Veikka Honkanen

# CLOUDFLARE DNS -MIKROPALVELUN HALLINNOINTI KÄYTTÄEN PALVELI-METONTA INFRASTRUKTUURIA

Opinnäytetyö

Liiketalouden ammattikorkeakoulututkinto

Tietojenkäsittelyn koulutus

2021

Kaakkois-Suomen ammattikorkeakoulu

| | |
|---|---|
| Tutkintonimike | Tradenomi (AMK) |
| Tekijä/Tekijät | Veikka Honkanen |
| Työn nimi | Cloudflare DNS -mikropalvelun hallinnointi käyttäen palvelimetonta infrastruktuuria |
| Toimeksiantaja | Metatavu Oy |
| Vuosi | 2021 |
| Sivut | 41 sivua |
| Työn ohjaaja | Jukka Selin |

## TIIVISTELMÄ


Toimeksiantajayritys Metatavu Oy haluaa toteuttaa hallintaliittymän, jonka kautta pystyttäisiin hallinnoimaan useita eri mikropalveluita. Hallintaliittymän keskipisteenä on tapahtumaväylä, jonka on tarkoitus välittää tiedot käyttäjältä mikropalveluihin ja takaisin.

Tämän työn ensimmäisenä tavoitteena on selvittää Apache Kafka -tapahtumaväylän sekä jonkin toisen tapahtumaväylän konsepti ja toiminnallisuus sekä suorittaa näiden vertailu. Työn toisena tavoitteena on ottaa selvää nimenomaisesti Cloudflare Workersista sekä AWS Lambdasta ja vertailla näiden ominaisuuksia ja hinnoittelua keskenään. Ensimmäisen selvityksen tavoitteena on löytää ominaisuuksiltaan paras tapahtumaväylä hallintaliittymää varten ja toisen selvityksen tavoite on löytää parempi FaaS-infrastruktuuri pilvifunktion käyttämiseksi hallintaliittymässä. Lopuksi työssä luonnostellaan teoreettinen datan kulku tuotantokäytössä olevasta tapahtumaväylästä Cloudflaren palvelimelle saakka.

Hallinnointi itsessään käyttää eräänlaista palvelimetonta rakennetta. Työn käytännön toteutuksen aiheena on suunnitella ja rakentaa valmiit Lambda- ja Workers-pilvifunktiot Cloudflare DNS -mikropalvelun hallinnointia varten sekä todistaa onnistunut datan kulku funktionaalisella testauksella.

Ensimmäisessä selvityksessä saatiin tulokseksi, että Kafka on hallintaliittymään soveltuvin vaihtoehto lokihistoriansa ja laajennettavuutensa takia. Paikallisessa testiajossa todettiin Kafkan sisältävän ominaisuuksia, joilla voitiin säädellä mm. palvelinten määrää ja replikointia. Lokit välittyivät julkaisijalta tilaajalle suunnitellusti. Lisäksi saatiin selville, että Workers on Lambdaa nopeampi, kustannustehokkaampi ja kevyempi käyttää. Lambdan mahdollisuus piilottaa se AWS-verkon sisäpuolelle on kuitenkin tärkeä ominaisuus turvallisuuden kannalta, eikä Workersilla ole tarjota samankaltaista ominaisuutta, minkä vuoksi voitiin todeta Lambdan soveltuvan paremmin hallintaliittymän käyttöön.

Työssä myös pohdittiin erilaisia vaihtoehtoisia tapoja käyttöönottaa tuleva hallintaliittymä tuotantoympäristöön. Näistä voitiin todeta parhaaksi vaihtoehdoksi AWS-verkon sisälle piilotettu Kubernetes-klusteri. Klusteri ei ainoastaan ole tämän hetken turvallisin ratkaisu, vaan sallii myös orkestroinnin hallintaliittymän monille eri palasille, joita aiotaan lisätä myöhemmin useita kappaleita.



**Asiasanat:** pilvipalvelut, ohjelmistokehitys, ohjelmistoarkkitehtuuri

South-Eastern Finland University of Applied Sciences

| | |
|---|---|
| Degree | Bachelor of Business Administration |
| Author (authors) | Veikka Honkanen |
| Thesis title | Managing Cloudflare DNS microservice using serverless infrastructure |
| Commissioned by | Metatavu Oy |
| Time | November 2021 |
| Pages | 41 pages |
| Supervisor | Jukka Selin |

## ABSTRACT


The commissioner company Metatavu Oy wanted to build a management system for managing several microservices. The middlemost part of the management system was going to be an event bus which was supposed to transfer data from the user to the microservices and back.

The first aim of the work was to find out about the concept and functionality of the Apache Kafka event bus and some other relevant event buses as well as compare these with each other. The objective was to determine the best possible event bus for the management system. The second aim of the work was to find information on Cloudflare Workers and AWS Lambda as well as to compare their attributes and pricing. The objective here was to determine which would be the better FaaS infrastructure for using the cloud function in the management system. Finally, a draft of the theorical flow of the data from the event bus to the Cloudflare server in production environment was being made.

The management itself is using a serverless structure. The topic of the practical implementation of the work was to design and build complete Lambda and Workers cloud functions for the management of the Cloudflare DNS microservice and to prove the successful flow of data by functional testing.

The result of the first objective was that Kafka was the most suitable option for the management interface due to its log history and extensibility. The local test run showed that Kafka contains properties that can be used to control number of servers and replication, for example. The logs were transmitted from the publisher to the subscriber as intended. The results from the second objective showed that Workers was faster, more cost-effective, and lighter to use than Lambda. However, Lambda's ability to hide itself within the AWS network was an important security feature whereas Workers did not provide any similar features. Therefore, it was found that Lambda was better suited for the management system.

The work also involved considering various ways of deploying the upcoming management system into production, of which a Kubernetes cluster hidden within the AWS network was found the best alternative. The cluster is not only the safest solution currently, but also allows for an orchestrated management for many different pieces that will extend the system in the future.

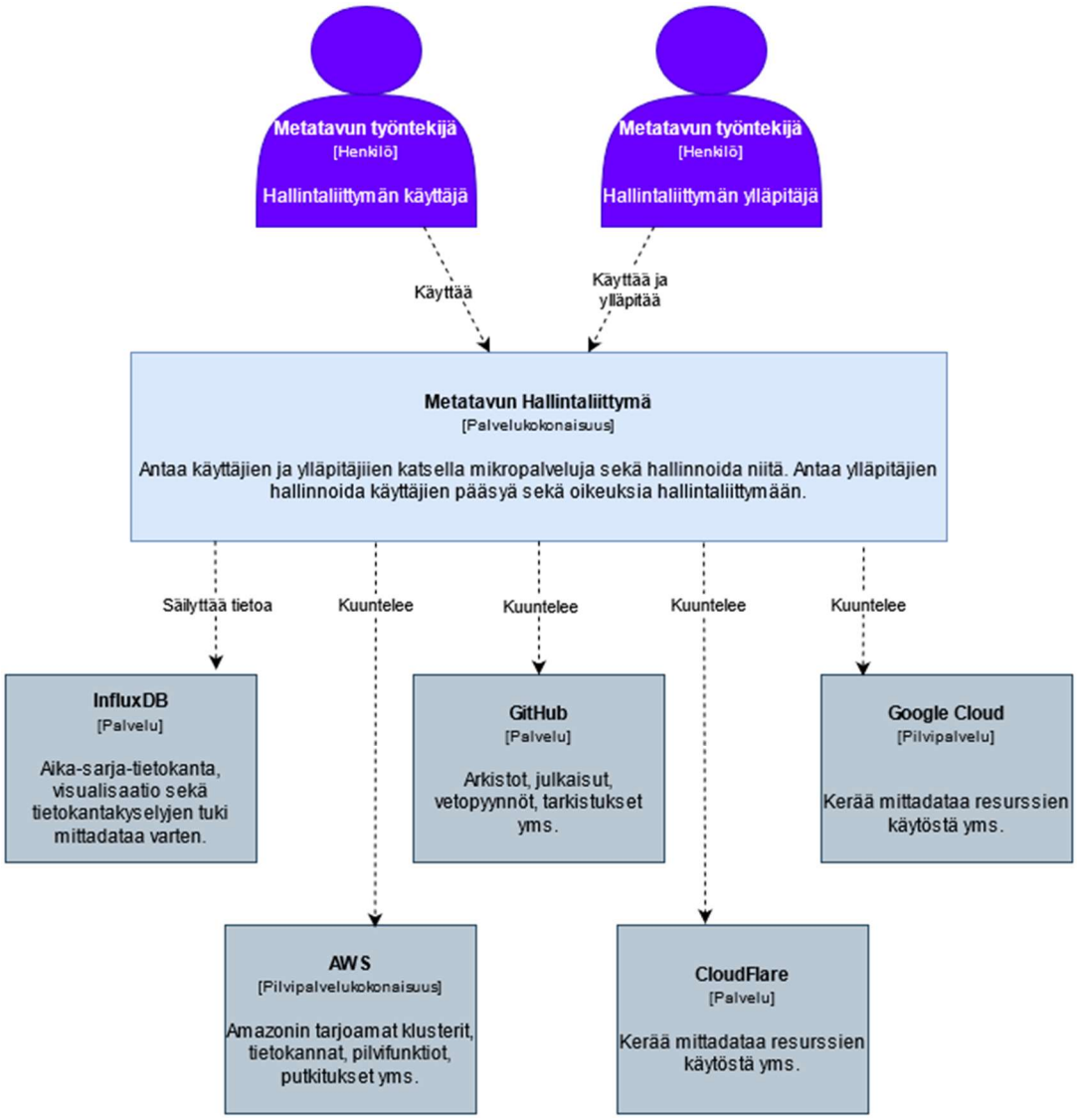


**Keywords:** cloud services, software development, software architecture

**SISÄLLYS**

# 1 JOHDANTO

Tutustuin harjoittelujakson aikana Metatavu Oy:llä pilviympäristöihin ja tämän tuomiin mahdollisuuksiin, kuten resurssien skaalattavuuteen sekä automaation toteuttamiseen funktioilla. Metatavu antoi mahdollisuuden laajentaa osaamista ja tietoutta kyseisellä osa-alueella: ohjelmistokehittäjänä tunnettu yritys tarvitsisi hallintaliittymän, joka niputtaa eri palveluiden alla tarvittavien resurssien käsittelyn (kuva 1). Hallinnointi olisi tarkoitus pystyä tekemään tulevaisuudessa sekä käyttöliittymästä että komentoriviltä.




Kuva 1. Hallintaliittymän systeemin konteksti, josta työssä rakennetaan Cloudflare-osuus.

Toimeksiannon kehittämisongelmina ovat valmiit pilvifunktiot Cloudflare DNS -hallinnointia varten sekä tutkimus tapahtumaväylän konseptista hallintaliitty-

mässä. Hallintaliittymän eri osat voisivat kommunikoida keskenään tapahtumaväylän välityksellä. Cloudflaren hallinnointi itsessään käyttäisi eräänlaista palvelimetonta rakennetta.

Ensimmäisenä tavoitteena on ottaa selvää yleisesti tapahtumaväylästä ja sen konseptista. Lisäksi tavoitteena on selvittää Apache Kafka -tapahtumaväylän sekä myös jonkin muun tapahtumaväylän toiminta (esim. VMware RabbitMQ). Työn toisena tavoitteena on ottaa tarkemmin selvää nimenomaisesti Cloudflare Workersista sekä AWS Lambdasta ja vertailla näiden ominaisuuksia sekä hinnoittelua keskenään. Tavoitteena on löytää parempi tapahtumaväylä hallintaliittymän keskipisteeksi sekä parempi FaaS-infrastruktuuri pilvifunktion käyttämiseksi.

Näiden lisäksi työn tavoitteena on soveltaa parhaiksi osoittautuneita ratkaisuja ja luonnostella teoreettinen datan kulku tuotantokäytössä olevasta tapahtumaväylästä Cloudflaren palvelimelle saakka. Käytännön toteutuksen tavoitteena työssä on suunnitella ja rakentaa onnistunut datan kulku pilvifunktion avulla Cloudflare DNS -mikropalveluun sekä osoittaa sen toimivuus funktionaalisella testauksella.

## 2 TYÖN KESKEISET TEKNIIKAT JA MENETELMÄT

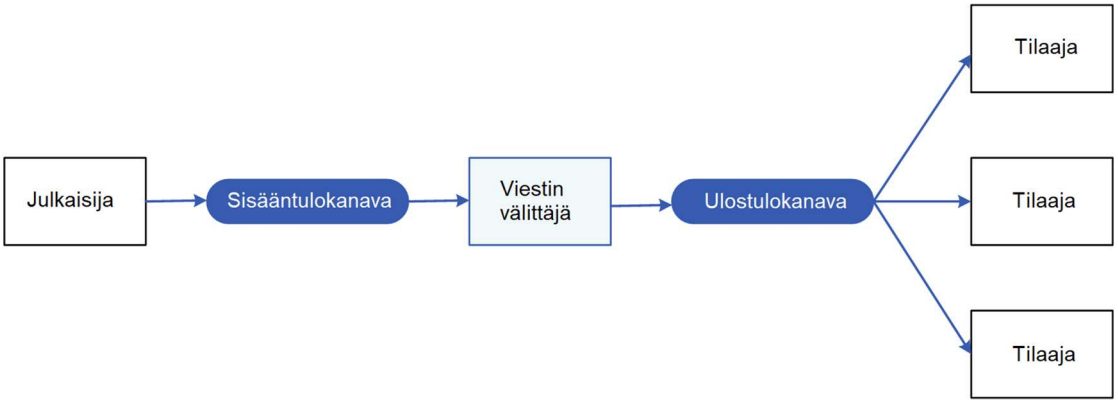
Ennen käytännön työn suunnittelua ja toteutusta pyritään hankkimaan käsitys ja kokonaisnäkemys työn ohjelmistoarkkitehtuurillisesta puolesta. Tämä on tarpeen, jotta päästäisiin sekä mahdollisimman tehokkaaseen että järkevään lopputulokseen. Erilaisten ratkaisuvaihtoehtojen tutkimisen jälkeen on päädytty tapahtumavetoiseen arkkitehtuuriin. Mitä enemmän perehdytään tapahtumavetoiseen arkkitehtuuriin ja sen erilaisiin metodeihin, sitä useammin vastaan tulee tiettyjä termejä ja käsitteitä, joihin ei välttämättä ole törmännyt opintojen aikana tai edes töissä IT-alalla. Tämän vuoksi niistä kerrotaan seuraavaksi vähän tarkemmin.

