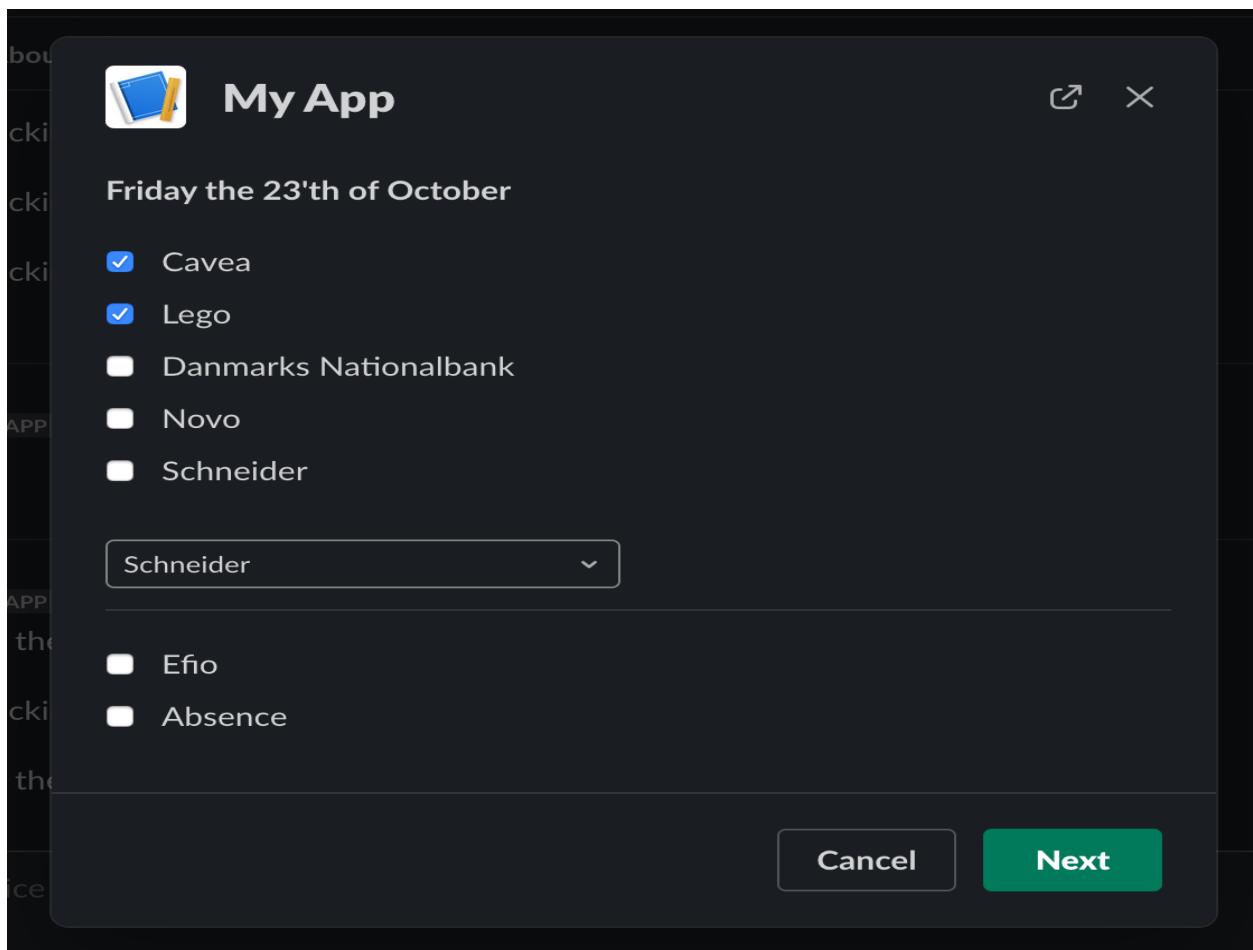
Derved fik vi altså mulighed for at bevæge os frit rundt inden for disse krav, som vi ville. Dette gav anledning til at afprøve alt, hvad vi havde lært under vores uddannelse som datamatiker, samt udfordre os selv ved at vælge teknologier, der ligger uden for pensum. Vi havde derfor også mulighed for selv at bestemme diverse teknologier, som vores system skulle gøre brug af f.eks. programmeringssprog. Vores product owner havde tidligere forsøgt sig med at udvikle et TUI-system (Tekstbaseret User Interface) og kunne derfor give os et indblik i, hvordan han selv gerne ville have systemet skulle se ud og derved inspirere os gennem dette. Vi valgte dog i sidste ende at bruge Block Kit, som er en nyere del af Slack APIet. Vi vil redegøre for valget bag Block Kit i afsnittet *Valg af teknologier* under kapitlet *produkt*. Med dette prøvede vi på en mere imødekommande måde at nå ud til konsulenterne og få dem engageret i den daglige tidsregistrering. Det første vi byggede under praktikperioden var en CI/CD-pipeline, som havde stor betydning for både praktik- og hovedopgaveforløbet. Vi vil redegøre for CI/CD-pipeline i afsnittet *Valg af arkitektur* under kapitlet *produkt*. CI/CD-pipelinen dannede grundlaget for den fremtidige udvikling af applikationen, eftersom al kode der blev udviklet efterfølgende, skulle igennem pipelinien, før det kunne publiceres. Derudover gav pipelinen også mulighed for en automatiseret gennemgang af kodestandarder i forhold til Python og Cloudformation (En del af Amazon Web Service), samt sikre os Behave- og unitest blev kørt igennem. Efter færdiggørelse af vores CI/CD-Pipeline begyndte vi at kigge på det overordnede design. Her kom vores valg af Block Kit UI ind i billede, hvor vi lavede et par dummy billeder af, hvordan layoutet eventuelt kunne se ud. Her fik product owner et indblik i, hvad Block Kit egentlig kunne, hvor vi så efterfølgende fik udleveret et overordnet design af, hvordan applikationen skulle se ud.

Under udviklingen af applikationen stod det ret hurtigt klart, at vi var nødt til at have en knap for overhovedet at kunne aktivere en modal fra Block Kit UI. Med et ønske fra product owners side om at få notificeret sine konsulenter kl. 9.00 sommertid, valgte vi derfor at sende en knap afsted på dette tidspunkt til hver enkelt konsulent, så de kunne starte den daglige timeregistrering.



Figur 2: Påmindelse fra slack til hver enkelt konsulent om den daglige timeregistrering.

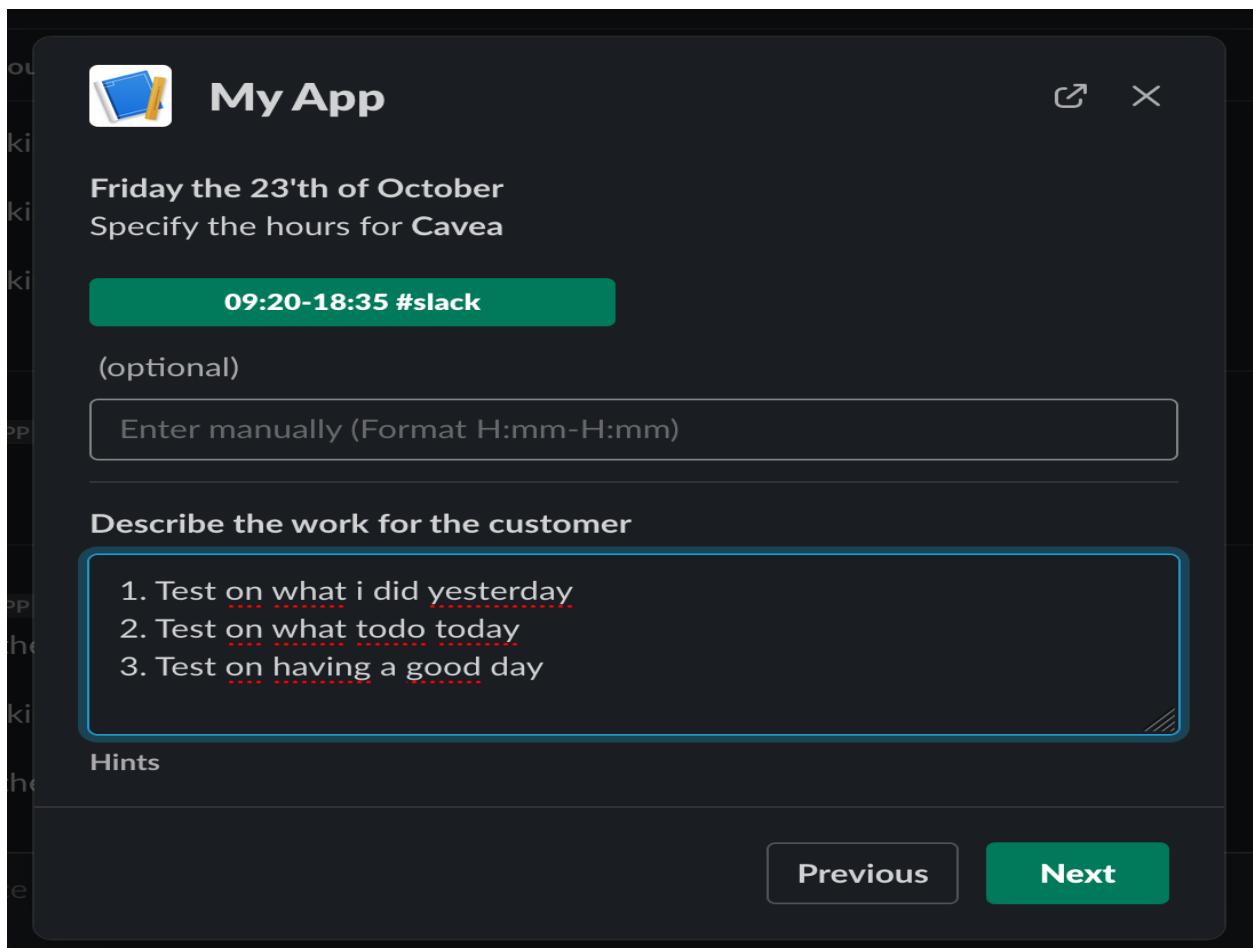
Da vi først kunne aktivere en modal, kunne vi begynde at implementere de forskellige dele af systemet. Første del var at give den enkelte konsulent muligheden for at vælge imellem aktive kunder. Alle kunder er på nuværende tidspunkt dummy data, som skal illustrere rigtige kunder. Hver kunde har også en sandsynlighed tilknyttet, hvor denne skal beskrive sandsynligheden for at konsulenten arbejdede for den bestemte kunde på den givne dag. Under praktikforløbet var alle kunders sandsynlighed forudbestemt. Hvis konsulenten ikke er ude hos en af de fire kunder med størst sandsynlighed, er der en mulighed for at vælge iblandt de resterende kunder i databasen. Når en kunde er valgt, der ikke er iblandt de 4 kunder med højest sandsynlighed, bliver denne kunde tilføjet som en tjekboks, så konsulenten får mulighed for at vælge denne kunde. Derudover tilføjede vi også muligheden for at vælge fravær og Efio-arbejde på denne modal. Fravær skal give en konsulent mulighed for at melde sig syg eller indberette ferie, hvor Efio-arbejde f.eks. kan være internt arbejde i virksomheden eller undervisning af praktikanter. Der blev ikke tilføjet noget funktionalitet til hverken fravær eller Efio-arbejde i første omgang.



Figur 3: Første modal af Sterling applikationen med mulighed for kunde valg, samt tjekbokse for Efio-arbejde og fravær.

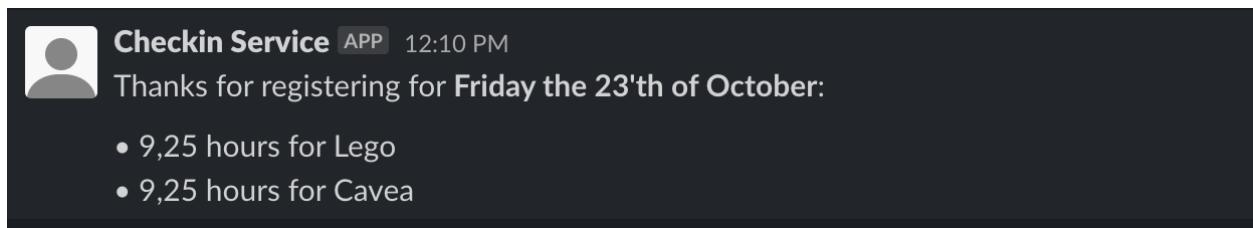
Løbende blev der tilføjet funktioner til systemet i form af f.eks. generering af ens aktive tid på slack fra igår og mulighed for at se hints fra idag og igår hentet fra en af Efio's slackkanaler kaldet *standup*. (Det er desværre ikke muligt at illustrere de hentede hints, da billederne i dette afsnit er taget under praktikperioden og applikationen har ændret sig siden). Desuden skal konsulenten også have muligheden for at redegøre for den brugte tid hos den enkelte kunde og derfor opdateres vores modal fra før (Se figur 3) til en modal med nyt udseende (Se figur 4). Modalen bliver således opdateret x antal gange, i forhold til x antal valgte kunder fra før. Dette vil sige, at hvis Cavea og Lego er blevet valgt som kunder på første modal (Se figur 3) så opdateres denne modal først med henblik på Cavea som kunde og dernæst med henblik på Lego som kunde (Se figur 4). På den nye modal er der mulighed for at vælge tiden, som er hentet fra slack og hvis denne tid ikke er korrekt, så er der også en mulighed for at skrive tiden ind selv. Derudover får konsulenten også mulighed for at redegøre for, hvad de brugte deres tid på hos kunden. Til sidst gives der et hint, som er hentet fra *standup*-kanalen. *Standup* er en intern kanal på slack hos Efio, som bruges til at beskrive følgende:

1. Hvad lavede jeg igår?
2. Hvad skal jeg lave i dag?
3. Andre bemærkninger? Eks. Jeg har fødselsdag.



Figur 4: Anden modal af Sterling applikationen. Denne modal kommer x antal gange ud fra x antal valgte kunder.

Når konsulenten er færdig med at redegøre for den brugte tid hos en eller flere kunder lukkes modalen og den første besked fra slack, hvor timeregistreringen kunne startes, bliver nu slettet. Herudover får konsulenten en kvittering tilbage om, at deres tid nu er blevet registreret og hvad der blev registreret, så hvis en konsulent f.eks. har brugt 9,25 timer hos Lego, kommer dette tilbage i svaret.

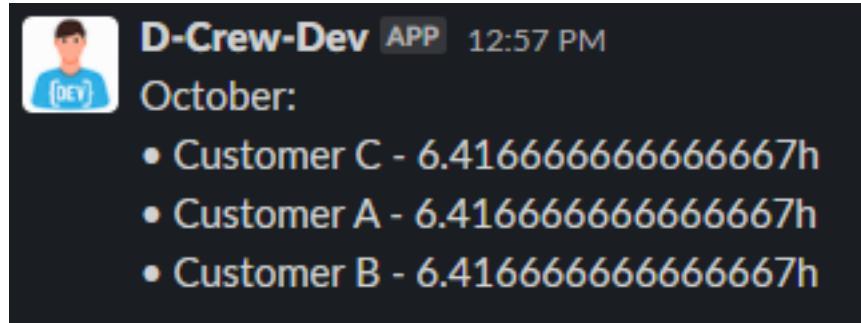


Figur 5: Kvittering fra gennemført timeregistrering med x antal registrerede timer for y kunde.

Udover den daglige timeregistrering udviklede vi også en ugentlig oversigt samt en månedlig oversigt. Begge oversigter bliver sendt afsted til hver enkelt konsulent og viser den enkelte konsulents registrerede timer. Det ugentlige overblik bliver sendt til konsulenten mandag morgen kl. 9.00 sommertid, hvorimod det månedlige overblik bliver sendt til konsulenten den sidste arbejdsdag i en given måned kl. 14:00 sommertid. Det er dog også muligt med et API-kald at få tilsendt ugentlig og månedlig rapport udenfor de fastsatte tidspunkter.



Figur 6: Ugentlig oversigt over registrerede timer.



Figur 7: Månedlig oversigt over registrerede timer.

Generelt set var applikationen ved at udforme sig til noget brugbart efter praktikforløbetets afslutning. Vi havde på dette tidspunkt bygget skallen af Sterling og kunne sagtens se mulighederne for videreudvikling af funktionaliteter i forhold til applikationen, hvilket dannede grundlaget for, at vi valgte at arbejde videre med applikationen i hovedopgaveforløbet.

4. Forretningsmodel

I følgende afsnit introduceres Efio som virksomhed, relevante strukturer i Efio i forhold til projektet, den forventede indflydelse projektet ville have og til sidst den opnåede forretningsværdi for projektet.

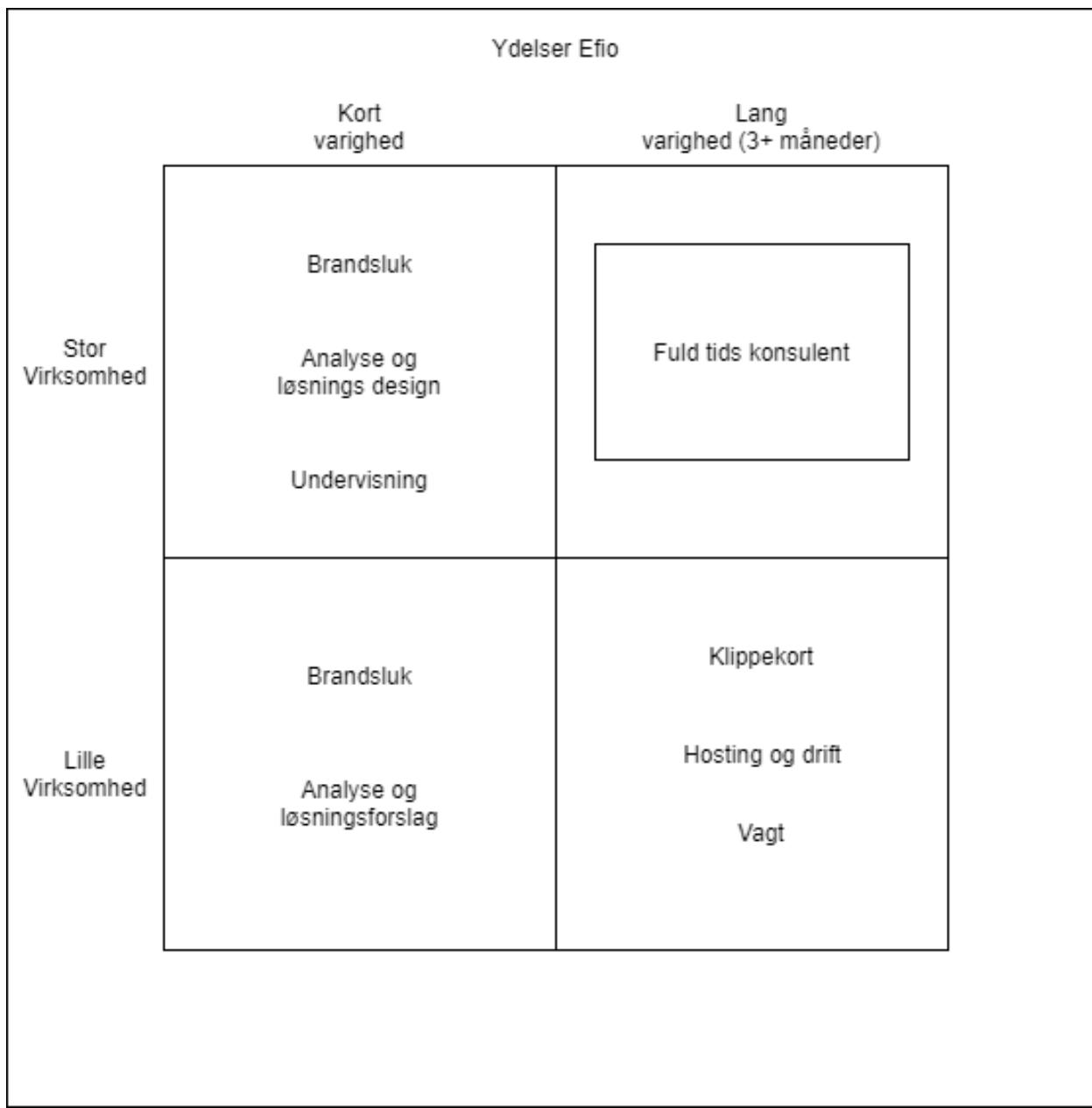
4.1 Efio

Efio er et konsulentfirma, der specialiserer sig i DevOps. DevOps indebærer automation, infrastruktur som kode, serverless first og nano services samt kontinuerlig integration- og levering. Alt dette kan de tilbyde i Cloud Computing form og har store kunder såsom DR, Danmarks National Bank, Coop, Novo og mange flere. Efio er et AWS (Amazon Web Service) Partner-selskab, hvilket vil sige de udbyder deres tjenester på AWS-platformen, og kan hjælpe med at få et eksisterende program eller hjemmeside til at køre igennem Cloud Computing hos AWS. Hos Efio finder vi seks Senior DevOps konsulenter og én DevOps konsulent, samt fem praktikanter i øjeblikket. Efio budgetterer for ca 35.000 kr. pr. praktikant til at dække for udgifter som undervisningsmateriale, udstyr, transport, mad og begivenheder. Det vil sige, at der er lagt budget med en udgift på omkring 100.000 kr. for en tremandsgruppe, der inkluderer både praktik- og hovedopgaveforløb.

Efio bruger lige nu Kala som tidsregistreringplatform. Med dette projekt har vi prøvet at opnå tilstrækkelig høj forretningsværdi for Efio's medarbejdere til potentelt at kunne erstatte Kala. Det der skal kunne stå til forskel fra det nuværende system er, at forkorte tiden det tager at registrere sine timer. Dette forsøger vi at opnå ved at komme med forudsigelser af, hvilke kunder konsulanten kunne have arbejdet for. Desuden bliver konsulanten påmindet op til to gange dagligt om, at indregistre deres timer og desuden ligger applikationen på en platform konsulenterne bruger dagligt. Dette medfører mindre risiko for, at der ikke bliver udfyldt timeregistrering, eller at timeregistrering bliver lavet forkert, fordi konsulanten måske har glemt, hvor lang tid han/hun var hos en kunde. På den måde går der ikke fakturerbare timer tabt. Den administrerende direktør for Efio har meldt ud, at der kan være tale om et tab i omsætningen på op til 100.000 kr. årligt på grund af manglende indregistrerede timer. Med dette projekt håbede han at kunne formindske den manglende indregistrering af timer, som konsulenterne bruger ude hos deres kunder.

4.2 Kundesegment

Efio har fire kundesegmente. På figur 8 kan man se, at hver kunde hører til et segment afhængigt af den pågældende virksomheds størrelse og varigheden på konsultationen.



Figur 8: Overblik over Efios kundesegment.

Stor virksomhed, kort varighed

I dette segment hjælper Efio med undervisning og præsentationer ud fra det, de specialiserer sig i. Dette kan handle om Cloud Computing, DevOps, sikkerhed mm., som den pågældende virksomhed kan have interesse i, at deres ansatte lærer om. Derudover er der også interesse for, at Efio laver analyser over virksomhedens system. Her handler det om at kigge på, hvad der kan forårsage et problem, finde roden i eventuelle problemer og finde bottlenecks osv. Her tilbyder Efio også et løsningsdesign med mere detaljerede løsningsforslag med arkitekturdiagrammer, beskrivelser mm. Dette varierer meget afhængigt af kundens behov. Til slut hjælper Efio også virksomheder med akut behov for hjælp, når der er store fejl i systemet, som skal rettes. Her kan det være store serverfejl, databaselækage mm., som kan være svært for virksomheden at tage sig af alene. Brandslukning er sjældent en robust løsning, men kan være nødvendigt som en midlertidig løsning.

Lille virksomhed, kort varighed

Analyse i små virksomheder, såvel som i store, handler om at finde de problematikker som f.eks. virksomhedens system, finde roden af problemet og hjælpe med at komme med løsningsforslag til virksomheden, der kan være vejledende for enten at fremme salg, optimering af systemet mm., som virksomheden selv kan arbejde med. For små virksomheder bliver der sjældent lavet løsningsdesign, da det kræver væsentligt mere tid og dermed koster mere. Efio hjælper også små virksomheder med brandslukning, da det kan være væsentligt sværere for små virksomheder at reparere katastrofale fejl i deres systemer, da små virksomheder ikke har nær samme bemanding som store virksomheder.

Stor virksomhed, lang varighed

Her bliver konsulenter ansat på fuldtid i virksomheden og arbejder på et enkelt eller flere projekter i virksomheden. Dette skaber en pålidelig og stabil indkomst over en længere periode, som gør dette segment til det mest foretrukne for Efio. Konsulenternes kontraktlængde varierer, men ligger normalt på mellem tre måneder og et år.

Lille virksomhed, lang varighed

Små virksomheder har sjældent mange ressourcer til at hyre konsulenter ind og derfor har Efio en klippekortsordning, således at de også har mulighed for at få hjælp. Klippekortsordningen fungerer på forskellige måder afhængigt af kunden. Man kan enten forudbetale for en vis mængde timer, som derefter kan genkøbes, når det antal timer, er brugt. Ellers kan virksomheden betale en fast pris månedligt for et vis antal timer. Dette hjælper virksomheden med at holde sig inden for det budget, som de har til rådighed til konsultation. Efio tilbyder også vagtordninger, som er service efter behov. F.eks. kan dette være telefonservice 24 timer i døgnet og reaktion på alarmer fra systemet. Vagtaftalen indeholder desuden, hvor hurtigt konsulenterne begynder at udbedre en fejl, eventuelt med henblik på brandslukning på kort sigt og eventuelt analyse og design på lang sigt for at undgå gentagelser af fejlen.

4.3 Slack

Den administrerende direktør for Efio har prøvet flere metoder til at få konsulenterne til at huske at få registreret deres timer. Alle ansatte i Efio bruger Slack til at kommunikere og lave deres daily standup. Ud fra dette kunne han se, at Slack er en platform, som alle rutinerede gør brug af, og fandt på den måde frem til, at applikationen skulle udvikles på Slack. Med denne platform vil vores applikation altid være tæt på, hvor konsulenterne er i forvejen, uden at de skal bevæge sig væk fra en platform, de allerede har i brug.

4.4 Opnået forretningsværdi

Vi havde ikke forventet et færdigt produkt, men som minimum et forslag på en prototype af, hvordan et endeligt produkt kunne se ud. Med vejledning fra product owner om, hvad han gerne ville have af design, kunne vi skabe et grundlæggende fundament til en applikation, som Efio vil kunne arbejde videre på efter vores praktikperiode. Prototypen har haft fokus på brugeroplevelsen ved design samt flow og oplevelsen gennem applikationen. Efio har tidligere prøvet at arbejde på, at få en prototype stykket sammen af den type applikation, som vi har udviklet, dog uden success. Med den feedback og rapportering der kom fra resten af virksomheden, ledte det til en grundlæggende designændring af den oprindelige idé. Oprindeligt havde product owner ønsket timeregistrering i et fra-til format, som nu er blevet til et totalformat i form af antal timer. (f.eks. 8.5 timer i stedet for 8.00-16.30). Formatet og det nye design bliver dokumenteret i designafsnittet under kapitlet *produkt*.

Efter vores afsluttede praktik- og hovedopgaveforløb har den administrerende direktør for Efio givet følgende afsluttende bemærkning: *"Der er generelt meget høj tilfredshed med jer som gruppe og I har indfriet vores forventninger trods svære forudsætninger situationen i betragtning"*.

Produkt

5. Kravspecifikation

Kapitlet indeholder en præsentation af vores produkt, hvor vi vil gennemgå følgende emner: Kravspecifikation, valg af teknologier og arkitektur, design, implementering, test og samlet evaluering heraf. I det følgende afsnit vil vi redeøre for omfanget af produktet, samt redeøre for og dokumentere de funktionelle- og ikke-funktionelle krav til produktet i form af user stories med tilhørende acceptkriterier.

