Juho Tarkkanen

Vapaasti sijoiteltavien ohjelmistojen koostaminen IoT-ympäristössä

Tietotekniikan pro gradu -tutkielma

4. maaliskuuta 2024

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Juho Tarkkanen

Yhteystiedot: juho.a.tarkkanen@student.jyu.fi

Ohjaaja: Tommi Mikkonen

Työn nimi: Vapaasti sijoiteltavien ohjelmistojen koostaminen IoT-ympäristössä

Title in English: Composing readily positionable software in an IoT environment

Työ: Pro gradu -tutkielma

Opintosuunta: Ohjelmisto- ja tietoliikennetekniikka

Sivumäärä: 55+4

Tiivistelmä: Reunalaskenta ja esineiden internet (IoT) lupaavat monenlaisia innovaatioita ja parannuksia niin ihmisten arjen kuin teollisuudenkin teknologiseen ympäristöön. Monenlaisten laitteiden kytkeytyessä toisiinsa tiheiksi ja laajoiksi kommunikaatioverkoiksi, kokonaisvaltaisen järjestelmän yhteistoimintaa pitää kyetä hallitsemaan luotettavasti, mutta ihmisen tekemä manuaalinen ylläpito kasvaa vaatimuksiltaan nopeasti mahdottomaksi. Manuaalisten hallintatoimien pitkälle jalostettua automatisointia eli orkestraatiota kaivataan laitteiden määrään ja heterogeenisyyteen vastaamiseksi. Lisäksi orkestraatio tukisi hajautettujen ohjelmistojen tehokasta liikkuvuutta, jonka katsotaan johtavan helpotuksiin älykkäiden laitteiden jokapäiväisessä käyttämisessä. Tässä tutkielmassa kehitettiin orkestraatiota ilmentävää ohjelmistoartefaktia, jonka avulla voidaan koostaa usean ohjelmiston tai mikropalvelun sekä laitteen yhteistoiminnasta muodostuva sovelluskokonaisuus. Tutkielman päätuloksena on artefakti, joka soveltuu orkestraation ja ohjelmistojen laitteesta toiseen siirtämisen tutkimiseen. Lisäksi tutkielma vahvistaa havaintoja siitä, että WebAssembly-teknologian voidaan sanoa olevan käytettävissä IoT-ympäristössä.

Avainsanat: Orkestraatio, Isomorfisuus, Liukas ohjelmisto, Hajautettu laskenta, Reunalaskenta, IoT, WebAssembly, Suunnittelutiede

Abstract: Edge computing and Internet of Things (IoT) promise many innovations and improvements into the technological environment of ordinary life and industry. When many

different kinds of devices create dense and large-scale communication networks the system requires robust control and management but manual efforts alone will not meet these ever-growing requirements. Orchestration as the sophisticated automation of these manual operations is needed in order to tackle the amount and heterogeneity of devices. In addition orchestration would support the efficient mobility of distributed software, making everyday interaction with smart devices less rigid. This thesis presents the development of a software artifact regarding orchestration. The resulting implementation enables composing together multiple cooperative software pieces or microservices and devices into some desired application. The main result of the study is a software artifact to support further research efforts into orchestration and the mobility of software between different devices. In addition the study reaffirms the notion of WebAssembly as a plausible technology in an IoT environment.

Keywords: Orchestration, Isomorphism, Liquid software, Distributed computing, Edge computing, IoT, WebAssembly, Design science

Esipuhe

Tutkielma tehtiin osana Jyväskylän yliopiston projekteja *Wasm-IoT* ja *LiquidAI 6G software bridge*, joista jälkimmäistä on tukenut Business Finland. Haluan kiittää koko joukkuetta. Projektit ja työtilat joihin pääsin työskentelemään, sekä etenkin ihmiset joihin näissä yhteyksissä tutustuin auttoivat minua suunnattomasti käynnistämään ja jaksamaan tämän työn loppuun asti. Tutkielmassa kuvaamani artefakti ja järjestelmä ovat usean henkilön yhteistyön tulos ja on minulle etuoikeus saada kertoa siitä tutkintoni lopputyötä varten. Kiitokset vielä sekä viralliselle että epävirallisille ohjaajilleni, StartupLab JYU:lle, ystäville, perheelle ja Vim-tekstieditorille.

