Aki Saastamoinen

Tikettien luonti Twitter-julkaisujen perusteella

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

2.12.2021

Tiivistelmä

Tekijä:	Aki Saastamoinen
Otsikko:	Tikettien luonti Twitter-julkaisujen perusteella
Sivumäärä:	34 sivua + 1 liite
Aika:	2.12.2021
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Ohjelmistotuotanto
Ohjaajat:	Lehtori Simo Silander

Insinöörityön tarkoituksena oli luoda palvelu, joka etsii Twitteristä julkaisuja, joissa mainitaan tietty yritys tai tuote. Maininnoista etsitään avainsanoja, jotka viittaavat ongelmiin yrityksen palvelussa tai tuotteessa. Näistä julkaisuista muodostetaan tikettejä Jira-palveluun.

Työn tavoite oli tutkia, olisiko tällainen palvelu mahdollinen toteuttaa kustannustehokkaasti ja toisiko se mahdollista lisäarvoa yritysten tuki- ja kehitystiimeille. Kustannustehokkuuden näkökulmasta palvelu päätettiin toteuttaa käyttäen Amazonin tarjoamaa AWS-pilvipalvelualustaa. Näiden palveluiden avulla työ pystyttiin toteuttamaan palvelimettomalla arkkitehtuurilla. Palvelimeton arkkitehtuuri mahdollisti palvelun ajamisen ilman erikseen konfiguroitavia ja ylläpidettäviä fyysisiä palvelimia.

Työssä käytiin yksityiskohtaisesti läpi sovelluksen määrittelyyn ja teknologioiden valintaan johtaneet vaiheet, käytetyt teknologiat sekä itse sovelluksen kehitys. Lopuksi havaittiin sovelluksen toteuttamisen kustannustehokkaasti olevan mahdollista.

Avainsanat: AWS, Twitter, Jira, pilvipalvelu, palvelimeton arkkitehtuuri

Abstract

Author:	Aki Saastamoinen
Title:	Creating Tickets Based on Tweets
Number of Pages:	34 pages + 1 appendix
Date:	2 December 2021

Degree:	Bachelor of Engineering
Degree Programme:	Information and Communication Technology
Professional Major:	Software Engineering
Supervisors:	Simo Silander, Senior Lecturer

The aim of the study was to create a service, which would search Twitter for tweets mentioning a specific company or product. Mentions would be checked for keywords referring to issues in the company's service or in the specified product. Jira tickets would then be created from the matching tweets.

The goal was to study whether this kind of service could be created in a cost efficient manner, and bring additional value to support and development teams of businesses. From the standpoint of cost efficiency, the project was decided to be implemented using Amazon's AWS cloud platform. With cloud services, the use of serverless architecture was possible. Serverless architecture enabled services to be executed without physical servers, which would have required manual installation and configuration.

The thesis takes a detailed look into how the software was defined, how technologies were selected, what the selected technologies were, and finally the development of the software itself. It was found out that the software could be developed and deployed cost efficiently.

Keywords: AWS, Twitter, Jira, cloud services, serverless architecture

Sisällys

Lyhenteet

1 Johdanto 1

2 Projektin määrittely 2

2.1 Tuotemäärittely 2
2.2 Vaatimusmäärittely 2
2.3 Teknologioiden valinta 3
2.4 Toteutussuunnitelma 4

3 Käytettyjen teknologioiden esittely 5

3.1 Twitter 5
3.2 Twitter-ohjelmointirajapinta 6
3.3 Jira 6
3.4 Jira-ohjelmointirajapinta 7
3.5 Pilvipalvelut ja AWS 8
3.5.1 AWS Lambda 11
3.5.2 AWS IAM 13
3.5.3 AWS EventBridge 14
3.5.4 AWS CloudWatch 14
3.5.5 AWS API Gateway 15

4 Toteutus 15

4.1 AWS-ympäristön käyttöönotto 16
4.2 Twitter-rajapinnan käyttöönotto 17
4.3 Jira-rajapinnan käyttöönotto 22
4.4 Jäsentimen toteutus 25

5 Yhteenveto 28

Lähteet 31

Liitteet

Liite 1: Sovelluksen lähdekoodi

Lyhenteet

AWS: Amazon Web Services. Amazonin tarjoama pilvipalvelualusta.

FaaS: Function as a Service. Funktiot palveluna.

HTTPS: Hypertext Transfer Protocol Secure. Hypertekstin siirtoprotokolla, jolla dataa siirretään suojatusti verkossa.

IaaS: Infrastructure as a Service. Infrastruktuuri palveluna.

JSON: JavaScript Object Notation. JavaScript-olioihin perustuva tiedonvälitysformaatti.

PaaS: Platform as a Service. Alusta palveluna.

REST: Representational State Transfer. Ohjelmointirajapintojen arkkitehtuurimalli.

SaaS: Software as a Service. Ohjelmisto palveluna.

URL: Uniform Resource Locator. Datan sijainnin kertova merkkijono.

1 Johdanto

Ohjelmistokehityksen hankalimpia osa-alueita voivat olla tuotannon virheiden havaitseminen ja korjaaminen. Pääasiallinen kanava virheiden löytämiseksi on käyttäjien tekemät virheilmoitukset. Monissa palveluissa on joko ilmoitettu sähköpostiosoite virheilmoitusten tekemistä varten tai palvelun sisällä tarjolla erillinen lomake, jonka avulla käyttäjä voi antaa yhteystietonsa ja kuvauksen virheestä. Monet käyttäjät saattavat kuitenkin jättää virheilmoitukset usein tekemättä, koska he eivät välttämättä koe virheen häiritsevän heidän käyttökokemustaan tarpeeksi tai eivät halua käyttää aikaansa ilmoituksen laatimiseen. Käyttäjät saattavat kuitenkin keskustella virheistä sosiaalisessa mediassa, ja etenkin videopelien maailmassa on kasvavassa määrin yleistymässä virheistä kirjoittaminen sosiaalisessa mediassa.

Sosiaalisessa mediassa mainittavien virheiden löytäminen ja kerääminen voi olla hankala ja työläs prosessi. Tämän lisäksi virheilmoituksen tiketöinti eli virheen tutkimiseksi ja korjaamiseksi vaativan tehtävän kirjaaminen projektinhallintajärjestelmään on myös mahdollisesti aikaa vievä työ tehdä manuaalisesti.

Työn tarkoitus on tehdä sovellus, joka automatisoi virheilmoitusten etsimisen sosiaalisesta mediasta ja niiden kirjaamisen projektinhallintajärjestelmään. Lisäksi työssä tarkastellaan tällaisen sovelluksen kannattavuutta ja sen tuomaa mahdollista lisäarvoa ohjelmistokehitystiimeille. Tavoitteena on myös tutkia, minkälainen kohderyhmä hyötyisi sovelluksen käytöstä.

Työssä käydään läpi projektin määrittely niin sisällön kuin teknologian kannalta, teknologioiden valintaprosessi sekä toteutussuunnitelma. Lisäksi työssä käydään läpi sovelluksessa käytetyt teknologiat sekä kuvataan koko kehitysprosessi.

2 Projektin määrittely

2.1 Tuotemäärittely

Koska projektin tarkoitus oli tutkia sosiaalisessa mediassa tehtyjen vikailmoitusten tiketöimisen automatisoinnin kannattavuutta, päätettiin sovellus toteuttaa käyttäen yleisimpiä palveluita sekä sosiaalisen median että tiketöinnin puolesta. Sosiaaliseksi mediaksi valittiin Twitter, sillä useilla yrityksillä ja tuotteilla on vahva näkyvyys kyseisessä palvelussa. Twitter myös mahdollistaa vahvasti keskustelukaltaisen kommunikaation yrityksen ja käyttäjien välille. Tiketöinnin puolesta palveluksi valittiin Atlassianin tuottama Jira-palvelu. Jira on yksi maailman käytetyimmistä palveluista projektien ja tikettien hallinnoimiseen, ja se tarjoaa myös kattavan kehitysrajapinnan. Sovelluksen tulisi hakea Twitterin rajapinnasta tietyin väliajoin julkaisuja, joissa asiakas tai asiakkaan tuote mainitaan, etsiä julkaisun tekstistä viittauksia ongelmiin asiakkaan palvelussa ja muodostaa julkaisuista tikettejä Jira-palveluun.

2.2 Vaatimusmäärittely

Jotta sovelluksen oikean maailman käyttötapauksia pystyttäisiin mahdollisimman tarkasti arvioimaan, päätettiin toteutus tehdä mahdollisen asiakkaan näkökulmasta myös teknologisesti houkuttelevaksi. Tämä tarkoitti sitä, että sovelluksen tulisi olla kustannustehokas ja helposti käyttöönotettavissa. Vaikka asiakkaaseen tai asiakkaan tuotteisiin liittyvien mainintojen etsiminen sosiaalisesta mediasta on tehtävänä yksinkertainen, päätettiin sovelluksesta silti tehdä mahdollisimman skaalautuva. Skaalautuvuus voi olla tärkeää asiakkaille, joilla on suuri ja aktiivinen käyttäjäkunta. Sovelluksen tulisi myös tukea sosiaalisen median ja tiketöinnin palveluiden vaihtamista mahdollisimman pienellä työmäärällä.

2.3 Teknologioiden valinta

Koska sovellus määriteltiin toteutettavaksi kustannustehokkaaksi ja skaalautuvaksi, ja sen on määrä hakea julkaisuja ajastetusti, oli pilvipalveluiden käyttäminen selkeä valinta [1]. Myös julkaisujen haun ajastettu suorittaminen ja se, että suurin osa sovelluksen toiminnallisuudesta on vain tekstin jäsentelyä ja siten erittäin nopeaa suorittaa, tekivät palvelimettoman arkkitehtuurin hyödyntämisen ilmiselväksi vaihtoehdoksi. Palvelimeton arkkitehtuuri poistaa tarpeen omien palvelinten hallintaan ja tarjoaa helpon tavan kasvattaa resurssien määrää laajempaa toimintaa varten. Palvelimeton Function as a Service -malli tarjoaa myös mahdollisuuden maksaa vain suoritukseen käytettävästä ajasta, kiinteän palvelinmaksun sijasta. [2.]

Vaihtoehtoja pilvipalveluille on nykyään useita. Suurimmat alan toimijat ovat Microsoft Azure-palvelullaan, Google Google Cloud -palvelullaan sekä Amazon Amazon Web Services (AWS) -palvelullaan. Jokaisella näistä palveluista on myös kattava tuki palvelimettomalle FaaS-arkkitehtuurille, joista kaikki ovat erittäin läheisesti verrattavissa toisiinsa hinnoittelun, toiminnallisuuksien ja ohjelmointikielien tuen puolesta. Koska löydettyjen vertailujen perusteella AWS oli kuitenkin kilpailijoihinsa verrattuna huomattavasti nopeampi funktioiden suoritusajassa [3], päätettiin toteutus tehdä käyttäen Amazonin AWS-palvelua.

AWS:n valinta palvelualustana tarkoitti sitä, että sovelluksen palvelimettomaan toteutukseen oli käytettävä AWS Lambda -funktioita. AWS Lambda -funktiot tarjoavat yksittäisten funktioiden ajamisen tapahtumien perusteella. Jokainen suoritus tapahtuu yksitellen [4] ja suoritusaikaa on rajatusti [5]. Suorituksesta maksetaan millisekunnin tarkkuudella [6], joten tehokkaasti toimiva funktio on erittäin kustannustehokas tapa suorittaa kevyitä operaatioita. Koska Lambda-funktioita ajetaan tapahtumien perusteella, piti julkaisujen ajastetun haun suorittamiseen toteuttaa ajastettu tapahtuman laukaisu. Tällaisia tapahtumien laukaisuja varten AWS tarjoaa EventBridge-palvelua.