5.1 Scope

Product owner har et problem med at få sine ansatte til at registrere det korrekte antal timer, de har brugt hos en kunde hver måned. Dette medfører som tidligere nævnt et stort tab i virksomhedens omsætning. Målet for vores applikation er derfor at formindske og helt optimalt forebygge tabt omsætning forårsaget af fejlagtig registrering af konsulentens timetal hos deres kunder. Den problemstilling er fundamentet for dette projekt og for at appellere til konsulenterne, bliver applikationen bygget på Slacks platform, hvor konsulenterne i forvejen befinner sig til at skrive deres daglige standup. De krav der blev stillet i hovedopgaveforløbet, var med til at få applikationen gjort mere dynamisk i forhold til at håndteringen af data f. eks., at kunder skulle være i forskellig rækkefølge i forhold til en given konsulent. Derudover blev der også stillet krav til udvidelse af systemet i forhold til andre tidsregistreringsformer f.eks. registrering af internt arbejde eller ferie. For at skabe en forståelse af det udførte arbejde i forbindelse med udviklingen af applikationen, vil vi i det næste afsnit gennemgå de funktionelle- og ikke-funktionelle krav, der er blevet stillet til applikationen.

5.2 Funktionelle krav

De funktionelle krav beskriver den måde applikationen opfører sig på, altså specifikke funktionaliteter der ligger til grund for applikationen. Nedenfor har vi opstillet de funktionelle krav i punktform:

1. Importer kunder og kontrakter fra Close CRM (Customer Relationship Management)
 - (a) Hvis en konsulent manuelt skriver egenskaber på en kontrakt, skal denne kontrakt overskrives, hvis den allerede eksisterer på Close CRM
 - (b) Der skal være en funktion, der matcher kunde-CVR, når der bliver registreret en ny eller søgt efter en kunde.
 - (c) Der skal være en funktion, der matcher kunders lead.status med en kunde fra Close CRM således, at kunden kan blive importeret, hvis vedkommende ikke eksisterer.
2. Personliggør kunder til timeregistrering vha. et vægtsystem.
 - (a) Det skal være muligt for den enkelte konsulent at få personliggjort alle kunder.
 - (b) Der skal være en funktion, der forudbestemmer de personlige kunder igennem et vægtsystem.
 - (c) Vægtsystem skal overholde følgende:

- i. Alle kontrakter er matchet med en konsulent.
 - ii. Kunder med ingen aktive kontrakter har vægt 1.
 - iii. Kunder med aktive kontrakter er vægtet x5.
 - iv. kunder med aktive kontrakter for den enkelte konsulent er vægtet x10.
 - v. Gårsdagens kunde er vægtet med x3.
 - vi. To dage gamle kunder er vægtet med x2.
 - vii. E-mail sendt til kunde er vægtet med x10.
 - viii. E-mail modtaget fra kunde er vægtet med x5.
3. Giv administrator mulighed for at skifte værdierne i vægtsystemet.
 - (a) Det skal være muligt at hente vægte fra en tabel i databasen, så administratoren kan bestemme vægtene og lære hen ad vejen.
 - (b) Vægtene skal kunne gå fra 1 til 100.
 4. Opdater emoji baseret på den indregistrerede tid.
 - (a) Den personlige emoji på slack skal opdateres, så den kan symbolisere fravær eller ferie.
 5. Mulighed for at specificere internt arbejde i form af Efio-arbejde.
 - (a) Det skal være muligt for en konsulent at arbejde internt i virksomheden.
 - (b) Der skal gives en kvittering for internt arbejde i virksomheden.
 6. Mulighed for at indregistrere fravær og give valget mellem sygdom og ferie.
 - (a) Det skal være muligt for en konsulent at sygemelde sig.
 - (b) Det skal være muligt for en konsulent at gå på ferie.
 - (c) Det skal være muligt for en konsulent at bestemme ferieperioden.
 7. Mulighed for at tidsregistrere før tid i tilfælde af sygdom eller ferie.
 - (a) Det skal være muligt for en konsulent at starte tidsregistrering med det samme i tilfælde af sygdom eller ferie.
 - (b) Det skal være muligt for konsulenten at tidsregistrere for flere dage ad gangen.
 8. Sæt en personlig tid for start af tidsregistrering.
 - (a) Det skal være muligt for den enkelte konsulent at bestemme påmindelsestidspunktet for timeregistreringen.
 - (b) Det skal være muligt for den enkelte konsulent at vælge mellem at registrere timer fra igår eller for idag.
 - (c) Tidspunktet for timeregistreringen skal gives med et 5 minutters interval.
 9. Beskrivelse skal være både krævet og valgfri i forhold til kontrakten.
 - (a) Det skal være krævet at udfylde beskrivelsesfeltet, hvis det er et krav i forhold til kontrakten fra Close CRM.
 - (b) Det skal være valgfrit at udfylde beskrivelsesfeltet, hvis det ikke er et krav i forhold til kontrakten fra Close CRM.
 10. Vores applikation skal integreres med Zenegy.
 - (a) Det skal være muligt for en konsulent, kun at skulle registrere ferie eller fravær på vores applikation, så konsulenten kan slippe for at logge ind på Zenegy for at gøre dette.
 11. Rette tidsformateringen fra praktikperioden på kvittering, ugentlig og månedlig rapport.

5.3 Ikke-funktionelle krav

De ikke-funktionelle krav beskriver de kvalitetsmæssige krav til applikationen f.eks. i form af svartider, arkitektur og brugervenlighed. Lad os tage et kig på de ikke-funktionelle krav opstillet i punktform:

1. Brugeren skal på ethvert tidspunkt, kunne stoppe sin tidsregistrering for ikke at være ved computeren, og kunne genoptage det efter 8 timer.
2. Brugeren skal altid starte fra toppen af ved at vælge alle aktiviteter for en given dag, før der går i detaljer med f.eks. timeforbrug og beskrivelse for en enkelt aktivitet.
3. Brugeren skal kunne gennemføre tidsregistreringen fra Slack app'en på en mobiltelefon.
4. Brugeren må ikke mødes af trin i flowet, der i sig selv ikke indsamler relevant data med undtagelse af afsluttende opsummering.
5. Brugeren må ikke mødes af tekster i flowet, der er generelle vejledninger til brugeren. Konkrete spørgsmål og navngivne actions er tilladt.
6. Nyt design for kunderegistrering.
 - (a) Det skal være muligt for en konsulent at tilføje et nyt projekt, så projektet bliver gemt i databasen og bliver vist på modalen.
 - (b) Det skal være muligt for en konsulent at se alle aktive projekter for en kunde under tidsregistreringen.

Alle de gennemgåede krav blev opfyldt på nær et af de funktionelle krav 10. *Vores applikation skal integreres med Zenegy*. Dette krav blev ikke opfyldt grundet manglende erfaring med Zenegy API'et og en langsom kundeservice fra Zenegy's side. Det skal dog siges, at vi havde fuld adgang til produktionsmiljøet ret hurtigt, men eftersom vi ikke havde tilladelse til at udvikle direkte i produktionsmiljøet, måtte vi vente på at få adgang til et testmiljø. Product owner havde tidligere fået tilsendt en API-token til Zenegy API'ets testmiljø i forbindelse med et andet projekt, men havde desværre fået forlagt denne. Hvis vi havde haft adgang til testmiljøet fra starten af, havde vi haft mulighed for at lære Zenegy API'et at kende meget hurtigere, men i stedet var vi nødt til at vente størstedelen af vores sidste sprint på at få adgang. Derudover skulle vi også lave en returnUrl til Zenegy for at kommunikere med deres API, hvilket også var ret besværligt. Zenegy har tilsyneladende ikke en måde hvorpå at man kan redigere returnUrl, som f.eks. Slack har. Dette betyder, at hver gang en af vores gruppemedlemmer opretter en ny stack, hvilket vi gør hver gang vi starter på at udvikle på en ny user story, skal Zenegy ind og godkende vores return-URL. Efter en længere samtale med forskellige ansatte fra Zenegy, blev vi til sidst omstillet til deres teknologichef, som endte med at sende en API-token til os, men på daværende tidspunkt havde vi kun et par dage tilbage i sprintet og vi kunne derfor ikke nå at opfylde kravet om, at vores applikation skal integreres med Zenegy. Udenfor kravet om Zenegy blev der også stillet to krav, der var påtænkt at skulle opfyldes, hvis vi havde ekstra tid tilbage i udviklingsperioden. Disse to krav var: *2.c.vii E-mail sendt til kunde er vægtet med x10* og *2.c.viii E-mail modtaget fra kunde er vægtet med x5*. I sidste ende havde vi ikke nok tid til at få implementeret disse krav i vores applikation og derved fik vi ikke opfyldt kravene. Applikationen var sådan set gjort klar til at løse kravene i form af, der var gjort plads til værdier i databasen samt metoden, der udregnede den samlede værdi.

5.4 Dokumentation af kravene

Alle ikke-funktionelle krav til applikationen blev præsenteret før udviklingsperioden startede i vores praktikforløb udover 6. *Nyt design for kunderegistrering*, der i stedet blev præsenteret for os i starten af hovedopgaveforløbet. Alle krav der er blevet stillet til applikationen i hovedopgaveforløbet, er blevet stillet igennem user stories og acceptkriterier. Alle user stories med tilhørende acceptkriterier kan findes i Bilag 1. Vi vil nu tage et kig på en user story med tilhørende acceptkriterier for at illustrere, hvordan de er blevet stillet fra product owners side:

As a consultant

I want individual possibilities on customers

So that checkin can be done faster

Given a consultant

When time registering

Then I get personal customers

Given a function

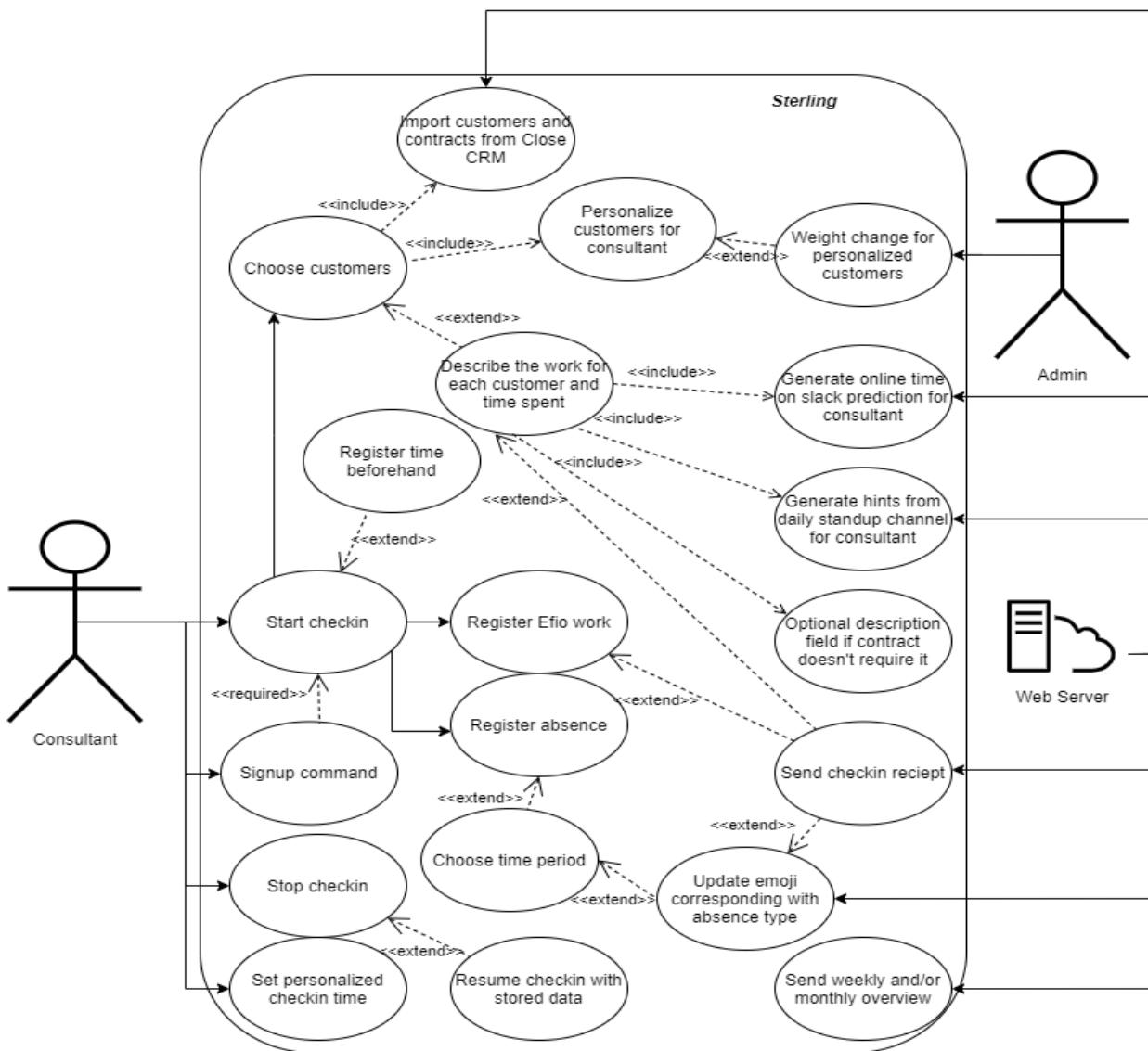
When predicting personal customers

Then use weighted system

Weighted system:

- All contracts is matched with a consultant
- Customers with no active contracts have weight 1
- Customers with active contracts have weight x5
- Customers with active contract for consultant have weight x10
- Yesterday's customers have weight x3
- Two days ago customers have weight x2
- Email sent to customer have weight x10
- Email received from customer have weight x5

Ud fra den givne user story har vi forsøgt at omskrive denne til et krav, som vi gennemgik i punktform under funktionelle krav: 2. *Personliggør kunder til timeregistrering vha. et vægtsystem*. Dette krav skulle erstatte vores hidtil konstante sandsynligheder for bestemmelse af kunderækkefølgen for den enkelte konsulent. Ydermere gav dette krav også mulighed for at udvikle et andet krav nemlig 3. *Giv administrator mulighed for at skifte værdierne i vægtsystemet*, hvor de konstante værdier i vægtsystemet blev udskiftet med værdier fra en tabel i databasen. Nedenfor ses et Use Case Diagram over applikationen efter vores afsluttede udviklingsforløb i hovedopgaveperioden.



Figur 9: Use Case Diagram over Sterling applikationen efter afsluttet udviklingsperiode i hovedopgaven.

Som det kan ses på det nuværende Use Case Diagram fra hovedopgaveperioden (Se figur 9), er der sket en del ændringer i forhold til use case diagrammet fra praktikperioden (Se figur 1). Ændringerne ligger bl.a. i funktioner, der er med til at skabe en dynamisk applikation, så applikationen tilpasser sig den givne konsulent. To konsulenter skal ikke have samme rækkefølge af kunder, hvis hver konsulent sidst arbejdede for forskellige kunder. Vores dummy kunder fra tidligere blev også udskiftet med rigtige kunder, der er hentet fra Close CRM. Derudover kom der en stor ændring i designet, hvor det nu blev muligt at beskrive gårdsdagens arbejde og arbejdstid på samme linje samt, hvis en konsulents kontrakt kræver, at der udfyldes en beskrivelse af dagens arbejde, bliver konsulanten tvunget til at skrive dette. Logikken bag hints blev også ændret til, at konsulanten f. eks. kan se et forslag på brugt arbejdstid med udgangspunkt i, hvor længe konsulanten var aktiv på Slack. Derudover blev applikationen udvidet i form af, det nu blev muligt at registrere fravær i form af sygdom eller ferie, samt det blev muligt at tidsregistrere for fremtiden. I tilfælde af indregistreret fravær bliver den personlige emoji også ændret, så ens kollegaer kan se, at du er sygemeldt eller er på ferie.

5.5 Vurdering

I bund og grund fik vi løst de fleste krav. Det eneste krav der ikke er opfyldt er som nævnt tidligere: *10. Vores applikation skal Integreres med Zenegy* og de to krav: *2.c.vii E-mail sendt til kunde er vægtet med x10* og *2.c.viii E-mail modtaget fra kunde er vægtet med x5*, da disse var påtænkt at være ekstra arbejde i forhold til, hvis vi var hurtige om at løse de resterende krav. Hvis vi ser tilbage på disponeringen af vores tid, havde det klart været en fordel, hvis kravet om Zenegy var kommet i det første sprint, eftersom det ville give os mulighed for at tage hele korrespondancen med Zenegy sideløbende med udviklingen af de resterende krav. Det havde i det hele taget været svært for os at nå de to ekstra krav, eftersom vi var tidspresset til sidst. Vi vurderer selv, at det har været muligt at identificere og dokumentere alle krav i forhold til, at de fleste af kravene er blevet opstillet igennem user stories. Eftersom kravene løbende blev stillet af product owner, havde vi mulighed for i samarbejde med ham at danne os et billede af hans forventninger til applikationen, og derved har vi haft god mulighed for at opfylde virksomhedens mål til systemet.

6. Valg af teknologier

I dette afsnit vil vi redegøre for de teknologier vi har brugt til at udvikle applikationen og hvorfor vi har valgt disse teknologier frem for andre. Ydermere vil vi begrunde vores valg af teknologier ved at se på fordele og ulemper ved den enkelte teknologi.

6.1 Slack

Slack er et kommunikationsredskab brugt af mange virksomheder men også af privatpersoner. De tilbyder en bred vifte af tjenester og funktioner som er styret med IRC: Internet Relay Chat i tankerne. Chatrum, private grupper, gruppeopkald og afstemninger er bare nogle af de mange funktioner, som Slack tilbyder. Udover alle IRC-funktionerne tilbyder Slack også at udviklere kan udgive applikationer, der kan forbedre brugen af produktet. Applikationsudvikling foregår gennem API-kald og der bruges de samme designelementer i alle apps, så Slack får et mere unikt look på hele deres platform. Der findes to store kommunikationsplatforme; Slack og Teams. De tilbyder på ydersiden det samme, så valget mellem dem handler om, om man i forvejen f.eks. bruger Microsofts tjenester, da det derved vil være nemmere at integrere Teams i virksomheden. Slack har dog været på markedet i længere tid og har dermed flere integrationer med andre tjenester. At arbejde med Slack var anderledes, da vi tidligere ikke har udviklet integrationer til allerede eksisterende software. Webudvikling eller Programmer har været hovedpunkterne i uddannelsen, så at bruge andres materiale og bruge det som de har tænkt, var udfordrende men spændende.

Block Kit

Block Kit er et UI framework bygget til udvikling af Slack Applikationer. Med Block Kit får du prædefinerede UI elementer såsom, knapper, selectors, dato/tidsvælgere og mange flere. Grundet dette UI framework vil Slack og alle dets applikationer have samme udseende. Et UI framework er fornøjelse for programmører at arbejde med, da vi ikke behøver at tænke på design. Med brugen af Block Kit kan vi som gruppe koncentrere os mere om backend og lære, hvordan vi bedst muligt kan opsætte den ved brug af AWS og dets tjenester. Vi vil ikke kunne have nået at lave applikationen så stor, som den er blevet, hvis vi også selv skulle have designet udseendet til applikationen i for eksempel CSS (Cascading Style Sheet).

6.2 AWS

AWS er et datterselskab af Amazon, som tilbyder Cloud Computing-platforme og API'er. Med AWS betaler man for det, man bruger, og på den måde kan man også hurtigt skalere ind og ud efter efterspørgsel. AWS tilbyder en bred vifte af forskellige tjenester lig fra opbevaring til automatisering. AWS og Azure er de to store cloud computing-platforme. På papiret kan de stortset det samme, men alligevel er der ting, som de hver især er bedre til. AWS går meget op i hastighed, hvor Azure går mere op i konsistens af deres data. Prisen afhænger meget af produktet, du vælger hos dem, for eksempel er en Windows Server samt SQL server billigere hos Azure, da det er Microsoft der ejer både Windows og Azure. AWS arbejder med NoSQL databaser for bedre skalerbarhed og derfor koster SQL mere. Efio har valgt AWS til deres firma og derfor satte product owner et krav om brugen af AWS. Hvis der ikke var blevet

sat dette krav fra product owner, ville vi som gruppe have valgt Azure, da de har en studieløsning, hvor de tilbyder \$100 til brug på deres tjeneste. AWS har nemlig ingen studieløsning, men er ofte billigere til små systemer, da man kun betaler for det, der bliver brugt.

CodePipeline

CodePipeline er en automatiseret kontinuerlig leveringstjeneste, som kan bruges til at modellere og visualisere de forskellige trin, der skal til for at frigive software. Vores applikation har to pipelines i udviklingsmiljøet og en i produktionsmiljøet. Den første pipeline i udviklingsmiljøet er vores kontinuerlige integrationspipeline, som sørger for at køre statisk kodeanalyse, teste unittest samt acceptance test og skrive forskellige statuskoder tilbage til Github. Den anden pipeline, der også er gældende i produktionsmiljøet, er vores kontinuerlige implementering, som sørgede for at deploye applikationstjenester såsom database og API'er. CodePipeline gjorde det muligt for os at arbejde hurtigere, da alt test og deployment var automatiseret og vi derfor kunne arbejde videre med vores kode imens. Uden CodePipeline skulle vi have brugt meget tid på at køre test og deploye vores kode manuelt til alle de forskellige AWS-tjenester.

CodeBuild

CodeBuild er en fuld administrativ build-tjeneste, som gør det muligt at kompilere og teste kildekode samt producere artefakter af kildekode. CodeBuild kommer med et færdigpakket miljø, så man ikke selv skal opretholde og skalere en server. Med dette kunne vi hurtigt opsætte en server, der både kunne teste, bygge og deploye vores applikation. Vi satte CodeBuild op til at bruge Python, hvorefter CodeBuild selv installerer de nødvendige pakker og opsætter et miljø, der kan køre vores kildekode. Vi kan uddover de indbyggede pakker selv, gennem CloudFormation, installere flere og fortælle, hvilke kommandoer der skal køres for at teste og bygge applikation. Ved at bruge CodeBuild kunne vi opsætte præcis, hvilke kommandoer der skulle køres for at teste og deploye vores kildekode. Uden denne mulighed ville det have taget meget længere tid at deploye, da vi dermed skulle huske mange flere kommandoer end bare et push til Github.

Lambda

Lambda er Amazons svar på serverless computing. Med lambda kan der køres kode uden opsætning eller administration af servere. Koden der køres på Lambda, kører kun når der er brug for det, og der bliver skaleret ud automatisk, hvis efterspørgslen for eksempel går fra et par gange om dagen til et par gange i sekundet. Prisen for en Lambda afhænger af kørselstiden, hvilket vil sige at så længe koden ikke kører, er den gratis og når den kører betales der en rate pr. 100ms. Hele vores app bygger på Lambda-funktioner, som er koblet op til et REST endpoint. Dette gør det nemt for os at lave nye endpoints, uden at skulle administrere og vedligeholde en server. Med Lambda er det muligt i vores CloudFormation-fil at definere endpoints, hvilken fil endpointsene skal køre og hvilket sprog filen er skrevet i. Uddover REST endpoints er det også Lambda, som sender buildstatus til Github, hvor det er et CodeBuild-event i stedet for REST event, der starter Lambdaen. Hvis vi ikke havde Lambda, vil vi skulle spinne en server op, hvor i vi kunne køre API'et og dets moduler, hvilket også vil kunne lade sigøre hos AWS i deres tjeneste EC2. Det kan argumenteres for, at moduler der kører konstant, vil være bedre at køre i en EC2, da det i længden vil være billigere. Vores applikation havde dog kun moduler, der kørte mindre end et sekund og de kørte heller ikke mere end én gang om dagen pr. konsulent.

CloudFormation

CloudFormation er en tjeneste, der ved hjælp af en skabelon, kan modellere og opsætte andre AWS-tjenester. CloudFormation bliver brugt til at udvikle ud fra konceptet infrastructure as code, som betyder at al infrastrukturen til applikationen, skal kunne skabes ud fra en kodebase. CloudFormation kan opsættes med YAML (Yet Another Multicolumn Layout) eller JSON (JavaScript Object Notation), hvor vi valgte at bruge YAML, da det er det mest

robuste og mest læsbare markup-sprog. CloudFormation gjorde det nemt og hurtigt for os at opsætte vores CI/CD-pipeline. Først kunne vi i en YAML-fil opsætte tilladelser for pipelinen, så den har adgang til at hente kildekode fra Github osv. CodePipelinen kan herefter sættes op med forskellige lag af tjenester, som f.eks. CodeBuild, der henter kildekoden og tester den. Med CloudFormation har vi kunne arbejde mere i dybden med koden, da vi ikke har skulle lægge vores fokus på at opsætte AWS-tjenester manuelt. Der findes en masse skabeloner, der kan opsætte præcis de teknologier, man skal bruge. Dette betød, at vi ikke manuelt skulle oprette alle tjenesterne og linke dem sammen efterfølgende. CloudFormation har gjort at vi inden for nogle få uger, kunne påbegynde udviklingen af selve applikationen. Uden dette havde vi for hvert release, manuelt skulle køre test og manuelt deploye vores kode til AWS, som kan skabe mange problemer, hvis to udviklere gør det på forskellige måder.

Serverless Framework CLI

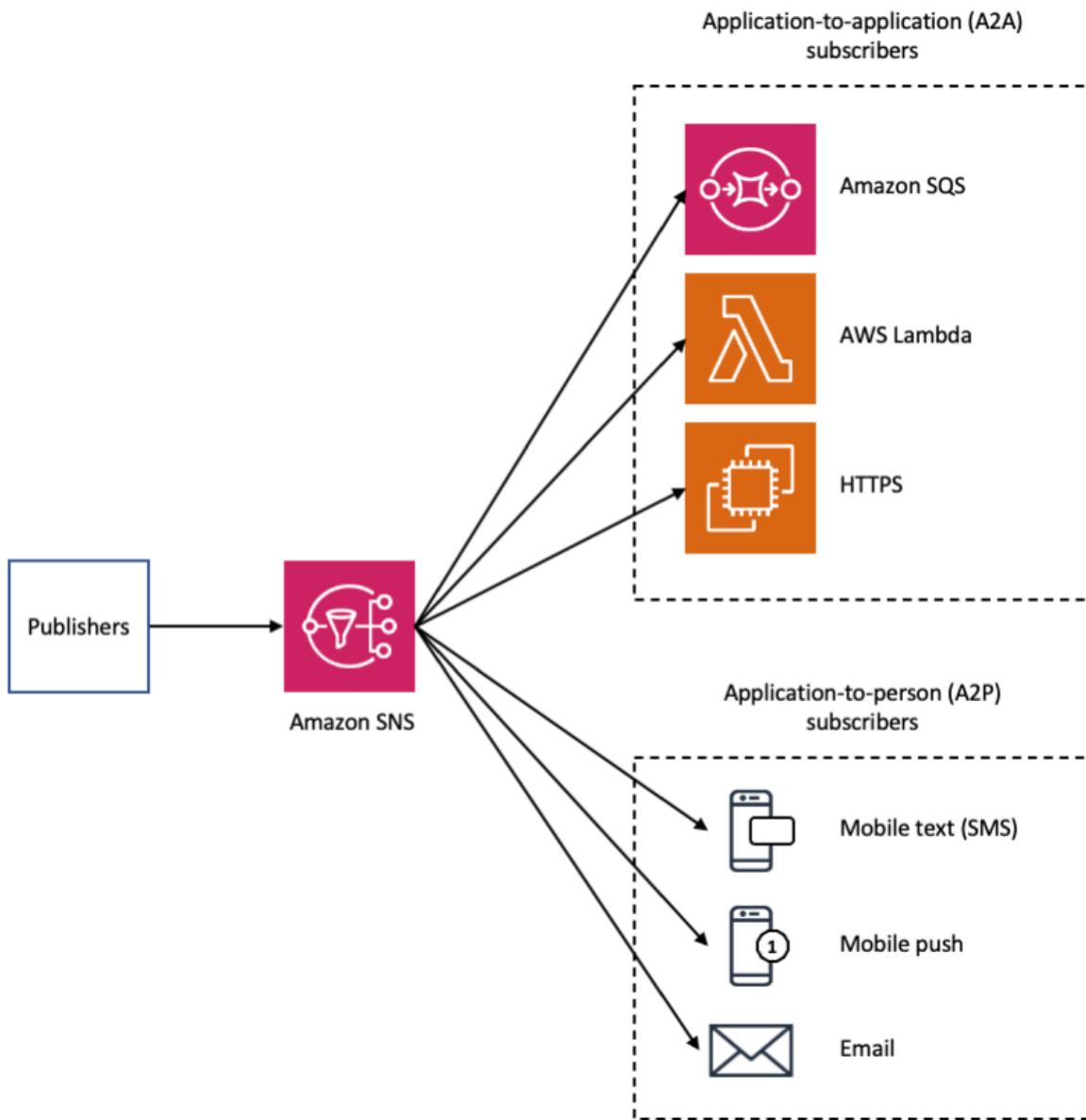
Serverless er en udvidelse af CloudFormation bygget til at gøre opsætning af Lambda-funktioner nemmere og hurtigere. Alle Lambda-funktioner bliver oprettet i sit eget dokument, hvor events såsom API eller SNS bliver sat på, og derefter skriver Serverless CLI filen om til et CloudFormation-template, som kan blive deployed på AWS. Ved at bruge Serverless kunne vi som udviklere hurtigere komme i gang med udvikling af kildekode fremfor at sidde fast i udviklingen af API'er og Lambda-funktioner gennem CloudFormation. I CloudFormation vil vi skulle skrive omtrent tre gange så meget for at få samme resultat, som Serverless CLI gør med en kommando.