Tässä luvussa käydään läpi muutamia termejä ja käsitteitä, jotka ovat keskeisiä työn teknisen toteutuksen sekä teorian kannalta. Niiden ymmärtäminen edesauttaa työn sisällön omaksumista. Termejä ja käsitteitä ei esitellä kovin syvällisesti, mutta ne esitellään työn kannalta riittävässä laajuudessa.

## 2.1 EDA

EDA (Event-driven architecture, tapahtumavetoinen arkkitehtuuri) on ohjelmistoarkkitehtuuri sekä malli sovelluksen suunnittelulle (Red Hat 2019). Kyseisellä arkkitehtuurilla on kolme keskeistä osaa; tapahtuman kuluttajat (event consumers), jotka tilaavat tapahtumia, välikappaleet (middleware), jotka suodattavat tapahtumia ja työntävät ne tapahtuman kuluttajille sekä tapahtuman tuottajat (event producers), jotka julkaisevat tapahtumia välikappaleeseen (What is an Event-Driven Architecture? s.a.; Red Hat 2019). EDA on löysästi yhdistetty, eli tapahtuman tuottajat eivät ole tietoisia siitä, mitkä tapahtuman kuluttajat kuuntelevat julkaistua tapahtumaa (Bui 2020; Red Hat 2019). Tapahtuman tuottajat ja tapahtuman kuluttajat ovat toisistaan irrallisia osia (What is an Event-Driven Architecture? s.a.; Bui 2020; Gamma ym. 1994, 296).

Tyypilliset mallit EDA:lle ovat Pub/sub-malli ja tapahtumien suoratoistomalli (event streaming model). Tapahtumien suoratoistomallissa tapahtumat kirjoitetaan lokiin (Bui 2020; Red Hat 2019), josta tapahtuman kuluttajat käyvät lukemassa tiedon (Red Hat 2019). Pub/sub-mallissa puolestaan julkaisija työntää tapahtuman jonoon tai virtaan, josta kyseisen tapahtuman tilaajat noutavat tiedon (Google Cloud s.a.; Red Hat 2019). Kuten kuvassa 2 näkyy, julkaisija ja tilaaja käyttävät viestin välittäjän eri kanavia tiedon viemiseksi eteenpäin. Pub/sub-mallissa tapahtuman julkaisijan ei tarvitse tietää, mitkä osapuolet tilaavat tapahtumaa. Tämän lisäksi yksittäisellä tapahtumalla voi olla kuinka monta tilaajaa tahansa. (Gamma ym. 1994, 294.) Pub/sub-malli tunnetaan myös nimellä Observer (mts. 293).




Kuva 2. Pub/sub-mallin loogiset komponentit (käännetty lähteestä Microsoft 2018)

EDA:n vahvuuksia ovat palasten irtonaisuus (yhteensopivuus), skaalautuvuus, nopeus ja virheensietokyky (What is an Event-Driven Architecture? s.a.; Bui 2020). EDA:a voidaan myös laajentaa pilkkomalla se pienemmiksi askeleiksi ja lisäämällä niihin virheenkäsittely noudattaen saga-mallia. Saga on virheenhallintamalli, joka toteuttaa tapahtumat askel kerrallaan (Saga pattern s.a.; Microsoft s.a.). Jos jokin askeleista epäonnistuu, saga-malli toimeenpanee kompensoivia vastatoimenpiteitä aikaisemmille toimenpiteille (Microsoft s.a.). Saga-mallin ohjelmistovirheitä on kuitenkin vaikea selvittää ja se monimutkaistuu lisää mikropalveluiden lisääntyessä (Saga pattern s.a.).

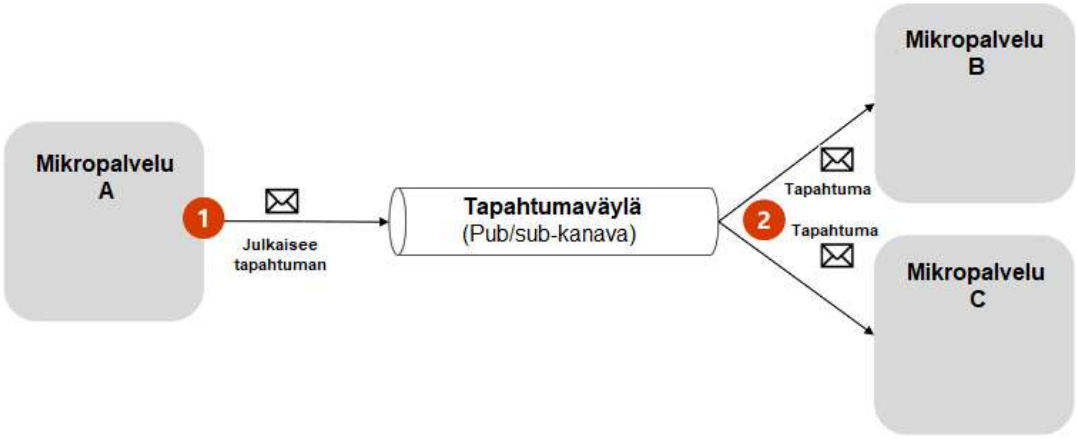
## 2.2 Mikropalveluarkkitehtuuri

Mikropalveluiden (microservices) arkkitehtuurillisena ideana on rakentaa palvelinapplikaatio pienten palveluiden joukkona. Kukin palvelu ajaa omaa prosessiaan ja kommunikoi muiden prosessien kanssa protokollien välityksellä. Kukin mikropalvelu toimeenpanee joko palvelimen bisnestason toiminnallisuuden (De la Torre ym. 2020, 25; IBM Cloud Education 2021a) tai tietyn päätepisteen verkkotunnuksessa (De la Torre ym. 2020, 25). Jokainen mikropalvelu on itsenäisesti käyttöönotettavissa (IBM Cloud Education 2021a; What are microservices? s.a.).

Mikropalvelut voivat kommunikoida toistensa kanssa synkronisesti jääden odottamaan vastausta tai asynkronisesti, jolloin ei jäädä odottamaan vastausta, vaan protokolla lähettää viestin heti esim. tapahtumaväylään (De la Torre ym. 2020, 45; IBM Cloud Education 2021a). Tapahtumaväylä (event bus) on rajapinta, johon sisältyy API. API:n avulla tilataan ja lopetetaan tilauksia (subscribe, unsubscribe) tapahtumiin sekä julkaistaan tapahtumia. (De la Torre ym. 2020, 134.) Tapahtumaväylä liittyy Observer-malliin sekä Pub/sub-malliin (mts. 137) antaen mikropalveluiden kommunikoida keskenään Pub/sub-tyylisesti ilman, että komponentteja vaaditaan olemaan tietoisia toisistaan (De la Torre ym. 2020, 136; Google Cloud s.a.).

Mikropalvelut voivat toimia tapahtuman julkaisijoina ja/tai tilaajina ja tapahtumaväylä toimii viestin välittäjänä eteenpäin. Kuvassa 3 vaiheessa 1 mikropal-

velu A julkaisee tapahtuman (publish) tapahtumaväylään, minkä jälkeen vaiheessa 2 tämän tapahtuman tilanneet (subscribe) mikropalvelut B ja C käyvät noutamassa tapahtuman tiedot väylästä.




Kuva 3. Pub/sub-perusteet tapahtumaväylän kanssa (käännetty lähteestä De la Torre ym. 2020, 137)

Yksittäisen palvelun tasot tai osat voidaan kukin jakaa myös yhdelle mikropalvelulle, jolloin muodostetaan niin kutsuttu monoliittinen applikaatio. Monoliittisen applikaation mallissa ”pinotaan” konteissa olevia applikaation osia yhdeksi suuremmaksi kokonaisuudeksi. Monoliittinen applikaatio -mallin ongelma on skaalautuvuus. (De la Torre ym. 2020, 19.) Applikaatiossa vain muutaman palasen tarvitessa resursseja skaalausta ei voida suorittaa, sillä yksittäistä komponenttia ei voida skaalata itsenäisesti. Palasten vaatimukset voivat myös olla erilaisia esimerkiksi prosessoritehon tai muistin osalta. (Microservices.io s.a.)

## 2.3 Virtualisointi

Virtualisointi käyttää ohjelmistoa luodakseen varsinaisesta laitteistosta eriävän abstrakiotason (What is Virtualization? s.a.). Laitteiston resurssit voidaan jakaa useaan eri virtuaalikoneeseen (virtual machine eli VM), jolla voidaan mahdollistaa tehokkaampi ja taloudellisempi laitteistoresurssien käyttö, kuten pilvipalveluiden skaalautuvuus (IBM Cloud Education 2019b). Kontti ja virtuaalikone eroavat toisistaan siten, että kontin tarkoituksena on ajaa vain yksittäinen ohjelma (De la Torre ym. 2020, 4; What is Virtualization? s.a.). Virtualisoituja Docker-kontteja (containers) on hyvä käyttää yhdessä mikropalveluarkkitehtuurin kanssa (De la Torre ym. 2020, 11; What are microservices? s.a.).

Vaikka virtualisoidut kontit tuovat mahdollisuuksia ja sopivat hyvin mikropalveluiden ajamiseen, ne eivät ole pakollisia mikropalveluiden arkkitehtuurille (De la Torre ym. 2020, 18; What are microservices? s.a.).

Kontteja on helppo ottaa käyttöön ja hallita manuaalisesti, jos niiden määrä on pieni, mutta määrän kasvaessa tämä täytyy automatisoida (IBM Cloud Education 2021b), sillä konttien määrä lisää hallinnan monimutkaisuutta (What is Container Orchestration? s.a.). Konttien orkestrointi automatisoi konttien käyttöönoton, verkon, skaalauksen, saatavuuden ja elinkaaren hallinnan (IBM Cloud Education 2021b) tehden niiden hallinnasta yksinkertaisempaa ja joustavampaa (What is Container Orchestration? s.a.). Kubernetes on suosittu konttien orkestroimiseen tarkoitettu alusta, joka tarjoaa laajan sarjan erilaisia työkaluja (IBM Cloud Education 2021b; What is Container Orchestration? s.a.). Näitä ovat mm. työkalut konttien käyttöönottoa, verkkoon tai muihin konteihin yhdistämistä ja konttien käyttämän tallennustilan varaamista varten (IBM Cloud Education 2021b).

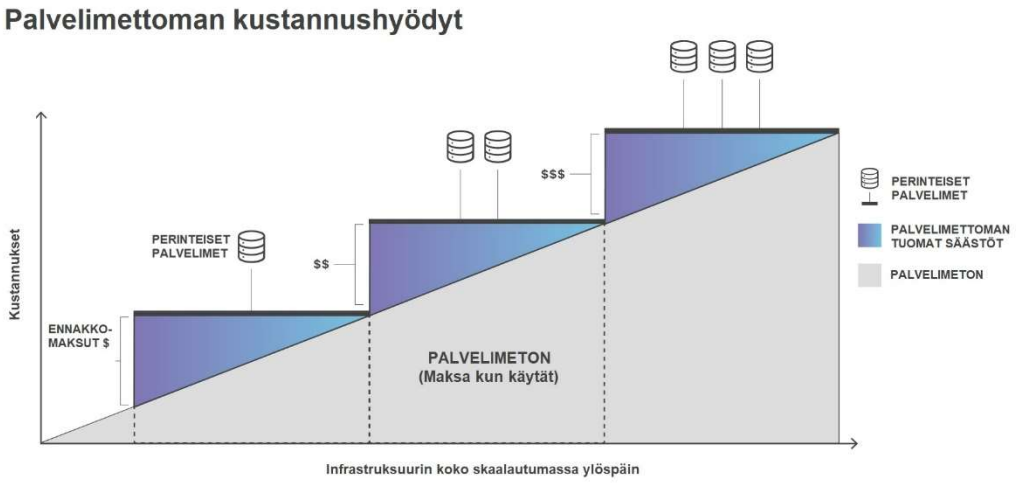
Kubernetes-klusteri on joukko solmukoneita (node machines) eli solmuja, mikä suorittaa kontteihin suljettuja sovelluksia (Red Hat 2020; What is a Kubernetes cluster? s.a.). Klusteri sisältää ohjaustason (Red Hat 2020), eli isäntäsolmun (What is a Kubernetes cluster? s.a.). Isäntäsolmu kontrolloi klusterin tilaa halutunlaiseksi (Red Hat 2020; What is a Kubernetes cluster? s.a.). Klusteri sisältää myös solmut, joiden tehtävänä on ajaa sovelluksia ja toteuttaa tehtäviä (Red Hat 2020; What is a Kubernetes cluster? s.a.), joita isäntäsolmu antaa (What is a Kubernetes cluster? s.a.). Klusterin haluttu tila voi sisältää erilaisia toiminnallisia elementtejä, kuten ajettavat sovellukset, sovellusten käyttämät kuvat, sovellusten resurssit sekä kopioiden (replikojen) määrä (What is a Kubernetes cluster? s.a.). Jotta klusteri saataisiin toimivaksi, tarvitaan vähintään yksi isäntäsolmu ja vähintään yksi työtä suorittava solmu (What is a Kubernetes cluster? s.a.).

## 2.4 Palvelimeton tietojenkäsittely

Palvelimeton tietojenkäsittely (serverless computing) on tapa tarjota resursseja laskentaa ja muistia varten käytön mukaan. Palvelimeton tietojenkäsittely mahdollistaa palveluiden automaattisen skaalautumisen siten, että käyttäjän

tarvitsee maksaa vain niistä resursseista, joita käyttää. (What is serverless computing? s.a.; Fruhlinger 2019.)

Kuten kuvassa 4 näkyy, palvelimettoman tietojenkäsittelyn skaalautuvuus tuo säästöjä verrattuna perinteiseen malliin, jossa ylläpidetään palvelimia itse. Kun palvelimia ei käytetä, tai ei niitä käytetä tarpeeksi tehokkaasti, resursseja menee hukkaan.