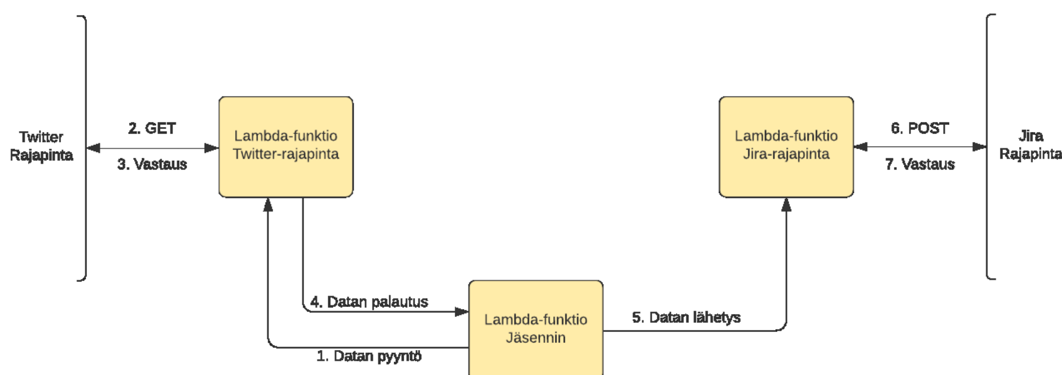
Funktioiden ajoympäristöksi valittiin Node.js, jonka avulla sovellus pystyttiin ohjelmoimaan JavaScript-kielellä, joka on yksi maailman käytetyimmistä ohjelmointikielistä ja jota on helppo kirjoittaa ilman kehittynyttä ohjelmointiympäristöä. Tämä mahdollistaa sovelluksen helpon käyttöönoton ja ylläpidon, sillä on todennäköistä, että mahdollisella asiakkaalla on jo valmiiksi JavaScriptiä osaavaa henkilöstöä. Tämän lisäksi sovelluksen lähdekoodia voidaan muokata suoraan AWS:n sisäänrakennetulla ohjelmointiympäristöllä. AWS:n sisäänrakennetun ohjelmointiympäristön käyttäminen poistaa myös tarpeen pakata lokaalisti kirjoitetut funktiot esimerkiksi .zip-tiedostoiksi, jotka ladattaisiin AWS-tilille joko AWS-verkkopalvelun kautta tai erikseen asennettavan komentoliittymän kautta.

Twitter- ja Jira-palveluiden valinta johti siihen, että rajapintoihin oli käytettävä REST-arkkitehtuurimallia. Twitter-rajapinta käyttää autentikointiin OAuth 2.0 -protokollan mukaista tunnistetta [7], kun taas Jira käyttää HTTP-protokollan mukaista Basic-autentikaatioskeemaa [8]. Molempien palveluiden rajapinnat käyttävät myös hyvien käytänteiden mukaisesti JSON-formaattia tiedon välittämiseen HTTP-kutsujen hyötykuormana.

2.4 Toteutussuunnitelma

Toteutus pystyttiin jakamaan kolmeen selkeään osa-alueeseen: Twitter-rajapintaa käyttävään funktioon, Jira-rajapintaa käyttävään funktioon ja jäsennin-funktioon, joka etsisi Twitter-julkaisuista avainsanoja ja välittäisi tiketin luomista varten tarvittavat tiedot Jira-rajapintaa käyttävälle funktiolle. Rajapintojen käyttöönotto oli molempien palveluiden kannalta samankaltainen prosessi: molempiin tuli luoda kehittäjäkäyttäjätunnukset, tutustua rajapintadokumentaatioon ja testata rajapinnan toimivuus. Funktioiden toteutusta varten tuli luoda käyttäjätunnukset AWS-palveluun, valita siellä sovellukselle sopiva palvelualue ja luoda AWS Lambda -funktiot. AWS Lambdan osalta päätettiin toteuttaa kolme erillistä funktiota kuvan 1 mukaisesti. Yksi funktio kutsuisi Twitter-rajapintaa, toinen toimisi varsinaisena jäsentimenä, joka käsittelisi Twitter-rajapinnasta saatujen julkaisujen datan muotoon, jonka perusteella voitaisiin luoda julkaisusta tiketti

Jiraan, ja kolmas funktio muodostaisi tiketin jäsentimeltä saadun datan perusteella ja kutsuisi Jira-rajapintaa. Funktioiden jako yhteen jäsentimeen ja erilisiin rajapintoja kutsuviin funktioihin eristää toimintalogiikan selkeiksi kokonaisuuksiksi. Tämän johdosta Jira-palvelun voisi vaihtaa toiseen tiketöintipalveluun muuttamatta jäsennintä tai Twitter-rajapintaa kutsuvaa funktiota. Jako erilisiin funktioihin helpottaa myös ylläpitoa ja ongelmien paikantamista.



Kuva 1. Sovelluksen suunniteltu arkkitehtuurimalli.

Toteutus päätettiin aloittaa rajapintojen käyttöönotolla. Kaikkiin palveluihin luotiin käyttäjätunnukset, jonka jälkeen keskityttiin Twitter-rajapinnan käyttöönottoon. Kun sen käyttöönottoa varten oli luotu alustava funktio AWS-palveluun, tehtiin sama Jira-rajapinnalle. Kun molemmat rajapinnat oli saatu otettua käyttöön ja kutsujen onnistuminen AWS-ympäristön ja rajapintojen välillä oli saatu varmistettua, pystyttiin aloittamaan jäsentimen toteutus.

3 Käytettyjen teknologioiden esittely

3.1 Twitter

Twitter on sosiaalisen median palvelu, jossa käyttäjät voivat luoda julkaisuja, vastata toisten julkaisuihin, jakaa toisten julkaisuja, seurata toisia käyttäjiä ja seurata keskustelunaiheita. Palvelu perustettiin vuonna 2006 ja siitä on nope-

asti kasvanut yksi maailman suurimmista sosiaalisen median palveluista. Palvelulla on arviolta 350 miljoonasta 400 miljoonaan aktiivista käyttäjää kuukaudessa [9].

Monilla yrityksillä on vahva näkyvyys Twitterissä. Useat yritykset luovat myös yksittäisiä tilejä tuotteilleen, jolloin heidän on helpompi kommunikoida suoraan tuotteen käyttäjien kanssa. Twitter on myös yksi suurimpia alustoja, joilla yritykset ilmoittavat toimintaansa ja tuotteisiinsa liittyvistä uutisista ja tapahtumista. Monet käyttäjät myös antavat vikailmoituksia ja palautetta yrityksille Twitterin välityksellä.

3.2 Twitter-ohjelmointirajapinta

Twitter tarjoaa laajan ohjelmointirajapinnan palvelunsa käyttämiseen ohjelmallisesti. Rajapinnan kautta on mahdollista muun muassa hakea ja luoda julkaisuja sekä etsiä käyttäjiä ja mainintoja käyttäjistä. Rajapintaa hyödyntäen on tehty myös useita kolmannen osapuolen sovelluksia Twitterin käyttämiseksi. Myös niin sanonut Twitter-botit, eli automaattiset Twitter-tilit, jotka seuraavat tiettyjä julkaisuja ja luovat omia niiden perusteella, ovat suosittuja projekteja ohjelmoijien keskuudessa. Näihin kuuluvat muun muassa Twitter-tilit @MuseumBot, joka julkaisee kuvia satunnaisista esineistä Metropolitan Museum of Artin arkistosta, ja @Mothgenerator, joka luo kuvia kuvitteellisista yöperhosista ja julkaisee ne Twitteriin antaen niille keksittyjä tieteellisiä nimiä.

3.3 Jira

Jira on australialaisen Atlassianin vuonna 2002 julkaisema tehtävienhallintaohjelmisto, jossa yksittäisistä tehtävistä voidaan luoda tikettejä, jotta niiden kehityksen vaihetta voidaan seurata. Tehtävänhallintaa hyödynnetään etenkin ketterässä ohjelmistokehityksessä, mutta myös esimerkiksi tukipyyntöjen ja virheilmoitusten seurannassa. Yksittäisestä tehtävästä luodaan Jira-ohjelmistoon tiketti, johon kirjataan yleensä vähintään tehtävän laatu, kuvaus ja laajuusarvio.

Tätä prosessia kutsutaan tiketöinniksi. Tämän jälkeen tehtävän kehitystä seurataan tiketin tilan perusteella, sekä keskustelulla, jota tiketin kommenttiosiossa käydään. Jiran tarjoaman kaltainen tehtävänhallinta on avainasemassa ketterässä ohjelmistokehityksessä, etenkin Scrum- ja Kanban-projektinhallinnan viitekehyksissä, mutta sitä hyödynnetään nykyään myös vanhemmankaltaisissa vesiputousmalleissakin.

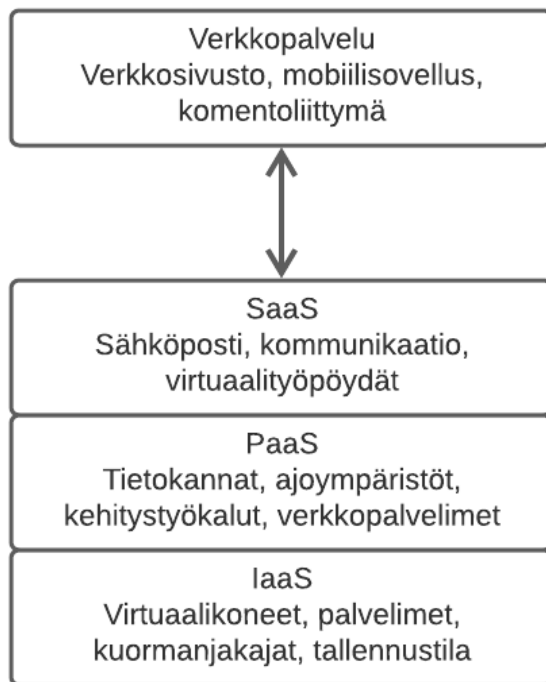
3.4 Jira-ohjelmointirajapinta

Twitterin tavoin Jira tarjoaa kattavan ohjelmointirajapinnan palveluunsa. Rajapinnan kautta pystyy muun muassa luomaan ja hallinnoimaan projekteja, sekä luomaan, muokkaamaan ja poistamaan tikettejä. Yleisiä käyttötapauksia rajapinnan hyödyntämiselle ovat esimerkiksi tikettien päivittäminen ulkoisen järjestelmän tapahtumien perusteella ja raporttien laatiminen tikettien kirjausten perusteella.

Jirasta on tarjolla kaksi versiota: Jira Cloud ja Jira Server. Jira Server on yrityksen omalle palvelimelle asennettava versio Jirasta, ja tarjoaa siten laajat mahdollisuudet palvelun mukauttamiselle projektien tarpeiden mukaiseksi. Jira Server tukee ohjelmointirajapinnan lisäksi myös laajaa kirjoa kolmannen osapuolen tahojen tekemiä lisäsovelluksia. Näihin kuuluu muun muassa ajankäytön raportointityökaluja, automaatiotestaustyökaluja ja integraatioita muihin ympäristöihin, kuten Atlassianin BitBucket git-repositorio -palveluun. Jira Cloud puolestaan on Atlassianin isännöimä pilvipalveluversio Jirasta. Jira Server -sovellukseen verrattuna Jira Cloud on edullisempi, ei tarvitse erillistä konfiguraatiota palvelimille ja on aina uusimmassa Jira-versiossa, sillä Atlassian itse isännöi palvelua. Jira Cloud ei kuitenkaan tue kolmannen osapuolen lisäsovelluksia yhtä laajasti kuin Jira Server, mutta ohjelmointirajapinnan tuki on molemmissa sama.