SQS

Simple Queue Service eller SQS er en sikker og robust løsning på en hosted kø. SQS understøtter to typer af køer: Standard kø giver bedst mulig sortering, da den næsten ubegrænsede kapacitet kan gøre, at nogle beskeder ikke bliver kørt i samme rækkefølge, som de kom ind. Dog giver dette mulighed for næsten ubegrænsede API-kald pr. sekund. FIFO kø: "First-in-first-out", sikrer at rækkefølgen af beskeder, der kommer ind, bliver bevaret og kørt i rækkefølgen. Dette gør dog at de ubegrænsede API-kald, bliver til 3000 kald pr. sekund ved brug af batching eller 300 kald uden. Vores applikation bruger SQS til at sikre datakonsistens, da alle kald der skal gemme data i DynamoDB, bliver lagt i en FIFO kø. Med denne FIFO kø kan vi sikre os at hver besked sendt til API'et, i sidste ende bliver kørt også, hvis tjenesten der skal læse beskederne, midlertidigt er nede.

SNS

Simple Notification Service eller SNS er en notifikationstjeneste. SNS kan bruges i to former: A2A (Application-to-application), hvor notifikationen fra en udbyder bliver sendt til en anden applikation eller tjeneste, som abonnerer på den. A2P (Application-to-person), hvor notifikationen bliver sendt til en person i form af SMS, e-mail eller andet. De to former bruges vidt forskelligt. A2A bliver oftest brugt til at starte en tjeneste såsom en Lambda-funktion, hvor A2P bliver brugt til at underrette en person, om der for eksempel er sket en fejl. De to former for SNS kan ses illustreret nedenfor.



Figur 10: SNS mulige arbejdsgange.¹

I vores applikation brugte vi SNS til at starte vores forudsigelser omkring konsulentens arbejdstider og hvilke kunder, de har arbejdet hos. Ved at bruge SNS sender vi en besked ud til alle sandsynlighedstjenesterne om, at de skal starte. Hvis en af dem er nede, eller et API-kald som en af dem kalder er nede, kører applikation videre uden dataen. SNS sikrer os at vi sender beskeden, men ikke at den bliver modtaget. Dette gør at vores applikation ikke vil hænge og vente, men istedet bare køre videre med den data, der var mulig at få. SNS gjorde det muligt for os at modtage beskeder på vores mobil, f.eks. når en test fejlede. Ulempen ved dette var, at der blev sendt beskeder omkring alles fejl og ikke kun udviklerens egne fejl, så det blev derfor hurtigt slået fra igen. SNS er en meget robust løsning til alt inden for notifikationer, men der findes også løsninger som Twilio, der tilbyder et API til at sende SMS og Email. For os var det en nem løsning at vælge SNS da vi i forvejen arbejdede med AWS og tjenesten var indbygget heri.

¹<https://docs.aws.amazon.com/sns/latest/dg/welcome.html>

CloudWatch

Amazon CloudWatch er en monitoreringstjeneste, som bruges af DevOps-ingeniører og udviklere til at se logs og events. CloudWatch kan bruges til at finde uregelmæssigheder, visualisere log, sætte alarmer op til at sende e-mails eller SMS'er ved samarbejde med SNS eller til fejlfinding af problemer i Lambda-funktioner. Da vores app bygger på Lambda-funktioner, er CloudWatch en fantastisk tjeneste på grund af dens måde at opdele logs for hvert push til Github og dets meget informationsrige logs, er en nødvendighed, når der ved bare ét tryk i applikationen, bliver kørt et til ti forskellige Lambda-funktioner. Brugen af CloudWatch var nødvendigt i arbejdet med AWS, da der ikke var nogen anden mulighed for at se logfiler over den implementerede software. AWS-tjenesterne kan nemlig kun blive kørt fra AWS selv og kan derfor kun blive logget i CloudWatch-tjenesten. Uden logs ville vores gruppe ikke have fået lige så langt i udviklingen, da de nye AWS-tjenester ofte havde mange fejl samt, at AWS var en ny og svær teknologi at lære.

DynamoDB

DynamoDB betegnes som en NoSQL database, der understøtter nøgle-værdipar. DynamoDB arbejder ikke med relationer, da hastighed er topprioritet hos AWS. DynamoDB gør brug af en primærnøgle, som skal være skalerbar. Primærnøglen kan tage form som en enkelt nøgle kaldet partitionsnøgle, som er hashable, eller som to nøgler, hvor den første stadig er en partitionsnøgle, som er hashable og hvor den anden nøgle er en sorteringsnøgle. Ved brug af sorteringsnøglen behøver partitionsnøglen ikke at være unik. Fordelen ved DynamoDB er dens hastighed, idet der ikke er nogen relationer og at partitionsnøglerne, kan hashes. Konsekvensen af dette medfører dog at dataen ikke altid er konsistent, da DynamoDB skalerer ud på flere servere, når der kommer et stort pres på den. DynamoDB var et godt valg for vores applikation, da hastighed var højere prioriteret end konsistens. Vi vidste, at med tiden ville vores data i databasen være konsistent, men i situationen, hvor det skulle bruges, var det vigtigt at den indgående data, blev skrevet hurtigt til serveren, så konsulenten skulle bruge mindst muligt tid i applikationen. Arbejdet med DynamoDB var anderledes, eftersom vi skulle tænke over fordeling af partitionsnøgler, så vi fik mindst mulige kollisioner. Tidligere har vi arbejdet med MySQL, hvor nøglerne har automatisk forøgelse, og man derved også kan lave relationer.

S3

S3 også kaldet Amazon Simple Storage Service er en cloud storage-tjeneste med automatisk skalering. Med S3 kan man til hver en tid alle steder fra, hente filer til brug på for eksempel hjemmesider. Vores applikation gør ikke specielt meget brug af S3, men al vores kildekode bliver zippet og gemt her, når CloudBuild henter det fra Github. På den måde kan vi sikre os, at vi har versionskontrol, da alle pushes har sin egen unikke ZIP-fil gemt i vores S3 bucket. Nedenfor kan ses et udsnit af vores S3 bucket, hvor kildekoden bliver gemt med unikke versionsnumre.

Version ID	Type	Last modified
GgUwR2Vaa3AmmKyawfVF47Ip3AM.YI.2 (Current version)	zip	December 11, 2020, 16:05 (UTC+01:00)
GklAuWAfo.Nnt9bXrAyn0KK9lBhJQUYD	zip	December 11, 2020, 11:53 (UTC+01:00)
j3bjbDI6pz7udbuPE1KKtjQkhQcuFH0h	zip	December 8, 2020, 14:51 (UTC+01:00)
NMG_INp4LDH7Ymz_9QkOoDrRTEefsZ	zip	December 8, 2020, 12:38 (UTC+01:00)
Hb1PJ51Q7Pc4Xhk6u.gBl8Hkbfls3oc8	zip	December 8, 2020, 11:31 (UTC+01:00)
xm9gfcEzcxTxtlOOTJM9CNmTGlsFaKwK	zip	December 8, 2020, 11:23 (UTC+01:00)
EShR2W1hEv1TXySmdTlczTshRmq7wuHJ	zip	December 7, 2020, 16:02 (UTC+01:00)
yUJ6Zxqz0LCIzrn66gWAeMxE0I9eEJKD	zip	December 7, 2020, 14:15 (UTC+01:00)
tcCwwyWPlyO8.N3QGGuoAzbnl_9D2EDU	zip	December 7, 2020, 14:00 (UTC+01:00)

Figur 11: Versionsstyring i S3 Bucket.

Udover S3 brugte vi også Github til versionsstyring, og vi kunne dermed have undladt S3 versionsstyring. Vi valgte at gøre AWS så fri fra Github som muligt, så vi principielt kunne skifte Github-arkivet ud, og stadig have tidlige versioner af både udviklings- og produktionsmiljøet liggende.

6.3 Python 3.8

Python er et high-level, platformsuafhængigt, Open Source programmeringssprog, udviklet af Guido van Rossum i 1991. Med Python er der rig mulighed for at bruge forskellige programmeringsparadigmer såsom objektorienteret programmering eller funktionel programmering. I vores projekt brugte vi Python version 3.8, som er indbygget med en bedre type deklaration, hvilket også giver os mulighed for både at arbejde funktionelt og objektorienteret. Valget om at bruge Python tog vi som gruppe, hvilket bunder i at vi lige inden praktikken, havde haft Python som valgfag, som medførte at sproget var friskt i vores erindring. Efio havde interesse i at udviklingen af applikation skete i Node.js, da product owner havde mest erfaring inden for dette, men efter en kort debat fandt vi frem til, at ud fra vores kompetencer, ville det egne sig bedst at udvikle applikationen i Python. De Python biblioteker som vi endte med at bruge er:

- pynamodb, til at gemme og hente data fra DynamoDB
- requests, til at sende HTTP requests
- injector, til at implementere Dependency Injection
- pytz, til at ændre tidszone på datetimes
- ruamel.yaml, til at læse og ændre i YAML
- behave, til at automatisere acceptance test
- pylint, til at følge python kodenstandarde
- boto3 til at snakke med AWS tjenester
- botocore til at opsætte AWS til brug i boto3

Set i bakspejlet af vores valg kunne det være, vi skulle have brugt Node.js, da sproget er mere udviklet inden for AWS' tjenester. Bibliotekerne til AWS eri Node.js bedre vedligeholdt end i Python, men med vores viden og kompetencer inden for Python fik vi tingene til at fungere, ved selv at videreudvikle på bibliotekerne.

6.4 Vurdering

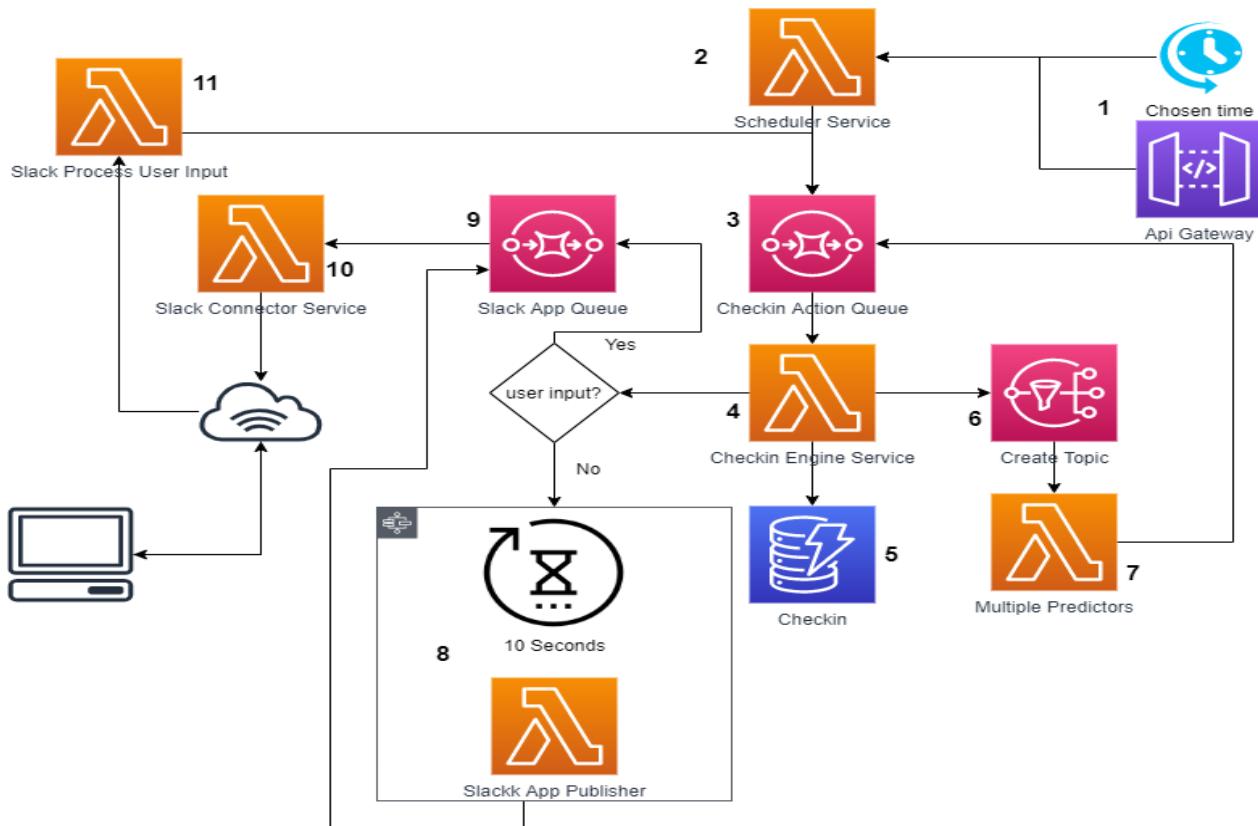
Valget af teknologier var op til os som gruppe, men Efio's erfaring indenfor faget gjorde, at vi hurtigt kunne tillære os nye teknologier. Python var et valg, vi tog ud fra tidligere erfaring, mens brugen af AWS var et krav fra Efio. Efter det afsluttede praktikforløb stod det klart for os, at AWS tilbød services, der var til gavn for vores udvikling af applikationen. Derfor var det et oplagt valg at fortsætte vores udvikling i samme stil i hovedopgaveforløbet og derved benytte de samme services, som applikationen allerede var bygget på. Vi har som gruppe haft mange store udfordringer undervejs, hvilket har givet os anledning til at gå mere i dybden med de forskellige services AWS udbyder. Vi opnåede ret gode resultater med AWS og hvis vi skulle udvikle applikationen på ny, ville vores valg af teknologier stadig forblive uændret.

7. Valg af arkitektur

I dette afsnit vil vi redegøre for den valgte arkitektur i applikationen. Ydermere vil vi forklare om arkitekturen bag vores CI/CD-pipeline samt arbejdsflowet med Github og de status'er, der bliver sendt mellem Github og AWS.

7.1 Backend

Systemet er sat op til at skulle køre ud fra et bestemt tidspunkt. Under praktikperioden var tiden fastsat til kl. 9.00 sommertid, men i løbet af hovedopgaveperioden er dette blevet ændret til, at konsulenten selv kan vælge, hvilket tidspunkt de vil modtage deres notifikation om deres daglige tidsregistrering. Der er også sat en API-gateway op til at lave en tidsregistrerings knap til alle, dog bruges denne kun til udvikling og test. En illustration af systemets backend kan ses forneden, hvor tallene på billedet illustrerer, hvilken del i systemet der bliver talt om.



Figur 12: Backend Arkitektur.

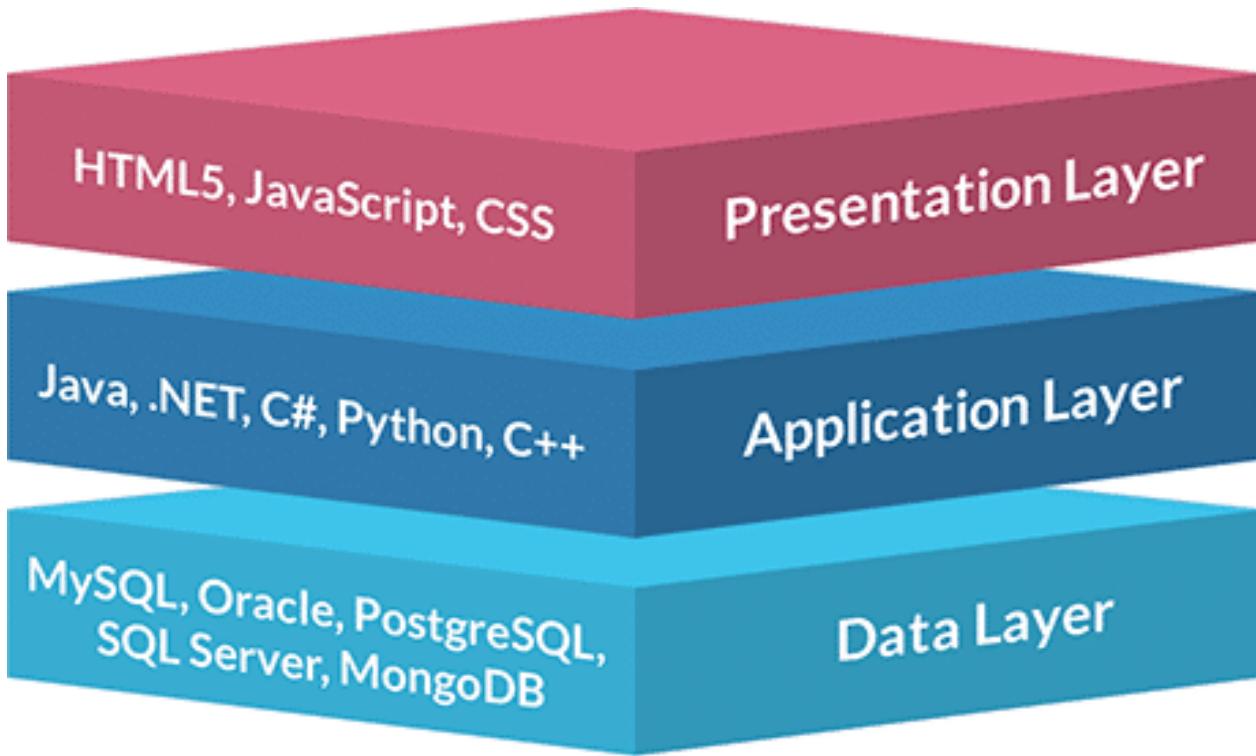
Flowet i backenden på figur 12 fungerer således at, hvert 5. minut køres der et event (1) der starter Scheduler Service (2). Hvis en konsulent f. eks. har valgt at få sin daglige tidsregistrering kl. 12.04 sommertid, vil en besked om, at konsulenten kan starte sin tidsregistrering, blive lagt ind i SQS (3) (Checkin Action Queue) kl. 12.05 sommertid. Checkin Engine Servicen (4) abonnerer på køen og modtager dermed beskeden, hvori der findes et "type" felt, der fortæller tjenesten, hvad der skal gøres med beskeden. Typen her er "Scheduled", som fortæller tjenesten, at der skal oprettes et ny tidsregistrering i databasen (5) med tomme felter. Når den nye tidsregistrering er oprettet sendes en besked til Create Topic-tjenesten (6) (SNS). Multiple Predictions (7) er flere Lambdafunktioner, der lytter efter en besked fra Create Topic. Når de modtager en besked, går de igang med at lave forudsigelser på estimeret arbejdstid og potentielle kunder. Disse forudsigelser bliver derefter sendt til Checkin Action Queue(3), hvorefter Checkin Engine Servicen (4) modtager beskeden, hvori typen er "Prediction". Typen "Prediction" fortæller at forudsigelserne skal gemmes i databasen (5).

Imens den ovenstående sektion har kørt, har Checkin Engine Service også sat gang i en stepfunktion (8), som venter 10 sekunder, før den sender en besked til Slack App Queue (9). Den aktiverer herefter Slack Connector Service (10), som til sidst sender en knap til konsulenterne, hvor de kan påbegynde deres tidsregistrering.

Alt ovenstående er alene for at oprette et checkin for en konsulent og efter dette kan brugeren svare tilbage ved at bruge applikationen. Når brugeren trykker på knappen, eller generelt interagere med applikationen, sender Slack et API-kald til Slack Process User Input (11), der håndterer alt, hvad brugeren trykker på. Derefter sender den en besked til Checkin Action Queue (3) med den korresponderende type i forhold til, hvad brugeren har gjort. Checkin Engine Service håndterer beskeden og gemmer brugerens input i databasen (5).

7.2 Trelags-arkitektur

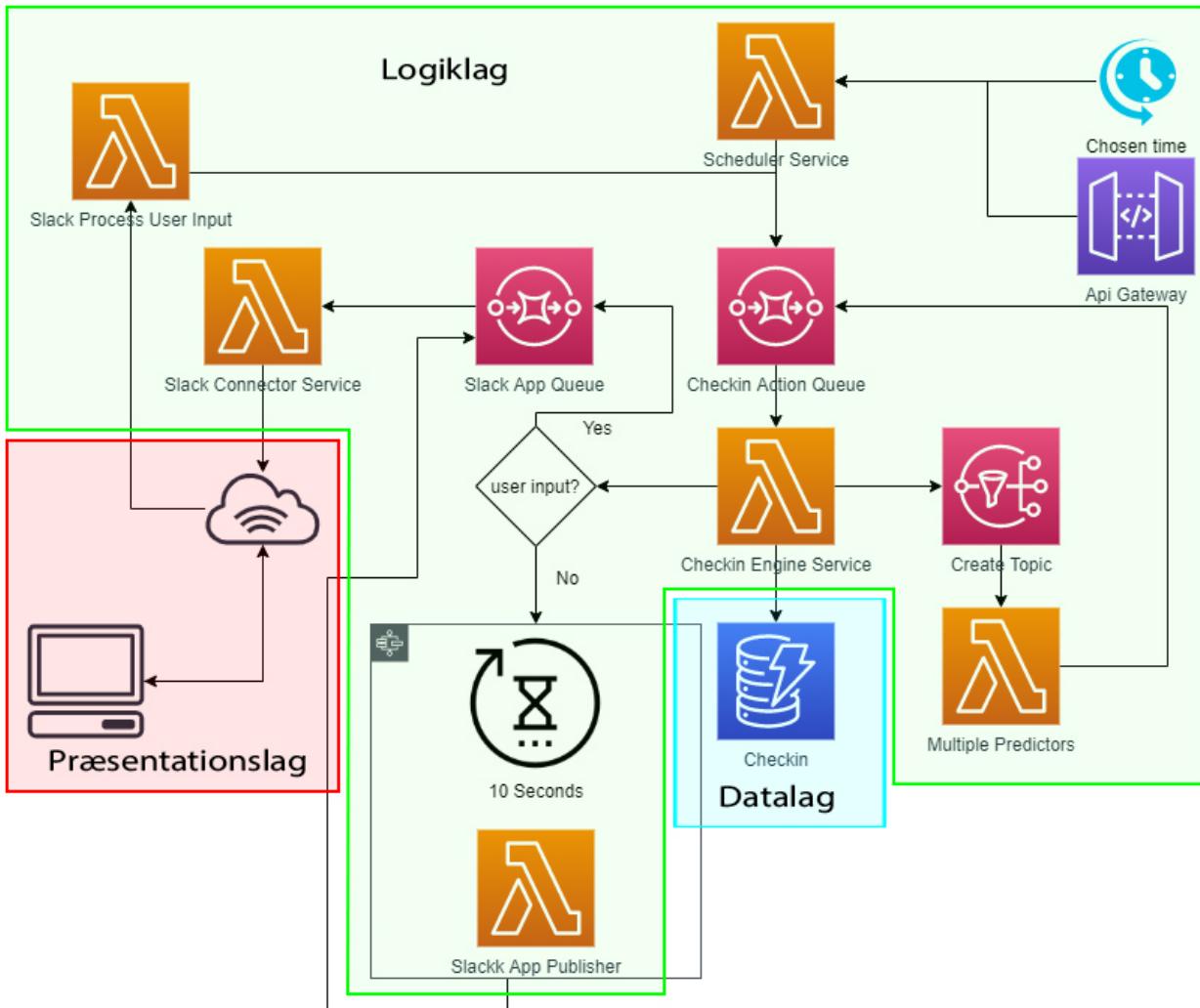
Trelags-arkitektur er en softwarearkitektur, der består af tre lag. Det første lag set oppefra er præsentationslaget, som indeholder frontend-delen af et system. Dette lag indeholder en brugergrænseflade som viser indhold og information, der er brugbar for brugeren. Det næste lag er logiklaget, hvilket består af den funktionelle forretningslogik, der står for at køre applikationens kernefunktioner. Til sidst har vi datalaget, der består af databaser samt dataopbevaring i form af lagringssystemer. Nedenfor ses en illustration af de tre lag, hvor man også kan se de forskellige programmeringssprog, lagene oftest er skrevet i.



Figur 13: Model af Trelags-arkitekturen.²

Vores applikation blev udviklet ud fra Trelags-arkitekturen. Til datalaget bruger applikationen DynamoDB tjenesten hos AWS, som gør det nemt og overskueligt at gemme data både i Python men også gennem deres indbyggede API. Logiklaget er det største lag af vores applikation og indeholder alle Lambda-funktioner, REST API'er, Step Functions, SNS og SQS. Hele logiklaget kører på AWS og kan snakke sammen indbyrdes gennem deres event system. Da vores applikation er en integration til Slack, er vores præsentationslag hostet og udviklet af Slack. Dette gør at vi gennem logiklaget, kan sammensætte en template i JSON og sende til Slack, som derefter håndterer rendering af vores app ud fra templatet. Ved at dele applikationen op i disse tre dele, ville vi nemt og hurtigt kunne udskifte et lag. Hvis Slack valgte at lukke ned, kunne vi for eksempel hurtigt sætte en hjemmeside op, der kan rendere brugergrænsefladen ud fra vores JSON filer og derved erstatte præsentationslaget. Nedenfor ses en illustration af vores backend opdelt i de tre arkitekturlag.