Kuva 4. Palvelimettoman tietojenkäsittelyn kustannushyödyt (käännetty lähteestä What is serverless computing? s.a.)

Palvelimettomassa tietojenkäsittelyssä käyttäjien ei tarvitse huolehtia palvelun takana olevasta infrastruktuurista, eikä heidän tarvitse myöskään olla tietoisia fyysisistä palvelimista, joilla palvelu toimii (What is serverless computing? s.a.; Fruhlinger 2019). Virtualisoidut kontit ja palvelimeton laskenta tulevat lähes varmasti olemaan olemassa yhtä aikaa tulevaisuudessa (Fruhlinger 2019). Moni palvelimettoman tietojenkäsittelyn palveluntarjoaja tarjoaakin tietokanta- ja tallennuspalveluiden lisäksi FaaS-alustoja (What is serverless computing? s.a.).

Function-as-a-Service (FaaS) on malli, jossa virtualisoiduista koneista käynnistetyt kontit ajavat käyttäjän määrittämiä funktioita (Kim & Lee 2019, 502). FaaS käsitetään palvelimettoman arkkitehtuurin osajoukkona, jota ajetaan vain tapahtumien tai pyyntöjen yhteydessä (IBM Cloud Education 2019a). FaaS:ia tarjoavia palvelimettomia rakenteita (serverless framework) ovat mm. AWS Lambda ja Cloudflare Workers.

# 3 ERI RATKAISUVAIHTOEHTOJEN TEKNISTEN TIETOJEN VERTAILU

Koska työssä on tarkoituksena perehtyä pilvifunktioihin ja tapahtumaväyliin, on myös hyvä tehdä vertailua eri palveluntarjoajien ratkaisujen välillä, jotta saataisiin tietoa niiden ominaisuuksista sekä voidaan tehdä päätöksiä siitä, mitä ratkaisua työn toteutuksessa olisi parasta käyttää. Tässä luvussa kerrotaan AWS Lambdasta sekä Cloudflare Workersista, jotka tarjoavat mahdollisuuden suorittaa pilvifunktioita. Lisäksi kerrotaan Apache Kafkasta ja VMware RabbitMQ:sta, jotka ovat kumpikin relevantteja tapahtumaväyliä hallintaliittymän käytettäväksi. Teoreettisten ominaisuuksien vertailun päätteeksi katsotaan, mitä ratkaisuja päädytään käyttämään työssä ja minkä vuoksi.

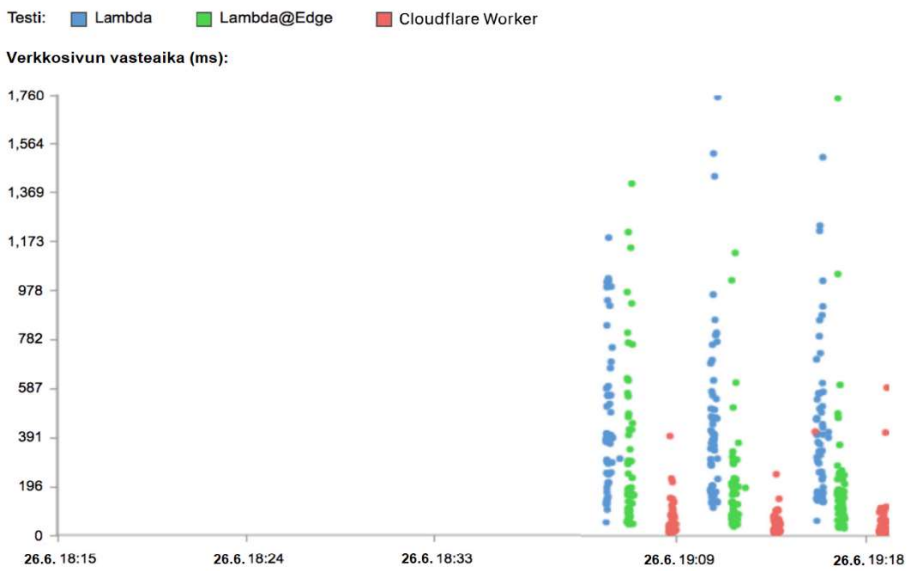
## 3.1 Lambda ja Workers

AWS Lambda on infrastruktuuri, joka suorittaa koodia vain tarvittaessa ja se tukee lukuisia ohjelmointikieliä (AWS Lambda Features s.a.; How can serverless improve performance? s.a.). Lambdassa on Java-, Go-, PowerShell-, Node.js-, C#-, Python- ja Ruby-tuki sekä tuki mille tahansa kolmannen osapuolen ohjelmakirjastoille. Lambda skaalaa automaattisesti tukeakseen saapuvia kutsuja ilman erillisiä asetuksia. Tyypillisesti koodia aletaan ajaa millisekunteja tapahtuman jälkeen. Lambda on myös liitettävissä moniin muihin AWS:n tarjoamiin palveluihin, mikä tuo mahdollisuudet käyttää sitä yhdessä tietokantojen kanssa tai vaikkapa tehdä kustomoitu backend-palvelu. (AWS Lambda Features s.a.) AWS tarjoaa alueita (regions) ympäri maailmaa ja tyypillisesti Lambdaa ajetaan vain yhdellä näistä alueista (How can serverless improve performance? s.a.).

AWS Lambda@Edge on Lambda, joka on käyttöönotettu kaikilla AWS:n tarjoamilla alueilla yhden maantieteellisen sijainnin sijaan (How can serverless improve performance? s.a.). Kun Lambda@Edge-funktiota kutsutaan, se ajetaan käyttäjää lähimpänä olevalla alueella, jolloin vasteaika on pienin mahdollinen (Lambda@Edge s.a.; How can serverless improve performance? s.a.). Lambda@Edge toimii samalla tavalla kuin Workers, jota isännöidään datakeskuksissa ympäri maailmaa (How can serverless improve performance? s.a.).

Cloudflare Workers tukee JavaScript-, TypeScript-, Rust-, C-, Cobol-, Kotlin-, Dart-, Python-, Scala-, Reason/OCaml-, Perl-, PHP- ja FSharp-kielillä kirjoitettua koodia. Workersiin tulevien kutsujen määrälle ei ole rajaa, vaikkakin näillä on tietyt kokorajoitukset, jotka riippuvat tilauksen tyypistä. Vastausten määrällä ei myöskään ole rajaa, mutta Cloudflaren sisällönjakeluverkkoon (Content Delivery Network, CDN) kohdistuva välimuistin käyttö on rajallista. (Platform s.a.) Workers vastaa tyypillisesti alle 200 ms sisällä kylmäkäynnistyksessä. Tämä johtuu siitä, että Workers ajetaan Chrome V8:lla Node.js:n sijaan ja se käyttää siten myös vähemmän muistia. (How can serverless improve performance? s.a.)

Cloudflare suoritti testin, jossa vertailtiin Lambdan, Lambda@Edgen ja Workersin nopeuksia. Kuvassa 5 näkyy vasteajat näille kolmelle. Lambdaa edustaa sininen, Lambda@Edgeä vihreä ja Workersia punainen väri (kuva 5). Testissä Lambdafunktiot isännöitiin alueella US-East-1. (How can serverless improve performance? s.a.) Testin perusteella Workers vastaa lähes poikkeuksetta puolen sekunnin (500 ms) sisällä, kun taas puolestaan Lambdan vastaavat funktiot suoriutuvat suurimman osan ajasta sekunnin sisällä (1000 ms). Vaikka AWS:llä on palvelimia ympäri maailmaa, Cloudflarella on enemmän reunapalvelimia ja siten enemmän läsnäolopisteitä (Point of Presence, PoP), joiden määrä vaikuttaa vasteaikaan (How can serverless improve performance? s.a.).




Kuva 5. Verkkosivun vasteaika eri pilvifunktioilla (käännetty lähteestä How can serverless improve performance? s.a.)

Koska nopeuksien vertailua näiden kolmen FaaS-ratkaisun välillä löytyi vain yhdestä lähteestä, päätettiin toteuttaa omat testifunktiot sekä Lambdalle että Workersille (luku 4.1) ja tämän jälkeen kokeilla suoritusnopeutta sekä käytännön toimivuutta funktionaalisella testauksella. Toisena syynä tälle kokeilulle on aiempi kokemattomuus Workersin käytöstä; on vaikeaa arvioida, toimisivatko kutsut Cloudflaren palvelimelle suunnitellusti, sillä Workers ei tue XmlHttp-kutsuja. Tämän lisäksi Workers toimii yksittäisen kooditiedoston avulla tyypillisen kansiorakenteen sijasta, joten jonkinlainen kääntäjä olisi tarpeen.

Lambdalla, Lambda@Edgellä ja Workersilla on toisiinsa nähden myös hintaeroja. Hinnoittelua kunkin osalta yhtä miljoonaa kutsua kohden on havainnollistettu taulukossa 1. Hintoihin vaikuttavat myös muut tekijät, kuten käytetty muisti, joka lasketaan GB/s kohti taulukon 1 mukaan. Workersilla on vielä erikseen määritelty sitoumukseton poistumistiedonsiirto, mikä käytännössä tarkoittaa lisämaksua siirrettäessä dataa kuukauden aikana enemmän kuin 5 GB:n edestä muille kuin Cloudflaren Bandwidth Alliancen kumppaneille (Pricing s.a.). Workersin maksulliseen suunnitelmaan kuuluu myös 5 $/kk minimimaksu, johon sisältyy miljoona kutsua, 400 000 GB-sekuntia varattua muistia ja 5 GB sitoumuksetonta poistumistiedonsiirtoa (Pricing s.a.). Näiden rajojen ylittyminen kuukauden aikana laskutetaan taulukon 1 mukaisesti.

Taulukko 1. Lambdan, Lambda@Edgen ja Workersin hinnoittelut ovat aseteltu rinnakkain (käännetty lähteistä AWS Lambda Pricing s.a.; Pricing s.a.).

| | Lambda | Lambda@Edge | Workers |
|---|---|---|---|
| Miljoona kutsua ($) | 0,20 | 0,60 | 0,15 |
| Varattu muisti / GB-sekunti ($) | 0,0000167 | 0,00005001 | 0,0000125 |
| Sitoumukseton poistumistiedonsiirto / GB ($) | | | 0,045 |

AWS puolestaan tarjoaa ilmaisen maksusuunnitelman Lambdan käyttäjälle. Siihen sisältyy miljoona kutsua ja 400 000 GB-sekuntia varattua muistia. Nämä määrät vähennetään aina loppusummasta, mikäli ilmaisrajat ylittyvät. Lambda@Edge ei puolestaan tarjoa ilmaista käyttöä. Tavalliseen Lambdaan täytyy aina varata vähintään 128 MB muistia. (AWS Lambda Pricing s.a.)

Koska kyseessä on vähän muistia käyttävä funktio (noin 70-80 MB), sen suorittamiseen varataan 128 MB ja ajaminen kymmenen miljoonaa kertaa kuukaudessa tulisi maksamaan (10 000 000 s * 128 MB / 1024 MB - 400 000 GB/s) * 0,0000167 $ = 14,195 $ käyttäen Lambdaa, mikäli sen suorittamisessa kuluu sekunti. Tämän lisäksi Lambdan kutsuminen maksaa 0,20 $ miljoonaa kutsua kohden, joista ensimmäiset miljoona kutsua kuukaudessa ovat ilmaisia (AWS Lambda Pricing s.a.). Tämä lisää loppusummaan (10 000 000 - 1 000 000) * 0,0000002 $ = 1,80 $. Nämä tekisivät yhteensä 15,995 $, joka on lopullinen hinta, mikäli Lambdaa kutsutaan AWS:n sisäisessä verkossa.

Mikäli Lambdaa täytyisi pystyä kutsumaan muualta Internetistä, siihen täytyy liittää AWS API Gateway, jolla on oma erillinen hinnoittelunsa. API Gateway on ilmainen kuitenkin vain uusille AWS-käyttäjille vuoden ajan (Amazon API Gateway Pricing s.a.). Kunkin kuukauden ensimmäiset 333 miljoonaa välitettyä pyyntöä maksavat 3,50 $ jokaista miljoonaa kutsua kohti (Amazon API Gateway Pricing s.a.), eli hintaan lisättäköön 10 * 3,50 $ = 35 $, mikä tekee loppusummaksi 50,995 $ kuussa, mikäli Lambdaa kutsutaan ulkoisesta verkosta käsin.

Workers ei tarvitse erillistä ”etuovea”, jotta funktioita voitaisiin kutsua ulkoisesta verkosta käsin, vaan kuhunkin funktioon sisältyy ilmainen URL-osoite, josta niitä voidaan kutsua koska tahansa. Kun Workersilla ajetaan sama määrä kuukaudessa sekunnin funktioita, hinnaksi tulee 5 $ perusmaksu ja lisäksi (10 000 000 - 1000 000) * 0,15 $ / 1 000 000 = 1,35 $ kutsuista sekä (10 000 000 s * 128 MB / 1024 MB - 400 000 GB/s) * 0,0000125 = 10,625 $ muistista, jotka tekevät yhteensä 16,975 $.

Näistä kaikista tiedoista voimme päätellä, että edullisin vaihtoehto käyttää ulkoisesta verkosta käsin suurilla kutsumäärillä on Workers. Workers suorittaa lisäksi funktioita teoreettisesti nopeammin kevyemmän V8-pohjansa vuoksi, joten vastaavan funktion ajaminen veisi vähemmän aikaa ja siksi kävisi vielä edullisemmaksi. Vähäisessä käytössä edullisin vaihtoehto puolestaan olisi Lambda, mikäli tätä kutsuttaisiin AWS:n sisäisestä verkosta käsin. Tässä tapauksessa hintaero jäisi kuitenkin melko pieneksi. Lopulliset laskelmat tapahtumia käsittelevän funktion suorittamisesta voidaan työssä tehdä vasta siinä vaiheessa, kun käytännön osuus on toteutettu ja ajettavissa.