3.5 Pilvipalvelut ja AWS

Kommunikaation ja palveluiden siirtyessä yhä enemmän ja enemmän internet-pohjaisiksi, myös perinteiset palvelinratkaisut alkavat siirtyä internetin välityksellä hajautetuiksi pilvipalveluiksi. Pilvipalveluita on saatavilla useilla erilaisilla kokonaisuustasoilla, joita kutsutaan yleisesti ”as a Service” -palvelumalleiksi. Pilvipalveluiden tarjoamat palvelumallit jaetaan sisältönsä mukaisiksi kokonaisuuksiksi, joista yleisimmät ovat Infrastructure as a Service (IaaS), Platform as a Service (PaaS) ja Software as a Service (SaaS). IaaS-mallissa pilvipalvelu tarjoaa samoja palveluita kuin perinteiset palvelinratkaisut, sekä samankaltaiset mahdollisuudet niiden konfiguroimiselle. Palveluihin kuuluu muun muassa palvelimien, tallennustilan, kuorman jakajien ja virtuaalikoneiden tarjoaminen [10]. PaaS-mallissa tarjotaan vastaavasti ylemmän tason palveluita, kuten ohjelmistojen ajoympäristöjä ja kehitystyökaluja, tietokantoja ja verkkopalvelimia, ilman niiden normaalisti vaatimien taustajärjestelmien hankkimista tai konfiguraatioita [11]. Perinteisessä palvelinratkaisussa PaaS-mallin sisältämät palvelut pitäisi rakentaa IaaS-mallia vastaavien ratkaisujen päälle, jolloin ne pitäisi myös itse hankkia, asentaa ja konfiguroida. SaaS-mallissa tarjotaan vielä astetta ylemmän tason palveluita, kuten sähköposti- ja kommunikaatiopalveluita, virtuaalityöpöytiä ja asiakkuudenhallintatyökaluja [12]. Näitä palveluita tarjottaisiin perinteisessä mallissa PaaS-mallin tarjoamien palveluiden avulla. Palvelumallien sisällöt seuraavatkin siis suoraan perinteisiä arkkitehtuuritasoja, mutta tarjoavat mahdollisuuden niiden käyttämiselle ja hallinnoimiselle suoraan verkkopalveluiden välityksellä ilman, että palveluiden ostajalla on tietoa tarvitsemaansa alemman tason järjestelmistä. Tämä jaottelu mahdollistaa omasta laitteistosta luopumisen ja pienentää järjestelmien ylläpitämisen työmäärää, jonka ansiosta laitteistoon ja siihen liittyviin työtehtäviin meneviä kuluja saadaan vähennettyä. (Kuva 2.)



Kuva 2. Pilvipalvelumallit arkkitehtuuritason mukaisessa järjestyksessä.

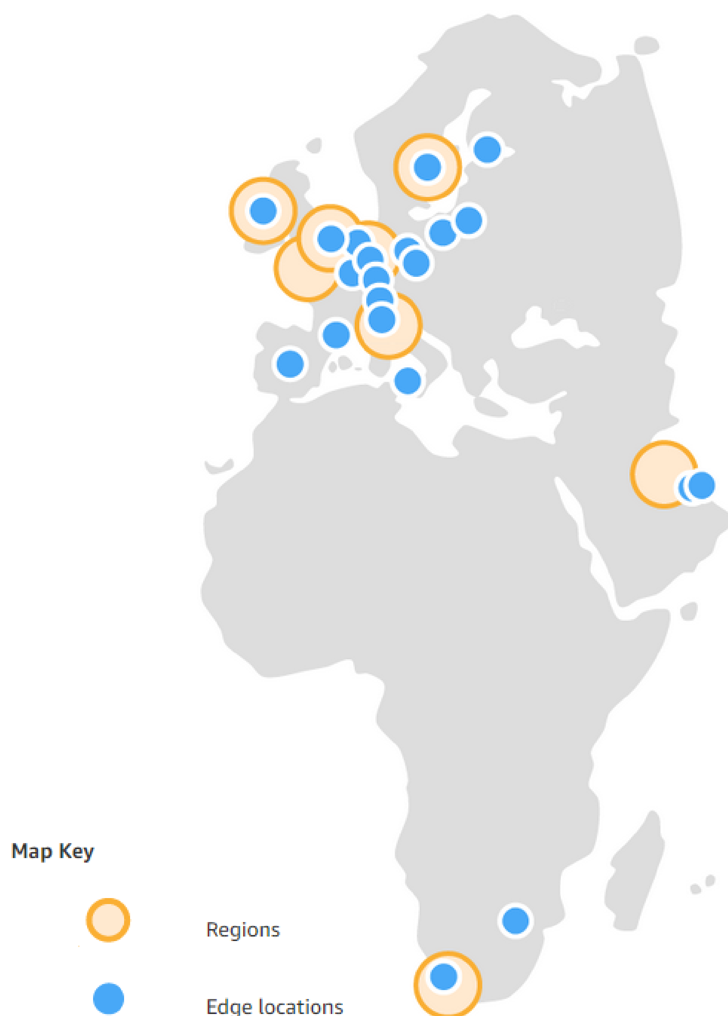
Yleisten palvelumallien lisäksi monet pilvipalvelut tarjoavat myös pienten sovellustoiminnallisuuksien kehitystä ja suoritusta varten Function as a Service (FaaS) -mallia. FaaS-mallissa kehittäjien ei tarvitse itse hallinnoida infrastruktuuria, jonka päällä sovellustoiminnallisuuksia ajetaan, vaan he voivat keskittyä pelkän ohjelman kehitykseen ja ylläpitämiseen. FaaS-mallin nimen mukaisesti sovellustoiminnallisuuksilla tarkoitetaan yksittäisiä pieniä funktioita, jotka suoritetaan kertaluontoisesti tapahtumien perusteella. Näitä voivat olla esimerkiksi lokitapahtumat tai rajapintatapahtumat, joiden perusteella funktio käsittelee dataa tietokannassa, käsittelee tapahtuman dataa ja palauttaa vastauksen rajapintaa kutsuneelle järjestelmälle tai tallentaa tapahtuman mukana tulleen hyötykuorman palvelimelle. FaaS-mallissa toiminnallisuuksien suorittaminen on erittäin kustannustehokasta, sillä käyttäjä maksaa vain funktion suoritukseen kuluvasta ajasta. Laskutus tapahtuu millisekunnin tarkkuudella, ja koska funktiot ovat tarkoituksenmukaisesti pieniä, on yksittäinen ajoaika rajoitettu palveluntarjoajan puolesta yleensä muutamiin sekunteihin. Esimerkkikoodi 1 on AWS

CloudWatch -palveluun kirjattu rivi sovelluksen jäsenninfunktion suorituksesta, josta näkyy suoritusaika, käytettävissä olevan ja käytetyn muistin määrä sekä laskutettu aika.

```
REPORT RequestId: c1f5a25c-adbb-4b6a-8695-d3c2c5f8bb90 Duration: 1601.70 ms    Billed Duration: 1602 ms    Memory Size: 128 MB    Max Memory Used: 76 MB Init Duration: 402.14 ms
```

Esimerkkikoodi 1. AWS CloudWatch -lokimerkintä jäsenninfunktion suorituksesta.

AWS on Amazonin vuonna 2002 julkaisema ja nykyään maailman suurin pilvipalveluiden tarjoaja. Vuodesta 2020 lähtien AWS on vastuussa noin 33 % kaikista IaaS-, PaaS- ja SaaS-palveluista maailmassa, kun taas sen suurimmat kilpailijat, Microsoft Azure ja Google Cloud Platform tarjoavat palveluista 18% ja 9% vastaavasti [13]. AWS:n Palvelut tarjotaan useista maantieteellisistä alueista (Region), jotka koostuvat niin kutsutuista saavutettavuusalueista (Availability Zone) ja lähialueista (Local Zone) ympäri maailman [14] (kuva 3). Yksi saavutettavuusalue koostuu yhdestä tai useammasta erillisestä ja itsenäisestä palvelinkeskuksesta. Koska palvelut voidaan tarjota saavutettavuusalueittain eikä palvelinkeskuksittain, ei yhden palvelinkeskuksen alueella tapahtuva sähkökatkos tai muu tapahtuma vaikuta palveluiden saatavuuteen. Saavutettavuusalueiden välille on rakennettu pienen latenssin ja suuren kaistan omaava datayhteys, joka mahdollistaa helpon ja luotettavan kuormanjaon maantieteellisen alueen sisällä. Lähialueet mahdollistavat laskennan suorituksen ja datan säilytyksen tietyllä pienellä maantieteellisellä alueella, jolloin paikallinen tuki esimerkiksi pienen latenssin palveluille on mahdollista. Näiden lisäksi AWS tarjoaa myös raja-alueita (Edge Location), jotka eivät suoraan tarjoa yleisiä AWS-palveluita, mutta toimivat AWS CloudFront -sisällönjakeluverkon yhteyspisteinä [15]. CloudFront mahdollistaa palveluiden varastoimisen välimuistiin, jolloin niihin on nopea pääsy raja-alueilta. Yhteyksien nopeuttamiseksi raja-alueilla on myös tarjolla AWS Route 53 -verkkotunnusjärjestelmäpalvelu, joka mahdollistaa yhteyksien luomisen nopeammin kuin ulkoisten palveluiden kautta.



Kuva 3. AWS:n palveluiden maantieteellisen sijainnit Euroopan ja Afrikan alueella.

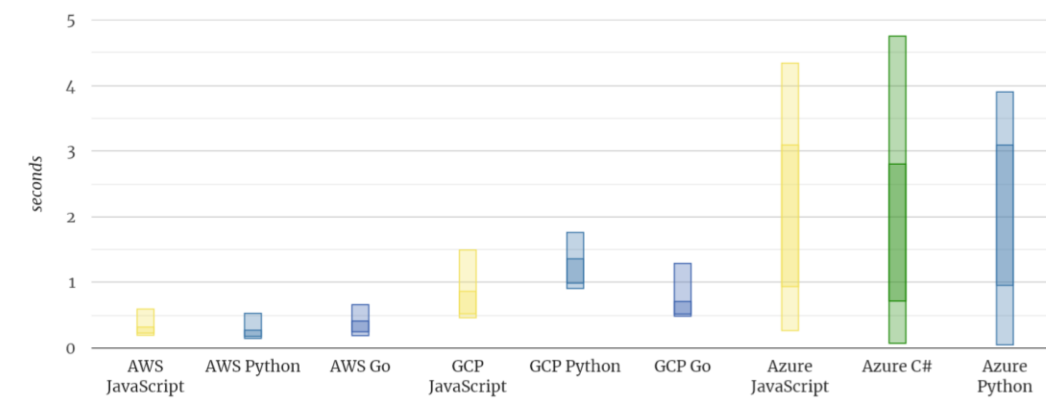
3.5.1 AWS Lambda

AWS tarjoaa FaaS-mallin mukaista funktioiden tapahtumapainotteista suoritusta Lambda-palvelullaan. AWS Lambda -funktiot ovat pieniä ohjelmistokokonaisuuksiaan, jotka suorittavat yhden yksinkertaisen toimenpiteen suorituksen käynnistävän tapahtuman perusteella. Yksittäisten funktioiden ajaminen tapahtumien perusteella mahdollistaa nopean ja kustannustehokkaan tavan toteuttaa pieniä operaatioita ilman tarvetta palvelimen ja ajoympäristön konfiguroinnille. AWS Lambda tukee useita eri ajoympäristöjä, kuten Node.js, Java, Python, Go ja .NET. Laajan ajoympäristötuen johdosta AWS Lambda tukee natiivisti Java-,

Go-, PowerShell-, JavaScript-, C#-, Python-, ja Ruby-kieliä, joiden lisäksi mahdollisuus lisätä omia ajoympäristöjä kasvattaa kielituen lähes rajattomaksi [16].