²<https://www.jinfonet.com/resources/bi-defined/3-tier-architecture-complete-overview/>

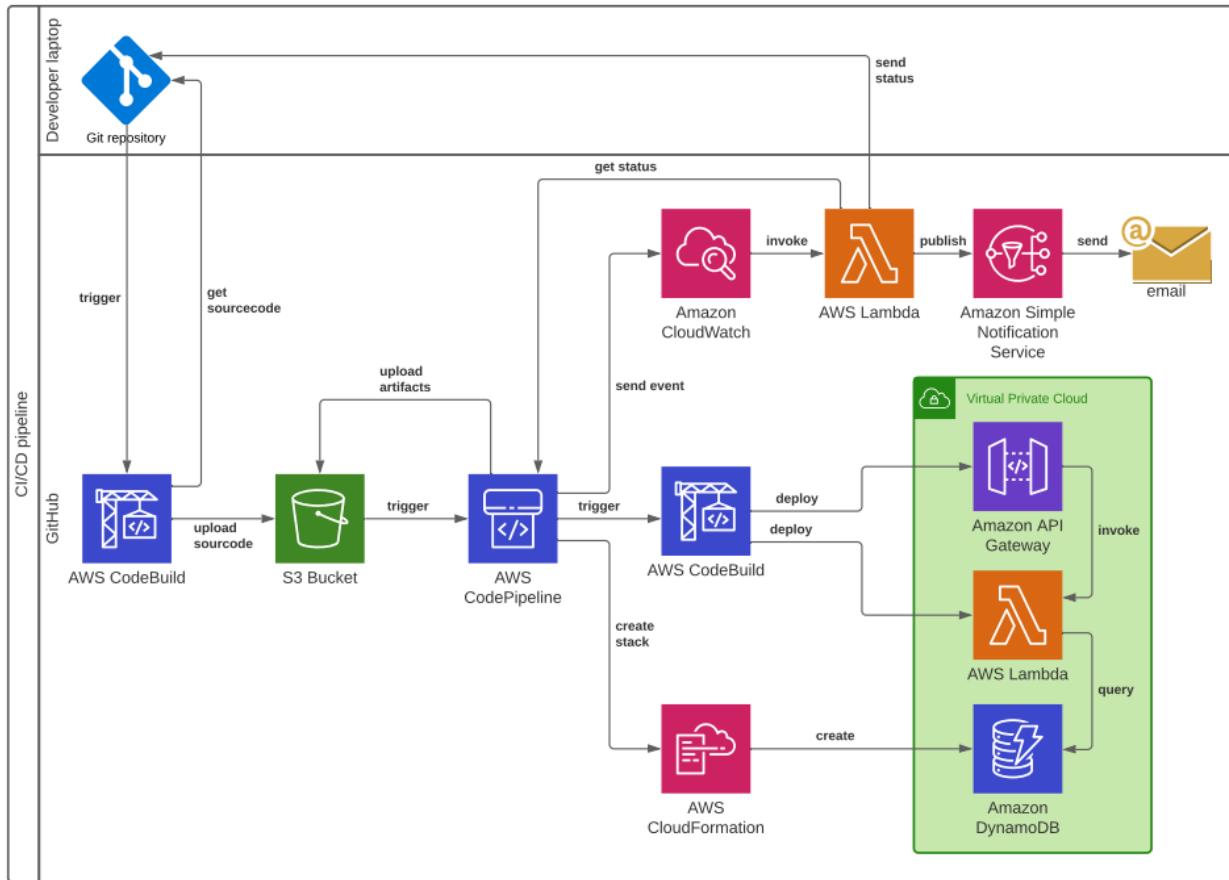


Figur 14: Applikationens backend opdelt i de tre arkitektur lag.

7.3 Microservice

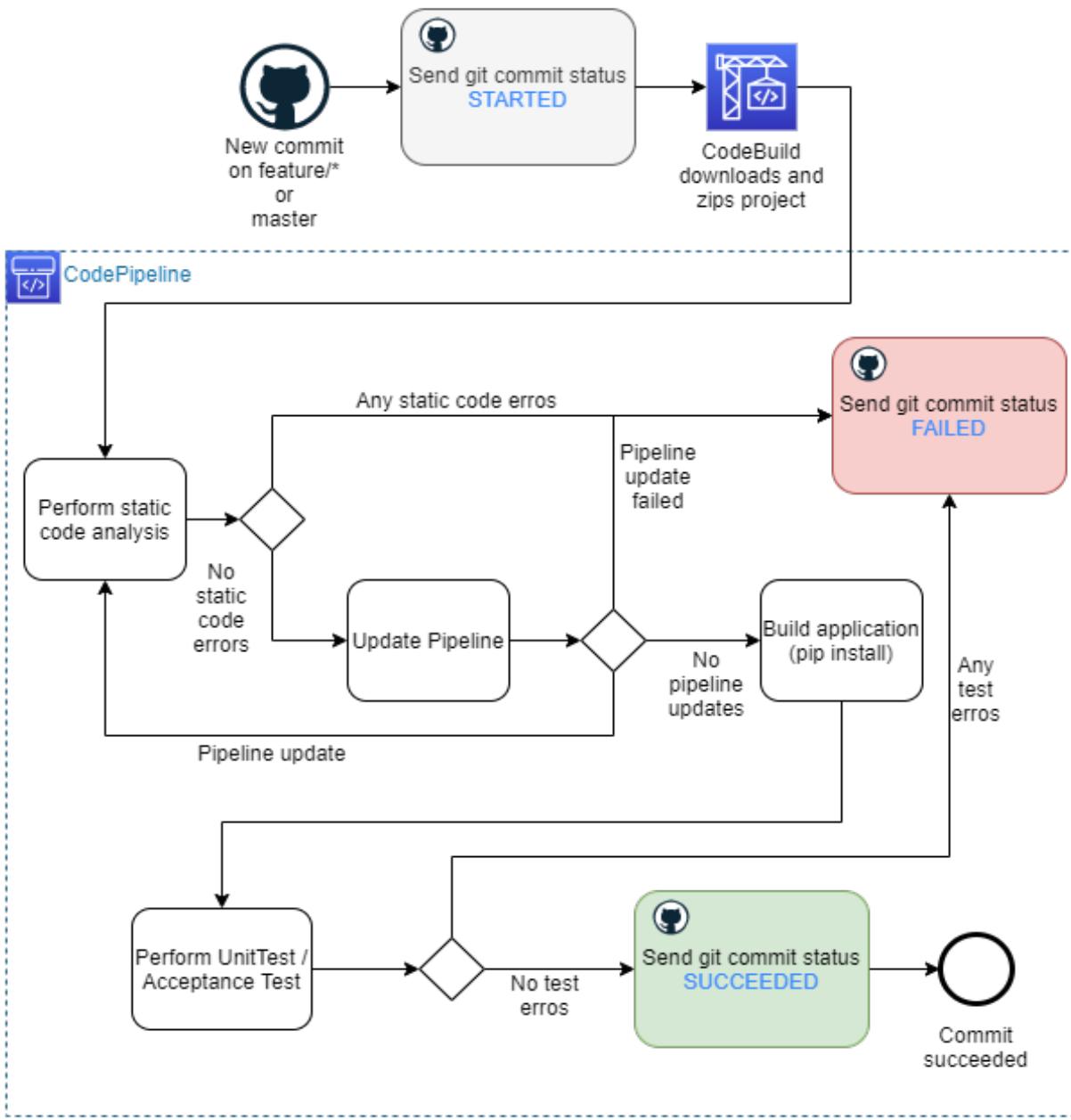
Hele vores applikation er opbygget af microservices, som er en arkitektonisk tilgang til softwareudvikling, hvor softwaren bliver organiseret og delt op i små og uafhængige moduler, som hver har et veldefineret API. Ved at dele softwaren op i små moduler øges effektiviteten og gør dem mere skalerbare. Ved et normalt system vil hele systemet skaleres ud, hvis der kommer en stigende efterspørgsel på et modul, hvorimod med microservices vil det kun være det enkelte modul, der skaleres ud. Vores applikation gjorde kun brug af microserviceprincippet, da alle Lambda-funktioner er deres eget lille modul, som tager sig af hver sin ting. Funktionerne har deres eget event tilknyttet, som f.eks. kan være et API eller at abonnere på en kø. Uden microservices ville hele vores system skulle køre på en EC2-server, hvor hele den server skal skaleres ud efter forespørgsel på applikationen. Det har været spændende at arbejde med microservices og det har virkelig givet et godt indblik i, hvor brugbart det er. Microservices er helt klart noget af det vi vil tage med videre efter hovedopgaveforløbet.

7.4 CI/CD-pipeline



Figur 15: Implementering af CI/CD-pipeline på AWS.

For at have et godt udviklingsflow til applikationen valgte vi at opsætte en kontinuerlig integrations- og en kontinuerlig implementerings pipeline. Disse pipelines gør det muligt for os at automatisere alt indenfor test og implementering. Som det ses illustreret på figur 15, er vores CI/CD-pipeline bygget op af en masse tjenester på AWS. Flowet starter med et Github Push fra en udvikler, som udløser en besked til AWS CodeBuild. CodeBuild vil zippe kildekoden og gemme det i en S3 bucket. Herfra vil vores Pipeline begynde at teste kildekoden og hvis der sker en fejl, vil der blive sendt en besked til Github og en SNS-tjeneste, vil sende en mail til udvikleren. Er der ingen fejl, vil CodeBuild og CloudFormation bygge og deploye alle Lambda-funktioner, køer, API'er og databaser. Nedenfor ses også et flow af selve pipeline-delen, fra når det kommer ind i CodeBuild fra Github, til det enten mislykkes eller lykkes og sender en besked til Github.



Figur 16: Illustration af Github besked flowet.

7.4 Vurdering

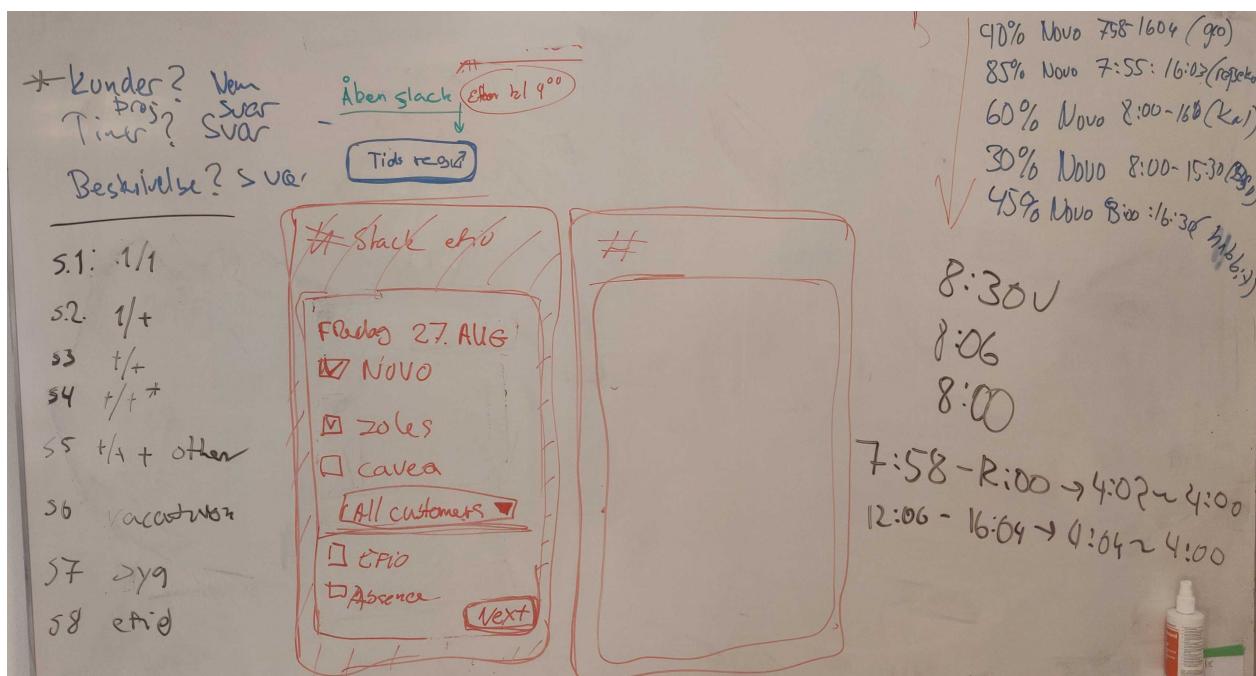
Gennem projektet har vores arkitektur udviklet sig og vi har lært at gøre brug af mange forskellige tjenester inden for microservices. Den arkitektur vi er endt med, hvor vores applikation er delt op i tre arkitektoniske lag, har gjort applikationen robust. Vi kan udskifte et helt lag, hvis vi f.eks. skifter cloud-løsning, men vi kan også udskifte enkelte microservices, hvis vi finder mere optimale løsninger. Hvis vi som gruppe ikke havde haft krav om arkitekturen fra product owner, ville vores arkitektur sandsynligvis ikke benytte sig af hverken microservices eller Github status. Derfor er den valgte arkitektur meget fordelagtig for applikationen da Efio, efter vores endte forløb, selv kan videreudvikle og udskifte microservices, hvis de finder på bedre løsninger.

8. Design

I dette afsnit vil vi præsentere, dokumentere og argumentere for vores valgte design i form af brugergrænseflade- og database design.

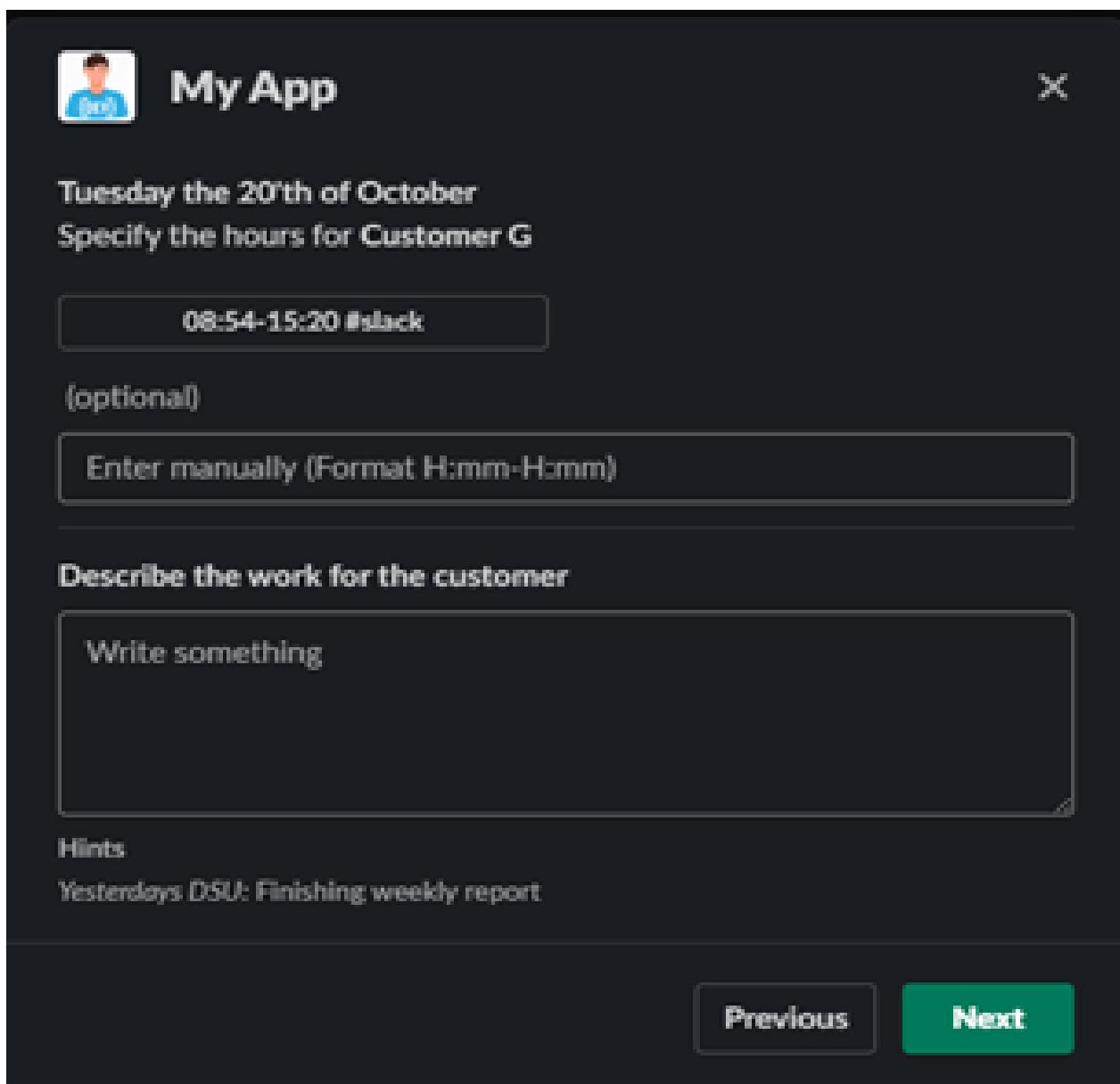
8.1 Brugergrænseflade

Vi har siden praktikperioden ændret designet på tidsregistrering og beskrivelsesinput på brugergrænsefladen. Dette gjorde vi, da nogle kunder kunne have både ét eller flere projekter. Dermed har vi skulle udvikle et design, der kunne håndtere flere projekter på den enkelte kunde, hvor vi stadig skulle sørge for en god brugeroplevelse. I det originale design fra praktikperioden, fyldte konsulenten sin tid ud med et start- og sluttidspunkt og derefter en beskrivelse af gårdsdagens arbejde i et andet felt. Dette design har ikke været noget, som konsulenterne var vant til og det er derfor, vi i hovedopgaveforløbet ændrede designet. Nedenfor ses et billede tegnet af product owner, hvor han har illustreret designet for valg af kunder og hvordan timerne skal udregnes ud fra start- og slut tidspunkt.



Figur 17: Illustration af design og beregning af timetal.

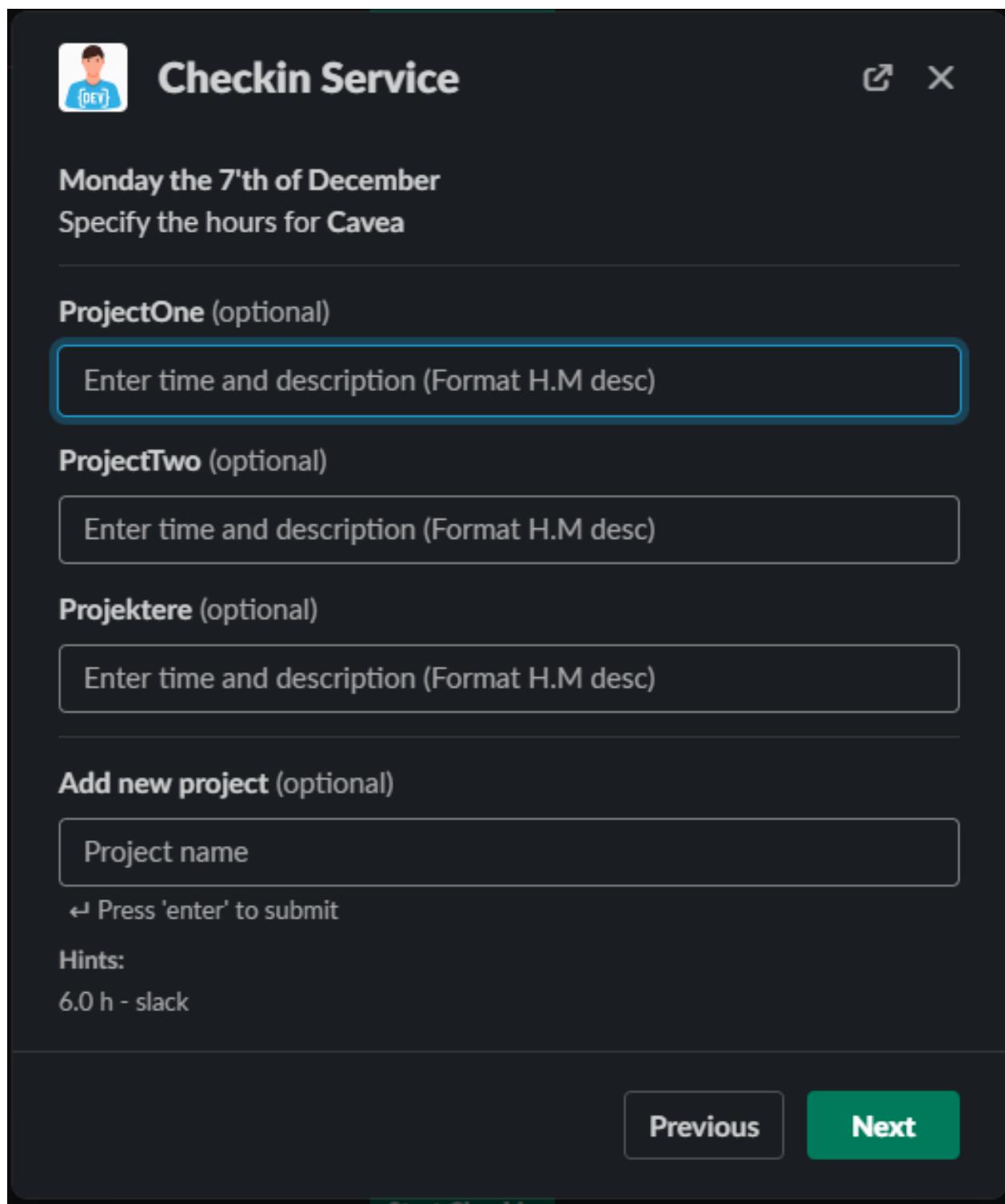
Gennem praktikforløbet stødte vi på et problem med de begrænsninger, der er i Block Kit, hvilket gjorde det svært at være extraordinært kreativt. Dermed skulle designet kunne tage imod en masse information, men på samme tid være simpelt og brugervenligt. Nedenfor ses et billede af designet som det så ud efter praktikforløbet.



Figur 18: Gammelt design fra praktikforløb.

Det første udkast af et nyt design i hovedopgaveforløbet, lignede meget det gamle. Her ville vi dog i stedet lave et nyt billede, for hvert projekt kunden havde. Dermed ville vi holde fast i det design, konsulenterne allerede kendte. Konsulenten vil så kunne se projektnavnet i toppen ligesom kundenavnet og have mulighed for at registrere tid og beskrivelse til projektet, hvis de har arbejdet på det og ellers bare trykke videre. Dette design blev kasseret, da det giver en unødvendig lang process for en konsulent, at skulle registrere tid for en enkelt kunde, hvis de eksempelvis kun har arbejdet på et projekt den pågældende dag. Derudover ville det blive meget ensartet og samtidig, vil der være stor risiko for, at en konsulent får registreret sin tid forkert ved ikke at være opmærksom på navnet af enten projektet eller kunden.

Product owner valgte i stedet at gå med det design, som kan ses herunder. Det er lidt mere kompakt da både tidsregistrering og beskrivelse, bliver skrevet i samme felt. På den måde kan vi have alle aktive projekter for en kunde på samme side, da de fylder så lidt som muligt. Dette design blev valgt, da vi vurderede, at det gav et bedre overblik og konsulenterne skulle bruge færre "klik" til at gennemføre tidsregistrering.



The image shows a mobile application interface titled "Checkin Service". At the top left is a user icon with the text "{DEV}". The title "Checkin Service" is centered at the top. On the right side are a refresh icon and a close button ("X").

Monday the 7'th of December
Specify the hours for **Cavea**

ProjectOne (optional)
Enter time and description (Format H.M desc)

ProjectTwo (optional)
Enter time and description (Format H.M desc)

Projektere (optional)
Enter time and description (Format H.M desc)

Add new project (optional)

Project name
↔ Press 'enter' to submit

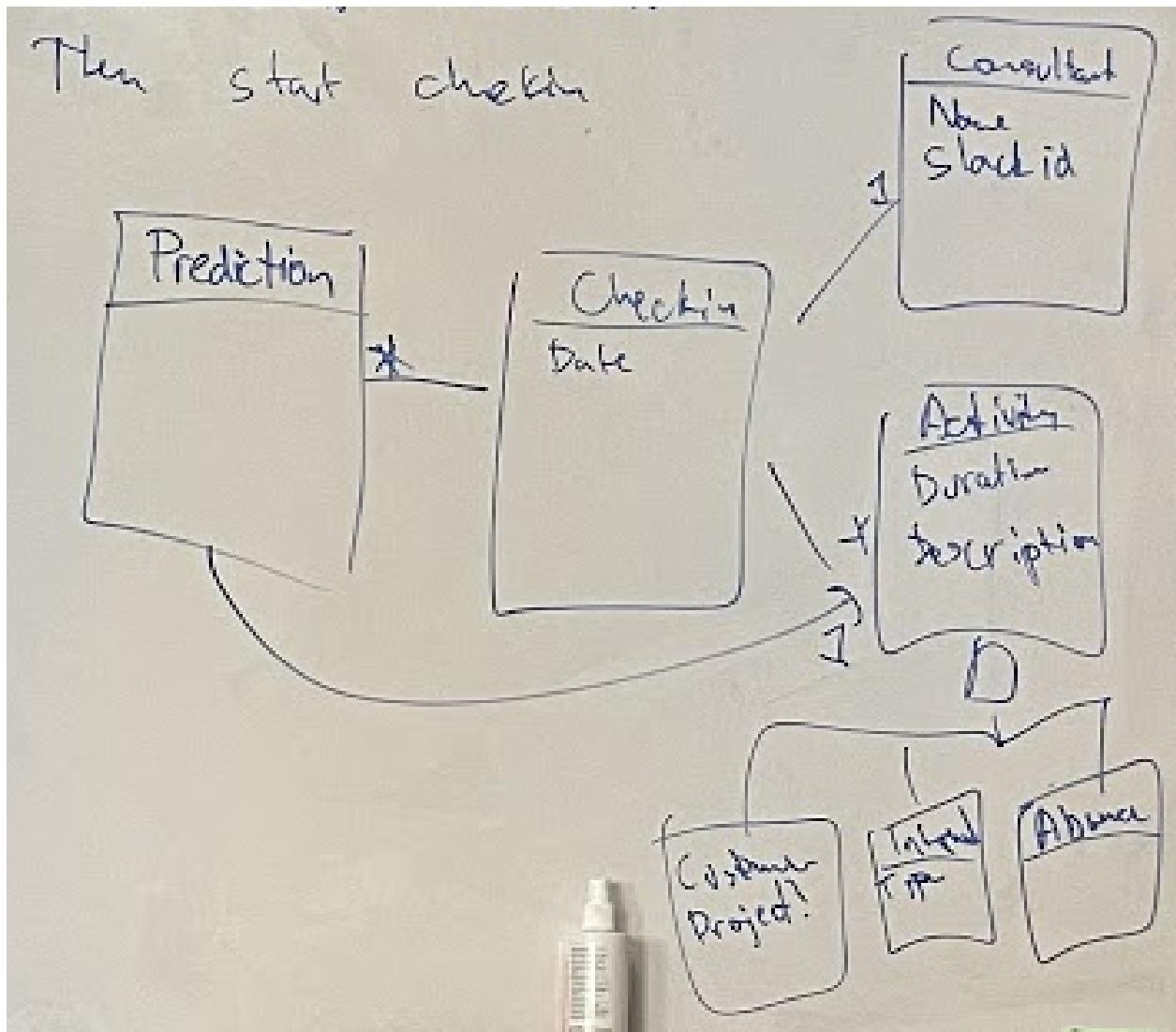
Hints:
6.0 h - slack

At the bottom right are "Previous" and "Next" buttons. The "Next" button is highlighted in green.

Figur 19: Nyt design fra hovedopgave forløbet.

8.2 Databasedesign

Databasen blev designet af product owner, men med inputs fra os i form af bedre eller alternative tilgange. Brugen af NoSQL betød for vores design, at vi ikke arbejdede med relationer, men i stedet gjorde brug af JSON-formaterede celler. Dette gjorde hastigheden for databasen hurtigere, da der ikke skulle laves nogle interne kald for at få alle relationer med. Nedenfor ses et billede af product owners første udkast til databasedesignet, hvor en "checkin" har en "consultant" og flere "predictions" som består af flere "actions". Designet er tegnet med relationer for at visualisere designet bedre, men relationerne vil blive lavet som JSON-formaterede celler.



Figur 20: Første udkast til database design fra product owner.

Det endelige design gjorde brug af flere tabeller med unikke ID'er, så vi kunne hente konsulenter til checkins ud fra ideerne. Dette gjorde at vi stadig ikke havde relationer men i stedet havde et link mellem tabellerne, som vi kunne bruge, hvis der var behov for det. Nedenfor ses det endelige databasedesign, hvor tabellerne har et *uuid*, som er det unikke ID, der kobler dem sammen, markeret ved en nøgle. Tabellen "online_statuses" gør også brug af en sorteringsnøgle, markeret med to pile som betyder, at vi ikke bruger et unikt ID, men hvert ID i stedet har en unik sorteringsnøgle i stedet. Der er også tre JSON-formaterede celler, beskrevet med tuborgklammer omkring tabelnavnet.