Jos hallintaliittymän Kafka-palvelin käyttöönotetaan AWS-klusterissa, Lambdan kutsuminen sisäisestä verkosta mahdollistuu. Etuna tässä olisi se, että AWS API Gateway:tä ei tarvita funktion käyttämiseen, eikä siten paljasta funktiota myöskään Internetiin kaikkien muidenkin käytettäväksi lisäten turvallisuutta ja pienentäen käytöstä tulevaa maksua. Kutsu Kafkasta Lambdaan tapahtuisi myös pienemmällä viiveellä niiden sijaitessa samassa palvelinkokonaisuudessa. Workers puolestaan ottaa vastaan kaikki kutsut eri puolilta Internetiä joka tapauksessa ja joutuisi siten täysin luottamaan ohjelmoijan toteuttamaan suojaukseen. Turvallisuuden näkökulmasta katsottuna AWS Lambda olisi varmempi vaihtoehto, sillä sisäisestä verkosta kutsuminen on varmempi tapa välttyä tietosuojariskeiltä.

### 3.2 Kafka ja RabbitMQ

Apache Kafka on tapahtumien suoratoistoalusta, joka antaa julkaista, tilata, tallentaa ja käsitellä tapahtumien suoratoistoja (Apache s.a.) reaaliajassa korkean suorituskykynsä ansiosta (Apache s.a.; Garg 2013, 22–23). Kafka on jaoteltu systeemi, joka koostuu palvelimista ja asiakasohjelmista (Apache s.a.; Garg 2013, 22). Ne kommunikoivat keskenään TCP-protokollan avulla. Kafka on alustana myös viansietokykyinen ja turvallinen. (Apache s.a.)

VMware RabbitMQ on Erlang-kielellä toteutettu AMQP-viestinvälittäjä (Dossot 2014, 30; McClain 2020). AMQP eli Advanced Message Queuing Protocol on avoin standardi, joka määrittelee protokollan järjestelmienväliseen tapaan vaihtaa viestejä (Dossot 2014, 27). RabbitMQ:ta on sittemmin laajennettu tukemaan myös muita protokollia, kuten esimerkiksi STOMP:ia (Streaming Text Oriented Messaging Protocol) ja MQTT:ä (MQ Telemetry Transport) (Which protocols does RabbitMQ support? s.a.). Kaikki RabbitMQ:n tukemat protokollat ovat TCP-pohjaisia ja käyttävät oletuksena pitkäikäisiä yhteyksiä säilyttääkseen tehokkuuden (Connections s.a.).

RabbitMQ:n viestintätapa on reitittää viestit perinteisiin viestijonoihin (Instaclustr 2021; McClain 2020), kun taas Kafkan viestintätapa on Pub/sub-tyylinen (Instaclustr 2021). Molempia pystyy käyttämään mikropalveluiden kommuni-

kointiin (Instaclustr 2021; McClain 2020) sekä viestien puskurointiin (Instaclustr 2021). Ominaisuuksiensa puolesta tällä hetkellä Kafkan vahvuuksia ovat suorituskyky sekä suoratoiston prosessointi, mukaan lukien pääsy suoratoiston historiaan (Instaclustr 2021; McClain 2020). RabbitMQ:n vahvuuksia ovat puolestaan viestien reitittäminen (Instaclustr 2021; McClain 2020) ja viestien välitysnopeus (McClain 2021).

Edellisestä voimme päätellä, että yleisesti parempi vaihtoehto on RabbitMQ viestien välittämiseen useisiin eri mikropalveluihin reitityskykynsä vuoksi, kun taas Kafka suoriutuu paremmin tilanteissa, joissa siirretään suuria tietomassoja tai vaihtoehtoisesti tarvitaan viestihistoriaa (Instaclustr 2021; McClain 2020) vaikkapa esimerkiksi saga-vianhallintamallin toteuttamiseksi.

Koska hallintaliittymässä ei käytetä kahta erillistä funktiota tilaamaan ja julkaisemaan (muun muassa Cloudflaren puutteellisten webhookien takia), on tarkoituksena hakea ajastetusti uusin tieto Cloudflaren palvelimelta, mikä olisi mahdollista toteuttaa Kafkan Connector-luokkaa hyödyntävällä asiakasohjelmalla. Mikäli Cloudflaren palvelin ei olisi sillä hetkellä saavutettavissa, olisi tärkeää, että toiseksi uusin tieto olisi kaikesta huolimatta saatavilla. Kafkan viestihistoria mahdollistaa tämän tiedon noutamisen kyseisessä tilanteessa, toisin kuin RabbitMQ. Tämän vuoksi työssä päädytään käyttämään Kafkaa.

# 4 DNS-HALLINNOINNIN TOTEUTUS JA TESTAUS

Tässä esitellään käytännön työssä toteutettu tapahtumaväylän kokoonpano yhdessä pilvifunktioiden kanssa, jotka ovat tarkoitettu Cloudflare DNS -hallinnointia varten. DNS eli Domain Name System on maailmanlaajuisesti hajautettu tietokanta, joka säilyttää muistissaan IP-osoitteita sekä näihin liitettyjä verkkoliitäntöjen nimiä (Dostálek & Kabelová 2006, 19). Koska testaaminen tapahtuu manuaalisesti, työtä helpotetaan käyttämällä saatavilla olevia testaustyökaluja: paikallinen ajo Webpackin tarjoamalla preview-ominaisuudella ja Cloudflaren oma Workers-ympäristö sekä näiden lisäksi AWS:n Lambda-ympäristö, jota tässä tapauksessa ajetaan toimeksiantajayrityksen AWS-tilin kautta.

### 4.1 Lambda- ja Workers-funktioiden ohjelmallinen toteutus

Työn käytännön toteutuksessa keskeinen aihe on tilaajafunktio, joka tässä tapauksessa toimii myös julkaisijafunktiona. Julkaisu, eli tiedon päivitys tapahtumaväylään tapahtuu joko käyttäjän kysyttyä uutta tietoa Cloudflaresta tai vaihtoehtoisesti ajastetusti. Jotta Kafkan tilaajat ja julkaisijat saisivat yhteyden Cloudflareen, tarvitaan kuitenkin väliin tietoa välittävä funktio. Tätä varten tehdään AWS Lambda -pilvifunktion TypeScriptillä (Node.js), johon löytyi NPM-paketti nimeltä ”Cloudflare Node.js bindings.” Tämä paketti tuo mukanaan API:n jolla on mahdollista toteuttaa autentikointi Cloudflareen joko API-avaimen (API Key) tai API-tunnuksen (API Token) avulla. Metatavu antoi käyttöön API-tunnuksen sekä alueen ID:n metatavu.com-verkkotunnukseen työtä varten. Tunnus sekä ID tallennetaan molemmat ympäristömuuttujiin näin aluksi.

Kun yllä mainitun paketin toimivuutta kokeiltiin, niin lopputulos oli, että paketin tyyppimäärittelyt funktioiden palautustyypeissä olivat epäselvät ja jäivät jokseenkin puutteellisiksi (esim. tyyppinä object). Niinpä oli tarpeen luoda oma luokka DNS-toiminnoille nimeltä DNSActions, jossa on kullekin toiminnolle oma funktio palautustyyppeineen. Tässä vaiheessa luodaan myös erillinen tiedosto tyypeille. Tiedostoon on lisätty rajapinta (interface) kuvaamaan yksittäistä DNS-tietuetta (record).

Kuvassa 6 näkyy DNSActions-luokan alle luotu julkinen staattinen funktio browse, joka palauttaa kaikki DNS-tietueet kyselyn alueelta. Mikäli kysely onnistuu, palautuu sieltä taulu täynnä tietueita. Jos virhe tapahtuu Cloudfalren päässä, palautuksena tulee objekti, jossa on virheen kuvaus. Mikäli jokin muu virhe tapahtuu kyselyä tehtäessä, ohjataan virhe DNSActions-luokan yksityiseen funktioon solveErrorMessage, joka selvittää virheen viestin ja palauttaa sen takaisin käyttäjälle.

Projektin juuressa funktioita ajaa index.ts, jota tässä vaiheessa käytetään toistaiseksi vielä toiminnallisuuden kokeiluun. Kun eri toiminnot on kokeiltu manuaalisesti, siirrytään suunnittelemaan Kafkan lähettämän viestin rakennetta.

```
/**
 * Browses a zone
 *
 * @param zoneId zone id
 */
public static browse = async (zoneId: string): Promise<Record[] | object> => {
  try {
    const response: any = await cloudflare.dnsRecords.browse(zoneId);
    return Promise.resolve(response.result);
  } catch (error: Response | Error | undefined | unknown) {
    return Promise.reject(await DNSActions.solveErrorMessage(error));
  }
}
```

Kuva 6. Browse-funktio pyrkii hakemaan alueen DNS-tietueet tai palauttaa virheen, mikäli sellainen tapahtuu kyselyn aikana.

Koska HTTPS-kutsun body on JSON-tyyppinen, voisi tähän käyttää rajapintana yksinkertaista oliota. Oliolle tulee nimeksi kuvaavasti IncomingEvent, jonka ominaisuuksia ovat alueen tunniste (zoneId), tietueen tunniste (id), toiminto (action) ja tietue (record) (kuva 7). Tietueen tunniste ja tietue ovat vaihtoehtoisia ominaisuuksia, sillä niitä ei tarvita osassa kutsuja.

```
/**
 * Rajapinta, joka kuvaa Kafkasta saapuvan viestin rakennetta
 */
export interface IncomingEvent {
  zoneId: string;
  id?: string;
  action: "edit" | "browse" | "export" | "delete" | "read" | "add";
  record?: CloudFlareModule.DnsRecord;
}
```

Kuva 7. IncomingEvent-rajapinta (alkuperäinen kuvaus englanniksi)

Tässä kohtaa index.ts:än ja DNSActions-luokan väliin lisätään uusi komponentti actionRouter.ts, joka sisältää funktion kutsujen reitittämistä varten. Funktio komponentin actionRouter sisällä on nimetty kuvaavasti nimellä routeAndExecuteAction. Kyseinen funktio jaottelee switch case-tyylisesti saapuvat viestit niiden toiminto-ominaisuuden tyypin mukaan (kuva 8) eri funktioihin luokassa DNSActions ja kutsuu ne.

Funktioon sisältyy myös tarkistuksia käyttäjän syötteen syntaksille, sillä Cloudflaren API:lla on olemassa omia tiettyjä sääntöjä, joita täytyy noudattaa kutsuja tehtäessä. Mikäli kutsu ei ole sääntöjen mukainen, ei kutsua lähdetä

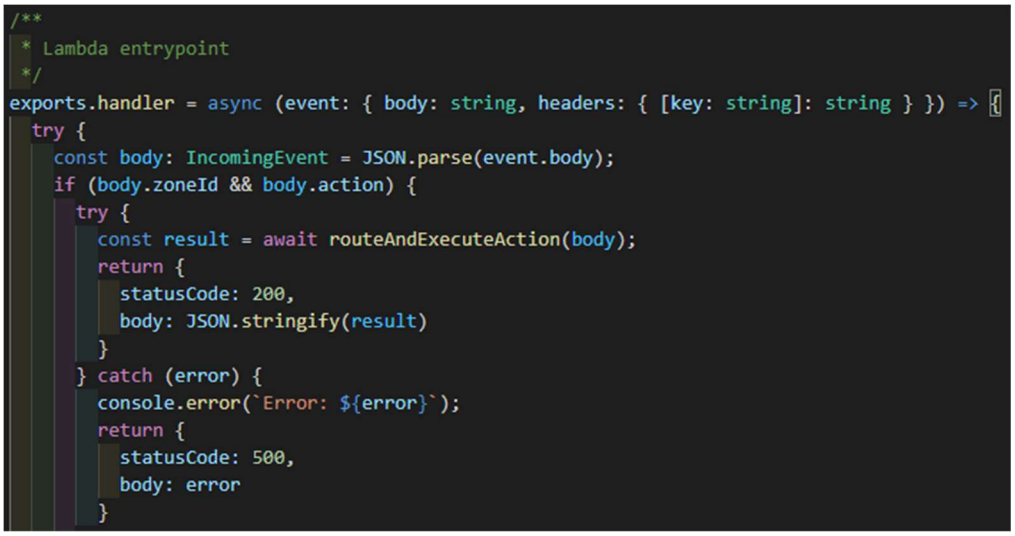
viemään eteenpäin Cloudflarelle, vaan palautetaan virhettä kuvaava viesti käyttäjän päähän (kuva 8). Tämä helpottaa hallinnointia ja pienentää funktion suoritusaikaa virhetilanteissa.

```
/**
 * Routes actions by type and executes them, returning either server response or error message
 *
 * @param body incoming event body
 */
export const routeAndExecuteAction = async (body: IncomingEvent): Promise<string | Record | Record[] | object | RouterError> => {
  switch(body.action) {
    case "edit":
      if (body.id && body.record) {
        const record: any = body.record;
        if (!record.name || (record.name as string).match(" ") != null) {
          return { error: "Record name cannot contain spaces" };
        }
        if (record.type == "CNAME" && record.name == record.content) {
          return { error: "A CNAME record's name may not match it's content" };
        }
        return await DNSActions.edit(body.zoneId, body.id, body.record);
      } else {
        return { error: "Missing record ID or edited record object" };
      }

    case "browse":
```

Kuva 8. Funktion routeAndExecuteAction palautustyypit ja toiminnallisuutta

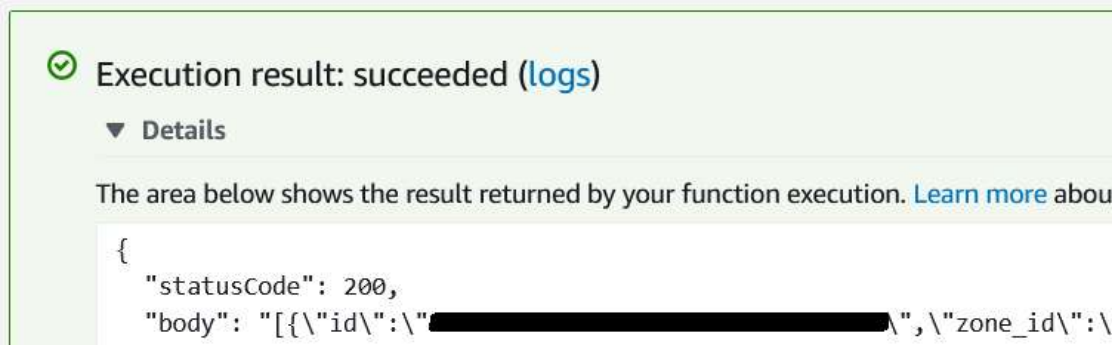
Tämän jälkeen paikalliseen ajoon tehty koodi korvataan Jest-testeillä: Jest-testit kattavat koodista 75 % ja löysivät vielä muutaman ongelman, jotka saatiin korjattua. Lambdalle luodaan vielä sisääntulopiste (entrypoint) määrittelemällä exports.handler, joka vastaanottaa kutsun kuten kuvassa 9 on nähtävissä. Sisääntulopiste ottaa vastaan tulevat kutsut, joiden oikeellisuus on helppo vahvistaa sekä antaa vastaus oikeellisuuden mukaan.



```
/**
 * Lambda entrypoint
 */
exports.handler = async (event: { body: string, headers: { [key: string]: string } }) => {
  try {
    const body: IncomingEvent = JSON.parse(event.body);
    if (body.zoneId && body.action) {
      try {
        const result = await routeAndExecuteAction(body);
        return {
          statusCode: 200,
          body: JSON.stringify(result)
        }
      } catch (error) {
        console.error(`Error: ${error}`);
        return {
          statusCode: 500,
          body: error
        }
```

Kuva 9. Lambdan sisääntulopiste

Testifunktio ei tällä hetkellä ole varsinaisesti turvallinen, sillä siinä ei ole vahvistusta kutsun alkuperästä ja se voi jakaa tietoa kelle tahansa, joka sitä kutsuu. Se kuitenkin riittää testaamiseen tässä vaiheessa, joten on aika katsoa, toimisiko luotu funktio sekä viestirakenne toivotusti palvelimettomassa ympäristössä. Virheitä tuli vastaan mm. pakettien osalta, mutta käsin tehty kutsu meni läpi ja palautti haluttua DNS-dataa Cloudflaren palvelimelta (kuva 10).