AWS Lambda -funktiot eivät nimestään poiketen ole itsessään funktioita, vaan instansseja ohjelmistokonteista [17], joissa Lambdalle tallennettu funktio suoritetaan. Ohjelmakontit pitävät sisällään version AWS:n omasta Linux-distribuutiosta, funktiokoodin suoritusta varten tarvittavan ajoympäristön ja valmiiksi konfiguroidut avoimet portit, jotta funktio pystyy vastaanottamaan ja lähettämään dataa kontin ulkopuolelle. Lambda on AWS:n tarjoaman palvelun tuotenimi, eikä sitä tule sekoittaa useimpien ohjelmointikielien tarjoamiin, funktionaaliseen ohjelmointiin liittyviin anonyymeihin funktioihin, joita usein myös kutsutaan lambda-funktioiksi.

Tapahtumapainotteisesta luonteestaan johtuen funktiot käynnistetään aina tapahtuman yhteydessä. Funktion käynnistäminen on kuitenkin verrattain hidasta, joten käynnistyksen ja suorituksen jälkeen funktio jää odotustilaan seuraavaa tapahtumaa varten, jolloin suoritus voidaan aloittaa heti ilman uutta käynnistystä. Eri palveluilla on eri odotusajat: AWS Lambda pitää funktioita päällä tavallisesti viidestä seitsemään minuuttia, Google Cloud Functions 15 minuuttia, ja Azure Function puolestaan pitää funktiot päällä yleisimmin 20-30 minuutin ajan. Koska funktiot eivät ole koko ajan päällä, on käynnistysnopeus tärkeä metriikka FaaS-palvelua valitessa. Käynnistysaikojen vertailuissa AWS on selkeä voittaja hyvin tasaisella kielestä riippumattomalla yleisesti alle puolen sekunnin käynnistysajalla. Google Cloud Functions sen sijaan suoriutuu toiseksi parhaiten, yleisesti 0,5-1 sekunnin käynnistysajalla JavaScript-ja Go -kielillä, mutta nousee useimmissa tapauksissa yli sekuntiin Python-funktioiden käynnistyksessä. Azure Functions häviää vertailun reilusti, mitaten JavaScript- ja Python-kielillä yleisimmät käynnistysajat noin 1-3 sekunnin välille. Kuvan 4 kuvaajassa tummemmat alueet kuvaavat 67% käynnistyksistä, ja vaalea alue kuvastaa 95% käynnistyksistä. [3.]



Kuva 4. Vertailu käynnistysajoista eri kielillä AWS Lambdan, Google Cloud Functionsin ja Azure Functionsin välillä [3].

Sovelluksen toteutuksessa päätettiin kustannustehokkuuden johdosta käyttää AWS Lambda -palvelua, sillä sovellus hakee julkaisuja Twitteristä ajastetusti tunnin välein. Tästä syystä käytännössä jokainen suoritus alkaa funktion käynnistyksellä, joten käynnistysaika vaikuttaa merkittävästi sovelluksen suoritusnopeuteen ja käytetyn pilvipalvelun laskutusasteeseen.

3.5.2 AWS IAM

AWS Identity and Access Management (IAM) on AWS:n tarjoama identiteetin- ja pääsynhallinnan työkalu. Sen avulla pystyy hallinnoimaan pääsyä AWS-palveluihin ja resursseihin luomalla käyttäjiä ja käyttäjäryhmiä, joille voi määritellä erilaisia käyttöoikeuksia ja käyttäjärooleja. AWS IAM mahdollistaa yksilöidyn ja erittäin tarkan pääsyoikeusmäärittelyn, monivaiheisen tunnistautumisen sekä pääsyvaltuusanalysoinnin, joka mahdollistaa valtuuksien rajaamisen käytön perusteella [18]. Käyttäjärooleja voi määritellä normaalien käyttäjien lisäksi myös sovelluksille ja funktioille, joten esimerkiksi funktion kykyä kutsua muita palveluita ja sovelluksia pystytään rajaamaan. AWS IAM kannustaa vähimpien valtuuksien periaatteen (Principle of least privilege, PoLP) mukaiseen valtuutukseen. PoPL:n mukaisesti käyttäjän tulee päästä käsiksi vain käyttötarkoituksiinsa nähden välttämättömiin resursseihin palvelussa [19]. AWS IAM tarjoaa

myös mahdollisuuden auktorisoida rajapintakutsuja allekirjoitusten perusteella. Sovelluksen toteutuksessa käytettiin IAM-rooleja antamaan funktioille oikeudet kutsua toisiaan.

3.5.3 AWS EventBridge

AWS EventBridge on tapahtumapohjaisen reitityksen työkalu. EventBridge pystyy vastaanottamaan, havainnoimaan ja reitittämään tapahtumia rajapintojen, AWS:n sisäisten palveluiden tai integroitujen SaaS-palveluiden välillä. EventBridgen avulla pystyy luomaan esimerkiksi viestin välitystä, jossa yhdessä palvelussa tapahtuva muutos käynnistää tapahtuman EventBridgessä, jonka seurauksena tapahtuman tiedot lähetetään kehitystiimin käyttämään chat-palveluun [20]. EventBridge mahdollistaa myös ajastettujen tapahtumien laukaisun, samaan tapaan kuin Unix-pohjaisten käyttöjärjestelmien cron-palvelu. Sovelluksessa käytettiin EventBridgen ajastettuja tapahtumia laukaisemaan julkaisujen haku ja käsittely aina tunnin välein.

3.5.4 AWS CloudWatch

AWS CloudWatch tarjoaa lokitusta ja metriikkaa AWS-ympäristössä suoritettavista sovelluksista. Sen avulla pystytään muun muassa optimoimaan resurssien hyödyntämistä ja reagoimaan järjestelmänlaajuisiin suoritusmuutoksiin. CloudWatchin kautta pystytään automaattisesti luomaan hälytyksiä poikkeuksellisen toiminnan tai normaalista poikkeavan suorituksen tapahtuessa, käynnistämään automaattisia toimenpiteitä reaktiona tapahtumiin järjestelmässä, etsimään virheitä sovellusten toiminnassa sekä visualisoimaan lokimerkintöjä ja niihin liittyvää metriikkaa [21]. CloudWatchia käytettiin sovelluksen kehityksessä virheiden etsintään sekä funktioiden välillä liikkuvan datan tarkasteluun.

3.5.5 AWS API Gateway

API Gatewayn avulla on helppo luoda esimerkiksi REST- tai Websocket-rajapintoja sekä ohjata liikennettä sisäisten palveluiden välillä. API Gateway mahdollistaa helpon tavan huolehtia liikenteen hallinnasta, Cross-Origin Resource Support (CORS) -tuesta, kutsujen auktorisoinnista, monitoroinnista, sekä versionhallinnasta [22]. Hyvien käytänteiden mukaisesti AWS Lambda -funktioita tulisi kutsua API Gateway -rajapinnan avulla. Koska Lambda-funktioiden kutsuminen suoraan tarkoittaa sitä, että kutsuvan funktion tulee tietää kutsuttavan Lambdan nimi, rajapinnan kautta kutsuminen estää virhetilanteita, jotka voivat johtua funktioiden muuttamisesta. Jos funktion nimi muuttuu tai laajan funktion toiminnallisuudet jaetaan useampaan pieneen funktioon, täytyy suoraan kutsuvien funktioiden koodia myös muuttaa tukemaan näitä muutoksia. Tästä käytännöstä poikkeaminen on kuitenkin perusteltua joissain tapauksissa, kuten pienissä projekteissa, joissa tiedetään, että funktiot tulevat pysymään pieninä eikä niitä tarvitse siksi jakaa useampaan osaan, tai joissa kehitystiimi on niin pieni, että muutoksien koordinoiminen ja niihin mukautuminen ei aiheuta ongelmia. API Gatewayn käyttö päätettiin jättää toteuttamatta sovellukseen, sillä nämä mainitut ehdot toteutuvat, eikä sovelluksella muutenkaan ole tarvetta olla kutsuttavissa ulkoisesta rajapinnasta [23].

4 Toteutus

Sovellus kirjoitettiin JavaScript-ohjelmointikielellä hyödyntäen Node.js-ajoympäristöä. AWS:n sisäänrakennetun ohjelmointiympäristön lisäksi toteutuksessa käytettiin Microsoftin Visual Studio Code -ohjelmointiympäristöä koodin rakenteen ja funktioiden dokumentaation selkeämpään tarkasteluun sekä Postman-rajapintatestaustyökalua.

Postman on joko pilvipalveluna tai lokaalisti tietokoneella käytettävä sovellus, jonka avulla REST-rajapintoja on helppo testata. Se tarjoaa kattavan tuen erilaisten kutsujen lähettämiselle, joita on myös helppo muokata juuri sellaiseen

muotoon kuin käyttäjä haluaa. Postmanilla on myös mahdollista tehdä kokonaisia testauskokonaisuuksia, joita voi tallentaa tiedostoihin ja jakaa esimerkiksi testaustiimin kesken [24]. Postmanin käyttö rajapintojen testaamiseen mahdollisti nopean tutustumisen kummankin rajapinnan tarjoamiin toiminnallisuuksiin sekä niiden datamalleihin. Myös mahdolliset virhetilanteet testattiin lähettämällä rajapintoihin väärin formatoituja kyselyitä ja tarkastelemalla rajapintojen reaktioita niihin.

Visual Studio Coden LSP (Language Server Protocol) -kielipalvelutuki helpotti huomattavasti koodin kirjoittamista. Kielipalvelut kertovat ohjelmointiympäristölle ohjelmointikielelle spesifeistä ominaisuuksista, jotta ohjelmointiympäristö pystyy tarjoamaan ohjelmoijalle tietoja käytetyistä funktioista ja kirjastoista, ehdottamaan funktioita ja koodin automaattista täydennystä sekä värikoodaamaan koodin osia tehden koodista helppolukuisempaa [25]. Funktioehdotukset, koodin automattinen täydennys ja suoraan ohjelmointiympäristössä saatava tieto funktioista ja kirjastoista vähentävät tarvetta dokumentaation lukemiselle ja helpottavat hahmottamaan kirjastojen toimintaa käytännönläheisestä näkökulmasta. Näiden toiminnallisuuksien avulla löytää helposti funktioita ja ominaisuuksia, jotka tarjoavat uusia ja elegantteja ratkaisuja ongelmiin.