Jyväskylässä 4. maaliskuuta 2024

Juho Tarkkanen

Termiluettelo

API	*Application Programming Interface*; Ohjelmointirajapinta, joka sille määritellyin kutsumuodoin tarjoaa rajattuja toimintoja tai resursseja ilman, että kutsujan tarvitsee tietää sisäistä toteutusta.
Esineiden internet	Englanniksi *Internet of Things*; hajautettujen ja oleellisesti heterogeenisten laitteiden yhteistoimintaa ja arkipäiväisten esineiden varustamista ohjelmoitavilla tietokoneilla datankeruuta tai erityisiä sovelluksia varten.
IoT	Esineiden internet.
Isomorfisuus	"Samalla muodolla"; ohjelmointikielten yhteydessä mahdollisuus suorittaa samaa ohjelmaa tai uudelleenkäyttää osia siitä järjestelmän eri puolilla esimerkiksi sekä palvelimella että asiakkaalla.
Kontti	Ohjelmistokontti.
Laite	Lähiverkossa toimintojaan mainostava IoT-laite jolla sijaitsevaa orkestraatiolle yhteensopivaa palvelinta voidaan pyytää suorittamaan WebAssembly-ohjelmia (tämän tutkielman kokeellisen osion yhteydessä).
Liukkaus	Ohjelman tai sen osan suorittamista ja liikkumista erilaisten laitteiden välillä käyttäjälleen nähden soljuvasti ja saumattomasti.
Moduuli	Ohjelmistokonttiin verrattava tiedosto- ja metadatakokoelma hiekkalaatikoidun WebAssembly-ohjelmakoodin suorittamiseen (tämän tutkielman kokeellisen osion yhteydessä).
Ohjelmistokontti	Englanniksi *container*; Käyttöjärjestelmän ominaisuuksia ja paketteja kokoava kevyt ja eristetty yksikkö kattamaan sovellukselle sopiva suoritusympäristö.
Orkestraatio	Usean ohjelmistokehityksessä sovelluksen toimittamiseen ja ylläpitoon liittyvien perinteisesti manuaalisten toimien kokonais-

	valtainen automatisointi.
ReST	Lyhenne englanninkielisistä sanoista *Representational State Transfer*; arkkitehtuurityyli, joka painottaa mm. kutsujen välistä tilattomuutta.
ReSTful API	API, joka seuraa ReST-tyylin periaatteita; usein käytössä WWW-palvelimissa.
Reunalaskenta	Englanniksi *edge computing*; Keskitettyjen palvelinsalien sijaan lähempänä verkon reunaa oleville laitteille sijoitettua laskentaa.
Rypäs	Englanniksi *cluster*; joukko solmuja tai laitteita (abstrakteja tai fyysisiä) jotka yhdessä kuuluvat osaksi tai muodostavat jonkin suuremman resurssi-, ohjelmisto- tai sovelluskokonaisuuden.
Skripti	Komentojen sarja, joka usein kuvataan jollakin korkean tason täsmä- (engl. domain-specific) tai ohjelmointikielellä.
Toimeenpano	Englanniksi *deployment*; sovellukseen kuuluvien ohjelmaosien siirtäminen niille tarkoitettuun toimintaympäristöön suoritettavaksi.
Toimeenpanon manifesti	Kuvaus halutunlaisesta sovelluksesta saatavilla olevia resursseja käyttäen (tämän tutkielman kokeellisen osion yhteydessä).
Toimeenpanon ratkaisu	Riippuvuuksien saatavuuden ja yhteensopivuuden varmistaminen sekä laitekohtaisten ohjeiden valmistelu sovellusta varten (tämän tutkielman kokeellisen osion yhteydessä).
Toimeenpanon sijoitus	Valmisteltua ratkaisua seuraten asianomaisten laitteiden konfigurointi tarvittavien ohjelmallisten osasten ja yhteyksien käyttämistä varten (tämän tutkielman kokeellisen osion yhteydessä).
WebAssembly	Alunperin WWW-selaimia varten kehitetty binääriformaatti, jota on mahdollista sekä kääntää että kutsua useasta eri ohjelmointikielestä.