Kuva 10. AWS Lambdan palauttamaa dataa suoraan Cloudflaren palvelimelta (ID sensoroitu).

Seuraavaksi kehitetään keino vahvistaa kutsujen alkuperä, joten mukaan kuvioon tuodaan myös salainen tunnus (secret token), jota käytetään vertailuun tulevien kutsujen otsikkotiedoissa (headers). Jotta saataisiin ylimääräinen kerros suojausta, käytetään HMAC-algoritmiä, johon syötetään SHA512-salausalgoritmi sekä salainen tunnus kuten kuvassa 11 on nähtävissä. Lopuksi algoritmiin lisätään siihen tulevan kutsun body kuvan 11 mukaisesti. Kutsujen vaihdellessa salauksen laskutoimituksen lopputulos vaihtelee antaen hieman vahvempaa suojaa kutsumattomia vieraita vastaan.

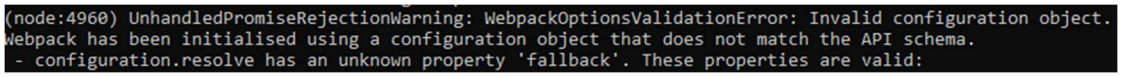
```
/**
 * Lambda entrypoint
 */
exports.handler = async (event: { body: any, headers: { [key: string]: string } }) => {
  let body: IncomingEvent = null;
  typeof event.body == "string" ? body = JSON.parse(event.body) : body = event.body;
  const hmac = crypto.createHmac("sha512", SECRET_TOKEN);
  const expectedSignature = hmac.update(JSON.stringify(body)).digest("hex");
  const signature = event.headers["Signature-512"];
  console.log(expectedSignature, signature);

  if (signature) {
    if (signature === expectedSignature) {
      try {
        if (body.zoneId && body.action) {
```

Kuva 11. Yhdistetty HMAC- ja SHA-512-salaus

Tämän jälkeen tehdään samanlainen Workers-funktio, mutta se ei olekaan yhtä suoraviivaista, sillä npm-paketteja ei suoraan tueta kyseisellä alustalla. Lisäksi yhdessä Workers-funktiossa voi olla vain yksi tiedosto, joten koko projekti tulisi kääntää ja niputtaa käyttämällä apuna Webpackia. Paikalliseen kehittämiseen tarvitaan myös Cloudflare Wrangler -työkalu.

Kun paikallinen projekti oli saatu rakennettua, Wrangler ei kelpuuttanutkaan lopputulosta. Wrangler alkoi toistaa virhettä epäsopivan asetusobjektin takia, kuten kuvassa 12 näkyy.




Kuva 12. Wranglerin API-rajapinta menee ristiin Webpack 5:en konfiguraation kanssa.

Rajapinnat menivät ristiin asetusten fallback-osion kanssa. Osio tarvitaan Webpackin uusimmassa versiossa, jotta npm-paketteja sisältävä projekti suostuisi menemään kääntäjästä läpi. Asiaa tutkittaessa kävi ilmi, että Webpackin versiossa 4 nämä fallbackit ovat sisäänrakennettuja, eikä niitä siksi tarvita. Npmjs.com-sivu myös kertoo, että tällä vuoden vanhalla versiolla on vielä tänäkin päivänä massiivinen määrä latauksia verrattuna sen uudempaan vastineeseen.

Työkalun korvaamisen sekä TypeScript- ja Webpack-konfiguroinnin jälkeen oli vielä välttämätöntä saada vaihdettua TypeScript-yhteensopiva loader (lataaja, jonka Webpack tarvitsee erikseen) vanhempaan versioon, joka vielä tukee Webpack 4:ää. Tämän jälkeen uusi Workers-funktio saadaan julkaistua, joskin siihen täytyy vielä lisätä sisääntulopiste.

Workersin sisääntulopiste on aina nimellä addEventListener, josta kutsutaan käsittelijäfunktiota, joka käsittelee kutsun (kuva 13). Kutsun käsittely tapahtuu Workersissa automaattisesti asynkronisesti, mikäli käsittelijäfunktio on asynkroninen, eikä addEventListener-funktiota tarvitse määritellä sellaiseksi erikseen.

Kuvan 13 käsittelijäfunktio handle vastaanottaa sisääntulopisteestä tapahtumaolion, joka sisältää kutsun (event.request). Kutsu voidaan parsia käytettäväksi tästä olion ominaisuudesta kutsumalla funktio event.request.json, joka palauttaa kutsun JSON-muodossa, kuten kuvassa 13 on nähtävissä.

```
/**
 * Workers entrypoint
 */
addEventListener("fetch", (event: any) => {
  event.respondWith(handle(event));
});

/**
 * Event request handler
 *
 * @param event event
 */
const handle = async (event: any) => {
  const request = await event.request.json();
  let body: IncomingEvent = request ? request : null;
      try {
        if (body.zoneId && body.action) {
          try {
            const result = await routeAndExecuteAction(body);
            return new Response(JSON.stringify(result), {
              status: (result as any).error ? 400 : 200,
              statusText: (result as any).error ? (result as any).error : "OK"
            });
          } catch (error) {
            return new Response(error, {
              status: 500,
              statusText: error
            });
          }
```

Kuva 13. Workers-funktion sisääntulopiste

Seuraavaksi funktiota voidaan testata käyttäen Postmania laittamalla POST-kutsuja Workers-funktion osoitteeseen. Toiminta kuitenkin pysähtyy virheeseen, josta nähdään, että Cloudflaren npm-paketti käyttää XML HTTP-kutsuja, joita ei tueta. Kutsut tulisi sittenkin toteuttaa itse käyttämällä Fetch API:a.

Fetch on yksinkertainen kutsu, joka noudattaa oikeastaan pitkälti samoja periaatteita kuin tavallinen HTTPS-protokollalla tehty kutsu. Node.js:än sisältämillä HTTPS- sekä Fetch-paketeilla on myös samankaltainen käyttösyntaksi, joka helpottaa kutsun tekemistä entisestään: kutsuun tarvitsisi sisällyttää URL-osoite, metodi sekä kutsun headers, johon sisällytetään autentikointi.

Koska kaikki kutsut luokassa DNSActions käyttävät Fetch-metodia, metodille tehdään oma yksityinen staattinen funktio doRequest, jotta voitaisiin välttää

saman koodin toistumista. Kyseinen funktio ottaa vastaan siis kolme parametriä. Kuvassa 14 ovat nähtävissä nämä parametrit, jotka ovat osoite, metodi ja kutsun body (vaihtoehtoinen parametri, jota ei tarvita suurimmassa osassa kutsuja). Funktioon on sisällytetty tarvittavat otsikkotiedot (headers) yhteyden muodostamiseksi. Otsikkotiedoista tärkein on Authorization, sillä se sisältää pääsytunnuksen. Varsinaisen kutsun rakenne riippuu siitä, onko siihen lisätty body. Mikäli body sisältyy kutsuun, on kyseessä joko POST- tai PUT-tyyppinen kutsu. Tässä tapauksessa headersiin sisällytetään mahdollinen sisällön pituus (Content-Length) (kuva 14).

```
/**
 * Performs a Fetch API request to the Cloudflare server using user params
 *
 * @param url url
 * @param method POST | READ | PUT | DELETE
 * @param body request body, if any
 */
private static doRequest = async (url: string, method: string, body?: string) => {
  const headers = {
    "Accept": "*/*",
    "Cache-Control": "no-cache",
    "Host": "api.cloudflare.com",
    "Connection": "keep-alive",
    "Content-Type": "application/json;charset=UTF-8",
    "Authorization": `Bearer ${API_TOKEN}`,
  };
  const request = body ?
    {
      method: method,
      headers: { ...headers, "Content-Length" : `${JSON.stringify(body).length}` },
      body: JSON.stringify(body)
    } :
    {
      method: method,
      headers: headers
    }

  try {
    const response = await fetch(url, request);

    return Promise.resolve(response);

  } catch (error: Response | Error | undefined | unknown) {
    return Promise.reject(await DNSActions.solveErrorMessage(error));
  }
}
```

Kuva 14. Yksityinen staattinen funktio doRequest, jonka tehtävä on tehdä Fetch-kutsuja.

Funktio doRequest on näkyvissä muille funktioille luokan DNSActions sisällä ja on siis kutsuttavissa sisältä käsin. Kuvassa 15 näkyy Fetch API:a käyttäen toteutettu vastaava browse-funktio luokan DNSAction sisällä, joka noutaa kaikki tietyn alueen DNS-tietueet käyttäen aiemmin luotua doRequest-funktiota.

```
/**
 * Browses a zone
 *
 * @param zoneId zone id
 */
public static browse = async (zoneId: string): Promise<Record[]> => {
  try {
    const response = await DNSActions.doRequest(
      `https://api.cloudflare.com/client/v4/zones/${zoneId}/dns_records`,
      "GET"
    );

    const responseBody = await response.json();
    return Promise.resolve(responseBody.result);

  } catch (error: Response | Error | undefined | unknown) {
    return Promise.reject(await DNSActions.solveErrorMessage(error));
  }
}
```

Kuva 15. Browse-funktio

Kun funktiot olivat korvattu fetch-funktioilla, uusien kutsujen vastaukseksi Cloudflaren palvelimelta tuli 400 Bad request. Syyksi paljastui Cloudflare Workersin oma tapa käyttää ympäristömuuttujia, jotka sekoittuivat dotenv-moduulin kanssa. Sekaannus aiheutti pääsytunnuksen puuttumisen kutsun headerseista. Kun ympäristömuuttujien määrittelyt olivat korjattu oikeanlaisiksi, menivät kutsut läpi ilman virheitä ja palauttivat haluttua DNS-dataa Cloudflarelta funktion kautta Postmanille, kuten kuvassa 16 näkyy.


Kuva 16. Workers-funktion palauttamaa dataa (ID sensuroitu)

Viimeinen lisäys Workers-funktiolle on samanlainen suojaus kuin AWS Lambdalla, eli HMAC-algoritmi, johon on lisätty SHA-512-salauksella tunnus sekä kutsun body (kuva 17). Workers ei käsitä kutsun rakennetta samalla tavoin kuin Lambda, vaan kutsu onkin oikeastaan vastaanotetun objektin event sisällä. Toinen eroavaisuus on, ettei headers-objektin ominaisuuksien arvoja voi lukea suoraan, vaan siihen täytyy käyttää omaa erillistä get-metodia. Tämän vuoksi kutsun oikeellisuutta tarkistettaessa salattu avain käydään hakemassa kutsumalla event.request.headers.get, kuten kuvassa 17 näkyy.

```
/**
 * Event request handler
 *
 * @param event event
 */
const handle = async (event: any) => {
  const request = await event.request.json();
  let body: IncomingEvent = request ? request : null;
  const hmac = crypto.createHmac("sha512", SECRET_TOKEN);
  const expectedSignature = hmac.update(JSON.stringify(body)).digest("hex");
  const signature = event.request.headers.get("Signature-512");

  if (signature) {
    if (signature == expectedSignature) {
      try {
        if (body.zoneId && body.action) {
```

Kuva 17. Tapahtuman käsittelijä tekee samankaltaisen avainten vertailun kuin Lambda-funktiossa sulauttamalla kutsun bodyn HMAC-algoritmiä ja SHA512-salattua avainta käyttäen.

Lopuksi funktio täytyy vielä ottaa pois käytöstä julkisesta verkkopäätteestä, jotta siihen ei toistaiseksi päästäisi käsiksi. Tämä on helppo suorittaa Cloudflaren asetuksista selaimella. Teknisen toteutuksen ja toiminnallisen testauksen jälkeen funktiot annetaan aina sovitun prosessin mukaan Metatavun vastaavalle työntekijälle koodikatselmointia varten. Tämä tehdään aina ennen kuin funktio julkaistaan jossain osoitteessa käyttöä varten. Koodikatselmoinnin tarkoituksena ei ole vain pelkkä selkeyden ja formaatin tarkistus, vaan myös turvallisuuden varmistus.