4.1 AWS-ympäristön käyttöönotto

AWS:n käyttäminen vaatii tilin rekisteröimistä sekä pankki- tai luottokortin liittämistä tilille laskutusta varten. Suurin osa AWS:n palveluista ovat ilmaisia ensimmäisen vuoden ajan tietyillä rajoituksilla, kuten tallennustilan tai kutsujen määrän maksimirajoilla. Monet palveluista ovat myös aina ilmaisia vastaavin rajoituksin. Rajoitukset ilmaisella tasolla ovat usein melko korkeita. Esimerkiksi AWS S3 -tallennuspalvelun saa ilmaiseksi käyttöön ensimmäiseksi 12 kuukaudeksi 5 gigatavun tallennuskapasitettilla. DynamoDB-tietokantapalvelun käyttö on aina ilmaista 25 gigatavun tallennuskapasiteetilla, ja Lambda-funktiota voi käyttää aina ilmaiseksi miljoonan kuukausittaisen suorituksen käynnistävän kutsun rajoituksella [26]. Projektia varten tarvittavat palvelut olivat kaikki ilmaisia.

Tilin luomisen jälkeen luotiin valmiiksi kolme tyhjää AWS Lambda -funktiota Jira-rajapintaa, Twitter-rajapintaa ja jäsennintä varten. Ennen funktioiden luomista piti valita, minkä maantieteellisen alueen palvelimilla ne suoritetaan. Lambda-funktion luomisen yhteydessä täytyy valita funktiokoodin suoritukseen tarvittava ajoympäristö sekä valita tai luoda AWS IAM -rooli, joka antaa oikeudet funktion kutsumiseen. AWS antaa vaihtoehdon luoda IAM-roolin itse, tai antaa AWS:n luoda rooli automaattisesti. Projektin tarpeita varten riitti, että käytettiin AWS:n automaattisesti luomaa roolia. Kun kaikki kolme funktiota ja niiden käyttöoikeusroolit oli luotu valmiiksi odottamaan toteutusta, voitiin siirtyä toteuttamaan rajapintojen käyttöönottoa.

4.2 Twitter-rajapinnan käyttöönotto

Twitter-rajapintaa varten tuli luoda kehittäjätunnus. Tunnuksen saaminen edellyttää kattavan hakemusprosessin läpikäymistä. Hakemuksessa täytyi ilmoittaa, minkälaista kehittäjätunnusta haetaan. Tunnukset jaetaan useisiin eri käyttötarkoituksiin: ammattilaisiin, harrastelijoihin ja akateemisiin käyttäjiin. Ammattilaistunnukset on tarkoitettu muun muassa yritystenvälisten sovellusten kehittämiseen, kaupallisten kuluttajapalveluiden kehittämiseen ja mainosten ohjelmalliseen julkaisuun. Harrastelijatunnukset ovat tarkoitettu yksittäisille kehittäjille, jotka luovat Twitter-botteja, työkaluja käyttäjille, tai jotka haluavat vain tutkia rajapintaa ja sen tarjoamia toiminnallisuuksia. Akateemiset tunnukset on tarkoitettu tutkijoille, opettajille ja opiskelijoille [27]. Projektia varten haettiin akateemisia tunnuksia opiskelijana.

Tämän jälkeen hakemukseen tuli tarkentaa, miten rajapintaa tultaisiin käyttämään. Tarkennuksessa piti ensin kuvata rajapinnan ja Twitterin datan käyttöä omin sanoin. Kaikissa kohdissa kehotetaan olemaan mahdollisimman yksityiskohtainen, jotta hakemuksen katselmuksessa ei jäisi tulkinnanvaraisuuksia ja hyväksyntä voitaisiin antaa mahdollisimman nopeasti. Omasanaisen kuvauksen jälkeen piti ilmoittaa, aiotaanko Twitterin dataa analysoida; lähettää, uudelleen jakaa, tykätä julkaisuista, seurata käyttäjiä tai lähettää viestejä käyttäjille; näyttää julkaisuja tai kerätä dataa Twitterin sisällöstä Twitterin ulkopuolelle; Twitterin

sisältöä tai siitä johdettua dataa antaa valtiollisille toimijoille. Sovellusta varten ilmoitettiin aikomukseksi näyttää julkaisuja Twitterin ulkopuolella. Tämän jälkeen hakemus lähetettiin hyväksyttäväksi. Kehittäjätunnukset saatiin käyttöön heti lähetyksen jälkeen, eli hyväksymisprosessin ajan oli mahdollista jo käyttää rajapintaa. Koska Twitter on tarkka siitä, kenelle ja mihin käyttötarkoituksiin antaa kehittäjätunnuksia, tehtiin heti tunnusten saamisen jälkeen Postmanilla projektia varten tarvittava kysely rajapintaan. Tällä tavoin saatiin Twitterin vastauksen mukainen datamalli talteen siltä varalta, että hakemus olisi hylätty ja myöhempi pääsy rajapintaan evätty. Datamallin perusteella pystyttäisiin luomaan oma testirajapinta, joka palauttaisi oikean mallista testidataa, jotta projekti voitaisiin silti toteuttaa, vaikka viralliseen Twitter-rajapintaan ei olisikaan pääsyä. Testirajapintaa ei kuitenkaan tarvinnut toteuttaa, sillä kehittäjätunnushakemus hyväksyttiin ongelmitta.

Kehittäjätunnuksien saannin jälkeen luotiin kaksi uutta Twitter-tiliä: yksi kuvitteelliselle yritykselle, jonka mainintoja Twitteristä haettaisiin, ja toinen kuvitteellisen yrityksen palvelun käyttäjälle. Twitter-tilin luominen on yksinkertaista, eikä tarvitse muuta kuin sähköpostin tai puhelinnumeron. Yrityksen tiliksi luotiin Hypothetical Company Ltd. - @HypotheticalLtd, ja asiakkaan tiliksi Hypothetical Customer - @NeedHypothetics.

Testitilien luomisen jälkeen alettiin toteuttaa ensimmäistä versiota AWS Lambda -funktiosta, joka hakisi Twitterin rajapinnasta julkaisuja joissa @HypotheticalLtd-tili mainitaan. Toteutus aloitettiin lisäämällä tarvittavat auktorisointitiedot ympäristömuuttujaan. Twitter-rajapinta käyttää OAuth 2.0 -protokollaa, jonka mukaisesti piti kyselyiden header-tietojen Authorization-kenttään lisätä OAuth 2.0 -protokollan mukainen Bearer-tieto, joka piti sisällään kehittäjätunnuksen auktorisaatiotunnisteen. Tunniste saatiin Twitterin kehittäjäportaalista, ja tallennettiin AWS Lambda -instanssin ympäristömuuttujaksi. Seuraavaksi toteutettiin yksinkertainen HTTPS GET -kysely, jonka funktio lähetti Twitterin rajapintaan osoitteeseen: https://api.twitter.com/2/users/:id/mentions, jossa ID-numerona käytet-

tiin @HypotheticalLtd käyttäjän ID-numeroa. Vastauksena saatiin esimerkkikoodi 2:n mukainen lista kaikista julkaisuista, joissa @HypotheticalLtd-käyttäjä mainitaan.


```
{
  "data": [
    {
      "id": "1452274051855040524",
      "text": "@HypotheticalLtd I noticed a bug while using your won-
derful service."
    },
    {
      "id": "1440592110063738889",
      "text": "@HypotheticalLtd your otherwise wonderful hypothetical
service is unfortunately down at the moment."
    },
    {
      "id": "1435262063971082246",
      "text": "@HypotheticalLtd Your hypothetical service is down."
    }
  ],
  "meta": {
    "oldest_id": "1435262063971082246",
    "newest_id": "1452274051855040524",
    "result_count": 6
  }
}
```


Esimerkkikoodi 2. Twitter-rajapinnan vastaus julkaisujen hakuun käyttäjämaininnan perusteella.

Ensimmäisen version kysely toimi hyvin, mutta vastauksesta puuttui tiketin kuvauksen luomiseen tarvittavia tietoja, kuten julkaisun kirjoittaneen käyttäjän käyttäjänimi sekä julkaisun luonnin aikaleima. Näitä varten piti kyselyyn liittää lisämääreitä, kuten kuten created_at ja username. Twitter-rajapinta ei kuitenkaan palauta julkaisun tietojen mukana suoraan kirjoittajan käyttäjänimeä vaan pelkästään käyttäjän ID-numeron. Käyttäjänimen ja muut tiedot Twitter-rajapinta palauttaa omana julkaisusta erillisenä kenttänään, joten myös käyttäjän ID-numeron haku oli lisättävä lisämääreeksi kyselyyn. Lisämääreet lisättiin HTTPS GET -kutsun URL-parametreiksi, joten kyselyt tuli johtaa uuteen osoitteeseen: https://api.twitter.com/2/users/:id/mentions?expansions=author_id&tweet.fields=created_at&user.fields=username. Uusi kysely palautti esimerkkikoodin 3 mukaisen palautuksen:

```
{
  "data": [
    {
      "id": "1452274051855040524",
      "created_at": "2021-10-24T14:01:30.000Z",
      "text": "@HypotheticalLtd I noticed a bug while using your won-
derful service.",
      "author_id": "1435260438929936386"
    },
    {
      "id": "1440592110063738889",
      "created_at": "2021-09-22T08:21:38.000Z",
      "text": "@HypotheticalLtd your otherwise wonderful hypothetical
service is unfortunately down at the moment.",
      "author_id": "1435260438929936386"
    },
    {
      "id": "1435262063971082246",
      "created_at": "2021-09-07T15:21:56.000Z",
      "text": "@HypotheticalLtd Your hypothetical service is down.",
      "author_id": "1435260438929936386"
    }
  ],
  "includes": {
    "users": [
      {
        "id": "1435260438929936386",
        "name": "Hypothetical Customer",
        "username": "NeedHypothetics"
      }
    ]
  },
  "meta": {
    "oldest_id": "1435262063971082246",
    "newest_id": "1452274051855040524",
    "result_count": 6
  }
}
```


Esimerkkikoodi 3. Twitter-rajapinnan vastaus julkaisujen hakuun käyttäjämaininnan ja lisämääreiden perusteella.

Lisämääreiden lisäyksen jälkeen tarvittavat tiedot palautuivat rajapinnasta. Kuten aiemmin mainittiin, käyttäjän tiedot palautuivat omana listanaan. Tämä monimutkaisti palautuksen käsittelyä, sillä käyttäjänimi olisi haettava erillisestä listasta ID-numeron perusteella. Vaikka tämä ei suurta datakäsittelyä vaadi, piti tiedon jäsentelylle keksiä jokin ratkaisu. Twitterin palauttama datamalli oli kuitenkin muuten tarpeeksi hyvä sovelluksen tarpeisiin nähden, joten kehityksen aikana todettiin, että oman datamallin luominen ei toisi tarpeeksi suurta lisähyötyä datamallin muuttamiseksi Twitter-rajapintaa kutsuvan Lambda-funktion sisällä. Käyttäjänimen hakeminen erillisestä listasta päätettiin jättää jäsennin-

funktion tehtäväksi, sillä tikettien luomista varten tarvittava datamalli pitäisi kuitenkin toteuttaa kyseisessä funktiossa, joten todettiin, että erillistä välimuotoa datamallista ei ole järkevää toteuttaa.