Kuviot

Kuvio 1. Kontit verrattuna virtuaalikoneisiin 7
Kuvio 2. Kubernetesin komponenttien jaottelu 12
Kuvio 3. Ehdotettu isomorfinen IoT-arkkitehtuuri 18
Kuvio 4. Artefaktin *looginen näkymä* luokkakaaviona 21
Kuvio 5. Yleisnäkymä artefaktin keskenään kommunikoivista tehtävistä 24
Kuvio 6. Sekvenssikaavio laitteiden terveyskyselyiden tekemisestä 25
Kuvio 7. Kommunikaatiokaavio toimeenpanon tapahtumista 26
Kuvio 8. *Orkestraattorin* kehitysnäkymäkuvaus komponenttikaaviona 27
Kuvio 9. *ReSTful API*:n kehitysnäkymäkuvaus komponenttikaaviona 28
Kuvio 10. Sarja kuvankaappauksia artefaktin toimintaa demonstroivasta sovelluksesta ... 31

Taulukot

Taulukko 1. Artefaktissa käytetyt teknologiat (1/2) 50
Taulukko 2. Artefaktissa käytetyt teknologiat (2/2) 51

Sisällys

1 JOHDANTO 1

2 TAUSTAA 3
2.1 Reunalaskenta 3
2.2 Ohjelmistokontit 6
2.3 Orkestraatio 9
2.3.1 Peruspiirteet ja -ominaisuudet 9
2.3.2 Kubernetes 10
2.3.3 Orkestraatio reunalaskennassa 12
2.4 Liukkaat ohjelmistot 13
2.4.1 Määritelmä 13
2.4.2 Teknologisia lähestymistapoja - isomorfisuus ja WebAssembly 15

3 TUTKIMUSASETELMA 17
3.1 Tutkimuskysymykset 17
3.2 Tutkimusmenetelmä 19

4 ARTEFAKTI 20
4.1 Ominaisuudet 20
4.1.1 Ohjelmistoyksiköt ja toimeenpano 20
4.1.2 Monitorointi ja ylläpito 23
4.1.3 Dynaamisuus ja liukkaus 23
4.1.4 Käyttöliittymä ja rajapinnat 23
4.2 Samanaikaisuus 24
4.3 Ohjelmistomoduulit 25
4.3.1 Orkestraattori 26
4.3.2 ReSTful API 28
4.3.3 Resurssihakemisto 29
4.4 Käyttöönotto 29
4.5 Demonstraatio 30

5 POHDINTA 32
5.1 Tutkimuskysymyksiin vastaaminen 32
5.2 Kritiikki ja jatkotutkimuskohteita 35

6 YHTEENVETO 39

LÄHTEET 40

LIITTEET 48
A Moduulikuvasdokumentin esimerkki 48
B Toimeenpanon manifestin esimerkki 49
C Tutkielman ulkoiset teknologiat 50

1 Johdanto

Esineiden internetin tai IoT:n (engl. Internet of Things) ja ihmisten arjelle läpitunkevan laskennan myötä kasvaa kehittäjäystävällisten teknologioiden tarve yhä enemmän. IoT-ympäristö on nykypäivän korkealla tasolla toimimaan tottuneille ohjelmistokehittäjille haasteellinen ja ala vaatii yhä monimuotoisempaa osaamista. Helpotusta tässä ympäristössä kohdattaviin haasteisiin voisi tuoda ohjelmistokontteja (engl. container) mukailevan kevyen virtualisointitekniikan käyttäminen nykypäivän aiempaa suorituskykyisemmillä IoT-laitteilla. (Mikkonen, Pautasso ja Taivalsaari 2021)