### 4.2 Funktionaalinen testaus

Toimeksiantajayritys Metatavu Oy tahtoo selvityksen funktioiden varsinaisesta käytännön toiminnallisuudesta. Keskeisinä asioina mainittakoon nopeus ja hinta. AWS Lambdan testaus suoritetaan käyttämällä AWS Management Consolen tarjoamaa testaustyökalua, jolla voidaan lähettää kutsu suoraan Lambdan sisääntulopisteeseen. Workersin testaus puolestaan suoritetaan tekemällä POST-tyyppinen HTTPS-kutsu Postmanin avulla. Funktioiden suorittamisen tiedot saadaan Lambdaa varten AWS CloudWatchista ja Workersia varten Cloudflare Dashboardilta.

Kumpaakin funktiota ajetaan sekä kylmäkäynnistyksessä että lämpimässä käynnistyksessä kymmeniä kertoja. Lämpimässä käynnistyksessä funktion kutsu tehdään alle viidessä minuutissa edellisestä kutsusta, jolloin alustan ei

tarvitse valmistautua lataamalla koodia ja varaamalla resursseja sen suorittamiseksi. Kylmäkäynnistyksessä puolestaan kestää pitempään, sillä koodin lataus ja resurssien varaus tapahtuu ennen koodin suorittamista.

Kutsuja suoritettaessa molemmat funktiot pystyivät kommunikoimaan Cloudflaren palvelimen kanssa moitteetta noutaen dataa DNS-tietueista suunnitellusti, eikä virheitä palautunut lainkaan. Funktioita ajettaessa kävi ilmi, että Lambdan suorittamiseen kului aikaa huomattavasti enemmän, vaikka kutsu ei varsinaisesti koskaan lähtenyt AWS:n palvelimen ulkopuolelta. Workers suoriutui tehtävästään erittäin nopeasti, eikä jäänyt hidastelemaan edes kylmäkäynnistyksissä. AWS CloudWatch kertoo, että Lambdan ajamiseen menee vähimmillään 1262 ms ja enimmillään 1959 ms. Suoritettujen funktioiden keston keskiarvoksi tuli noin 1400 ms. Cloudflare Dashboardin mittadata kertoo, että Workers-funktion suorittamiseen meni keskimääräisesti aikaa noin 2,5 ms ja muistia käytettiin noin 0,07 GB/s.

Jos Lambda-funktiota ajettaisiin kymmenen miljoonaa kertaa kuukaudessa ja sen kesto olisi arviolta noin 1400 ms, sen käytön hinnaksi tulisi (10 000 000 * 1,4 s * 128 MB / 1024 MB - 400 000 GB/s) * 0,0000167 $) + (10 000 000 - 1000 000) * 0,0000002 $ = 24,345 $ kuussa.  Jos vastaavaa Workers-funktiota ajettaisiin kymmenen miljoonaa kertaa kuukaudessa ja sen kesto olisi noin 2,5 ms sekä muistin käyttö 0,07 GB/s, niin käytön hinnaksi tulisi 5 $ perusmaksusta, (10 000 000 - 1000 000) * 0,15 $ / 1 000 000 = 1,35 $ kutsuista sekä (10 000 000 * 0,0025 s * 0,07 GB/s - 400 000 GB/s) * 0,0000125 $ = 0 $ muistista. Nämä tekevät yhteensä 6,35 $.  Workers-funktion muisti ei maksa ylimääräistä, sillä käytetyn muistin määrä jää 1 750 GB/s, joka ei ylitä läheskään kuukausimaksuun sisältyvää 400 000 GB/s:n rajaa.

Saaduista tuloksista voimme päätellä, että Workers ei joudu varaamaan funktion suorittamista varten yhtä paljon muistia ja suoriutuu tehtävästään huomattavasti Lambdaa nopeammin. Workers on tämän lisäksi myös selkeästi kustannustehokkaampi FaaS-tarjoaja.

# 5 TAPAHTUMAVÄYLÄN KONSEPTI HALLINTALIITTYMÄSSÄ

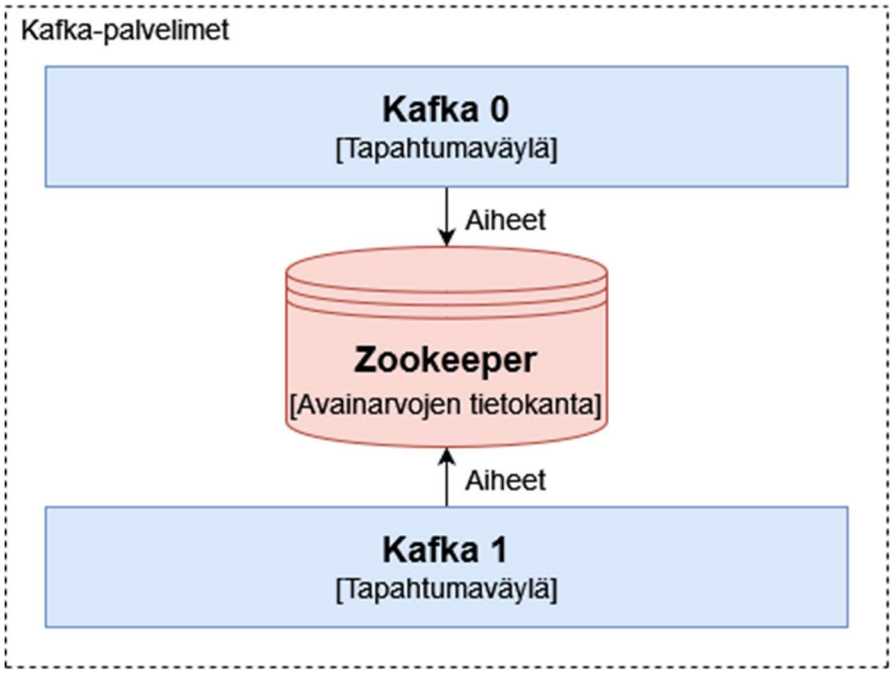
Tässä luvussa tutkitaan tapahtumaväylää, sen konseptia ja potentiaalia toimia hallintaliittymän keskeisenä palasena viestien (tässä tapauksessa lokien) välittäjänä eri mikropalveluihin. Kafka-tapahtumaväylä käynnistetään ja testataan sen toiminnallisuutta välittäessä lokit käyttäjältä julkaisijan kautta tilaajalle. Tämän lisäksi esitellään ohjelmallisesti toteutetut Kafkan osat, jotka välittävät tietoa Kafkasta pilvifunktiolle ja takaisin. Luvussa myös pohdiskellaan, kuinka näitä hallintaliittymän osia tulisi käyttää, jotta niiden toimivuus tuotantoympäristössä olisi mahdollisimman hyvä.

## 5.1 Kafka-palvelimet

Testauksessa käytetään Kafkan mukana tulevia paikallisajon työkaluja, jotka ovat helpoin tapa käyttää Kafkaa testausmielessä. Kafka 3.0 on työn tekohetkellä Kafkan uusin versio, jossa on mahdollista käyttää aiheiden tallentamiseen joko Zookeeperiä tai KRaftia. Zookeeper omaa kuitenkin paremman tuen sekä yhteensopivuuden, minkä vuoksi Kafkan käyttöönotto tehdään käyttämällä Zookeeperiä.

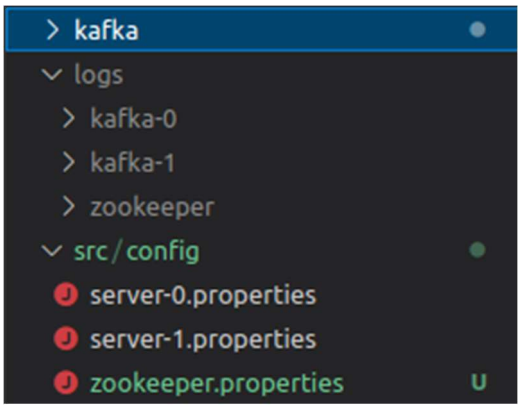
Yksi tapahtumaväylän tärkeimmistä ominaisuuksista on jatkuva saatavuus (high availability, HA), eli käytäntö, jolla pyritään siihen, että järjestelmä on aina käytettävissä. Tämä voidaan varmistaa tekemällä kaksi Kafka-palvelinta valvomaan muutoksia yhdessä Zookeeperissä replikoimalla kuvan 18 havainnollistamalla tavalla. Käytännössä loki menee perille kullekin tapahtuman tilaajalle kuitenkin vain kerran, mikäli ylimääräisiä asetuksia ei tehdä. Jos ensimmäinen Kafka-palvelin kaatuu, toinen jatkaa lokien välittämistä eteenpäin yksin.

Testissä ajetaan paikallisesti molempia Kafka-palvelimia, jotka varsinaisessa tuotantoympäristössä voisivat sijaita eri palvelimilla palvelunkatkosten estämiseksi. Palvelimia voisi olla tuotantoympäristössä myös kolme tai tarvittaessa useampi.




Kuva 18. Kaksi Kafka-palvelinta valvoo Zookeeperissä tapahtuvia muutoksia.

Kun Kafka on ladattu, tehdään projekti sen yläkansioon, jonne voidaan tallentaa muokattuja asetustiedostoja. Nämä ovat properties-tiedostoja, joista yksi on Zookeeperiä varten, toinen ensimmäistä Kafka-palvelinta varten ja kolmas toista Kafka-palvelinta varten. Kafkan juurikansio kannattaa uudelleennimetä esimerkiksi “kafka”-nimiseksi. Tämä lyhentää komentojen syntakseja. Asetukset ovat tallennettu kansioon src/config (kuva 19). Paikallisajoon tarkoitetun projektin kansiorakenne pysyy varsin selkeänä, kuten kuvassa 19 näkyy.




Kuva 19. Lokit ja asetustiedostot ovat lajiteltu omiin kansioihinsa projektin juurikansion alle.

Kansiorakenteen ja tiedostojen valmistelujen jälkeen avataan muutama välilehti terminaaliin, joista ensimmäiseen ajetaan komento kafka/bin/zookeeper-server-start.sh src/config/zookeeper.properties. Tämä komento käynnistää Zookeeperin, jonka oletusasetuksiin on vaihdettu datan tallennussijainniksi

logs/zookeeper. Molempien Kafka-palvelinten lokien sijainti on myös vaihdettu logs-kansion alle (kuva 19).

Toiseen terminaaliin ajetaan Kafkan käynnistymiseksi komento, joka on kafka/bin/kafka-server-start.sh src/config/server-0.properties ja kolmanteen terminaali-ikkunaan ajetaan vastaavanlainen komento, jossa konfiguraatioksi on asetettu server-1.properties. Näissä asetustiedostoissa lokien sijainnin lisäksi kummallekin Kafka-palvelimelle on määritelty omat broker ID:t 0 ja 1 sekä oma portti, johon se käynnistyy.

Kun Zookeeper ja tähän kytketyt Kafkat ovat käynnissä, tarvitaan vielä aihe, joka on replikoitu näiden kahden Kafka-palvelimen kesken. Replikoidun aiheen voi luoda ajamalla komennon kafka/bin/kafka-topics.sh --create --replication-factor 2 --partitions 1 --topic cloudflare-dns-user --bootstrap-server localhost:9092. Kohtaan replication-factor on määritelty kaksi replikaa, kohtaan partitions yksi osio, kohtaan topic aiheen nimi sekä kohtaan bootstrap-server määritellään toinen Kafka-palvelimista. Määritellylle Kafka-palvelimelle lähetetään tämä komento. Osioita on määritelty vain yksi, sillä lokien oikeellinen ajan mukainen järjestys on taattu vain osion sisällä. Mikäli osioita olisi useampi, lokien järjestys menisi sekaisin.

Kun kaikki esivalmistelut Kafkan käyttämiseksi ovat saatu valmiiksi, siirrytään tarkastelemaan luotua aihetta ja Kafkan toiminnallisuutta. Avaamalla uusi terminaali-ikkuna ja ajamalla komento kafka/bin/kafka-topics.sh --describe --topic cloudflare-dns-user --bootstrap-server localhost:9092,localhost:9093 pyydetään aiheen kuvausta ja palautuvasta tiedosta nähdään, että aihe cloudflare-dns-user:lla on yksi osio. Osion johtajana (Leader) on 0 (ensimmäinen käynnistetty Kafka, jonka broker ID on 0) ja replikat löytyvät sekä 0 että 1 broker ID:n omaavista Kafka-palvelimista.

Jos halutaan saada aikaiseksi hallintaliittymää varten toimiva tuotantoympäristö, luodaan vielä toinen aihe nimeltä cloudflare-dns-server. Tämä tulee tehdä siksi, että palvelimen vastaukset käyttäjälle mukaan lukien automaattinen tiedon haku tulee julkaista toiseen aiheeseen. Jos näin ei tehdä, käyttäjän komentoja kuunteleva tapahtuman kuluttaja ajautuu loputtomien kutsujen silmukkaan.

Lokeja voidaan lähettää Kafkaan käynnistämällä manuaaliseen ajoon tarkoitettu tuottajakonsoli avaamalla uusi terminaali-ikkuna ja ajamalla komento kafka/bin/kafka-console-producer.sh --broker-list localhost:9092,localhost:9093 --topic cloudflare-dns-user. Tähän komentoon annetaan lista käytössä olevien Kafka-palvelinten osoitteista sekä aihe, johon aiotaan tuottaa sisältöä.

Kun komento on ajettu, siirrytään yksinkertaisen näköiseen tuottajakonsoliin, jolla voidaan julkaista aiheeseen lokeja kirjoittamalla ne yhdelle riville ja painamalla Enter-näppäintä (kuva 20). Jotta lokien lähetystä ja vastaanottamista voitaisiin testata, konsoliin julkaistaan tässä vaiheessa kymmenen lokia, kuten kuvassa 20 näkyy.

Kuhunkin lokiin on kirjoitettu numero, jotta myöhemmässä vaiheessa niiden lähetysjärjestyksen oikeellisuus voitaisiin tarkistaa. Kaikissa tapauksissa lokien järjestyksellä ei ole välttämättä väliä, mutta hallintaliittymän kannalta tämä olisi olennaista.


Kuva 20. Konsoliin on julkaistu lokeja, joissa kussakin on numero.