Figur 21: Illustration af det endelige database design.

8.3 Patterns

Python er dynamisk skrevet, hvilket vil sige at typetjekket sker ved runtime og ikke ved kompilering. Derved kan et Python-program sagtens køre, selvom en variabel der bør være et heltal, måske er en streng. Fejlen vil først opstå, når programmet prøver at bruge variablen, hvor typen er forkert. For at gøre det nemmere for udviklere at se typen på en variabel, har Python i version 3.5 integreret typing som hints. Nedenfor ses én funktiondeklarering uden typing og én med typing. Begge disse funktioner vil virke og er lige gode til formålet, men den nederste med typing er nemmere at forstå, for udviklere der skal bruge funktionen.

```
def random_number(start, end):  
def random_number(start: int, end: int) -> int:
```

Figur 22: Funktion deklaration med og uden typing.

8.4 Vurdering

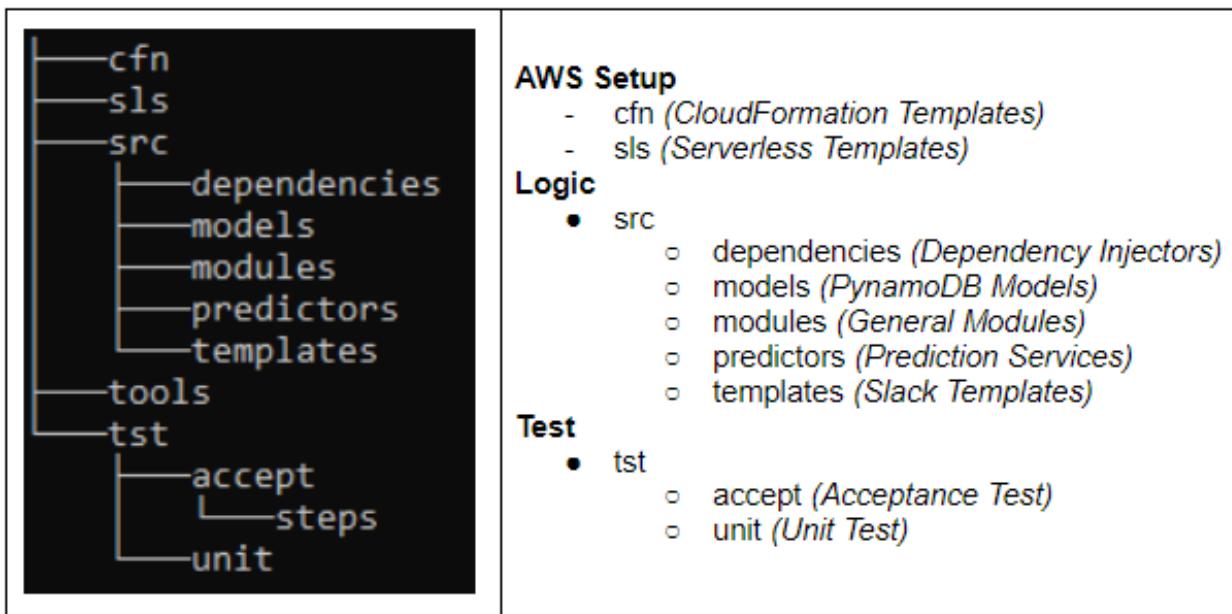
Designet er blevet lavet gennem flere iterationer, hvor hver iteration har gjort brugervenligheden bedre for konsulenterne. Ved at ændre designet efter feedback fra praktikperioden, synes konsulenterne at det er hurtigere at registrere deres tid. Vi som gruppe synes, at der er blevet opnået en tilstrækkelig brugervenlighed, da designet er enkelt og kun indeholder den information, der er nødvendig for at registrere sin tid som konsulent. Ydermere er brugervenligheden for udviklere, der skal videreudvikle på koden god, da vores patterns har gjort vores kode mere letlæselig og gennemskuelig.

9. Implementering

I dette afsnit vil vi fremvise spændende og lærerige dele af vores kode. Disse kodededele er valgt, da de viser en god struktur og et tilfredsstillende stykke arbejde i vores projekt. De viser også, hvordan vi har arbejdet med nye teknologier og hvordan vores arkitektur, har været gennem funktionerne.

9.1 Mappestruktur

Projektets mappestruktur blev lavet for at opdele AWS-tjenesters setupfiler fra kildekode til Slack kommunikation. I de to første mapper: cfn og sls findes der opsætningsfiler til Cloudformation og Serverless. I CloudFormation-mappen er der filer til CI- og CD pipeline, database samt SQS- og SNS-tjenester og Serverless mappen indeholder filer til opsætning af Lambda-funktioner og krævede pythonbiblioteker. Den tredje mappe er src-mappen, der indeholder alle vores filer til Slack applikationer. Alle disse filer er Python-filer med undtagelse af template-mappen, som består af JSON-templatefiler til præsentations laget i Slack. Tools-mappen er den fjerde mappe, der har filer til at gøre udviklingen nemmere. Den sidste mappe er tst-mappen, som indeholder alle vores automatiserede test. Disse test er både Unit Test og Acceptance test. Nedenfor ses et “Tree” af vores mappestruktur samt en kort liste over, hvad mapperne indeholder.



9.2 Pylint

Pylint er et værktøj, der finder stilproblemer og bugs i Pythonkode. Dette gøres ud fra en stilguide, hvilket der findes mange af på internettet. Nedenfor ses et eksempel på et Pythonmodul, der kan returnere et tilfældigt tal mellem en startværdi og en slutværdi. I modulet er nogle overflødige ting, såsom import af time og deklarering af numbers, men dette er ikke noget, der får modulet til at fejle, dog kan det gøre modulet langsommere.

```
import time
import random

def random_number(start, end):
    numbers = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23]

    return random.randint(start, end)

print(random_number(1,101))
```

Figur 23: Python Modul uden Pylint.

Hvis Pylint bliver kørt gennem kommandoen `pylint <file>.py`, vil resultatet nedenfor blive vist. Her fortæller den os, at vi mangler en blank linje til sidst, at vi mangler dokumentation på både dokumentet og funktionen og at der er to overflødige objekter.

```
***** Module test
test.py:9:0: C0304: Final newline missing (missing-final-newline)
test.py:1:0: C0114: Missing module docstring (missing-module-docstring)
test.py:4:0: C0116: Missing function or method docstring (missing-function-docstring)
test.py:5:4: W0612: Unused variable 'numbers' (unused-variable)
test.py:1:0: W0611: Unused import time (unused-import)

-----
Your code has been rated at 1.67/10 (previous run: 1.67/10, +0.00)
```

Figur 24: Pylint kommando kørt på test filen.

Hvis vi følger Pylint og omskriver modulet, vil modulet ud fra kodenstanden se ud som nedenfor. Her har vi fra toppen lavet en dokumentation for filen med en license, fjernet importen af time, lavet typedeklaration, lavet funktion dokumentation med parameter, fjernet numbersvariablen og tilføjet en blank linje i bunden af filen. Med alt dette lavet er koden meget mere læsbar og struktureret.

```
...
Random Number Generator
:license: MIT
...
import random

def random_number(start: int, end: int) -> int:
    ...
        Returns random number between start and end

    :param start: start integer
    :param end: end integer
    ...
    return random.randint(start, end)

print(random_number(1,101))
```

Figur 25: Python Modul med Pylint korrektion.

I de næste afsnit, vil der være udsnit af vores kildekode, hvor hver af dem er godkendt af Pylint med undtagelser af Dependency Injection, som er kodet ud fra “injector”-bibliotekets kodestandard. I afsnittene er dokument og moduldokumentation fra Pylint blevet fjernet for mere overskuelig fremvisning.

9.3 Dependency Injection Factory for Boto3SNS

I vores projekt gjorde vi brug af Dependency Injection, som er en praksis, hvor man erstatter moduler med simulerede moduler, så vi kan køre test i et simuleret testmiljø, hvor hvert modul simuleres til at kunne det samme som produktionsmodulet. Hvis vi ikke havde brugt Dependency Injection, ville en anden løsning være, at have endnu et miljø hos AWS, der kunne bruges som testmiljø. Dette testmiljø ville have de samme tjenester, som de andre miljøer og ville derfor også koste penge at køre test på. Derfor valgte vi at simulere miljøet både for at spare penge, men især også for at kunne teste vores software lokalt. Nedenfor ses et udsnit af et Dependency Injection Factory, som skifter mellem AWS' SNS-modul og et simuleret Mock object baseret på, hvad det nuværende miljø er. Dette er bare et af de mange Factories, vi har i applikationen.

1	Dependency Injection Factory for Boto3SNS
	<pre>from injector import Injector, Module, provider, singleton class SNSProvider(Module): @singleton @provider def provide_boto3sns(self) -> Boto3SNS: return boto3.client('sns') class SNSProviderTest(Module): @singleton @provider def provide_boto3sns(self) -> Boto3SNS: return Mock() def get sns provider(test=False): if test: injector = Injector([SNSProvider(), SNSProviderTest()]) else: injector = Injector(SNSProvider()) return injector.get(Boto3SNS)</pre>

- 1.1 @singleton betyder at metoden i klassen kun skal oprettes i memory én gang og genbruges derefter.
- 1.2 @provider fortæller biblioteket injector, at denne metode er en udbyder. En udbyder betyder, at metoden vil returnere et objekt af den givne type.
- 1.3 Metoden provide_boto3sns bliver defineret og typen af provider, bliver sat til Boto3SNS.
- 1.4 Metoden returnerer en ny boto3 client af typen1 sns.
- 1.5 Den samme metode med samme providertype bliver oprettet under testklassen.
- 1.6 Metoden returnerer nu et Mock object, som er et simuleret objekt, der efterligner handlingerne af reelle objekter på en kontrolleret måde.
- 1.7 Get-metoden bliver defineret, så vi kan få vores provider baseret på hvilket miljø vi er i.
- 1.8 Hvis vores miljø er et testmiljø, erstatter vi den normale provider med vores testprovider i injektoren.
- 1.9 Hvis det ikke er et test miljø, bliver provideren sat til den normale provider i injektoren.
- 1.10 Metoden returnerer en provider fra den gældende klasse af typen Boto3SNS

9.4 Time/Description input box decoder

I applikationen har vi et inputfelt, hvor konsulenten både skriver antal timer og beskrivelse af dagens arbejde. Dette valg tog vi for at formindske pladsen, der bliver brugt i applikationen. For at få informationen fra inputfeltet gemt i databasen, skal det splittes, så f.eks. tiden kan beregnes korrekt. Her brugte vi regex til at dele inputtet op i forskellige grupper, så vi kunne formater dataen til forskellige Pythonobjekter. Uden regex var vi blevet nødt til at dele felterne op så tal og bogstaver var adskilt i inputfelterne. Nedenfor ses modulet der ved hjælp af regex, grupperer og formaterer dataen fra inputfeltet.

2	Time/Description input box decoder
2.1	<pre>def decode_time_desc(time_desc: str, description: bool) -> str: regex = r'^([0-2]?d)(([.,\.])(\d+))?((\s+)([^\\d].*))?' try: match = re.match(regex, time_desc) time_formatet = match.groups()[0] if match.groups()[2] in [',', '.']: time_formatet = '{0}{1}'.format(match.groups()[0], match.groups()[1].replace(',', '.')) elif match.groups()[2] == ':': if int(match.groups()[3]) < 35 and int(match.groups()[4]) > 4: time_formatet = '{0}{1}'.format(match.groups()[0], '.5') elif int(match.groups()[3]) > 35: time_formatet = '{0}'.format(int(match.groups()[0]) + 1) else: time_formatet = '{0}'.format(match.groups()[0]) value = { 'input': time_desc, 'time': float(time_formatet) } if len(match.group()) == 6 and match.group()[6] is not None: value['description'] = match.groups()[6] elif description: return "Decode Error" return value except: return "Decode Error"</pre>

2.1 Regex som matcher på strenge såsom:

- “8.5 CloudFormation setup”
- “3 AWS technology reading”
- “7:45 Drinking some coffee”

2.2 De indkommende strenge matches med regexen, så vi får opdelt strengene til:

Gruppe 1 Index: 0	Gruppe 2 Index: 2	Gruppe 3 Index: 3	Gruppe 4 Index: 6
8	.	5	CloudFormation setup
3			AWS technology reading
7	:	45	Drinking some coffee

2.3 time_formattet bliver sat til gruppe 0, da dette er timetallet og altid skal være til stede.

2.4 Hvis gruppe 2 er et punktum eller komma, vil time_formattet blive lavet om til at indeholde gruppe 1 og gruppe 3 separeret med et punktum.

2.5 Hvis gruppe 2 er et kolon, ved vi, at personen har skrevet minutter i stedet for dele af en time. Derfor finder vi ud af, om gruppe 3 er mindre end 35 og større end 4, da Efio fakturerer for påbegyndte halve time, efter der er gået 5 minutter.

2.6 Hvis gruppe 3 er over 35, skal vi ligge en time til gruppe 1.

2.7 Hvis gruppe 3 er under 4 minutter, skal vi kun have gruppe 1 med.

2.8 Et nyt objekt bliver oprettet med original input og tiden formateret til en float.

2.9 Hvis gruppe 4 findes, vil description tilføjet til det nye objekt.

2.10 Hvis gruppe 4 ikke findes og beskrivelse er påkrævet, vil der blive returneret en fejlbesked.

2.11 Metoden vil returnere objektet med den udregnede information i.

2.12 Hvis regex matching fejler, vil en fejlbesked blive returneret.

9.5 Delete Message in Slack

Efter et checkin er blevet udfyldt, skal alle knapper, der hører til den dato slettes. Sletningen sker, så der ikke kan laves ændringer på et checkin, efter det er udfyldt og for at fjerne brugt indhold fra beskedboksen, så det kun er relevant information og/eller knapper, der ligger i beskederne. Beskederne bliver slettet gennem slacks API og modulet er delt op i tre metoder, for at få fat på alle relevante informationer. Alternativt kunne man have lavet en "ren" skærm ved at sende en masse tomme beskeder, som man f.eks. gør i Java. Dette ville dog have været en ufordelagt fremgangsmåde, da det sender mange kald til Slacks API og gør det svært for konsulenten, at kigge tilbage på tidligere beskeder sendt af applikationen.

3	Delete Message in Slack
3.1 3.2 3.3 3.4 3.5	<pre>def delete_message(user_id: str, requests_client: Requests, checkin_date: str) -> None: channel_id = get_conversation_id(user_id, requests_client) timestamps = get_timestamp(channel_id, requests_client, checkin_date) hed = {'Authorization': 'Bearer ' + os.environ['SlackAuth']}</pre> <p>for timestamp in timestamps: param = {'channel': channel_id, 'ts': timestamp} request = requests_client.get('https://slack.com/api/chat.delete', params=param, headers=hed)</p>

- 3.1 Channel-id er et id på kanalen mellem brugeren og applikationen. Metoden uddybes i næste kodeeksempel.
3.2 Timestamps er en liste af tidspunkter, som svarer til en besked, der skal slettes.
3.3 Header til GET-kaldet bliver sat, hvor autorisationen bliver sat til en Bearer token, som er genereret af slack og hentes her fra en miljøvariabel SlackAuth.
3.4 Hver timestamp bliver sat ind i sin egen parameter sammen med kanal id'et.
3.5 Et GET-kald bliver sendt til Slacks API chat.delete, som ved at modtage kanal-id og tidspunkt sletter en besked fra chatten.

9.5.1 Get Conversation Id Between User and Bot

For at kunne slette en besked, kræver det et samtale-id, som identificerer, hvilken samtale der skal slettes fra. Havde checkin-applikationen ligget i en kanal, kunne man i stedet bruge navnet på kanalen, eller alternativt kanalens id, som ikke ændrer sig. Men siden applikationen snakker i private beskeder med konsulenterne, skal id'et trækkes ud for den enkelte konsulent, hvilket metoden forneden udfører.

3.1	Get Conversation Id Between User and Bot
3.1.1 3.1.2 3.1.3	<pre>def get_conversation_id(user_id: str, requests_client: Requests): hed = {'Authorization': 'Bearer ' + os.environ['SlackAuth']}</pre> <p>param = {'user': user_id, 'types': 'im'} request = requests_client.get('https://slack.com/api/users.conversations', params=param, headers=hed) response = request.json() return response['channels'][0]['id']</p>

- 3.1.1 Parametrene til GET-kaldet bliver sat med bruger-id og typen "im", som er direkte beskeder mellem to personer eller én person og en applikation.
- 3.1.2 Et kald bliver sendt til Slacks API users.conversations, som ved at modtage bruger-id og type, sender samtale id'et tilbage.
- 3.1.3 Kanal id'et bliver trukket ud af GET-kaldets respons, som er et JSON-formateret objekt.

9.5.2 Get Timestamp of Message to Delete

Det sidste der er krævet for at slette en besked, er tidspunktet, beskeden er sendt. For at sikre at det er alle beskeder, der indeholder en knap for den valgte dato, bliver alle tidspunkter gemt, hvor beskeden indeholder "Start Checkin". Således kan man slette flere beskeder på én gang, i stedet for at skulle køre det hele igennem enkeltvis.

3.2	Get Timestamp of Message to Delete
3.2.1	<pre>def get_timestamp(channel_id: str, requests_client: Requests, checkin_date: str) -> List: hed = {'Authorization': 'Bearer ' + os.environ['SlackAuth']}</pre>
3.2.2	<pre> param = {'channel': channel_id}</pre>
3.2.3	<pre> request = requests_client.get('https://slack.com/api/conversations.history', params=param, headers=hed)</pre>
3.2.4	<pre> response = request.json() timestamps = []</pre>
3.2.5	<pre> checkin_date = datetime.datetime.strptime(checkin_date, "%Y-%m-%d") date = "{0}'th of {1}".format(checkin_date.strftime("%-d"), checkin_date.strftime("%B"))</pre>
3.2.6	<pre> for messages in response['messages']: if messages['text'] == "Start Checkin": if date in messages['blocks'][0]['text']['text']: timestamps.append(messages['ts'])</pre>
	<pre> return timestamps</pre>

- 3.2.1 Kanal id'et bliver sat som en parameter til GET-kaldet.
- 3.2.2 Et GET-kald bliver sendt til Slacks API conversations.history, som sender alle beskeder fra den seneste dag tilbage.
- 3.2.3 Checkin datoens bliver formateret fra en streng til et datetime-objekt.
- 3.2.4 Datoen bliver skrevet om til formatet, som kan findes i kanalens beskeder. "%-d" giver dagen i måneden som et tal uden at være nul-polstret og "%B", angiver måneden som fulde navn.
- 3.2.5 Alle beskeder fra responsen bliver skrevet og kort igennem, hvis beskeden indeholder teksten "Start Checkin", datoens findes og datoens, er den dato, vi tjekker for, vil timestampet blive tilføjet til listen timestamps.
- 3.2.6 Alle timestamps på beskeder der skal slettes, bliver returneret.

9.6 Create Work Time Hint

I forbindelse med det automatisk genererede hint brugte vi dette modul til bestemmelse af tiden. Derudover er tidsbestemmelsen brugt til den kvittering konsulenten får tilbage efter endt tidsregistrering, samt det ugentlige og månedlige overblik udregnet på samme måde. Meningen med dette modul er at bestemme den korrekte tid, konsulenten har brugt hos en kunde. Det er her vigtigt at få udregnet tiden efter påbegyndt halve time. En påbegyndt halv time vil sige, at der er gået 5 minutter af den halve time. (7 timer og 5 minutter skal skrives som 7 timer og 30 minutter, hvor 7 timer og 4 minutter skal skrives som 7 timer). Efter vi afsluttede udviklingsperioden, er det efterfølgende gået op for os, at der er en mindre fejl i denne kodestump. Hvis vi ser på linje 4.1 og 4.2, er disse kodelinjer et levn fra en tidligere udgave af modulet, idet vi på daværende tidspunkt skulle udskrive start- og sluttidspunkt og ikke differencen mellem de to tider, som vi gør nu. Det vil sige, at på det tidspunkt var vi nødt til at sætte den rigtige tidszone, for at kunne udskrive tiden korrekt. Dette betyder ikke noget for modulet nu, men det er en ændring, vi klart ville lave, hvis vi skulle fortsætte udviklingen. Derudover har der været et par udfordringer i forhold til, hvordan tiden egentlig skal beregnes og i sidste ende kom vi frem til en ret simpel løsning. Nu bruger vi deltatimes egen metode .seconds i stedet for at bruge deltatime til at lave en ny dato og så på den nye dato få returneret minutter og sekunder. Deltatime er det format man får ud, når man trækker to datoer fra hinanden. Ved nu at bruge deltatimes egen metode får vi differencen mellem de to datoer returneret i sekunder. Dette sekundtal kan vi så gøre brug af, hvilket kan ses nedenfor i det omtalte modul med en gennemgang af modulet.

4	Create Work Time Hint
4.1	<pre>def create_time_hint(starttime: str, endtime: str, source: str) -> List: starttime = datetime.strptime(starttime, '%Y-%m-%d %H:%M:%S.%f').replace(tzinfo=timezone.utc).astimezone(pytz.timezone('Europe/Berlin')) endtime = datetime.strptime(endtime, '%Y-%m-%d %H:%M:%S.%f').replace(tzinfo=timezone.utc).astimezone(pytz.timezone('Europe/Berlin'))</pre>
4.2	<pre> time = (endtime - starttime).seconds minute = time // 60 % 60</pre>
4.3	<pre> if minute >= 35: time += 3600 - (minute * 60)</pre>
4.4	<pre> elif minute >= 5: time += 1800 - (minute * 60)</pre>
4.5	<pre> else: time -= minute * 60</pre>
4.6	<pre> return '{0} h - {1}'.format(time/3600, source)</pre>
4.7	
4.8	

4.1 starttime (starttid) bliver formateret fra en streng til en datetime. Samtidig bliver tidszonen lavet om fra UTC til "Europe/Berlin" som er central europe.

4.2 endtime (sluttid) bliver også formateret og tidszonen ændret.

4.3 Differencen mellem slut- og starttid bliver udregnet som en timedelta, hvorefter vi bruger .second-metoden til få differencen returneret i sekunder.

4.4 Antal minutter bliver udregnet vha. gulvdelingsoperatøren og modulus. Vi gulvdeler antallet af sekunder med 60, fordi vi er interesseret i at kigge på antallet af hele minutter. Restproduktet efter en normal division er ikke vigtig for vores udregning og derfor, vælger vi at bruge gulvdeling. F. eks. $90 / 60 = 1,5$ hvor $90 // 60 = 1$. Efter vi har fundet det hele minuttal kan vi tage modulus af 60 på det for at have et restprodukt tilbage, der fortæller os, hvor mange minutter der ikke går op i en time.

4.5 Hvis minutterne er større end eller lig med 35, vil der blive lagt 3600 sekunder (svarende til en hel time) minus

antallet af de resterende minutter omregnet til sekunder til vores antal af sekunder. Altså hvis det resterende minuttal er 37 vil der blive lagt $3600 - 37 * 60 = 1380$ sekunder til.

4.6 Der sker præcis det samme her som i punkt 4.5, udover at der her er tale om 1800 sekunder (svarende til en halv time).

4.7 Hvis det resterende minuttal ikke er større end eller lig med enten 35 eller 5, vil det resterende minuttal i sekunder blive trukket fra antallet af sekunderne.

4.8 Nu kan vi så returnere det korrekte timetal, hvilket vi får ved at dividere antallet af sekunder med 3600 (svarende til en time). Grunden til normal division er, at vi er interesseret i et kommatal, hvis der er et. F. eks $30.600 / 3600 = 8,5$ time.

9.6 Vurdering

De valgte udsnit af kildekoden, synes vi giver et godt overblik over vores udvikling af applikationen. Der er både kode, som vi synes er skrevet rigtig godt, men også kode som vi kunne forbedre, hvis vi havde mere tid. Overordnet er udsnittene gode eksempler på vores læring både indenfor AWS og Python. Pylint har vi brugt gennem hele udviklingen af applikationen og det har været et fantastisk redskab til at skrive læsbar og forståelig kode. Vores mappestruktur kunne opdeles dybere, men er fyldestgørende i forhold til at finde rundt i de forskellige teknologier.

10. Test

I dette afsnit vil vi gerne redegøre for hvad systematisk test er og hvordan vi har brugt det til nemmere og hurtigere udvikling af vores projekt.

10.1 Software Kvalitet

Softwarekvalitet er vigtigt for virksomheder, da det giver færre fejl i koden og dermed en bedre brugeroplevelse. At have en god softwarekvalitet indebærer, at koden er forståelig, simpel og robust. Hvis koden er en gåde, vil det være svært at forstå for nye udviklere, men også for udvikleren selv efter noget tid. Ved for eksempel at sætte kodestandarder til et kodesprog, vil softwarekvaliteten blive øget, da alle udviklere vil følge guides til, hvordan koden skal formateres og styles. Situationen hvori softwaren bliver udviklet, kan afgøre, hvor vigtig kvaliteten er. For eksempel har en applikation, som har at gøre med en pandemi, sandsynligvis ikke behov for et robust testmiljø, da den hurtigst muligt skal ud til befolkningen og i brug. En applikation der derimod har faste brugere, hvis arbejde afhænger af at applikationen virker, har mere behov for et robust testmiljø, så der ikke forekommer fejl under brug. Vores applikation startede med at have en god softwarekvalitet med et godt testmiljø, men halvvejs i praktikken valgte product owner at skære ned på kvaliteten af test, for at få flere resultater. Dette endte ud i mere fejlfinding i produktionsapplikationen, da de nye funktioner ikke var gennemtestet.

10.2 Kodestandarder

Under udviklingen af applikationen var det vigtigt at have nogle kodestandarder. Vi valgte som gruppe, at kodestandarderne skulle overholdes ved at bruge forskellige biblioteker, der testede vores kode. Disse test blev kørt på CI- og CD pipelinen og ville fejle pushet til Github, hvis kodestandarden ikke var blevet overholdt. Til Python brugte vi Googles Python Style Standard³, der blev kørt gennem pylint. CloudFormation var et nyt værktøj for os og derfor valgte vi at bruge både en linter - der testede for kodestandard mod AWS CloudFormation resource specifications⁴- og en nag der testede for usikker infrastruktur i form af roller med for mange wildcards og eksponerede kodeord. Kodestandarderne gjorde, at vi alle var enige om, hvordan tingene blev skrevet, så der ikke var forvirring mellem udviklerne og at det var nemt at læse og forstå andres kode. Ved at teste vores CI- og CD pipeline for fejl kunne vi hurtigere lave vores templatefiler i YAML, så vi kunne komme gang med udviklingen af selve applikationen.

³<https://google.github.io/styleguide/pyguide.html>

⁴<https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cfn-resource-specification>

10.3 Test Driven Development

TDD (Test Driven Development⁵) er en praksis, hvor udvikleren udvikler en test case, der specifiserer, hvad koden skal kunne og validerer den. En test case bliver lavet til hver funktionalitet i softwaren, hvorefter den bliver testet. Hvis testen fejler, vil koden blive skrevet så testen kan bestå. På denne måde er man sikret simpel kode med minimale bugs. TDD sikrer således at udvikleren tænker på udkommet af en funktion, før den bliver skrevet. Nedenfor ses en model af workflowet i TDD.