Koska Twitter-rajapinta ei tue käyttäjämainintojen hakua suoraan käyttäjänimellä vaan vaatii käyttäjän ID-numeron, toteutettiin Lambda-funktioon kaksi erillistä funktiota: toinen hakee käyttäjän ID-numeron käyttäjänimen perusteella ja toinen käyttää haettua ID-numeroa hakemaan tähän kohdistuvat käyttäjämaininnat. Erillinen ID-numeron haku toteutettiin helppokäyttöisyyden johdosta, sillä Twitteristä on hankala saada käyttäjän ID-numeroa selville ilman rajapintaa, ja työssä toteutettavan sovelluksen loppukäyttäjälle haluttiin mahdolliseksi antaa sovellukselle vain etsittävän käyttäjän nimi. Käyttäjänimi tallennettiin helpon vaihtamisen mahdollistamiseksi AWS Lambdan ympäristömuuttajaksi. ID-numeron hakeva toiminnallisuus eristettiin omaksi getUserId()-funktiokseen (esimerkkikoodi 4), ja itse käyttäjämaininnat hakeva toiminnallisuus toteutettiin Lambda-funktion pääfunktioon. AWS Lambda -funktiot käyttävät pääfunktionaan ”handler”-nimistä funktiota, joka vastaanottaa Lambdan käynnistävän tapahtuman sisällön [28]. Handler-funktio toteutettiin hakemaan ensimmäiseksi getUserId()-funktiolta käyttäjän ID-numero, muodostamaan tuntia aikaisemmin oleva aikaleima ja käyttämään näitä tietoja käyttäjämainintojen hakuun tarvittavan rajapinnan URL-polun muodostamiseen (esimerkkikoodi 5). Tämän jälkeen handler-funktio toteuttaa yksinkertaisen HTTPS GET -kyselyn, ja palauttaa kyselyyn saadun vastauksen JSON-oliona Lambda-funktiota kutsuvalle taholle.

```
const optionsUserId = {
     hostname: 'api.twitter.com',
     port: 443,
     path: '/2/users/by/username/' + process.env.TwitterHandle,
     method: 'GET',
     headers: {
          'Authorization': 'Bearer ' + process.env.Authorization,
     }
};

function getUserId() {
     return new Promise((resolve, reject) => {
          https.request(optionsUserId, res => {
               let data = '';
               res.on('data', chunk => {
                    data += chunk;
               });

               res.on('end', () => {
                    resolve(JSON.parse(data));
               });
          }).on('error', (e) => {
               reject(Error(e));
          }).end();
     });
}
```

Esimerkkikoodi 4. Twitter-käyttäjän ID-numeron haku käyttäjänimen perusteella.

```
const tweets = await getUserId().then(userData => {
          let time = new Date();
          time.setHours(time.getHours() - process.env.TimeRange);
          optionsMentions['path'] = `/2/us-
ers/${userData['data']['id']}/mentions?expansions=au-
thor_id&tweet.fields=cre-
ated_at&user.fields=username&start_time=${time.toISOString()}`;
```

Esimerkkikoodi 5. Koodipätkä Twitter-rajapintaa kutsuvan Lambda-funktion handler-funktiosta, joka hakee etsittävän käyttäjän ID-numeron toiselta funktiolta, muodostaa aikaleiman etsittävän aikajakson alkuun ja muodostaa näiden perusteella Twitter-rajapinnan URL-polun käyttäjämainintahakua varten.

4.3 Jira-rajapinnan käyttöönotto

Jira-rajapinnan käyttäminen vaatii Jira-käyttäjätunnuksen perusteella luotua rajapinta-avainta. Avaimen generoiminen tapahtui suoraan Jiran käyttöliittymältä osoitteesta https://id.atlassian.com/manage-profile/security/api-tokens. Rajapinta-avaimen luomisen jälkeen tuli muodostaa autentikaatioavain käyttäen Jira-tilin sähköpostia sekä rajapinta-avainta. Autentikaatioavain oli Base64-muotoon

käännetty teksti, joka oli formaatissa <sähköposti>:<rajapinta-avain>. Autentikaatioavain sisällytettiin rajapintakutsujen Authorization-otsakkeeseen.

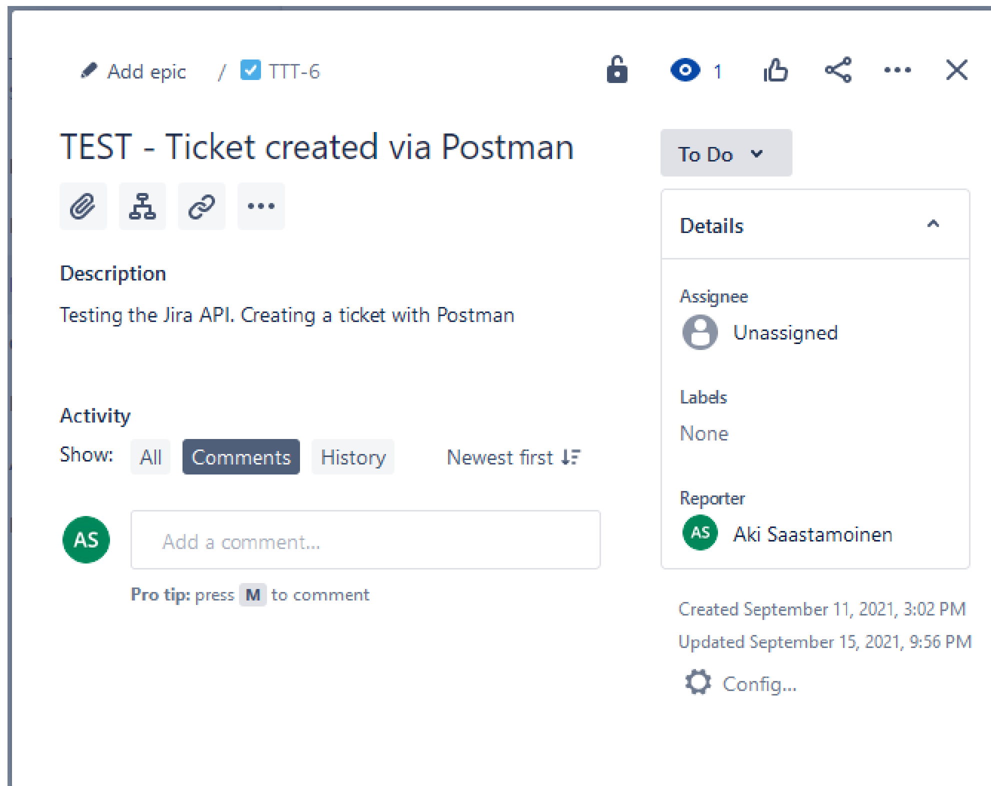
Avainten luomisen jälkeen alettiin rajapintaa testata Postmanin avulla. Jiran rajapinnan tietomalli oli helppokäyttöinen, sillä koko datamalli oli vain yksinkertainen kokoelma kenttiä ilman monimutkaisempaa rakenteellista hierarkiaa kahta tietuetta lukuun ottamatta, jotka olivat omia oliorakenteitaan. Nämäkin oliorakenteet olivat kuitenkin vain yhden kentän sisältäviä olioita, joten nekään eivät monimutkaistaneet koko datamallin rakennetta merkittävästi. Ensimmäisiä Postman-testejä varten luotiin yksinkertainen POST-rajapintakutsu, jonka hyötykuormaan kirjoitettiin esimerkkikoodi 6:n mukainen JSON-olio. Olio sisälsi yhden pääolion ”fields”, joka sisälsi tiedot projektista (project), yhteenvedosta (summary; puhutaan suomeksi usein tiketin otsikosta), selitteestä (description) sekä tikettilajista (issuetype). Projekti- ja tikettilajitiedot olivat itsessään myös olioita, joilla oli kentät projektin avaimelle (key) ja tikettilajin nimelle (name).

```
{
        fields: {
            project: {
                key: “TTT”
            },
            summary: “TEST - Ticket created via Postman”,
            description: “Testing the Jira API. Creating a ticket with 
Postman”,
            issuetype: {
                name: "Task"
            }
        }
}
```

Esimerkkikoodi 6. Ensimmäisen Postman-testin Jira-rajapintaan lähetetty hyötykuorma

Ensimmäinen testikutsu muodosti kuvan 5 mukaisen tiketin Jiraan. Kaikki kutsun mukana annetut tiedot kirjautuivat tiketille oikein, ja tiketin luojaksi kirjautui käyttäjä, jonka rajapinta-avainta käytettiin. Onnistuneen testin jälkeen suunniteltiin, mitä Twitter-julkaisuista saatavaa dataa tiketille haluttaisiin kirjata sekä missä muodossa ne haluttaisiin näytettävän. Haluttiin, että tiketteihin sisällytetään aiemmin lähetettyjen tietojen lisäksi nimike (label), joka kertoisi tiketin ole-

van luotu sovelluksen toimesta, sekä vaihtaa tikettilaji tehtävästä (task) virheeksi (bug). Suunnitelman mukaista lähetystä ei kuitenkaan ollut tässä vaiheessa tarpeellista erikseen testata, vaan oli aika alkaa toteuttaa AWS Lambda -funktiota, joka sovelluksen kokonaisuudessa hoitaisi tikettien luomisen rajapinnan välityksellä.



Kuva 5. Ensimmäisestä rajapintatestistä muodostunut tiketti Jira-palvelussa.

Kuten Twitter-rajapintaa kutsuvan Lambdan kanssa toimittiin, myös tähän Lambda-funktioon kirjoitettiin kaksi JavaScript-funktiota, joista toinen muodosti Jira-rajapinnan datamallin mukaisen hyötykuorman, ja toinen käsitteli HTTPS POST -kutsun. POST-kutsua käsitteleväksi funktioksi käytettiin handler-funktiota, sillä erillisen funktion kirjoittamiselle ei ollut tarvetta, ja se olisi vain lisännyt tarvittavaa käsittelyä ja monimutkaisuutta funktion rakenteeseen. Tiketin luomista varten tarvittavan hyötykuorman muodostavan funktion toteutuksessa pystyttiin hyödyntämään JavaScriptin Template Literals -menetelmän tarjoamaa tapaa interpoloida muuttujien sisältöä tekstiin. Tämä oli hyödyllistä, sillä sen avulla pystyttiin teksti kirjoittamaan normaalisti kokonaisina lauseina, joihin

muuttujien tiedot sisällytettiin käyttäen dollarimerkki-operaattoria (esimerkki-koodi 7).

```
function generateTicket(match) {
    return {
        fields: {
            project: {
                key: process.env.ProjectKey
            },
            summary: `Tweet containing keyword "${match.matchedWord}"
by ${match.username}`,
            labels:['Tweet2Ticket'],
            description: `Tweet by ${match.author} (${match.username})
at ${match.timestamp}\n\n${match.text}\n\n${match.url}`,
            issuetype: {
                name: "Bug"
            }
        }
    };
}
```

Esimerkkikoodi 7. Tiketin luontia varten tarvittavan hyötykuorman muodosta funktio. Summary ja description -kentissä on hyödynnetty JavaScriptin Template literals -menetelmää.