Kafkaan lähetettyjä lokeja voidaan lukea avaamalla vielä yksi uusi terminaali-ikkuna ja ajamalla siinä komento kafka/bin/kafka-console-consumer.sh --bootstrap-server localhost:9093,localhost:9092 --topic cloudflare-dns-user --from-beginning, joka avaa kuluttajakonsolin. Komentoon tulevat pitkälti edellisen komennon parametrit sekä näiden lisäksi vaihtoehtoinen parametri from-beginning, jonka avulla jo lähetettyjä lokeja voidaan lukea aiheen alusta saakka.

Konsolin avauduttua lähetetyt lokit ilmestyvät kronologisessa järjestyksessä konsoliin (kuva 21), koska oikea järjestys osion sisällä on taattu. Mikäli tuottajakonsoliin kirjoitetaan lokeja lisää, ilmestyvät ne kuluttajakonsoliin saman tien.


Kuva 21. Kuluttajakonsolin vastaanottamat lokit, jotka ovat peräsin luetusta aiheesta.

Mikäli ensimmäisenä käynnistetyn Kafka-palvelimen sammuttaa ja ajaa uudelleen komennon aiheen kuvausta varten, voi nähdä, että aiheen johtajaksi on vaihtunut 1. Kun tuottajakonsolista käsin lisää aiheeseen lokeja, ne saapuvat kuluttajakonsolille katkoksetta. Kun juuri sammutetun palvelimen käynnistää uudelleen, johtajana toimii edelleen 1. Käytännössä käyttäjä ei edes olisi koskaan huomannut ensimmäisen palvelimen kaatumista toisen palvelimen välittäessä lokit perille.

Näistä testeistä voimme päätellä, että Kafka toimisi hallintaliittymässä suunnitellusti. Viestintä Kafkan välityksellä toimii Pub/sub-tyylisesti ja luotettavasti. Kafkan aiheen lokihistoriaan pääsee käsiksi tapahtuman tilaajan avulla, eivätkä lokit tai lokihistoria katoa tilaajan vastaanottaessa lokia. Vaikka yksi Kafka-palvelin kaatuisi, toinen niistä pystyy jatkamaan lokien välittämistä eteenpäin taaten näin palvelun saatavuuden toisen palvelimen ollessa pois käytöstä.

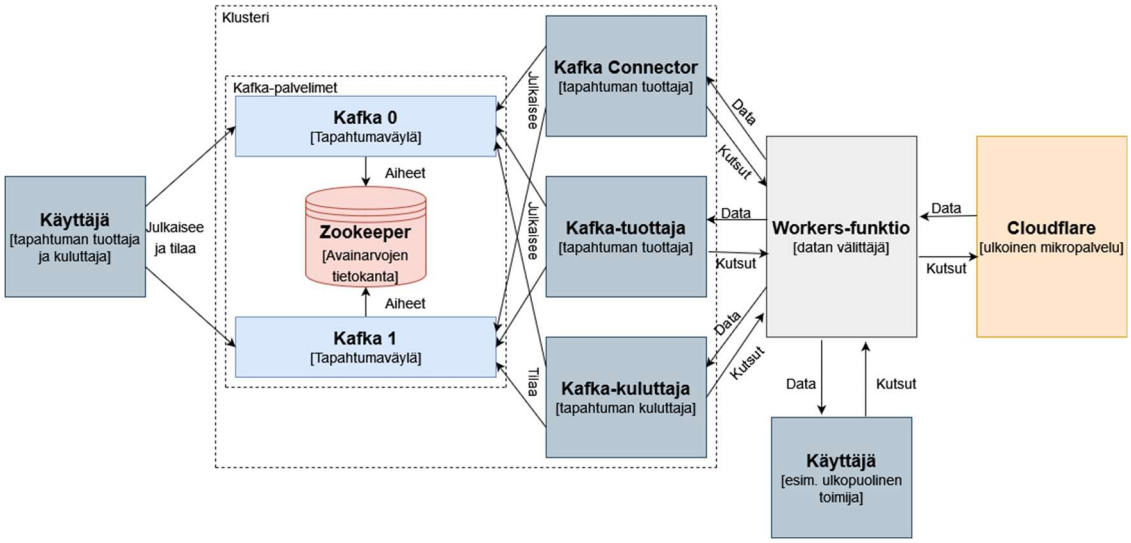
Varsinaisessa tuotantoympäristössä voidaan ajatella käyttäjän päähän sekä tapahtuman tuottaja aiheelle cloudflare-dns-user että tapahtuman kuluttaja aiheelle cloudflare-dns-server. Numeroiden sijaan käyttäjä julkaisisi aiheeseen luvussa 4.1 esitellyn rajapintaolion mukaisia lokeja, jotka välittyisivät toisesta

päästä asiakasohjelma-tyyppisen kuluttajan kautta eteenpäin pilvifunktiolle. Pilvifunktiosta tuleva vastaus puolestaan julkaistaan asiakasohjelma-tyyppisen julkaisijan kautta toiseen aiheeseen lokeina, joista käyttäjä voisi lukea mitta-dataa.

## 5.2 Kafkan ja pilvifunktion kommunikaatio

Varsinaista tuotantoympäristöä varten Zookeeper, molemmat Kafkat, Kafka-tuottaja, -julkaisija ja Connector voisivat esimerkiksi toimia samassa klusterissa. Mikäli tuotantoympäristössä haluttaisiin käyttää Workersia, niin Workers-funktio jäisi klusterin ulkopuolelle, jolloin se paljastuisi ulkoiselle verkolle kuten kuvassa 22 näkyy ja olisi siten kutsuttavissa myös ulkoa käsin. Mikäli Cloudflare DNS -mikropalveluun haluttaisiin päästä käsiksi hallintaliittymän lisäksi myös muilla tavoin, sille voitaisiin rakentaa erillinen asiakasohjelma.

Funktioon käsiksi pääseminen ulkoa ei kuitenkaan ole missään nimessä tarkoituksenmukaista hallintaliittymän kannalta, eikä kuvan 22 ulkopuolinen toimija välttämättä käyttäisi Workers-funktiota hyvin aikein. Workersin marginaalisen pieni hintaero Lambdaan nähden ei kompensoi tarpeeksi Internetin välityksellä Workersin kutsumisen tuomaa tietoturvariskiä.



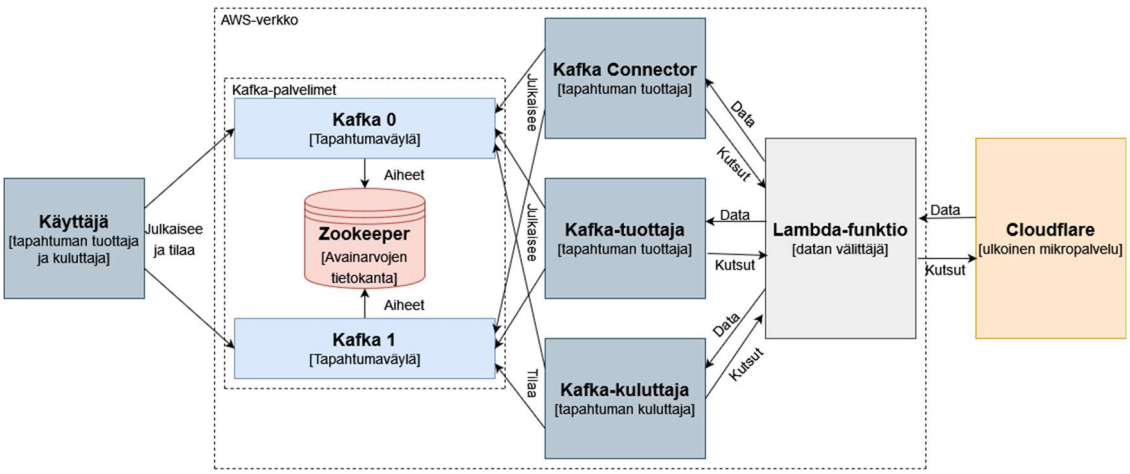

Kuva 22. Käyttäjän kommunikaatio Cloudflaren kanssa käyttäen hallintaliittymää tai vaihtoehtoisesti käyttäen suoraan Workers-funktiota.

Mikäli tuotantoympäristössä halutaan käyttää Lambdaa, niin Zookeeper, molemmat Kafkat, Kafka-tuottaja, -julkaisija ja Connector voitaisiin kaikki sijoittaa

AWS:n palvelimettomassa ympäristössä toimivaan Kubernetes-klusteriin. Tällöin näiden kanssa kommunikoiva Lambda toimisi klusterin kanssa yhdessä samassa sisäisessä AWS-verkossa. Tässä tapauksessa kaikki kommunikaatio hallintaliittymässä toimisi Kafka-palvelimilta Lambdaan saakka AWS-verkon sisällä (kuva 23). AWS:n Kubernetes-klusterissa on olemassa toinenkin vahva ominaisuus, eli mahdollisuus ajaa klusteria usealla eri alueella. Tämä poistaa palvelun riippuvuuden yksittäisen palvelimen toiminnallisuudesta ja siten myös vähentää mahdollisuutta tulla palvelukatkoksia.

Kuvan 22 ja 23 Kafka-tuottaja, -kuluttaja ja Connector ovat erikseen ohjelmallisesti toteutettavia Kafkaan liitettäviä palasia, tai oikeastaan Kafkan laajentamiseen tarkoitettuja asiakasohjelmia. Asiakasohjelmien tarkoituksena on linkittää pilvifunktion toiminnallisuus tapahtumaväylään ja siten mahdollistaa tiedon kulku näiden välillä.

Tapahtuman kuluttaja ja tuottaja ovat tässä tapauksessa ohjelmallisesti yhdistetty kokonaisuus, joka kuuntelee käyttäjäsyötteelle tarkoitettua tapahtumaa, poimii täältä käskyt sekä välittää ne eteenpäin pilvifunktiolle ja siten Cloudflarelle. Pilvifunktiosta tuleva Cloudflaren vastaus taas puolestaan julkaistaan tapahtuman tuottaja-tyylisesti toiseen palvelinsyötteelle tarkoitettuun aiheeseen yhdistetyn osan toimesta. Connector on oma palasensa, jonka ainoana tarkoituksena on hakea ajoitetusti kaikki DNS-tietueet Cloudflaresta ja julkaista ne Kafkaan palvelinsyötteelle varattuun aiheeseen tai vaihtoehtoisesti sille tarkoitettuun omaan aiheeseen. Tämä viimeisin tieto olisi aina valmiiksi noudettu Connectorin ansiosta, vaikka yhteyttä Cloudflareen ei jostakin syystä saataisi.



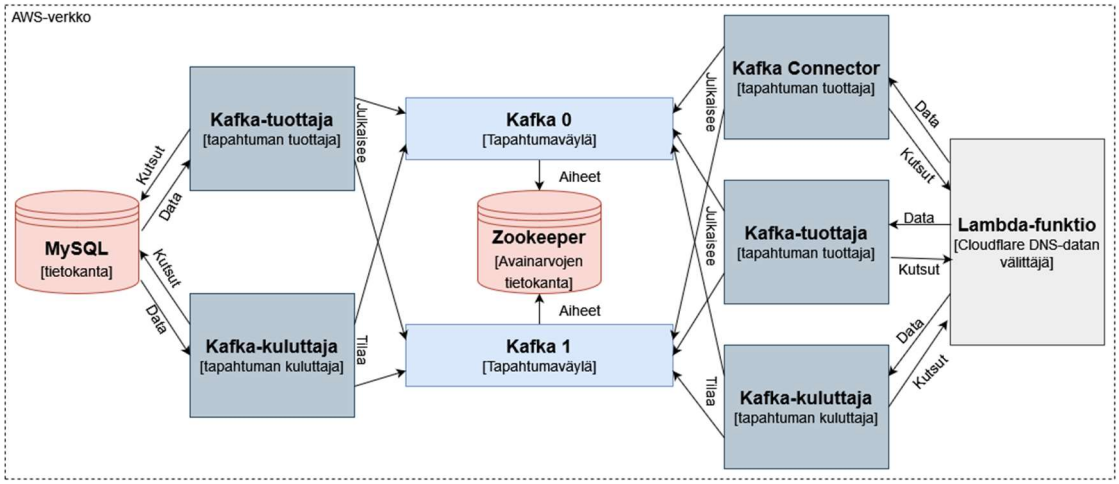

Kuva 23. Käyttäjän kommunikaatio Cloudflaren kanssa käyttäen hallintaliittymää.

Connectorin noutama tieto sekä mikropalvelujen datahistoria voitaisiin kuitenkin säilyttää lisäksi muilla tavoin, jotta kaikesta tiedosta olisi olemassa kopio. Zookeeper ei myöskään varsinaisesti ole tarkoitettu tavanomaiseksi tietokannaksi, joka säilöisi tietoa pitkän aikaa. Hallintaliittymässä olisi kuitenkin hyvä olla olemassa historia pidemmältä ajalta, josta näkee, mitä on tapahtunut milloinkin. Hallintaliittymää tulisi laajentaa lisäämällä siihen tärkeä osa, eli toisenlainen tietokanta.

Tiedot voitaisiin ohjata Kafka-palvelimilta jonkinlaiseen SQL-tietokantaan tai tietokantapalveluun kuten esimerkiksi AWS RDS:ään (Relational Database Service). Käytön perustelu RDS:lle olisi sama kuin Lambdalle, eli se pystyttäisiin myös piilottamaan AWS-verkon sisälle ja näin lisättäisiin turvallisuutta. Mikäli RDS:n tarjoamaa valmisratkaisua ei haluttaisikaan käyttää, voisi silti esimerkiksi jonkun SQL-palvelimen ottaa käyttöön klusterin sisälle.

Mitä hyvänsä vaihtoehtoa käytettäisiinkin, tulisi silti molempien Kafka-palvelinten ja tietokantapalvelimen / tietokantapalvelun väliin tehdä Kafka-tuottaja ja Kafka-kuluttaja, jotta tieto saataisiin liikkumaan edestakaisin, kuten pilvifunktionkin kohdalla (kuva 24). Kuvassa 24 on havainnollistettu tietokannan liittäminen hallintaliittymään käyttäen koodillisesti toteutettavia Kafkan asiakasohjelmia.