Figur 26: Test Driven Development Workflow.⁶

⁵<https://www.guru99.com/test-driven-development.html>

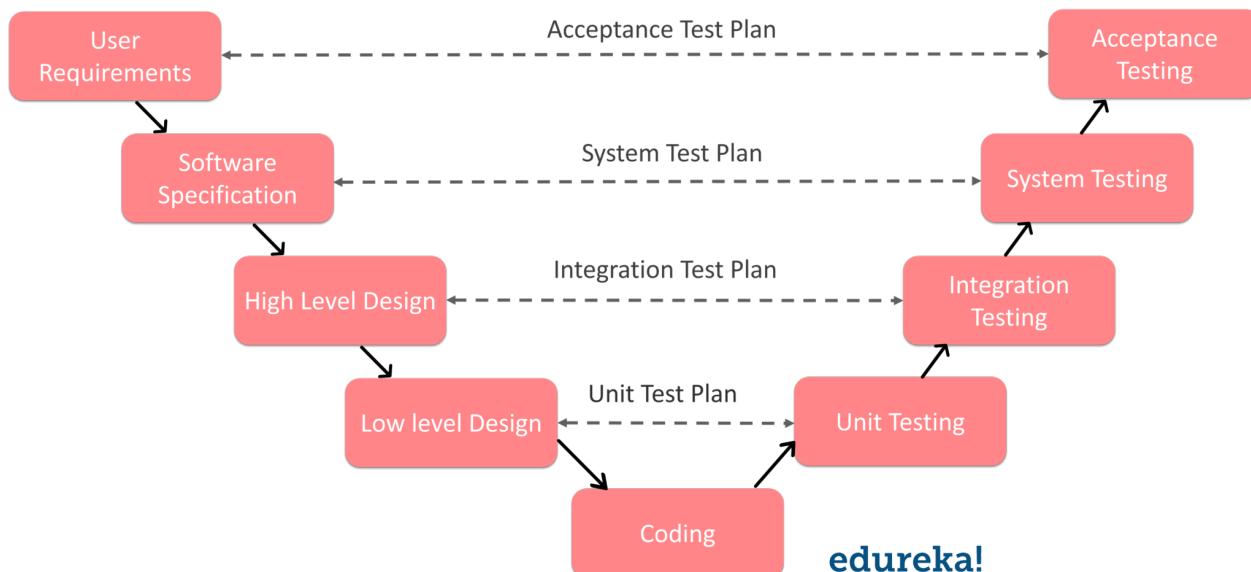
⁶<https://www.xenonstack.com/blog/test-driven-development-python/>

Acceptance TDD and Developer TDD

Der er to former for TDD: Acceptance TDD er en test, som specifiserer systemets adfærd og den bliver skrevet ud fra Acceptance Test. Denne test fokuserer på den overordnede adfærd af systemet. Developer TDD er en test, der fokusere på alle de små funktionaliteter i systemet. Testen bliver skrevet som en unit test og skal kun opfylde præcis den funktionalitet, der er tale om. Fordelen ved Acceptance- og Developer TDD er at øge effektiviteten ved kun at opfylde præcis det, som er nødvendigt for at få systemet til at fungere, og at product ownerens krav, der er stillet inden udviklingen påbegyndes, bliver overholdt. TDD sikrer desuden at koden bliver skrevet så minimalt som muligt, så det kun opfylder de krav, der er blevet stillet.

10.4 Software Testing

Software testing er en praksis, der undersøger og sikrer at et software / system er af højeste kvalitet igennem test. Der bruges både White Box og Black Box Testing⁷, som sammen sikrer verifikation af applikationen under test, også kendt som AUT. Der findes fire forskellige software testing-lag, som hver især tester og sikrer det bedste resultat af softwaren. Nedenfor ses en figur af de fire forskellige software testing-lag, der bruges til software testing.



Figur 27: Software Testing Layers Model.⁸

Black & White Box Testing

Inden for test findes der sort og hvid boks-testning. I den sorte boks kendes den indre struktur eller designet ikke af testeren, og testeren vil udføre inputs og verificere outputs ud fra det forventede resultat. I den hvide boks kender testeren til den indre struktur og designet, og testeren vil kigge på implementationen af koden og verificere outputs både fra lovlige- og ulovlige inputs. Med disse to bokse kan man teste systemet både fra brugerens- og udviklerens synspunkt. Nedenfor forklares de fire software testinglag, hvor Black- og White box Testing indgår.

⁷<https://softwaretestingfundamentals.com/black-box-testing>

⁸<https://www.edureka.co/blog/software-testing-tutorial/>

Unit Testing

Unittest er en test af de enkelte enheder af systemet. Testen udføres som White Box Testing, og bliver oftest gjort automatisk. Som det ses på figur 27, er Unit Testing⁹ et Low Level Design, som er der, hvor alle de enkelte komponenter bliver udviklet. Ved at lave unittests sikrer det, at alle enheder af systemet fungerer som de skal uden fejl. Den største fordel ved unittest er, at sikre sig programmet altid virker selv efter en ændring af koden. For at dette kan lade sig gøre, skal testene være skrevet fyldestgørende og optimalt. Udo over dette bidrager unittest til en mere effektiv og hurtigere udvikling, da man uden at køre programmet og skrive i et inputfelt, kan teste metoden, der ville blive kørt bagved.

Integration Testing

Integrationstest¹⁰ tester om forskellige moduler fungerer som forventet, når de er integreret med hinanden. Integrationstest kan udføres som Black- eller White box testing og kan køres manuelt eller automatisk. Som det ses på figur 27 er Integration Test et High Level Design, som er der arkitekturen af softwaren, bliver lavet. Integration Test tester derfor på større high-level kombinationer af enheder, der kaldes for moduler.

System Testing

En systemtest¹¹ er en test af hele softwaren, som oftest udføres af en professionel testagent. Testen er baseret på, at strukturen af softwaren er kendt af testeren, som det også fremgår af figur 27. Systemtest bliver udført som Black Box testing, da testeren ikke har nogen viden omkring den underliggende kode. Testen bliver udført manuelt og i vores projekt, blev det udført af konsulenterne, der i sidste ende også skulle bruge applikationen. Ved at bruge konsulenterne fik vi testet inputfelterne, som brugerne ville bruge dem. På den måde kunne vi få fejl frem, vi ikke selv ville kunne tænke os til, da vi som udviklerne har viden om den underliggende kode.

Acceptance Testing

En acceptance test er et krav til systemet, der bliver stillet inden udviklingen påbegyndes. Som det ses på figur 27, er Acceptance testing¹² en User Requirement, hvilket betyder at testen bliver lavet udfra et krav, der bliver stillet af brugeren. Testen bliver udført som en Black Box Test og er oftest udført manuelt. Der er dog blevet en trend de seneste år, at acceptance test bliver automatiseret i stedet. Dette gør, at kravene til systemet bliver overholdt gennem hele udviklingen. Vores projekt brugte Cucumber¹³, der bygger på adfærdsdrevet udvikling, hvilket vil sige at vores acceptance test blev skrevet som kode og blev testet i vores kontinuerlige integrations pipeline.

10.5 Vurdering

Gennem projektet har vi gjort brug af test på mange forskellige måder og det har hjulpet os til at komme i mål med et færdigt produkt, der kan bruges. Kvaliteten af vores applikation kunne være bedre, eftersom vores unit testing blev lagt lidt på hylden halvvejs i projektet, da product owner hellere ville se flere resultater. Konsulenternes feedback gennem hele forløbet hjalp os væsentligt, da vi kunne fokusere på at rette koden til ved områderne, hvor der var fejl. Vores kodestandarder hjalp os dog gennem hele projektet med at holde en god kodekvalitet, som både er letlæselig og veldokumenteret. Vi mener som gruppe, at vi har opnået at teste applikationen i et bredt nok omfang i forhold til de krav, der er blevet opstillet. Product owner valgte tidligt i forløbet at skære ned på test for at se

⁹<https://softwaretestingfundamentals.com/unit-testing/>

¹⁰<https://softwaretestingfundamentals.com/integration-testing/>

¹¹<https://softwaretestingfundamentals.com/system-testing/>

¹²<https://softwaretestingfundamentals.com/acceptance-testing/>

¹³<https://cucumber.io/>

flere resultater. Dette medførte at systemet ikke blev fuldstændig gennemtestet, da vi prioriterede vores tid mere i retning af resultater. Derfor mener vi ikke at applikationen opnåede, at blive testet i et bredt nok omfang.

11. Evaluering af produktet

I forløbet er der blevet sat mange krav til applikationen og de fleste af kravene, er blevet opfyldt efter bedste evne. Kravene har ændret sig og flere er blevet stillet gennem hele forløbet. Der er dog, som tidligere nævnt, særligt tre krav der ikke nåede at blive opfyldt. Det første krav var: *2.c.vii*, som omhandlede at vægte kunder ud fra, om en e-mail er sendt til dem. Det andet krav var: *2.c.viii*, som vedrører at vægte kunder ud fra modtagede e-mail. Og det sidste krav var: *10*, som omhandler integration med Zenegy, så fravær kunne blive registreret. Selvom disse krav ikke blev opfyldt, blev det endelige produkt sat i produktion og taget imod med gode resultater. Der har været mange fejl gennem udvikling af applikationen, og de har alle sänket udviklingen på hver deres måde. Vores CI/CD-pipeline hjalp med at minimere fejl og hvis større fejl opstod, at den derefter kunne konvertere tilbage til en version, der ikke havde fejlen. Ud over fejlene i systemet har der været mange uhensigtsmæssigheder i form af arbejdet hos Efio. Coronapandemien gjorde f.eks. at vi var nødsaget til at arbejde hjemmefra og det resulterede i at vores Scrum meetings ofte blev kortere end de ville have været på normalvis. Som gruppe har vi grundet omstændighederne heller ikke fået det autentiske indblik i virksomheden og hvordan konsulenterne arbejder med kunder, som vi havde ønsket. Oplevelsen hos Efio har på trods af omstændighederne stadig været god og vi har nydt at kunne lære fra dygtige fagpersoner. Vi ved at Efio har planer om at videreudvikle på applikationen, da vi efter forløbet fik følgende bemærkning fra Efio's direktør: *"Vi planlægger et hackathon for at arbejde videre med appen, når vi kan mødes face to face. Det der primært mangler er sikkerhed på API'et, databasen, backup, altså nogen af de mere tunge men nødvendige ting for idrifttagning"*. Heraf kan vi uddrage, at der er rig mulighed for at videreudvikle på produktet. Desuden er det muligt at videreudvikle uden at tage applikationen ud af drift, da hele systemet er bygget op af microservices, der kan udskiftes med mere sikre løsninger.

Proces

12. Projekt set-up

I dette kapitel vil vi give læseren indsigt i vores processer og udviklingsmetoder. Hertil følger, hvilke teknikker vi har gjort brug af under udviklingen samt, hvad vi har inddraget fra andre udviklingsmetoder. Kapitlet indeholder: Projekt set-up, valg af udviklingsmetode, planlægning, dokumentation af og refleksion over processen og evaluering. I det følgende afsnit kommer vi ind på projektets karakteristika. Hvor vi vil redegøre for projektets størrelse, vores kompetencer, hvilken type opgave det er, nye teknologier, og hvilke usikkerheder der ligger for projektet.

Efter vores afsluttede praktikperiode, valgte vi at afsætte fem uger til videreudvikling af applikationen. Her lå product owners fokus på at forbedre systemet ved at fjerne konstante værdier og i stedet begynde at bruge rigtige data som f.eks. kunder og personalisering af kunder. Product owner lagde også fokus på at udvikle et nyt design og integration af Zenegy, som skulle forbedre oplevelsen for den enkelte konsulent.

Zenegy er den eneste nye teknologi, som vi skulle arbejde med, der er notoriske for at være langsomme til at kommunikere med udviklere. Desuden skal man oprettes og godkendes af Zenegy, før man kan begynde at integrere deres program. Dermed skaber det stor usikkerhed i forhold til implementation af Zenegy, da vi ikke har nogen viden om, hvordan vi kan og skal bruge deres API og har generel mangel på information uden yderligere kontakt til dem før projektets start. Vi har dog tilegnet os kompetencer fra studiet i forhold til at bruge API-endpoints, og har yderlige udviklet disse under praktikperioden. Vi har valgt at fortsætte vores arbejde med AWS og deres tjenester, som vi har udviklet kompetencer indenfor under vores arbejde i praktikperioden i forbindelse med udviklingen af applikationen i hovedopgaven.

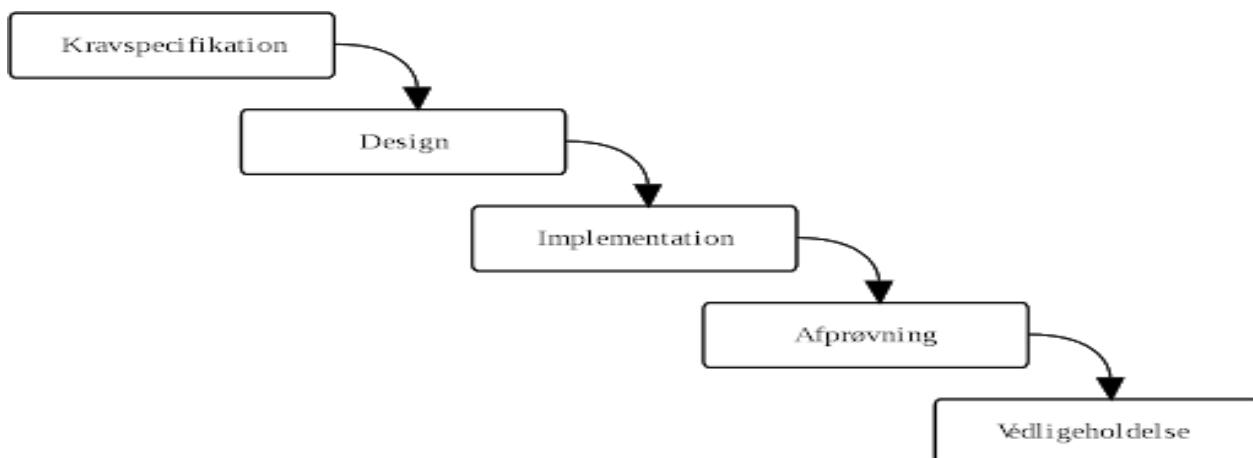
Da vores udviklingsperiode kun strakte sig over fem uger, og vi havde mange andre opgaver at arbejde med, har det svært at finde tid til at benytte os af spikes. Dette skaber endnu en usikkerhed med Zenegy, på grund af den begrænsede tid til at undersøge deres API, og at man skal igennem flere processer, for at kunne få adgang til dybere viden omkring deres API. Vi antager, at Zenegy er det eneste krav, der ville have behov for en spike, da det er en ukendt faktor i vores videreudvikling.

13. Valg af udviklingsmetode

Herunder introduceres fire af de udviklingsmetoder, vi kender til. Der vil blive redegjort for selve metoden, fordele og ulemper ved dem, og hvornår de er bedst egnet til at blive brugt. Vi laver en vurdering heraf, hvor vi til sidst vil sammenligne denne med Boehms og Turners Radar chart for at verificere, om vores vurdering er korrekt herudfra.

13.1 Udviklingsmetoder

Vandfaldsmodellen



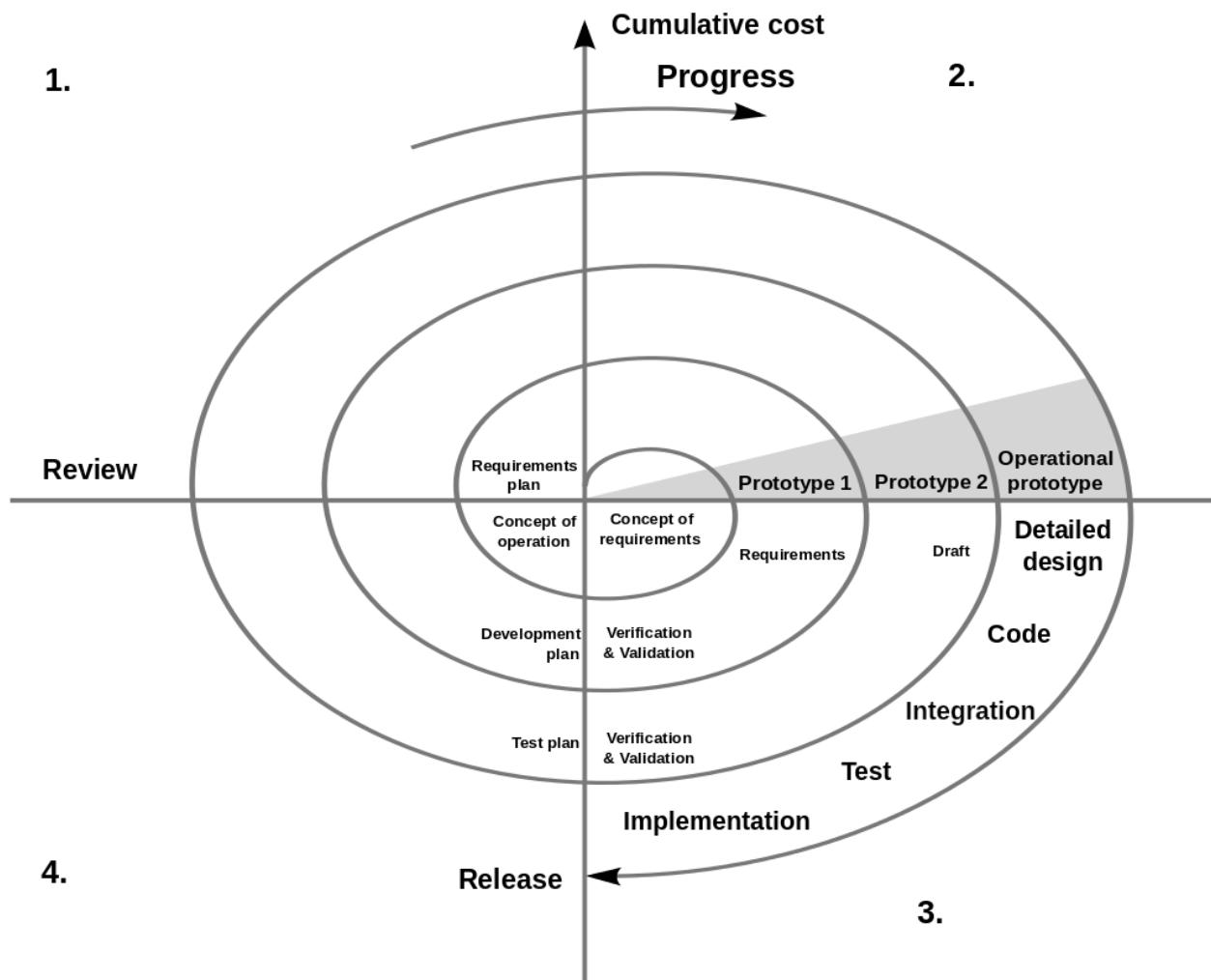
Figur 28: Illustration af vandfaldsmodellens proces.¹⁴

Vandfaldsmodellen er en udviklingsmetode med fem forskellige faser, som man går igennem strengt sekventielt. Det vil sige, når man starter et projekt, begynder man med kravspecifikationen, og man går ikke videre herfra, før det er helt færdigt. Når man er færdig med det, kan man så gå videre til design. Sådan fortsættes processen hele vejen ned, og der er ikke mulighed for at kunne gå tilbage til f.eks. kravspecifikation, når man er i gang med implementation eller design. For at kunne gøre brug af vandfaldsmodellen, kræver det, at man bruger meget tid på at gennemtænke hver fase. Foruden risikerer man at sidde fast i en senere proces, eller gå i stå, da der kan være manglende eller forkerte elementer fra en tidligere fase.

Vandfaldsmodellen egner sig bedst til større udviklingshold, hvor der er mange forskellige ekspertiser. For at kunne være sikker på, at man er færdig med en del af modellen, kræver det mange input og mange forskellige synspunkter.

¹⁴<https://da.wikipedia.org/wiki/Vandfaldsmodellen#/media/Fil:Vandfaldsmodellen.svg>

Spiralmodellen

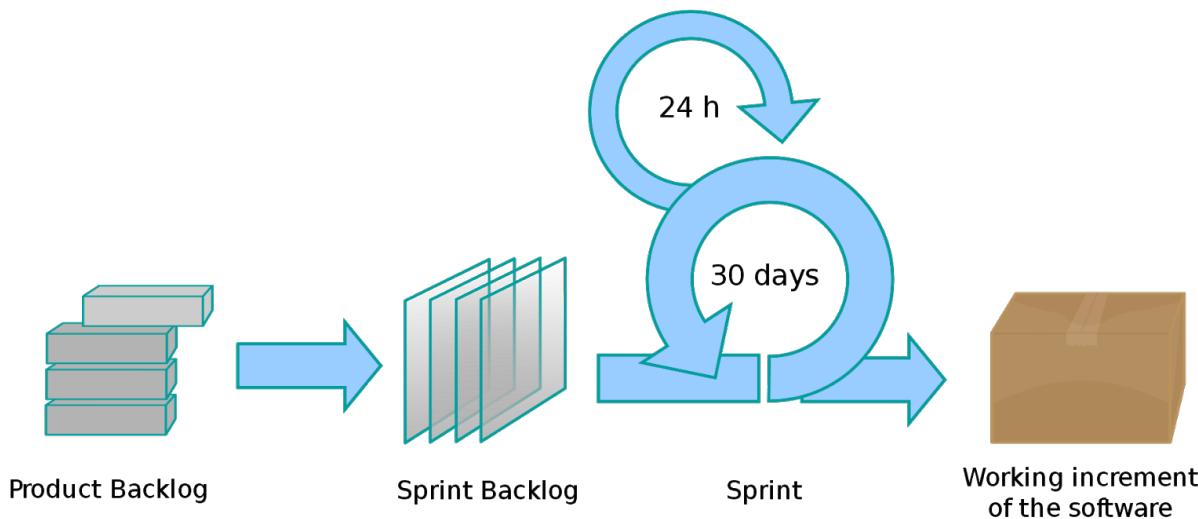


Figur 29: Illustration af processen for spiralmodellen.¹⁵

Spiralmodellen er en risikodrevet udviklingsmodel med fire forskellige faser. Første fase er planlægning, hvor der skal estimeres budget, tidshorisont og deadlines samt ressourcer for iterationen. Anden fase er risikoanalyse, som handler om at identificere potentielle risici for iterationen og eventuel håndtering af dem. Her bygger man også en prototype, der gøres brug af i tredje fase, hvor kompleksiteten af prototypen afhænger af, hvor langt i processen man er. Tredje fase er ingeniørarbejde, som handler om tests, kodning og implementering af produktet. Fjerde og sidste fase er evaluering. Her monitorerer og identificerer man risici inden for deadlines og budgetoverskridelse. Spiralmodellen er en god udviklingsmetode ved et stort projekt, hvor der er påkrævet analyser af både budget, deadlines mm. som kan være aktuelt for interesserter. Udvikling og implementering sker hurtigt og der er altid plads til feedback fra brugere eller interesserter, ved at man kører i iterationer. Dog fungerer spiralmodellen kun godt for store projekter med et stort udviklingshold, hvor der er folk med ekspertise i analyse, risikovurdering og estimering.

¹⁵[https://en.wikipedia.org/wiki/Spiral_model#/media/File:Spiral_model_\(Boehm,_1988\).svg](https://en.wikipedia.org/wiki/Spiral_model#/media/File:Spiral_model_(Boehm,_1988).svg) redigeret billede

Scrum



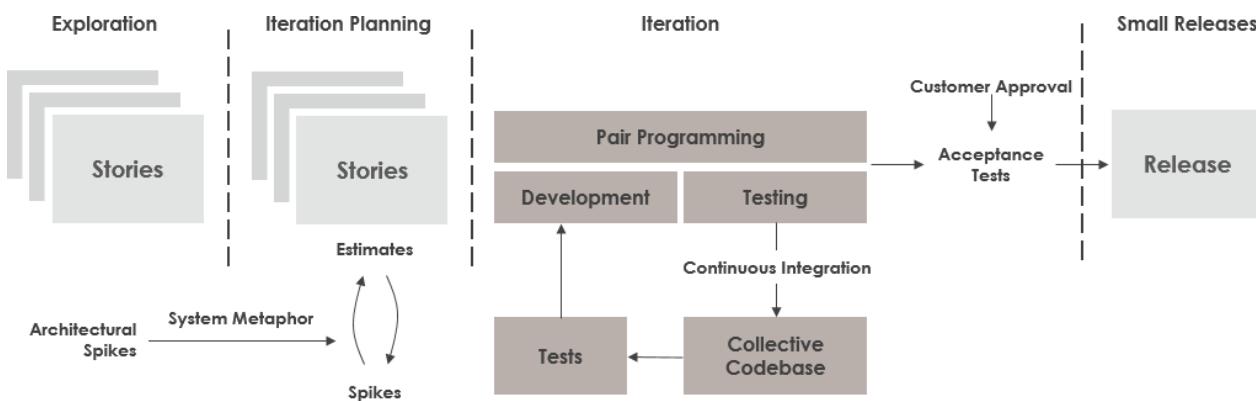
Figur 30: Illustration af scrum udviklingsprocess.¹⁶

Scrum er en agil udviklingsmetode. Her strukturer man arbejdsprocessen med, hvilke opgaver, der skal laves i en iteration. En iteration må maksimalt være en måned og er normalvis to uger langt. Sprintet bliver udarbejdet under sprint planning, hvor udviklingsholdet holder møde omkring, hvad der skal laves og samtidig bliver enige om, at de opgaver der skal udføres, er klar og kan udføres. Sprint planning bliver udført i starten af hvert nyt sprint. I sprintet udføres der dagligt daily stand-up, hvor hele holdet snakker om, hvad de foretager sig, og hvorvidt der er noget, der står i vejen for dem. Ved afsluttende sprint foretages der en sprint review og retrospektivt møde, hvor sprintet bedømmes ud fra, hvad der blev gennemført og er starten på en diskussion om, hvad der skal arbejdes på i næste sprint sammen med interesserter. Der er tre roller i scrum: product owner, scrum master og udviklere. Holdet skal kunne være selvforvaltende og kunne fokusere på tværs af ekspertiser mod sprintets mål.

Scrum fungerer bedst i små hold og bruges til udvikling, udgivelse og vedligeholdelse af komplekse systemer, hvor der er fokus på kontinuerlige og hyppige udgivelser. Der er meget kommunikation mellem alle roller, som prøver at sikre forståelse for opgaverne og hjælpe med løsninger. Scrum er også optimalt at bruge, når alle kravene ikke er kendt og klargjort, og hvis de har en stor risiko for at ændre sig i løbet af udviklingen. Scrum er dog ikke optimalt, hvis man er et stort hold. Det er muligt at arbejde med udviklingsmetoden, men scrum er mere intimt baseret med fokus på face-to-face kommunikation mellem udviklerne og product owner. Hvis virksomheden har meget fokus på deadlines og budget, egner scrum sig heller ikke. Det er muligt at lave en plan for, hvornår man estimerer at være færdig, men det er svært at forudsige, når man bruger scrum.

¹⁶[https://en.wikipedia.org/wiki/Scrum_\(software_development\)#/media/File:Scrum_process.svg](https://en.wikipedia.org/wiki/Scrum_(software_development)#/media/File:Scrum_process.svg)

Extreme programming



Figur 31: Illustration af extreme programming udviklingsprocess.¹⁷

Extreme programming er, ligesom scrum, en agil udviklingsmetode. Figur 31 giver et overblik over metodens udvikling. Extreme programming er bygget på at de positive elementer, der hører til traditionelle udviklingsmetoder, bliver taget til det ekstreme. Som eksempel er code review anerkendt som et positivt element i traditionel udvikling, hvis det tages til det ekstreme, kan kode kontinuerligt blive anmeldt i form af pair programming. Idet extreme programming er en agil udviklingsmetode, er der dermed hyppige udgivelser af projektet. Extreme programming har fem værdier: kommunikation, simplicitet, feedback, mod og respekt. Disse værdier er fundamentet til, hvordan et hold skal tænke og agere både over for hinanden og i deres udvikling.