4.4 Jäsentimen toteutus

Sovelluksen päätoiminnallisuus, eli Twitter-julkaisujen sisällön tulkitseminen ja tikettien luomiseen tarvittavan datan muodostaminen toteutettiin kolmanteen Lambda-funktioon. Kutsuakseen Twitter- ja Jira-rajapintoja käyttäviä Lambda-funktioita piti funktion hyödyntää AWS:n ohjelmistokehityspakettia. Koska Lambda-funktio toteutettiin suoraan AWS Lamdba -palvelun sisäisessä ohjelmistoympäristössä, pystyttiin ohjelmistokehityspaketti tuomaan suoraan funktiolle käyttöön ilman, että sitä piti erikseen paketoida ohjelmistokoodin mukaan. Lambda-funktioiden kutsumista varten piti luoda uusi Lambda-olio, jolle annettiin konstruktorissa tieto AWS-ympäristön maantieteellinen sijainti. Olion luomisen jälkeen sen kautta pystyttiin kutsumaan muita maantieteellisen sijainnin sisällä olevia Lambda-funktioita nimellä. Kutsu tapahtui lambda-olion invoke()-funktiolla. Funktion konstruktoriin annettiin parametriksi olio, joka piti sisällään tiedon kutsuttavan Lambdan nimestä. Jira-rajapintaa käyttävän Lambdan kutsussa määriteltiin myös kutsutyyppi (Invocation type) ja funktiolle annettava hyötykuorma. Kutsutyypillä määritellään, onko kyseessä vastausta odottava

kutsu (RequestResponse; oletustyyppi), suorituksen laukaiseva tapahtuma (Event) vai pelkästään testikutsu (DryRun). Jäsentimen toiminnan kannalta ei ollut tarpeellista tietää, mitä Jira-rajapinta vastaa tiketin luontiin, joten kutsutyypiksi valittiin Event. Koodin rakenteen selkeyttämiseksi molempien Lamdbojen kutsut eristettiin omiksi funktioikseen (esimerkkikoodi 8).

```
const AWS = require('aws-sdk');
const lambda = new AWS.Lambda({ region: 'us-east-2' });

const invokeTwitterAPIHandler = async () => {
    return lambda.invoke({
        FunctionName: 'TwitterAPIHandler'
    }).promise();
};

const invokeTicketAPIHandler = async (payload) => {
    return lambda.invoke({
        FunctionName: 'JiraAPIHandler',
        InvocationType: 'Event',
        Payload: payload
    }).promise();
};
```

Esimerkkikoodi 8. AWS-ohjelmistokehityspaketin tuonti, Lambda-olion luonti sekä Lambda-kutsujen funktiot.

Jäsentimen varsinainen toimintalogiikka kirjoitettiin filterTweets()-nimiseen funktioon (esimerkkikoodi 9). Funktio otti parametrinä Twitter-rajapintaa kutsuvan Lambda-funktion palauttaman olion, ja suodatti pois julkaisut, joissa ei esiintynyt ennalta määriteltyjä avainsanoja. Avainsanoiksi määriteltiin down, broken ja bug. Suodattimen avainsanavertailu kävi julkaisun läpi sana kerrallaan, poisti sanoista erikoismerkit ja numerot sekä muunsi kaikki kirjaimet pieniksi. Jäljelle jääneille julkaisuille tehtiin datamallimuunnos. Uudessa datamallissa tiketin luomiseen tarvittavat tiedot olivat rakenteellisesti yksinkertaisemmin saatavilla kuin Twitter-rajapinnan datamallissa, ja se suunniteltiin sisältämään vain Jira-rajapintaa kutsuvan Lambdan tarvitsemat tiedot. Datamallimuunnoksen yhteydessä julkaisun kirjoittajan käyttäjänimi haettiin käyttäjän ID-numeron perusteella Twitter-rajapinnan palauttamasta erillisestä listasta, ja aikaleima muunnettiin ISO-standardista ihmisluettavampaan muotoon ja Suomen aikavyöhykkeeseen. Näiden lisäksi julkaisun URL muodostettiin käyttäjänimen ja julkaisun ID-numeron perusteella.

```
const keywords = ['down', 'broken', 'bug'];

function filterTweets(twitterData) {
    return twitterData.data.filter(tweet => tweet.text.split(' ')
                .some(word => {
                    let index = keywords.indexOf(word.replace(/[^a-zA-
Z]/g, '').toLowerCase());
                    if (index < 0) return false;
                    else return tweet['matchedWord'] = keywords[in-
dex];
                })
            ).map(tweet => {
                const user = twitterData.includes.users.find(user =>
user.id == tweet.author_id);
                const tweetTime = new Date(tweet.created_at);
                console.log(tweetTime);
                return {
                    author: user.name,
                    username: '@' + user.username,
                    text: tweet.text,
                    matchedWord: tweet.matchedWord,
                    timestamp: `${tweetTime.toLocaleDateString('fi-
FI', {timeZone: 'Europe/Helsinki'})} ${tweetTime.toLocaleTime-
String('fi-FI', {timeZone: 'Europe/Helsinki'})}`,
                    url: `https://twitter.com/${user.username}/sta-
tus/${tweet.id}`
                };
            });
}
```

Esimerkkikoodi 9. Avainsanalista sekä suodatin- ja datamuunnosfunktio.

Kuten aiemmissakin Lambda-funktioissa, loput toteutuksesta kirjoitettiin suoraan handler-funktioon. Funktio kutsui Twitter-rajapintaa kutsuvaa Lambda-funktiota ja jäsensi sen paluuarvona tulleen JSON-datan JavaScript-olioksi. Seuraavaksi varmistettiin, että olio sisälsi Twitter-julkaisuja, ja se annettiin parametrina filterTweets()-funktiolle. Funktiolta paluuarvona saadun listan oliot annettiin yksitellen Jira-rajapintaa kutsuvan Lambda-funktion kutsun hyötykuormaksi (esimerkkikoodi 10).

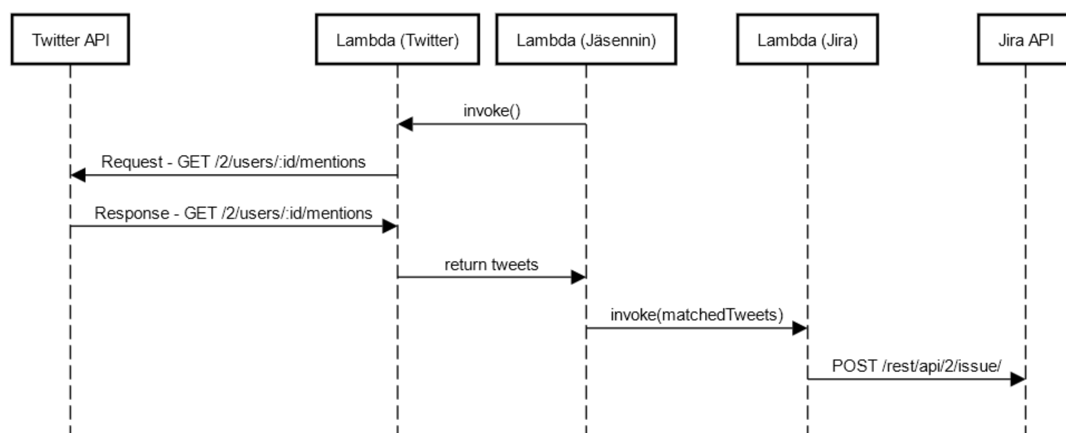
```
exports.handler = () => {
    invokeTwitterAPIHandler().then(res => {
        const payload = JSON.parse(res['Payload']);
        console.log(payload);
        if (payload.meta.result_count) {
            for (let match of filterTweets(payload)) {
                invokeTicketAPIHandler(JSON.stringify(match));
            }
        }
    });
};
```

Esimerkkikoodi 10. Jäsennin Lambda-funktion handler-funktio.

5 Yhteenveto

Projektin tarkoituksena oli tutkia, onko mahdollista luoda lyhyellä ajan jaksolla kustannustehokas ja helposti käyttöönotettava sovellus, joka etsii Twitteristä vikailmoituksina toimiva julkaisuja ja luo niistä Jira-tikettejä. Tarkoituksena oli myös arvioida kyseisen sovelluksen tarpeellisuutta.

Suunnitelman mukaisesti sovellus toteutettiin kolmessa osassa. Suorituksen aloitti pääfunktiona toimiva Jäsennin-funktio, joka käynnistettiin EventBridge-palvelun ajastetulla tapahtumalla tunnin välein. Jäsennin-funktio kutsui Twitter-rajapintaa kutsuvaa funktiota, joka sai haettua julkaisuja onnistuneesti. Jäsentimen suodatin ja datamallimuunnos saatiin toimimaan ongelmitta, ja se välitti uuden datamallin mukaisia olioita Jira-rajapintaa kutsuvalle funktiolle, joka sai tiketit luotua Jiraan (kuva 6).



Kuva 6. Sovelluksen sekvenssikaavio.

Twitter- ja Jira-rajapintojen sekä AWS-ympäristön käyttöönotto ja konfigurointi olivat hyvän dokumentaation johdosta selkeitä ja suoraviivaisia prosesseja. Näistä syistä sovelluksen toteutuksen yhteydessä ei kohdattu väärinkirjoituksista johtuvia virheitä suurempia ongelmia. Yksinkertaisuutensa ansiosta sovelluksen suoritus oli nopeaa, joten laskutettavaa laskenta-aikaa ei juurikaan kertynyt, eikä kolmen funktion suorittaminen tunnin välein riko miljoonan ilmaisen kuukausittaisen suorituksen rajaa. Tämän johdosta sovellus on AWS:n toistaiseksi voimassa olevan hinnoittelun mukaisesti ilmainen.

Työn tarpeellisuutta pohdittiin eri kohderyhmien näkökulmasta. Isot yritykset, joilla on todennäköisesti kattavat monitorointiohjelmistot käytössä tuotannon seurantaan, eivät todennäköisesti hyötyisi sovelluksen käytöstä. Lisäongelmaksi voisi muodostua myös isojen yritysten suuret käyttäjämäärät, jotka todennäköisesti lisäävät yrityksen mainintoja sosiaalisessa mediassa. Tämä voi aiheuttaa suuria prosessointiaikoja julkaisujen hakuun ja suodattamiseen. Yksittäiset kehittäjät tai pienet yritykset, joiden palveluiden käyttäjäkunnat ovat aktiivisia sosiaalisessa mediassa, voivat saada merkittävääkin lisähyötyä sovelluksen käytöstä. Tällaisilla kehittäjillä ja yrityksillä ei välttämättä ole resursseja tarkkailla palveluittensa tuotannon tilaa, joten nopeasti löydettävät käyttäjäilmoitukset voivat olla erittäin hyödyllisiä. Palveluiden kehitystiimeillä ei myöskään välttämättä ole riittävästi testausresursseja, joten vikaetsinnän joukkoistaminen käyttäjäkunnalle voi olla tehokas tapa löytää piileviä vikoja sovelluksessa. Peliteollisuudessa voi sosiaalisen median vikailmoituksilla olla myös lisäarvoa, sillä julkaisuihin usein liitetään videomateriaalia vikatilanteesta, joka helpottaa vianetsintää.

Koska sovelluksen tarkoitus oli pääsääntöisesti sen kannattavuuden tutkiminen, ei toteutuksessa otettu virhetilanteiden käsittelyä juurikaan huomioon. Jatkokehityksessä olisi kannattavaa ottaa käyttöön jonkinlainen tietokanta, jonne voitaisiin tallentaa Twitter-julkaisuja, joiden tiketöinti epäonnistuu. Täten edellisen tunnin julkaisujen haku ei aiheuttaisi puuttuvia julkaisuja. Myös julkaisujen haun yhteydessä tapahtuvia virhetilanteita voitaisiin kompensoida ottamalla talteen

epäonnistuneen haun aikaleima, jotta seuraavaa hakua voidaan laajentaa kattamaan epäonnistuneen haun aikajakso. Koska avainsanapohjainen suodatus ei ole välttämättä tarpeeksi tarkka, eikä ota huomioon samasta aiheesta tehtyjä useampia julkaisuja, voisi jatkossa tutkia koneoppimisen hyödyntämistä julkaisujen sisällön analyysissä ja tämän mahdollista vaikutusta kustannustehokkuuteen. Oman datamallin kehittäminen sosiaalisen median julkaisujen välittämiseksi jäsentelijälle, mahdollistaisi saman funktion käyttämisen muidenkin sosiaalisten median palveluiden julkaisujen tiketöimiseksi.