Connectoria ei tässä tapauksessa tarvitse käyttää, sillä koko tietokannan sisältöä ei tarvitse kysellä kerralla, vaan tietokantapalvelimen toiminnallisuutta voitaisiin ajatella Zookeeper-palvelimen tietojen varmuuskopiona. MySQL-palvelinta ei siis tarvitsisi käyttää normaalitilanteessa kovinkaan usein. Hallintaliittymä, johon on liitetty tietokanta, voisi nyt palvella täysmittaisesti Cloudflare DNS -mikropalvelua hallinnoitaessa. Koska perustoiminnallisuus on kunnossa, olisi hallintaliittymä valmis laajennettavaksi tästä eteenpäin muidenkin mikropalvelujen hallinnointiin.




Kuva 24. MySQL-palvelin osana hallintaliittymää

Tästä kaikesta voimme päätellä, että hallintaliittymä olisi tässä vaiheessa turvallinen käytettäväksi, olisi jatkuvasti saatavilla sekä pystyisi toimittamaan haluttua tietoa Cloudflare DNS -mikropalvelusta sekä ulkoisten että sisäisten katkosten aikana. Tämän lisäksi on selvää, että hallintaliittymään lisättävät palaset käyttäisivät kaikki vähintään ylimääräistä tuottaja- ja kuluttajapalasta, jotta ne voitaisiin liittää yhdeksi kokonaisuudeksi Kafka-palvelinten saataville. Tuottaja- ja kuluttajapalasten käyttäminen siis koskisi myös Lambda-funktion sekä MySQL:n lisäksi tulevia muita uusia palasia sekä tarvittavia Lambda-funktioita, jotka olisivat liitoksissa erilaisiin mikropalveluihin.

# 6 LOPPUPÄÄTELMÄT

Tapahtumaväylänä Kafka toimii tehokkaasti ja luotettavasti sekä tarjoaa käyttäjälleen vaikuttavan listan ominaisuuksia ja laajennusmahdollisuuksia itse koodattavien asiakasohjelmien takia. Kafkan korkea suorituskyky ja lokihistoria mahdollistavat sen käytön niin suoratoistossa kuin pub/sub-järjestelmissä kaiken toiminnallisuuden keskipisteenä ja niin ollen myös suunnitteilla olevassa hallintaliittymässä. RabbitMQ ei toistaiseksi säilytä lokihistoriaa miten-kään, mikä oli sen tärkein ero Kafkaan verrattuna hallintaliittymän tapahtumaväylän valintaa tehtäessä. Tilanne ei kuitenkaan olisi välttämättä sama, mikäli hallintaliittymää lähdettäisiin suunnittelemaan ja toteuttamaan vasta tulevaisuudessa, sillä molempien tapahtumaväylien kehitystyö on aktiivista.

RabbitMQ:lle on kehitteillä uudenlainen datarakennemalli, jossa tapahtumaa kulutettaessa lokit eivät tuhoudu, vaan olisivat tämänkin jälkeen käytettävissä.

Tämä paikkaisi sen ominaisuuksien puutteita ja pienentäisi RabbitMQ:n eroa verrattuna muihin tapahtumaväyliin. Kafka puolestaan luopuu Zookeeperin käytöstä versioon 4.0 mennessä, ja se on takaraja avaintietokannan vaihtamiseksi KRaftiin. KRaft on vielä kehityksen alla, mutta sen ideologia lokien tallentamiseksi itse Kafkaan ulkoisen palvelimen sijasta avaa uusia mahdollisuuksia Kafkan kehittymiselle sekä tapahtumaväylänä että ohjelmistona.

Mitä tulee työn käytännön toteutusosaan, niin Lambda- ja Workers-funktioiden toteutus sujui onnistuneesti, vaikkakin muutaman mutkan kautta. Vaikka kehitys Workersin alustalle oli lähtökohtaisesti hankalaa, oikeiden työkalujen löydyttyä funktion toteutus ja käyttöönotto kävivät sulavasti. Jos Wrangler-työkalu päivitettäisiin tukemaan Webpack 5:tä, kynnys Workersin käyttämiseksi voisi madaltua entisestään.

Pilvifunktioiden tarjoajien osalta selvitysten sekä testauksen tuloksena kävi ilmi useassa kohtaa, että Workers on palvelimettomana infrastruktuurina nopea, kustannustehokas ja resurssien käytön osalta kevyt. Näin ollen Workers olisi suurimmassa osassa tapauksia paras vaihtoehto, mikäli kutsujen ja datan välittäminen tapahtuisi ulkoisen verkon välityksellä tai tiedon kulun vaarantuminen ei aiheuttaisi isompia ongelmia.

Tuli myös pohdittua, ettei Hallintaliittymässä datan ja kutsujen välittäminen ulkoisessa verkossa kuitenkaan ole pakollista. Vaikka Lambda onkin toiminnaltaan kömpelömpi ja käyttöhinnaltaan kalliimpi kuin Workers, se mahdollistaa ylimääräisen tietoturvan tason kuuluessaan AWS:n palvelimettomien palvelujen joukkoon. Käytännössä tämä tarkoittaa sitä, että Lambdaa voitaisiin kutsua suoraan sisäisestä verkosta käsin.

Tämän hetken paras ratkaisuvaihtoehto lienee hallintaliittymän ja sen komponenttien niputtaminen yhdessä turvalliseen AWS:n verkkoon, Lambda mukaan lukien. Ylimääräisen tietokannan, kuten vaikkapa MySQL-palvelimen tai RDS-palvelun liittäminen osaksi hallintaliittymää parantaisi tiedon saatavuutta katkoksienkin aikana entisestään ja tämän lisäksi se säilöisi hallintaliittymässä tapahtuneet toiminnot sekä tiedot pidemmäksi aikaa. Nämä erilaiset osat voitaisiin sijoittaa klusteriin, jossa niiden ylläpito helpottuisi orkestroinnin ansiosta.

Eri osien ja niin ollen myös hallintaliittymän virheensietokyky pysyisi hyvänä, sillä klusteri ei ole riippuvainen yksittäisten palvelinten tilasta.

Vaikka klusteri ja palvelimettoman ympäristön klusteri ovat tällä hetkellä monella tapaa parhaita vaihtoehtoja, ovat ne silti kumpikin vain yksi mahdollisuus monien muiden mahdollisten ratkaisujen joukosta. Sekä klusterit, tapahtumaväylät että FaaS ovat olleet olemassa jo useita vuosia. Vaikka ne ovatkin ohjelmistokehityksessä keskeisiä ratkaisuja tänä päivänä, uusia ratkaisuja kehitetään jatkuvasti.

**LÄHTEET**

Amazon API Gateway Pricing s.a. AWS. WWW-dokumentti. Saatavissa: https://aws.amazon.com/api-gateway/pricing/ [viitattu 2.10.2021].

Apache s.a. Kafka 3.0 Documentation. WWW-dokumentti. Saatavissa: https://kafka.apache.org/documentation/ [viitattu 26.9.2021].

AWS Lambda Features s.a. AWS. WWW-dokumentti. Saatavissa: https://aws.amazon.com/lambda/features/ [viitattu 1.9.2021].

AWS Lambda Pricing s.a. AWS. WWW-dokumentti. Saatavissa: https://aws.amazon.com/lambda/pricing/ [viitattu 1.9.2021].

Bui, K. 2020. Introduction to Event-Driven Architecture. WWW-dokumentti. Päivitetty 11.2.2021. Saatavissa: https://medium.com/microservicegeeks/introduction-to-event-driven-architecture-e94ef442d824 [viitattu 22.5.2021].

Connections s.a. VMware. WWW-dokumentti. Saatavissa: https://www.rabbitmq.com/connections.html [viitattu 3.9.2021].

De la Torre, C., Wagner, B. & Rousos, M. 2020. .NET Microservices: Architecture for Contanerized .NET Applications. E-kirja. Redmond: Microsoft Developer Division, .NET and Visual Studio product teams. Saatavissa: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/ [viitattu 13.5.2021].

Dossot, D. 2014. RabbitMQ Essentials. E-kirja. Birmingham: Packt Publishing. Saatavissa: https://kaakkuri.finna.fi/ [viitattu 3.9.2021].

Dostálek, L. & Kabelová, A. 2006. DNS in Action. E-kirja. Birmingham: Packt Publishing. Saatavissa: https://kaakkuri.finna.fi/ [viitattu 13.5.2021].

Fruhlinger, J. 2019. What is serverless? Serverless computing explained. WWW-dokumentti. Päivitetty 15.7.2019. Saatavissa: https://www.infoworld.com/article/3406501/what-is-serverless-serverless-computing-explained.html [viitattu 29.5.2021].

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. 1994. Design patterns: elements of reusable object-oriented software. 11. painos. Wokingham: Addison-Wesley.

Garg, N. 2013. Apache Kafka. E-kirja. Birmingham: Packt Publishing. Saatavissa: https://kaakkuri.finna.fi/ [viitattu 16.5.2021].

Google Cloud s.a. Event-driven architecture with Pub/Sub. WWW-dokumentti. Saatavissa: https://cloud.google.com/solutions/event-driven-architecture-pubsub [viitattu 22.5.2021].

How can serverless improve performance? s.a. Cloudflare. WWW-dokumentti. Saatavissa: https://www.cloudflare.com/learning/serverless/serverless-performance/ [viitattu 1.9.2021].

IBM Cloud Education. 2019a. FaaS (Function-as-a-Service). WWW-dokumentti. Päivitetty 30.6.2019. Saatavissa: https://www.ibm.com/cloud/learn/faas [viitattu 16.5.2021].

IBM Cloud Education. 2021a. What are microservices? WWW-dokumentti. Päivitetty: 30.3.2021. Saatavissa: https://www.ibm.com/cloud/learn/microservices [viitattu 23.5.2021].

IBM Cloud Education. 2021b. What is Container Orchestration? WWW-dokumentti. Päivitetty: 27.5.2021. Saatavissa: https://www.ibm.com/cloud/learn/container-orchestration [viitattu 12.10.2021].

IBM Cloud Education. 2019b. What is Virtualization? WWW-dokumentti. Päivitetty 19.6.2019. Saatavissa: https://www.ibm.com/cloud/learn/virtualization-a-complete-guide [viitattu 16.5.2021].

Instaclustr. 2021. RabbitMQ vs. Apache Kafka: Key Differences and Use Cases. WWW-dokumentti. Päivitetty 9.5.2021. Saatavissa: https://www.instaclustr.com/rabbitmq-vs-kafka/ [viitattu 5.9.2021].

Kim J. & Lee K. 2019. FunctionBench: A Suite of Workloads for Serverless Cloud Function Service. Teoksessa IEEE (toim.) 2019 IEEE 12th International Conference on Cloud Computing (CLOUD) Milan: IEEE, 502–504. PDF-dokumentti. Saatavissa: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8814583&isnumber=8814488 [viitattu 14.5.2021].

Lambda@Edge s.a. AWS. WWW-dokumentti. Saatavissa: https://aws.amazon.com/lambda/edge/ [viitattu 2.9.2021].

Le, T. 2020. [Microservices Architecture] What is SAGA Pattern and How important is it? WWW-dokumentti. Päivitetty 30.3.2020. Saatavissa: https://medium.com/swlh/microservices-architecture-what-is-saga-pattern-and-how-important-is-it-55f56cfedd6b [viitattu 22.5.2021].

McClain, B. 2020. Understanding the Differences Between RabbitMQ vs Kafka. Blogi. Päivitetty 16.11.2020. Saatavissa: https://tanzu.vmware.com/developer/blog/understanding-the-differences-between-rabbitmq-vs-kafka/ [viitattu 5.9.2021].

Microservices.io s.a. Pattern: Monolithic Architecture. WWW-dokumentti. Saatavissa: https://microservices.io/patterns/monolithic.html [viitattu 29.7.2021].

Microsoft. 2018. Publisher-Subscriber pattern. WWW-dokumentti. Päivitetty 7.12.2018. Saatavissa: https://docs.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber [viitattu 26.8.2021].

Microsoft s.a. Saga distributed transactions – Azure Design Patterns. WWW-dokumentti. Saatavissa: https://docs.microsoft.com/en-us/azure/architecture/reference-architectures/saga/saga [viitattu 22.5.2021].

Platform s.a. Cloudflare. WWW-dokumentti. Saatavissa: https://developers.cloudflare.com/workers/platform [viitattu 1.9.2021].

Pricing s.a. Cloudflare. WWW-dokumentti. Saatavissa: https://developers.cloudflare.com/workers/platform/pricing [viitattu 22.9.2021].

Red Hat. 2020. What is a Kubernetes cluster? WWW-dokumentti. Päivitetty 15.1.2020. Saatavissa: https://www.redhat.com/en/topics/containers/what-is-a-kubernetes-cluster [viitattu 12.10.2021].

Red Hat. 2019. What is event-driven architecture? WWW-dokumentti. Päivitetty 27.9.2019. Saatavissa: https://www.redhat.com/en/topics/integration/what-is-event-driven-architecture [viitattu 22.5.2021].

Saga pattern s.a. AWS. WWW-dokumentti. Saatavissa: https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-data-persistence/saga-pattern.html [viitattu 22.5.2021].

What are microservices? s.a. OpenSource. WWW-dokumentti. Saatavissa: https://opensource.com/resources/what-are-microservices [viitattu 16.5.2021].

What is a Kubernetes cluster? s.a. VMware. WWW-dokumentti. Saatavissa: https://www.vmware.com/topics/glossary/content/kubernetes-cluster [viitattu 12.10.2021].

What is an Event-Driven Architecture? s.a. AWS. WWW-dokumentti. Saatavissa: https://aws.amazon.com/event-driven-architecture/ [viitattu 22.5.2021].

What is Container Orchestration? s.a. VMware. WWW-dokumentti. Saatavissa: https://www.vmware.com/topics/glossary/content/container-orchestration [viitattu 12.10.2021].

What is serverless computing? s.a. Cloudflare. WWW-dokumentti. Saatavissa: https://www.cloudflare.com/learning/serverless/what-is-serverless/ [viitattu 29.5.2021].

What is Virtualization? s.a. OpenSource. WWW-dokumentti. Saatavissa: https://opensource.com/resources/virtualization [viitattu 16.5.2021].

Which protocols does RabbitMQ support? s.a. VMware. WWW-dokumentti. Saatavissa: https://www.rabbitmq.com/protocols.html [viitattu 3.9.2021].