Extreme programming har fire aktiviteter som en del af udviklingsmetoden: kodning, test, at lytte efter og design. Kodning er den vigtigste aktivitet da foruden kode, er der intet produkt. Desuden er testing et af de centrale komponenter i extreme programming. Hvis man tester lidt, kan man eliminere nogle fejl og ligeledes, hvis man tester meget, kan man eliminere mange fejl. Derfor gør XP brug af test driven development, hvor man laver test, inden man laver resten. At lytte efter handler om at programmørerne skal lytte til, hvilken forretningslogik de skal bruge og kunne forstå logikken til at kunne give feedback til interessen på, hvordan problemet kan løses. Design handler om at sørge for at have en solid designstruktur over logikken i systemet, og på den måde undgå at have for mange afhængigheder i systemet, således at det ikke forårsager unødvendige ændringer i resten af systemet, når man laver ændringer i en del af systemet. Extreme programming fungerer bedst i små- eller mellemstore hold og har mange metoder og teknikker, som bruges til at op holde en god standard og morale både som hold og i selve koden. Da det er en agil udviklingsmetode, kræver det også, at der kan komme nye udgivelser hurtigt i en kontrolleret orden, hvor alt kan godkendes af andre og af de tests, der er lavet.

13.2 Vurdering

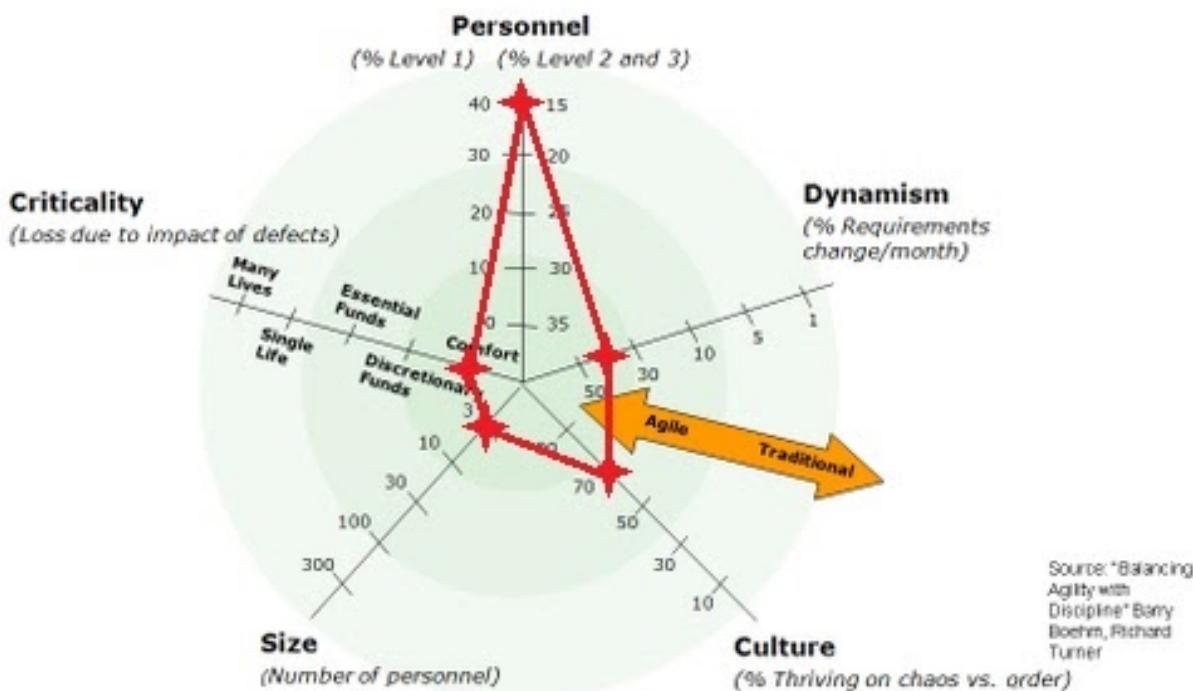
Efio har ikke stort fokus på budget for dette projekt, da der ikke er mange omkostninger i udviklingen af det. Hovedsagelige omkostninger ville komme fra AWS, som allerede er minimale. Product owner estimerer at et fuldt implementeret system vil koste mellem 1-2000 kr om måneden. Derudover er personaleomkostningerne minimale, da vi er praktikanter og der dermed kun er omkostninger, når andre end udviklerholdet hjælper til i projektet. Vi er et lille udviklerhold på tre mennesker med de samme kompetencer inden for web- og applikationsudvikling. Dette gør, at der ikke er en god mulighed for at kunne lave risikoanalyser, budgettering, målsætninger mm., som ellers er essentielle i spiralmodellen. Da vandfaltsmodellen egner sig bedst til store projektgrupper, af mennesker med forskellige kompetencer, falder brugbarheden af modellen også fra for dette hold. Ud fra dette synspunkt - at der ikke er stor fokus på budget og deadlines fra virksomheden, og at vi er en lille gruppe - egner vi os bedst til at gøre brug af enten scrum, extreme programming eller en kombination af de to frem for enten vandfaltsmodellen eller spiralmodellen.

¹⁷<https://www.visual-paradigm.com/scrum/extreme-programming-vs-scrum/>

Vi har valgt at bruge scrum som vores udviklingsmetode sammenkoblet med teknikker fra extreme programming. Dette bunder i, at vi har en positiv erfaring med scrum, og som hold arbejder vi bedst med små hyppige løsninger og de værdier, scrum har med selvforvaltning. Grunden til at vi også sammenkobler scrum med extreme programming, kommer af, at det var et krav fra virksomhedens side, at vi udviklede og benyttede en CI- og CD pipeline, som er en praksis i extreme programming. Derudover har vi gode erfaringer med nogle af extreme programmings teknikker, som vi gerne vil forbedre os på, hvoraf vi gør brug af pair programming, test driven development og small releases.

13.3 Radar chart

Boehms og Turners radar chart¹⁸er en model, der kan bruges til at give et overblik over, hvilken udviklingsmetode, der ville være optimal at bruge til et projekt. Modellen bliver normalt lavet af den eller de personer, der er ansvarlige for et projekt. Vi har brugt denne model til at verificere at brugen af en agil udviklingsmetode, var passende til projektet. På figur 32 ses radar charten lavet for denne gruppe.



Figur 32: Boehms og Turners radar chart for Sterlings udvikler hold.¹⁹

Personnel: 40%/15%

Personnel handler om den erfaring og færdighed, holdet har, som bunder i erhvervserfaring og kompetencer inden for feltet. Boehm og Turner mäter dette ved brug af Alistair Cockburns developer scale²⁰: level 1 (beginner), level 2 (intermediate) og level 3 (expert). Idet vi startede med at bruge mange nye teknologier i praktikperioden, og at vi aldrig har udviklet en applikation før, betød at vi alle startede i level 1. Vi har efter praktikperioden bevæget os mere hen i starten af level 2. I dette stadie af læring vurderer man ikke succes ud fra, hvor godt man kan følge med i

¹⁸https://leadinganswers.typepad.com/leading_answers/files/agile_suitability_filters.pdf

¹⁹https://leadinganswers.typepad.com/leading_answers/files/agile_suitability_filters.pdf

forståelse og udførelse som i level 1 men mere ud fra, hvordan man kan optimere både kode, processer og metoder, som man gør brug af.

Dynamism: 40%

Product owner ville gerne løbende, kunne se resultater. Ud fra disse resultater kunne der ske ændringer både på designet og systemet. Nogle af de features der skulle udvikles, var ikke undersøgt, hvilket kunne give udfordringer i forhold til udviklingen. Disse udfordringer kan undersøges og nogle udfordringer, kræver både helt fra milde til drastiske ændringer i systemet, mens andre kan vise sig ikke at være mulige. Vi oplevede ikke nogen udfordring, som ikke var mulig, men mødte udfordringer, der kræver væsentligt mere tid. Eksempelvis da vi arbejdede med at brugerens emoji opdateres - hvis konsulenten registrerer sygdom eller ferie - krævede det ændringer på Slackgruppens rettigheder således, at vi kunne få en administratortoken, som kan bruges til at udføre opdateringen af status-emoji. Løbende udfordringer og ændringer forekommer i udviklingen, da kravene for produktet blev sat efter hvert sprints udførelse, derfor lægger vi her langt mere mod origo end periferien.

Culture: 70%

Idet vi i høj grad arbejder med løbende udfordringer og prøver at finde løsninger hen ad vejen, drives gruppen meget på kaos af de mange ændringer både internt og eksternt. Internt kaos opstår på grund af de ændringer, der kommer fra product owner, som får nye idéer eller løsningsforslag, der kan forårsage større ændringer. Dette skaber forlænget arbejdstid på sprintet, som virksomheden har forståelse for. Eksternt kan der komme opdateringer fra de biblioteker, sprog og systemer, vi bruger og afhænger af, som gør, at vores program ikke længere virker. Eksempelvis kom AWS med en opdatering til brug af Serverless, som resulterede i, at vores system ikke kunne køre. Her var vi nødt til at bruge ekstra ressourcer og tid til at finde ud af en løsning for at få systemet til at virke igen. Der bliver ikke set negativt på, at noget kan tage længere tid på grund af ændringer, som gør at virksomheden lige så vel som vores gruppe kan arbejde med det kaos, der følger. Dette er dog ikke ensbetydende med, at vi er afhængige af kaos, da vi sagtens kan arbejde med orden, når der er mulighed for det.

Size: 3

Vi har arbejdet som et udviklerhold bestående af tre medlemmer, men vi har også haft et hold, der har hjulpet os mod målet. Vi har haft én person som product owner og business ekspert, der sørgede for nye user stories og kunne give relevant indsigt i forretningen samt, hvordan den fungerer. Vi har derudover haft én person som konsulent inden for UX, der har kunne bidrage med sin holdning til, hvad der er godt og dårligt design. Ham brugte vi under praktikperioden og hans idéer og tanker var in mente under udviklingen af det nye design. Desuden har vi haft seks konsulenter fra virksomheden, der har bidraget med at teste i produktion, hvor de gav feedback på det de oplevede og fejlrapporterede på de fejl, de fandt hen ad vejen.

Criticality: Comfort

Criticality består af, hvad resultatet af systemfejl forårsager af tab. Hvis der opstår systemfejl, risikerer vi ikke andet end tab af komfort, altså at konsulenterne mister tid på at systemet ikke virker. Konsulenten kan altid notere sig, hvad han har arbejdet på og registrere det i systemet senere, når det virker.

13.4 Verificering

Ud fra Boehms og Turners radar chart ville det tyde på at vi har taget den rigtige beslutning med at udvikle agilt.

²⁰Crystal Clear, a human-powered methodology for small teams, Alistair Cockburn, 2004, <https://bit.ly/3rXQ4td>

14. Planlægning

I dette afsnit vil vi dokumentere vores planlægning af udviklingsperioden i hovedopgaveforløbet. Vi vil bl.a. komme ind på sprint planning, estimering af tiden der skal bruges til en given user story og få et indblik i, hvordan vi prioriterer forskellige user stories.

I starten af hvert sprint mødes vi med product owner til sprint planning meeting, hvor product owner stiller de krav, han har til sprintet i form af user stories. Herfra laver product owner prioriteringer i form af MoSCoW, som står for Must have, Should have, Could have og Won't have. Dette betegner user stories i forhold til, hvor vigtige de er for applikationen, must have burde altid laves først, derefter should have og could have. Won't have betegner en story, som ikke må laves. På denne måde gav product owner os indsigt i, hvad vi skulle fokusere på, således at vi fik lavet det vigtigste før noget med en lavere prioritering. Herfra kunne product owner enten fortsætte i sprint planningen og lave acceptkriterier til user storiesene eller forlade mødet, hvor vi så selv skulle lave acceptkriterier og derefter bruges definition of ready til at klargøre user storyen til sprintet.

14.1 Sprint overblik

Sprint 1 - Varighed 2 uger (9 november - 21 november 2020)

Incomplete issues					
Key	Summary	Issue type	Status	Assignee	Story points
DEMO-169	Personalized customers for checkin (L)	Story	DONE	MF	8
DEMO-170	Option to change weights on customers (L)	Story	DONE	MF	2
DEMO-172	Register Absence (H)	Story	DONE		8
DEMO-176	Required and optional description field depending on contract (L)	Story	DONE	AS	3

Completed issues					
Key	Summary	Issue type	Status	Assignee	Story points
DEMO-168	Import customers and contracts from Close CRM (L)	Story	DONE	AS	5
DEMO-171	Register Efio work (H)	Story	DONE		8
DEMO-173	Time register beforehand in case of sickness or vacation (H)	Story	DONE		11
DEMO-174	Set personalized check in time (L)	Story	DONE		5

Figur 33: Resultat af 1. sprint.

På figur 33 kan resultatet af første sprint ses. Her fremvises færdiggjorte user stories nederst og ikke-afsluttede user stories øverst. Under sprint planning havde vi estimeret nogle user stories helt forkert og vi havde påtaget os flere

opgaver, end vi kunne klare, hvilket betød, at der var fire user stories, der ikke blev færdiglavet. Ude til højre i figur 33 kan der ses, hvor mange user story-points hver user story har til en total af 50, som allerede er meget i forhold til, at vi i praktikperioden arbejdede med 16 user story points pr. uge, altså et totalt af 18 user story points mere for et sprint, end vi har været vant til. Efter færdigt sprint blev de fire user stories lagt tilbage i product backlog.

Sprint 2 - Varighed 1 uge (23 november - 28 november 2020)

Incomplete issues					
Key	Summary	Issue type	Status	Assignee	Story points
DEMO-169	Personalized customers for checkin (L)	Story	DONE	MF	8
DEMO-170	Option to change weights on customers (L)	Story	DONE	MF	2
DEMO-172	Register Absence (H)	Story	DONE		8
DEMO-176	Required and optional description field depending on contract (L)	Story	DONE	AS	3

Figur 34: Resultat af 2. sprint.

På figur 34 ses resultater af andet sprint. Her er der blevet arbejdet videre på de user stories, som ikke blev færdiggjort fra det tidligere sprint. Som fremvist på figuren blev dette sprint heller ikke færdiggjort. De er næsten lavet færdigt, men opfylder enten ikke definition of done eller mangler at opfylde sidste acceptkriterier. User storiesne er ikke blevet genestimeret, da der skal kunne reflekteres over, hvor store user storiesne virkelig er, hvor vi efterfølgende kunne diskutere indbyrdes om, hvad der gik galt i estimeringen efter sprintets afslutning. Efter vores estimering ligger dette sprints totale user story points på 18, som er 2 mere end, hvad vi estimerer, at vi kan håndtere på en uge.

Sprint 3 - Varighed 2 uger (30 november - 12 december 2020)

Incomplete issues					
Key	Summary	Issue type	Status	Assignee	Story points
DEMO-216	Integrate Zenegy for absence and vacation	Story	TO DO		-
Completed issues					
Key	Summary	Issue type	Status	Assignee	Story points
DEMO-169	Personalized customers for checkin (L)	Story	DONE	MF	8
DEMO-170	Option to change weights on customers (L)	Story	DONE	MF	2
DEMO-172	Register Absence (H)	Story	DONE		8
DEMO-176	Required and optional description field depending on contract (L)	Story	DONE	AS	3
DEMO-214	New Design for customer registration	Story	DONE		-
DEMO-217	Update emoji based on checkin	Story	DONE	AS	2
DEMO-222	Time formatter	Bug	DONE	MF	-

Figur 35: Resultat af 3. sprint.

User stories fra sprint to er blevet flyttet over til sprint tre som set på figur 35. Her bliver de afsluttet kort efter sprintets begyndelse. Der er to stories med i sprintet, som ikke er estimeret; DEMO-214 krævede at product owner godkendte vores design, og vi kan derfor ikke estimere, hvor lang tid det tager for os at lave noget, der skal godkendes af ham. DEMO-216 afhæng af Zenegy, som vi var i kontakt med. Det nye design (214) blev først godkendt halvvejs igennem sprintet og kunne godt have blevet estimeret der, men blev glemt i udviklingsprocessen. Zenegy (216) blev aldrig til noget, på grund af at Zenegy har nogle krav til at kunne bruge deres API, som vi ikke nåede at kunne opfylde.

14.2 Vurdering

Den fejl der endte med at koste os rigtig meget tid, var forkert estimering, men samtidig tog vi også for mange opgaver på os allerede i første sprint, der gjorde at vi blev overvældet. I stedet skulle vi have delt det mere ligeligt op, så vi havde den mængde user story points, som vi har erfaret, at vi kan nå på en uge. Alle vores user stories er blevet prioriteret med MoSCoW, som tidligere beskrevet står for Must have, Should have, Could have og Won't have. Størstedelen af de user stories vi har fået fra product owner er blevet lagt som must have, hvor enkelte user stories som f.eks. DEMO-170 har været should have, som betyder at det ikke var nødvendigt for applikationen, men er godt at have.

15. Dokumentation af processen

I dette afsnit vil vi redegøre for de teknikker, vi har gjort brug af inden for scrum og extreme programming. Derudover kommer vi ind på, hvordan vi har struktureret arbejdet igennem en branch strategi og vil afslutningsvist inddrage brugerfeedback.

15.1 Scrum

Scrumroller og deres funktion

Product owner har skulle klargøre, hvad der skal arbejdes på i næste sprint og sørge for, at der er en klar forståelse af, hvad der er ønsket. Product owner repræsenterer virksomhedens konsulenter, der skal bruge applikationen og har indsigt i, hvad konsulenterne interesserer sig for inden for applikationen.

Vi fik tildelt en scrum master fra virksomheden, som afholdte en evaluering af sprintet, når halvdelen af sprintet var overstået. Her checker scrum master, om vi sidder fast i udviklingen, om vi har brug for noget hjælp og får en evaluering af, om sprintets opgaver kan nås inden deadline. Scrum master er der også til at sørge for at de interesser product owner har, bliver lavet og kan tydeliggøre det for udviklerne, hvis der skulle opstå tvivl under udviklingen, som kan forekomme af nye informationer eller indsigt i teknologier.

Vi udfylder rollen som udviklere, da vi tager os af udviklingen af projektet. Her er det vigtigt, at vi selv kan organisere arbejdet og udviklingen under sprintet, og har kontakt til både scrum master og product owner under forløbet, hvis der er behov for det. Hver dag laves der dagligt stand-up, hvor man fortæller hinanden indbyrdes på udviklingsholdet, hvad man lavede siden sidste stand-up, hvad man går i gang med og eventuelle forhindringer. På den måde får alle et overblik fra starten af dagen over, hvad der mangler og hvad der bliver udviklet.

Retrospektiv, sprint review og sprint planning

Efter afsluttet sprint holdte vi et retrospektivt møde før begyndelsen af næste sprint. Her gennemgik vi, hvad der blev lavet og fortalte om eventuelle udfordringer og forslag til fremtidig udvikling. Dette udføres sammen med scrum master for at tilpasse udviklingsmetoden bedst til udviklerne. Derefter er der et sprint review-møde sammen med product owner. Dette er til for, at product owner kan se, hvordan det går med udviklerne, og at udviklerne kan få indsigt i, hvordan det står til med produktet. For at product owner kan se, hvordan det står til med produktet, laves der en demo med fokus på, hvad der er blevet arbejdet på i foregående sprints. Derefter startes sprint planning meeting med product owner, hvilket i vores tilfælde kunne forløbe på tre forskellige måder. På den første måde kunne product owner præsentere kravene uden at lave dem til user stories. På den anden måde præsenterede product owner kravene i form af user stories og på den tredje måde, præsenterede product owner kravene i form af user stories ledsaget af acceptkriterier. Afhængigt af forløbet byggede vi derefter user storiesene op som forklaret i definition of ready.

15.2 Definition of ready

Under sprint planning meeting kigger vi på alle vores user stories individuelt. Før en user story kan komme med i et sprint, skal user storyen være helt klar først. I dette projekt har det betydet en klar formidling af, hvad user storyen skal udføre, tydelige acceptkriterier, samt at kunne opfylde INVEST-kriterierne.

Vi har under udviklingsperioden i hovedopgaveforløbet prøvet selv at lave user stories, og udfylde acceptkriterier afhængigt af, hvad der blev leveret af product owner under sprint planning. Hvis vi har skulle lave user stories eller acceptkriterier har vi siddet sammen som gruppe og fundet frem til, hvordan user storyen skal formidles. Hvis product owner selv har formidlet user stories, bliver det gjort med et **AS A, I WANT, SO THAT** format. Derfor følger vi selv dette og argumenterer for, hvem der vil have denne feature, og hvorfor de ville have den. Derefter har vi diskuteret acceptkriterier og hvad der skulle opnås, før en user story kunne accepteres som færdiggjort. Dette hjalp os med at få indsigt i, hvilke problemstillinger der kunne være forude. Hertil har det hjulpet os med at opstille et billede af, hvordan problemet skulle løses.

Til sidst skulle user stories kunne opfylde INVEST, som består af:

- Independent (Være uafhængig af andre user stories).
- Negotiable (Kunne ændres og diskuteres).
- Valuable (Være værdifuld enten til systemet, brugerne eller aktionærer).
- Estimable (Estimerbar).
- Small (Ikke være for stor til at kunne passe i et enkelt sprint).
- Testable (Kunne testes, hvilket bunder i, at der er nok information til, at der kan laves test driven development eller behaviour driven development).

For at kunne opfylde hvert punkt i INVEST, er hvert punkt blevet diskuteret og skulle være enstemmigt godkendt, før punktet er opfyldt. Som eksempelvis med independent der kigger vi på, hvorvidt udviklingen af user storyen er mulig alene, eller om den afhænger af andre user stories. Afhænger den ikke af andre antages dette punkt som opfyldt.

For at *estimable* kan blive opfyldt, gør vi brug af planning poker i sammenkobling med risk evaluation. Planning poker er en estimeringsmetode, hvor man ved brug af spillekort estimerer, hvor stor en user story er. På figur 36 ses et billede af, hvordan kortene ser ud, som vi har brugt. Her går det ud på, at man vælger et kort, som man mener repræsenterer størrelsen på storien og så ligger alle sit kort frem, så alle kan se. Hvis resultatet ikke er enstemmigt, argumenterer man for sit valg af kort. Herefter kan man så spille igen, til der er en enstemmig beslutning.



Figur 36: Planning poker kort.²¹

²¹https://images-na.ssl-images-amazon.com/images/I/61uQd%2BXyo%2BL._AC_SL1222_.jpg

I sammenkobling med planning poker har vi gjort brug af risk evaluation, som bygger på, hvor stor risiko der er for, at vores estimering er forkert. Dette bunder hovedsageligt i, hvis user storyen indeholder nye teknologier eller andre udfordringer, der kunne gøre, at estimatet enten er for stort eller for lille. Man vælger hertil et rødt kort til at repræsentere high risk, eller et grønt kort til at repræsentere low risk og lægger det sammen med sit planning poker kort. Illustration af kortene ses i figur 37.



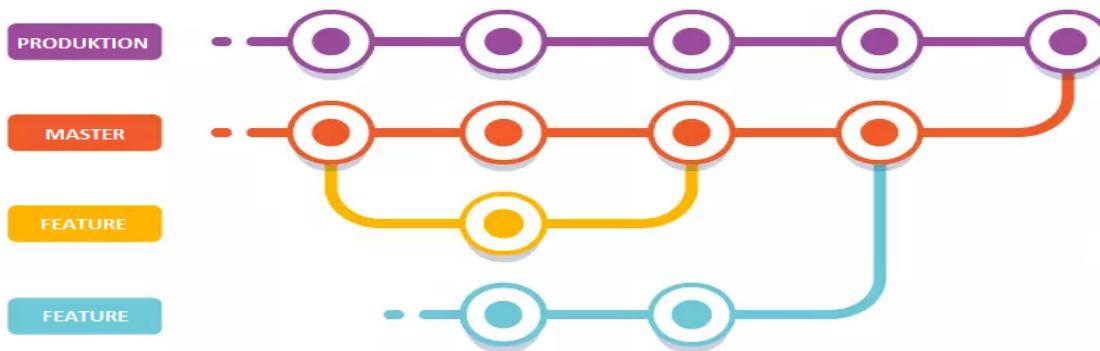
Figur 37: Team happiness kort, som blev brugt til risk evaluation.²²

Når alle punkterne af INVEST og øvrige kriterier er opfyldt, er user storyen klar og kan sættes ind i sprintet.

15.3 Branch strategi

Det har været et krav fra product owner, at vi skulle udvikle med en feature branch strategi. Som illustreret i figur 38 betyder det, at hver ny user story man arbejder på, udvikles i sin egen branch. Hvis man er færdig med udviklingen af en user story, skal det testes af systemet og verificeres af en anden udvikler end én selv, for at sikre sig at koden gør, hvad den skal. Derudover er det god skik at udvikleren også kører Pylint lokalt, før man kan pushe sin kode til master branchen. Teknisk set behøver udvikleren ikke at køre Pylint lokalt, idet vores CI/CD-pipeline gør det for os, men på denne måde øger vi chancen for at et push ikke fejler. Product owner har låst master branchen, således at der ikke kan pushes kode direkte op til branchen, men at det i stedet skal gå igennem pull requests. Efter et pull request er oprettet, anmelder man en anden udvikler om, at gennemgå de ændringer, man har lavet. Her er det den gennemgående udviklers rolle at sørge for, at sikkerhed mm. bliver overholdt og prøver at forhindre menneskelige fejl, f.eks. sørge for, at der ikke kommer unødvendige eller usikre filer og kode med ind i systemet. Det er derfor vigtigt at den gennemgående udvikler, kigger de ændrede filer nøje igennem og hvis der er ændringer, der er påkrævet før udvikleren kan godkende det, anmeldes der om disse ændringer og ellers kan udvikleren godkende pull requested. Herefter skal CI-pipelinen kører igennem uden fejl. Hvis koden er godkendt af både en udvikler og CI-pipelinen, kan branchen flettes sammen til master branchen.

²²https://images-na.ssl-images-amazon.com/images/I/61tD9vk5mXL._AC_SL1265_.jpg



Figur 38: Feature branch strategi, med tilføjet “produktions branch” for illustration af struktur.

Fordelen ved denne strategi er, at man kan arbejde adskilt på forskellige udgivelser samtidig, uden det giver store problemer og at man sikrer, at det man implementerer virker med master branchen, da man er nødt til at trække fra master branchen og sikre at det virker, før man kan pushe op. På den måde minimerer man risikoen for fejl på produktionen. Når en user story er blevet pushet til master branchen og udvikleren har sikret sig, at det som er udviklet også virker på master branchen, kan der derefter laves et nyt release til produktion.

Produktion branchen som vist på figur 38 er ikke en rigtig branch, men er der for at illustrere strukturen af branches og processen af udgivelser. Når man er klar til at udgive en ny version af applikationen, laves der et nyt release på github: Her følges semantisk versionering (en måde at vise versionsnummer på) og man inkrementerer enten major, minor eller patch afhængig af, hvad den nye release opfylder. Major bliver hævet med én, hvis der bliver lavet uforenelige ændringer til det kendte system, som ikke er bagudkompatibel med den eksisterende applikation. Minor bliver hævet, hvis der bliver lavet bagudkompatible ændringer. Patch bliver hævet ved bagudkompatible bugrettelser. Sammen med versionsnummeret laver man også en beskrivelse af udgivelsen og hvad den indeholder, som illustreret på figur 39. Efter man har publiceret udgivelsen sender github et request til AWS, som informerer om, at der er en ny udgivelse klar. Dette sætter CD-pipelinen i gang i produktionsmiljøet, hvor den henter koden fra github og laver opdateringer til databasen og/eller pipelinen, hvis der er nogen og derefter deployer CD-pipelinen kildekoden til Lambda funktionerne.



Figur 39: Sidste udgivelse af Sterling som ses på GitHub.

15.4 Definition of done

Efter user storyen er lagt ud på produktionsmiljøet, er det tid til at verificere om Definition of Done er opfyldt, så man kan lukke user storyen i jira. Definition of Done siger at hver user story har/er

1. Automatisk verificeret acceptkriterier.
2. Bliver bedømt og godkendt af en anden.
3. Bliver flettet sammen til master branchen.
4. Kommer op på produktionsmiljøet.

Det første punkt skal udføres ved brug af Cucumbers Behave, som automatisk kører acceptance test lavet i applikationen. De opsættes i en .feature fil, der giver grundlaget til et step, hvor koden bliver testet. Eksempel på feature ses på figur 40. Product owner skiftede senere hen definition of done til at ekskludere dette punkt, da han heller vil se resultater frem for opfyldning af krav.