Lähteet

1 Kirby, James. 2020. How Cloud Computing Differs from Traditional IT Infrastructure. Verkkoaineisto. Micropro. <https://micropro.com/blog/cloud-computing-vs-traditional-it-infrastructure/>. 27.11.2020. Luettu 18.10.2021.

2 FaaS (Function-as-a-Service). 2019. Verkkoaineisto. IBM Cloud Learn Hub. <https://www.ibm.com/cloud/learn/faas>. 30.7.2019. Luettu 18.10.2021.

3 Shilkov, Mikhail. 2021. Comparison of Cold Starts in Serverless Functions across AWS, Azure, and GCP. Verkkoaineisto. Mikhail Shilkov. <https://mikhail.io/serverless/coldstarts/big3/>. 5.1.2021. Luettu 1.11.2021.

4 Lambda function scaling. (n.d.). Verkkoaineisto. Amazon. <https://docs.aws.amazon.com/lambda/latest/dg/invocation-scaling.html>. Luettu 4.11.2021.

5 AWS Lambda enables functions that can run up to 15 minutes. 2018. Verkkoaineisto. Amazon. <https://aws.amazon.com/about-aws/whats-new/2018/10/aws-lambda-supports-functions-that-can-run-up-to-15-minutes/>. 10.10.2018. Luettu 4.11.2021.

6 AWS Lambda Pricing . (n.d.). Verkkoaineisto. Amazon. <https://aws.amazon.com/lambda/pricing/>. Luettu 10.9.2021.

7 Authentication. (n.d.). Verkkoaineisto. Twitter Developer Platform. <https://developer.twitter.com/en/docs/authentication/overview>. Luettu 10.9.2021.

8 Basic auth for REST APIs. (n.d.). Verkkoaineisto. Atlassian Developer. <https://developer.atlassian.com/cloud/jira/platform/basic-auth-for-rest-apis/>. Luettu 10.9.2021.

9 Iqbal, Mansoor. 2021. Twitter Revenue and Usage Statistics (2021). Verkkoaineisto. Business of Apps. <https://www.businessofapps.com/data/twitter-statistics/>. 5.7.2021. Luettu 13.10.2021.

10 What is IaaS? (n.d.). Verkkoaineisto. Azure. < ttps://azure.microsoft.com/en-us/overview/what-is-iaas/>. Luettu 4.11.2021.

11 What is PaaS? (n.d.). Verkkoaineisto. Azure. < ttps://azure.microsoft.com/en-us/overview/what-is-paas/>. Luettu 4.11.2021.

12 What is SaaS? (n.d.). Verkkoaineisto. Azure. < ttps://azure.microsoft.com/en-us/overview/what-is-saas/>. Luettu 4.11.2021.

13 Panettieri, Joe. 2020. Cloud Market Share 2020: Amazon AWS, Microsoft Azure, Google, IBM. Verkkoaineisto. ChannelE2E. <https://www.channele2e.com/channel-partners/csps/cloud-market-share-2020-amazon-aws-microsoft-azure-google-ibm/>. 3.11.2020. Luettu 1.11.2021.

14 Regions and Availability Zones. (n.d.). Verkkoaineisto. Amazon. <https://aws.amazon.com/about-aws/global-infrastructure/regions_az/>. Luettu 3.11.2021.

15 Amazon CloudFront Key Features. (n.d.). Verkkoaineisto. Amazon. <https://aws.amazon.com/cloudfront/features/?p=ugi&l=na&whats-new-cloudfront.sort-by=item.additionalFields.postDateTime&whats-new-cloudfront.sort-order=desc>. Luettu 3.11.2021.

16 Lambda Runtimes. (n.d.). Verkkoaineisto. Amazon. <https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html>. Luettu 3.11.2021.

17 AWS Lambda The Ultimate Guide. (n.d.). Verkkoaineisto. Serverless. <https://www.serverless.com/aws-lambda>. Luettu 4.11.2021.

18 AWS Identity and Access Management (IAM). (n.d.). Verkkoaineisto. Amazon. <https://aws.amazon.com/iam/>. Luettu 2.11.2021.

19 Lord, Nate. 2020. What is the Principle of Least Privilege (POLP)? A Best Practice for Information Security and Compliance. Verkkoaineisto. Digital Guardian. <https://digitalguardian.com/blog/what-principle-least-privilege-polp-best-practice-information-security-and-compliance>. 1.12.2020. Luettu 4.11.2021.

20 Amazon EventBridge. (n.d.). Verkkoaineisto. Amazon. <https://aws.amazon.com/eventbridge/>. Luettu 2.11.2021.

21 Amazon CloudWatch. (n.d.). Verkkoaineisto. Amazon. <https://aws.amazon.com/cloudwatch/>. Luettu 2.11.2021.

22 Amazon API Gateway. (n.d.). Verkkoaineisto. Amazon. <https://aws.amazon.com/api-gateway/>. Luettu 2.11.2021.

23 Cui, Yan. 2020. Are Lambda-to-Lambda calls really so bad? Verkkoaineisto. theburningmonk.com. <https://theburningmonk.com/2020/07/are-lambda-to-lambda-calls-really-so-bad/>. 12.7.2020. Luettu 15.9.2021.

24 Using Collections in Postman. (n.d.). Verkkoaineisto. Thinkster. <https://thinkster.io/tutorials/testing-backend-apis-with-postman/using-collections-in-postman>. Luettu 1.11.2021.

25 Krill, Paul. 2016. Microsoft-backed Language Server Protocol strives for language, tools interoperability. Verkkoaineisto. InfoWorld. <https://www.infoworld.com/article/3088698/microsoft-backed-langauge-server-protocol-strives-for-language-tools-interoperability.html>. 27.6.2016. Luettu 4.11.2021.

26 AWS Free Tier. (n.d.). Verkkoaineisto. Amazon. <https://aws.amazon.com/free/free/?all-free-tier.sort-by=item.additionalFields.SortRank&all-free-tier.sort-order=asc&awsf.Free%20Tier%20Types=*all&awsf.Free%20Tier%20Categories=*all&c=nhp&z=2&awswt=203b>. Luettu 15.10.2021.

27 #UseCases. (n.d.). Verkkoaineisto. Twitter Developer Portal. < https://developer.twitter.com/en/portal/petition/use-case>. Luettu 4.9.2021.

28 AWS Lambda function handler in Node.js. (n.d.). Verkkoaineisto. Amazon. <https://docs.aws.amazon.com/lambda/latest/dg/nodejs-handler.html>. Luettu 10.9.2021.

Twitter-rajapintaa kutsuvan Lambda-funktion ohjelmakoodi

```
const https = require('https');

const optionsUserId = {
	hostname: 'api.twitter.com',
	port: 443,
	path: '/2/users/by/username/' + process.env.TwitterHandle,
	method: 'GET',
	headers: {
		'Authorization': 'Bearer ' + process.env.Authorization,
	}
};

const optionsMentions = {
	hostname: 'api.twitter.com',
	port: 443,
	path: '',
	method: 'GET',
	headers: {
		'Authorization': 'Bearer ' + process.env.Authorization,
	}
};

function getUserId() {
	return new Promise((resolve, reject) => {
		https.request(optionsUserId, res => {
			let data = '';
			res.on('data', chunk => {
				data += chunk;
			});

			res.on('end', () => {
				resolve(JSON.parse(data));
			});
		}).on('error', (e) => {
			reject(Error(e));
		}).end();
	});
}

exports.handler = async () => {
	const tweets = await getUserId().then(userData => {
		let time = new Date();
		time.setHours(time.getHours() - process.env.TimeRange);
		optionsMentions['path'] = `/2/users/${userData['data']['id']}/menti-
ons?expansions=author_id&tweet.fields=created_at&user.fields=user-
name&start_time=${time.toISOString()}`;
		return new Promise((resolve, reject) => {
			const req = https.request(optionsMentions, res => {
				let response = '';
				res.on('data', (chunk) => {
					response += chunk;
				});
				res.on('end', () => {
					resolve(response);
				});
			});
			req.on('error', (e) => {
				reject(Error(e));
			});
			req.end();
		});
	});

	return JSON.parse(tweets);
```

Jira-rajapintaa kutsuvan Lambda-funktion ohjelmakoodi

```
const https = require('https');
const options = {
      hostname: 'akisaa.atlassian.net',
      port: 443,
      path: '/rest/api/2/issue/',
      method: 'POST',
      headers: {
            'Authorization': 'Basic ' + process.env.Authorization,
            'Content-Type': 'application/json',
      }
};


function generateTicket(match) {
    return {
        fields: {
            project: {
                key: process.env.ProjectKey
            },
            summary: `Tweet containing keyword "${match.matchedWord}" by
${match.username}`,
            labels:['Tweet2Ticket'],
            description: `Tweet by ${match.author} (${match.username}) at
${match.timestamp}\n\n${match.text}\n\n${match.url}`,
            issuetype: {
                name: "Bug"
            }
        }
    };
}

exports.handler = event => {
      return new Promise((resolve, reject) => {
            const req = https.request(options, (res) => {

                  let data = '';
                  res.on('data', (chunk) => {
                        console.log(`BODY: ${chunk}`);
                        data += chunk;
                  });

                  res.on('end', () => {
                        resolve(JSON.parse(data));
                  });
            });

            req.on('error', (e) => {
                  console.error(`problem with request: ${e.message}`);
            });
            console.log(JSON.stringify(generateTicket(event)));
            req.write(JSON.stringify(generateTicket(event)));

            req.end()
      })
}
```

Jäsennin Lambda-funktion ohjelmakoodi

```
const AWS = require('aws-sdk');
const lambda = new AWS.Lambda({ region: 'us-east-2' });

const keywords = ['down', 'broken', 'bug'];

const invokeTwitterAPIHandler = async () => {
    return lambda.invoke({
        FunctionName: 'TwitterAPIHandler'
    }).promise();
};

const invokeTicketAPIHandler = async (payload) => {
    return lambda.invoke({
        FunctionName: 'JiraAPIHandler',
        InvocationType: 'Event',
        Payload: payload
    }).promise();
};

exports.handler = () => {
    invokeTwitterAPIHandler().then(res => {
        const payload = JSON.parse(res['Payload']);
        console.log(payload);
        if (payload.meta.result_count) {
            for (let match of filterTweets(payload)) {
                invokeTicketAPIHandler(JSON.stringify(match));
            }
        }
    });
};

function filterTweets(twitterData) {
    return twitterData.data.filter(tweet => tweet.text.split(' ')
                .some(word => {
                    let index = keywords.indexOf(word.replace(/[^a-zA-Z]/g,
'').toLowerCase());
                    if (index < 0) return false;
                    else return tweet['matchedWord'] = keywords[index];
                })
            ).map(tweet => {
                const user = twitterData.includes.users.find(user => user.id
== tweet.author_id);
                const tweetTime = new Date(tweet.created_at);
                console.log(tweetTime);
                return {
                    author: user.name,
                    username: '@' + user.username,
                    text: tweet.text,
                    matchedWord: tweet.matchedWord,
                    timestamp: `${tweetTime.toLocaleDateString('fi-FI', {time-
Zone: 'Europe/Helsinki'})} ${tweetTime.toLocaleTimeString('fi-FI', {timeZone:
'Europe/Helsinki'})}`,
                    url: `https://twitter.com/${user.username}/sta-
tus/${tweet.id}`
                };
            });
}
```