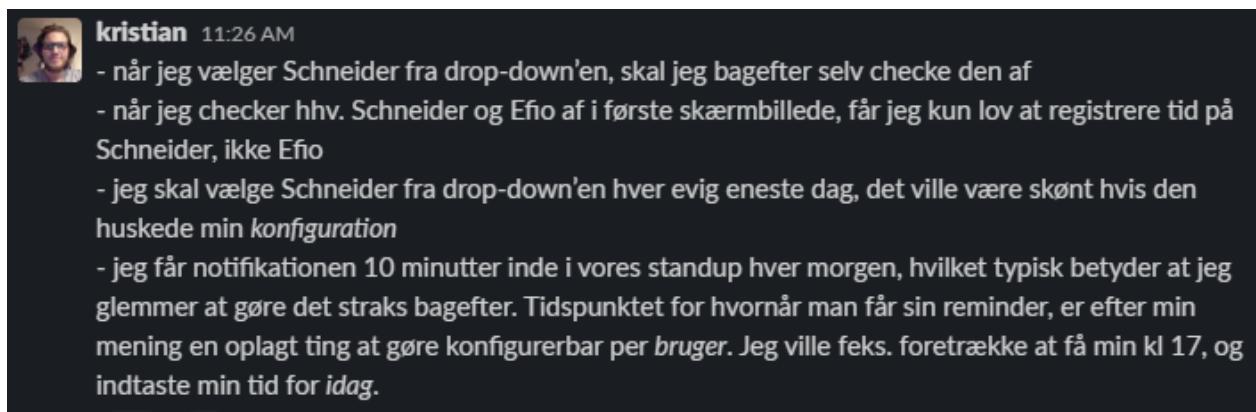
```
1  Feature: Slack Slash Command
2
3      Scenario: a user wants to register with slash Command
4          Given an unregistered consultant
5          When you send /signup Command
6          Then pull user data and publish to database
7
```

Figur 40: Cucumber behave .feature fil til acceptance test.

Master branchen er låst, hvilket vil sige, at der ikke kan pushes direkte op til master branchen uden, at det går igennem et pull request. Før et pull request kan blive godkendt, kræver det at både CI-pipelinen bliver godkendt, og at der er en anden, der kigger kodeændringerne igennem og godkender dem. Når dette er gjort er andet punkt opfyldt. Det tredje punkt opfyldes, efter et pull request er blevet godkendt og derefter kan flettes sammen med master branchen. I forbindelse med vores branch strategi bliver det sidste punkt opfyldt ved at oprette et nyt release på GitHub.

15.5 Brugerfeedback

Vi har løbende fået feedback fra konsulenterne hos Efio. Efter udgivelsen af vores applikation til konsulenterne i slutningen af praktikperioden, blev der oprettet en kanal dedikeret til fejlregistrering på applikationen. Konsulenterne havde også adgang til vores Jira og kunne oprette issues dertil. Dette gav os løbende et godt billede af, hvad der skulle forbedres og gjorde det nemmere for os at finde fejl i koden. Kanalen fungerede også som platform til idéer og gav os indsigt i, hvad den enkelte konsulent havde af ønsker til applikationen. I figur 41 ses et udsnit fra kanalen med både konstruktiv kritik og løsningsforslag.



Figur 41: Udsnit af #time-reg-incidents med besked fra en konsulent.

15.6 Extreme Programming

Vi har gjort brug af nogle extreme programming teknikker under udviklingen, heraf pair programming, collective ownership og coding standard. Pair programming er en udviklingsmetode fra Extreme Programming, som vi har benyttet os meget af. I normale omfang bruger man pair programming fysisk, men dette har dog ikke været muligt under udviklingen af dette projekt, og derfor har vi gjort det digitalt i stedet. Pair programming har vi gjort brug af, særligt når vi har stået foran komplekse problemer eller, hvor vi har stået over for en teknologi, som ens medstuderende har bedre kompetencer i og kendskab til. På denne måde lærer man både om enten en fremmed teknologi eller flere stykker kode, som man ikke har kendskab til i forvejen, og man kan dermed i fællesskab ende med et mere optimalt løsningsforslag ved at sparre med hinanden. Dette bidrager til en større forståelse og bedre overblik over hele applikationen, frem for kun at kende til det man selv udvikler. Vi har desuden brugt pair programming meget i forhold til udviklingen af det nye design, så vi alle har været inde over og er kommet med idéer til, hvad der fungerer godt og dårligt. Dette giver mulighed for at stille en løsning op, som ikke ville have været den samme, hvis man havde udviklet alene, og medvirker dermed til at formindske utilfredsstillende brugerdesign.

Vi har i mindre omfang gjort brug af continuous integration ved at efter hver afsluttet user story, skal den flettes sammen med master, i stedet for at flette det hele sammen efter hvert afsluttet sprint. Dette gør det nemmere at integrere ændringer, da man håndterer de løbende ændringer, der kommer fra andre under udviklingen i stedet for til sidst. Dette hjælper med fejlhåndtering og gør det nemmere for den enkelte udvikler at finde ud af, hvorfor det ikke virker. Vi har også i mindre omfang gjort brug af collective code ownership, der bunder i at alle kan lave ændringer i alles kode, og at man dermed ikke "ejer" nogle filer eller kodestykker, som man har udviklet.

16. Refleksion over processen

I dette afsnit vil vi gennemgå de tanker, vi har haft igennem processen af de forskellige teknikker, vi har gjort brug af og komme ind på, om vi skulle have gjort noget anderledes end det, vi gjorde.

16.1 Scrumroller

Normalvis på studiet har det altid været en fra studiegruppen, der har ageret scrum master, men det har været lærerigt at se, hvordan nogen ude i erhverveslivet udførte denne rolle. De teknikker som scrum master gjorde brug af med at lave evalueringer af sprintet halvvejs igennem, virker smart, når scrum masteren ikke er en, der også udvikler koden. Derudover har det været super godt at arbejde med en product owner, der ofte kan give svar tilbage omkring produktet i modsætning til studiet, hvor den tilgængelige samtale der har været med product owner, kom fra korte møder efter afluttet sprint.

16.2 Retrospektive møder

Sammen med scrum master har vi diskuteret, hvad der har fungeret godt og dårligt i forhold til scrum frameworkt. I studiet har det føltes overflødigt, da vi alligevel har diskuteret ting, vi synes er dårlige under et sprint. Her, hvor vi har haft en scrum master, som ikke er en del af udviklingsholdet, har det fungeret ret godt, og det har føltes som om at det giver bedre mening at have. Det kommer også af, at scrum master ikke hører de debatter, vi har løbende under et sprint og kan stille spørgsmål til emner, vi ikke selv har tænkt over.

16.3 Sprint review

Ligesom i studiet har vi her kunne få feedback på det udførte arbejde, ved at lave en demo af udarbejdede user stories. Det er også her, vi har kunne få gode råd fra product owner til eventuelle ændringer, rettelser, eller klargørelse af, hvordan det fungerer i en konsulents verden. Det har været god feedback at kunne få, og er en god praksis at benytte sig af.

16.4 Definition of ready

Idet vores product owner havde forskellige tilgange til sprint planning, resulterede det i at vi fik mulighed for at kunne bygge user stories og acceptkriterier op, og få feedback fra product owner, hvis de var blevet opbygget forkert eller, hvis vi havde misforstået noget. Ved at inddrage diskussion før oprettelse af user story igennem INVEST, har vi skulle sætte ord på user storyen og forsvare de holdninger, vi bringer ud. Dette bidrager til, at der ikke bliver igangsat udvikling i noget, som ikke er klar til at blive udviklet, og vi sparar dermed en masse tid og frustration. Under vores uddannelse har vi tidligere estimeret user stories henholdsvis small, medium, large og legendary og har nu prøvet, hvordan man kan tildele story points til user stories under praktik- og hovedopgaveperioden. Desværre har dette

ikke gjort vores estimeringsevner bedre, som blev demonstreret i sprint 1. Vores product owner fortalte os, at det er meget normalt at man i starten ender med at estimere forkert, da det er en svær opgave at ramme præcis estimering. Derfor skal vi for at blive bedre til fremtidige estimeringer, blive ved med at lære, hvorfor vores estimeringer ender med at være forkerte og fortsat prøve at estimere user stories. Vi fortsætter derfor også med at bruge risk evaluation, så vi kan sætte endnu flere ord på vores estimeringer. Det har været en god øvelse for os at kunne argumentere for, om der er høj eller lav risiko for estimeringen kan gå galt, og hvilke faktorer der kan påvirke dette.

16.5 Branch strategi

Tidligere har vi arbejdet med, at man udvikler på en developer branch, som er den man arbejder på alle sammen. Ved at vi har gjort brug af en ordentlig branch strategi, har det resulteret i, at vi har undgået mange og store merge konflikter, som plejer at være en hyppig udfordring ved tidligere branch strategi. Vi har kunne arbejde adskilt på den samme kode og koble det sammen uden store udfordringer, og kunne teste simpelt og nemt i et særskilt miljø.

16.6 Definition of done

Ved at bruge definition of done har vi også kunne sørge for at alle får udviklet på en uniform måde, skrevet dokumentation og sikret at når noget bliver anerkendt som færdigt, at det så reel er det. Desværre blev det første punkt i definition of done skåret fra, da product owner hellere ville have fokus på, at vi kunne levere flere resultater end at vi fik testet alt igennem, som dermed ændrede definition of done. Dette er ærgerligt da vi lige havde sat os ind i en ny teknologi og at selve konceptet med acceptance test er ret spændende. Det har givet os en basal viden om brugen af det, men desværre ikke en dybere viden. Derfra stoppede vi også med at bruge extreme programmings teknik test driven development, da der ikke længere var behov for det. Dette er også noget vi ærgrede os over, da vi føler at det er noget vi har manglet under vores uddannelse som datamatiker, og netop havde tænkt os at gøre brug af under praktik- og hovedopgaveperioden.

16.7 Brugerfeedback

Det har hjulpet os meget at vi har haft andre mennesker til at bruge vores applikation, hvortil de har haft et forum, hvor de kunne give deres holdning til systemet men også indrapportere fejl. Man kan sige at vi på en måde har outsourceret testing af applikationen til konsulenterne, som på en måde opvejer dét, at vi fjernede første punkt af definition of done.

16.8 Extreme Programmering

Pair programming er en teknik vi er meget tilfredse med og gør meget brug af. Desværre som situationen har stået til, har vi ikke kunne opnå det fulde potentiale af pair programming, da det fungerer bedst, når man sidder sammen face-to-face. Det er en super god teknik til at kunne kombinere flere hjerner til at takle et problem og er en god teknik at bruge, hvis man ikke selv har meget erfaring med en teknologi, som en anden måske har. Man kan derved lære, hvordan det skal bruges i praksis ved at man gør brug af pair programming. Da vi arbejdede på at udvikle det nye design til applikationen, hjalp pair programming os ved at opnå konsensus om, hvordan designet skulle se ud. Dette betød at vi udviklede et design hurtigere og mere effektivt, end hvis vi havde udviklet udkast individuelt og vist til hinanden.

Vi har også i et mindre omfang gjort brug af test driven development som fortalt i definition of done og continuous integration. Det har været meget nemmere at gøre brug af continuous integration, i takt med at vi har brugt den branch strategi, vi har, som har gjort teknikken endnu bedre. Det har fungeret godt, løbende at få integreret det arbejde, der er lavet ind i master branchen, da vi har undgået de store merge konflikter, der havde været, hvis

vi havde flettet det hele sammen i slutningen af sprintet. På den her måde kan vi også sikre, at det vi lægger op, fungerer som det skal, og ikke skal finde fejl i slutningen af sprintet og blive stresset over det.

17. Evaluering af processen

Overordnet set har vi haft en god proces, hvilket kommer af de teknikker, vi har gjort brug af, og som vi har arbejdet med under vores uddannelse. Vi har også tillært os nye processer under praktik- og hovedopgaveforløbet, som f.eks. definition of ready og definition of done. Idet vi har arbejdet videre på vores praktikprojekt, har vi dermed fortsat brugen af de samme teknikker som i praktikperioden. Vi er dermed blevet bedre til at gøre brug af de teknikker, der tilhører vores udviklingsmetode før påbegyndelsen af hovedopgaveforløbet. Det der dog stadig ikke har fungeret optimalt, er vores estimeringer, som skabte nogle problemer i første og andet sprint på trods af, at de user stories havde, havde været igennem hele definition of ready. Vores product owner er selv uddannet datamatiker og har dermed været god til, at formidle de krav der blev stillet på en måde, så det var forståeligt.

Vi har fortsat gjort brug af daily standup, hvilket har fungeret godt og givet os et overblik over dagen allerede fra start af. Ved at vores scrum master evaluerede sprints halvvejs igennem, har vi både kunne give ham en indsigt i udviklingen, men vi har også selv kunne få hjælp, hvis der var noget vi sad fast i. Scrum master og product owner har begge været tilgængelige over Slack, hvilket betød, at vi altid havde en kommunikationsvej med dem, så hvis der var noget der hastede kunne scrum master eller product owner hjælpe med udfordringen. At have så meget tilgængelighed til product owner og have en scrum master, der ikke er en del af udviklerne, har været positivt, da de kan se udfordringer og problemstillinger fra et andet perspektiv end os. Denne evaluering har primært foregået digitalt og vi har ikke haft mulighed for, at finde ud af, om det ville have fungeret bedre, hvis evalueringen havde fundet sted face-to-face. Til gengæld har der været en betydelig forskel på sprint review, retrospektive møder og sprint planning fra da vi gik fra face-to-face til digitale møder. Alle møderne blev væsentligt kortere og der var ikke samme mulighed for, at kunne illustrere eksempelvis vha. whiteboard eller gennem kode. Der var ydermere et mere personligt touch over møderne, da de var face-to-face og alle punkter, der hører til de møder, blev gået meget mere i dybde med, hvorimod det blev væsentligt kortere og mindre detaljeret, da det foregik digitalt. Som eksempel kunne de tre møder face-to-face vare seks timer, mens de digitale varede 1-2 timer. Overordnet har disse møder haft større værdi og mening, da de blev udført face-to-face. Vi har været heldige at være i en virksomhed, hvor konsulenterne var oprigtigt engageret, i det vi lavede og kom med forslag til videreudvikling. De var desuden til stor hjælp med at rapportere fejl, da de opstod, så de kunne blive rettet til.

Den måde vi har udviklet på og den proces, vi har arbejdet igennem, har givet et meget positivt flow i udviklingen og vigtigst af alt udførelsen. Vi har arbejdet lidt langsommere ved brug af disse processer på nogle fronter, men de har skabt et grundlag for, at vi har udviklet en applikation, der er mere robust, velbygget og dokumenteret på baggrund heraf. Denne måde at arbejde på ser vi meget positivt på, og vi vil så vidt muligt prøve at inddrage det til vores fremtidige udvikling. Havde vi skulle gøre noget anderledes til et fremtidigt projekt, så havde vi insisteret på vigtigheden i test driven development igennem hele projektet, i stedet for at skære det fra under udviklingen, da det ikke er givet, at der sidder en flok engagerede konsulenter klar til at gennemteste systemet for os. Scrum og XP er stensikre vindere som udviklingsmetoder for os, da vi efterhånden har afprøvet dem nogle gange og de har ikke slået fejl endnu. Dog skal det siges at vi stadig er i et tidligt stadiet i vores karriere som datamatiker, så derfor havde vi muligvis også afprøvet andre XP praktikker af og måske endda valgt en anden udviklingsproces, end dem vi har redegjort for, hvis vi skulle lave en ny systemudviklingsproces. Vi har alle tre under vores tid på uddannelsen som datamatiker, været glade for, at uddannelsen har stået inde for hands on learning og derfor handler det for os om, at få afprøvet nogle ting i praksis, så vi på den måde kan finde ud af, hvad der fungerer og ikke fungerer.

18. Konklusion

Vi vil i dette afsnit sammenfatte vores vurderinger og refleksioner, for at forsøge at komme frem til en besvarelse af vores problemstillinger.

I vores projekt ville vi undersøge, om det var muligt for os som studerende, med de kompetence og erfaringer vi havde, at løse vores opstillede problemstillinger. Efter vores vurdering er problemstillingerne løst og dokumenteret efter bedste evne. Vi har i denne rapport identificeret og dokumenteret alle krav, der blev opstillet igennem user stories og været i stand til, at danne os et overblik over product owners forventninger. Målene til systemet har vi også haft rig mulighed for at opfylde ud fra de opstillede krav.

Vi vurderer at de teknologierne, som vi endte med at bruge, har egnet sig bedst til produktet. Python og AWS som er de største teknologier vi har benyttet, har været til stor gavn for udviklingen og hjulpet os i mål med projektet. Med valget af teknologier fulgte en god arkitektur, hvor vi valgte at arbejde udfra trelags-arkitekturen, som vi syntes var fordelagtig for produktet, da vi arbejde med microservices. Med den valgte arkitektur kunne vi derved nemt udskifte funktioner både som hele lag, men også i form af de microservices vi havde bygget. CI/CD-pipelinens gjorde det nemt at teste, udskifte og publicere vores kode og er dermed fordelagtigt for projektet, da Efio har mulighed for at videreudvikle og udskifte microservices efter behov. Designet af brugergrænsefladen synes vi endte med at have en tilstrækkelig brugervenlighed, da designet er enkelt og konsulenterne nemt og hurtigt kan udfylde den information, der er krævet for at tidsregistrere.

Produktet syntes vi ikke er blevet testet i et bredt nok omfang, på baggrund af, at vi tidligt i forløbet fik stillet et krav af product owner om, at han hellere ville se flere resultater og vi dermed skulle skære ned på test. Ved at skære ned på test kunne vi ikke gennemteste systemet fuldstændigt og vi mener derfor ikke at systemet blev testet i det omfang, der i første omgang var forventet. Dette betød også at DoD i vores systemudviklingsproces, blev ændret. Vi har haft nemt ved at identificere product owners opstillede krav, men på nogle områder måske undervurderet nogle af kravene. Selvom vores estimering af de givne krav muligvis har halteret en smule, synes vi selv at vi har både planlagt og gennemført en god systemudviklingsproces.

19. Litteraturliste

Acceptance Testing. (s.d.). *Softwaretesting Fundamentals*. Lokaliseret den 29.december 2020 på:
<https://softwaretestingfundamentals.com/acceptance-testing>

Alistair Cockburn, A.C. (s.d.). Crystal clear. *ResearchGate*. Lokaliseret den 7.januar 2021 på:
https://www.researchgate.net/publication/234820806_Crystal_clear_a_human-powered_methodology_for_small_teams

Amazon SNS Billede. (s.d.). Amazon. Lokaliseret den 29.december 2020 på:
<https://docs.aws.amazon.com/sns/latest/dg/images/sns-overview-1.png>

Amazon. (s.d.). *AWS CloudFormation resource*. AWS. Lokaliseret den 29.december 2020 på:
<https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cfn-resource-specification>

Black Box Testing. (s.d.). *Softwaretesting Fundamentals*. Lokaliseret den 29.december 2020 på:
<https://softwaretestingfundamentals.com/black-box-testing>

Cucumber. (s.d.). *Cucumber*. Lokaliseret den 10.januar 2021 på:
<https://cucumber.io/>

Extreme Programming (XP) vs Scrum Billeder. (s.d.). Visual Paradigm. Lokaliseret den 7.januar 2021 på:
<https://www.visual-paradigm.com/servlet/editor-content/scrum/extreme-programming-vs-scrum/sites/7/2018/12/extreme-programming.png>

Google. (s.d.). *Google Python Style Guide*. Google Github. Lokaliseret den 29.december 2020 på:
<https://google.github.io/styleguide/pyguide>

Integration Testing. (s.d.). *Softwaretesting Fundamentals*. Lokaliseret den 29.december 2020 på:
<https://softwaretestingfundamentals.com/integration-testing>

Kanchan Kulkarni, K.K. (s.d.). Test Driven Development. *Guru99*. Lokaliseret den 2.januar 2021 på:
<https://www.guru99.com/test-driven-development>

Mike Griffiths, M.G. (s.d.). Boehms og Turners radar chart. *Typepad*. Lokaliseret den 7.januar 2021 på:
https://leadinganswers.typepad.com/leading_answers/files/agile_suitability_filters.pdf

Navdeep Singh Gill, N. G. (s.d.). *Test Driven Development Billeder*. Xenonstack. Lokaliseret den 29.december 2020 på:
<https://images.xenonstack.com/blog/test-driven-development-process-cycle.png>

Neha Vaidya, N. V. (s.d.). *Software Testing Billeder*. edureka!. Lokaliseret den 29.december 2020 på:
<https://d1jnx9ba8s6j9r.cloudfront.net/blog/wp-content/uploads/2019/02/V-V-model-Software-Testing-Tutorial-Edureka-768x353.png>

Planning Poker Billeder. (s.d.). Amazon. Lokaliseret den 7.januar 2021 på:
https://images-na.ssl-images-amazon.com/images/I/61uQd%2BXyo%2BL._AC_SL1222_.jpg

Scrum Udviklingsprocess. (s.d.). *Wikipedia*. Lokaliseret den 7.januar 2021 på:
[https://en.wikipedia.org/wiki/Scrum_\(software_development\)#/media/File:Scrum_process.svg](https://en.wikipedia.org/wiki/Scrum_(software_development)#/media/File:Scrum_process.svg)

Spiral Model Billede. (s.d.). Wikipedia. Lokaliseret den 7.januar 2021 på:
[https://en.wikipedia.org/wiki/Spiral_model#/media/File:Spiral_model_\(Boehm,_1988\).svg](https://en.wikipedia.org/wiki/Spiral_model#/media/File:Spiral_model_(Boehm,_1988).svg)

System Testing. (s.d.). *Softwaretesting Fundamentals*. Lokaliseret den 29.december 2020 på:
<https://softwaretestingfundamentals.com/system-testing>

Team Happiness Billede. (s.d.). Amazon. Lokaliseret den 7.januar 2021 på:
https://images-na.ssl-images-amazon.com/images/I/61tD9vk5mXL._AC_SL1265_.jpg

Trelags-arkitektur Billede. (s.d.). Logi REPORT. Lokaliseret den 8.januar 2021 på:
https://www.jinfonet.com/wp-content/uploads/2018/08/3-tier_architecture-460x275.png

Unit Testing. (s.d.). *Softwaretesting Fundamentals*. Lokaliseret den 29.december 2020 på:
<https://softwaretestingfundamentals.com/unit-testing>

Vandfaldsmodellen Billede. (s.d.). Wikipedia. Lokaliseret den 7.januar 2021 på:
<https://da.wikipedia.org/wiki/Vandfaldsmodellen#/media/Fil:Vandfaldsmodellen.svg>

20. Bilag

20.1 Bilag 1

Sprint 1 - varighed 2 uger:

As a consultant

I want customers and contracts imported from CRM

So that i can register hours on relevant customers

Given a consultant

When manually written properties are entered

Then Close CRM overrides properties if set in Close

Given a function

When registering or searching for customers

Then customers are matched on CVR

Given a function

When importing customers

Then customers lead.status = customer in Close

As a consultant

I want individual possibilities on customers

So that checkin can be done faster

Given a consultant

When time registering

Then I get personal customers

Given a function

When predicting personal customers

Then use weighted system

Weighted system:

- All contracts is matched with a consultant
- Customers with no active contracts have weight 1
- Customers with active contracts have weight x5
- Customers with active contract for consultant have weight x10
- Yesterday's customers have weight x3
- Two days ago customers have weight x2
- Email sent to customer have weight x10
- Email received from customer have weight x5

As a admin

I want option to change weights on customer prediction
So that we can learn as we go

Given a weighted system
When an admin changed the weights (between 1-100)
Then the changed weights are reflected in the application

As a consultant

I want ability to register time on Efio sales/administration/training/labd/other
So that my work is valued
And I won't see nagging reminders

Given a consultant
When registering Efio work
Then I receive a receipt

As a consultant

I want ability to register absence
So that I won't see nagging reminders

Given a consultant
When registered as sick
Then update emoji on slack

Given a consultant
When registered on vacation
Then update emoji on slack

Given a consultant
When I register for vacation
Then I can specify duration

More info:

- Status emoji for child sick

As a consultant

I want ability to register hours including absence before hand
So that I won't be bothered on vacation or sickness

Given a consultant
When sick or on vacation
Then be able to start checkin on demand

Given a consultant
When registering absence
Then be able to register for several days

More info:

- Ability to choose date myself

As a consultant

I want ability to configure checkin time individually
So that I won't be disturbed if I have a recurring activity

Given a consultant
When checking time is not suitable
Then be able to change checkin time

More info:

- Checkin time can be set with 5 min granularity

As a consultant

I want the admin to have ability to choose between required and optional description on contract level

So that i won't have to type meaningless comments

Given a consultant

When time registering for a customer without a required description according to contract on Close

Then be able to continue without typing comment

20.2 Bilag 2

Sprint 2 - varighed 1 uger:

As a consultant

I want individual possibilities on customers

So that checkin can be done faster

Given a consultant

When time registering

Then I get personal customers

Given a function

When predicting personal customers

Then use weighted system

Weighted system:

- All contracts is matched with a consultant
- Customers with no active contracts have weight 1
- Customers with active contracts have weight x5
- Customers with active contract for consultant have weight x10
- Yesterday's customers have weight x3
- Two days ago customers have weight x2
- Email sent to customer have weight x10
- Email received from customer have weight x5

As a admin

I want option to change weights on customer prediction

So that we can learn as we go

Given a weighted system

When an admin changed the weights (between 1-100)

Then the changed weights are reflected in the application

As a consultant

I want ability to register absence

So that i won't see nagging reminders

Given a consultant

When registered as sick

Then update emoji on slack

Given a consultant

When registered on vacation

Then update emoji on slack

Given a consultant

When i register for vacation

Then i can specify duration

More info:

- Status emoji for child sick

As a consultant

I want the admin to have ability to choose between required and optional description on contract level

So that i won't have to type meaningless comments

Given a consultant

When time registering for a customer without a required description according to contract on Close
Then be able to continue without typing comment

20.3 Bilag 3

Sprint 3 - varighed 2 uger:

As a consultant

I want individual possibilities on customers

So that checkin can be done faster

Given a consultant

When time registering

Then I get personal customers

Given a function

When predicting personal customers

Then use weighted system

Weighted system:

- All contracts is matched with a consultant
- Customers with no active contracts have weight 1
- Customers with active contracts have weight x5
- Customers with active contract for consultant have weight x10
- Yesterday's customers have weight x3
- Two days ago customers have weight x2
- Email sent to customer have weight x10
- Email received from customer have weight x5

As a admin

I want option to change weights on customer prediction

So that we can learn as we go

Given a weighted system

When an admin changed the weights (between 1-100)

Then the changed weights are reflected in the application

As a consultant

I want ability to register absence

So that i won't see nagging reminders

Given a consultant

When registered as sick

Then update emoji on slack

Given a consultant

When registered on vacation

Then update emoji on slack

Given a consultant

When i register for vacation

Then i can specify duration

More info:

- Status emoji for child sick

As a consultant

I want the admin to have ability to choose between required and optional description on contract level

So that i won't have to type meaningless comments

Given a consultant

When time registering for a customer without a required description according to contract on Close
Then be able to continue without typing comment

As a consultant

I want to be able to register time on existing project and / or create new projects
So that i can register hours on the correct projects

Given a consultant

When a new project is submitted

Then the project is added to the database, and is shown in the modal

Given a consultant

When registering time for a customer

Then all active projects on customers contract are shown

As a consultant

I want to register vacation or absence in checkin

So that i don't have to log into Zenegy as well

Given a consultant

When registering vacation or absence in checkin

Then create absence or vacation in Zenegy

As a consultant

I want my checkin to update my Slack Status

So that others can know if I'm sick or on vacation

Given a consultant

When registering absence or vacation

Then update Slack Status for the given period

As a consultant

I want my time formatted correctly

So that i can get a better prediction of my time