Kontit ja niitä paljolti soveltavat pilvipalvelut tukevat monenlaisilta kanteilta hyödyllisiksi katsottuja ohjelmistoarkkitehtuurien ja -hallinnoinnin ominaisuuksia. Sovellusten suorittamiseen tarvittavat yhdenmukaistavat ympäristöt voidaan esimerkiksi konttien avulla toteuttaa virtuaalikoneita kevyemmin. Erityisesti yhdenmukaisuus tukee tässä yhteydessä ohjelmistokehittäjien työtä mahdollistamalla sekä saumattoman kehittäjienvälisen yhteistyön, että pitkälle jalostettavan automatisaation sellaisten toistuvien hallintatoimien kuten päivitysten tai laskentaresurssien varaamisen suhteen. (Casalicchio ja Iannucci 2020; Bernstein 2014; Dua, Raja ja Kakadia 2014)

Koska IoT-laitteet ovat erilaisiin käyttötapauksiinsa erikoistuneina luontaisesti keskenään heterogeenisiä, voitaisiin lisäksi koettaa kääriä alustojen fyysiset ominaisuudet, kuten kamerat, sensorit tai grafiikkasuorittimet, yhteiseen korkeamman tason rajapintaan. Kaikenlaisesta yhtenäistämisestä huolimatta fyysisesti kauemmaksi pilven laskentakeskuksista ja lähemmäksi verkon reunaa sijoittuvien laitteiden valtava lukumäärä on toinen ratkaistava haaste.

Laajan ja hajautetun järjestelmän rutiinitoimenpiteiden automatisointi voi olla jopa ehdotonta sovellusten ylläpidolle. Tällainen pilvilaskennassa kitevöitynyt *orkestraatioksi* (engl. orchestration) nimitetty toiminta voidaan tunnistaa yhteiseksi tekijäksi tukemaan sekä IoT:n että sovellusten liikkuvuuden lupaamia innovaatioita ja käyttäjäkokemusta. Orkestraatio hakee nyt omaa muotoaan verkon reunalla ja heterogeenisissä laiteympäristöissä. (Pan ja McElhannon 2018; Vaño ym. 2023; Taivalsaari ja Mikkonen 2017).

Tässä tutkielmassa esitellään toteutus orkestraatiota tukevasta ohjelmistoartefaktista, joka

osana suurempaa järjestelmää on tarkoitettu mahdollistamaan joitakin IoT:stä ja ohjelmistojen liikkuvuudesta johdettuja yksinkertaisia käyttötapauksia. Artefakti on kehitetty perustuen aiempaan isomorfisen (engl. isomorphic) ohjelmakoodin ja liukkaiden ohjelmistojen (engl. liquid software) visiosta kumpuavaan arkkitehtuurisuunnitelmaan ja teknologisiin valintoihin (Kotilainen ym. 2022; Kotilainen ym. 2023).

Tutkielma alkaa Luvusta 2, jossa esitellään taustakirjallisuuteen perustuen tutkielman aiheen ympäristöä ja käsitteitä. Luvussa 3 eritellään tutkielman tarkoitus ja tutkimuskysymykset sekä -menetelmä. Luvussa 4 kuvataan tutkielmassa kehitetyn ohjelmistoartefaktin arkkitehtuuri, teknisiä toteutusyksityiskohtia ja toimintoja. Luvussa 5 tarkastellaan kuinka artefaktin toteutus vastaa esitettyihin tutkimuskysymyksiin sekä esitetään kritiikkiä ja jatkotutkimuskohteita. Luku 6 nitoo lyhyesti tutkielman sisällön yhteen.

2 Taustaa

Tässä luvussa käydään läpi tutkielmalle keskeisten käsitteiden ja tekniikoiden taustaa ja määritelmiä perustuen kirjallisuuteen. Luvussa esitellään tutkielman näkökulmasta pilvilaskentaan ja orkestraatioon liittyviä aiheita kuten reunalaskentaa ja ohjelmistokontteja, sekä avataan käsitettä saumattomasta laskennasta ja liukkaiden ohjelmistojen visiosta. Taustakirjallisuutta käytetään käsitteistön määrittelyn lisäksi myös perustelemaan tutkielman tavoitteita.

2.1 Reunalaskenta

Reunalaskennan (engl. edge computing) määrittelemiseksi on sopivaa avata pienoisesti käsitteen muodostumisen historiaa ja taustalla olevia vaikutuksia. Lyhyesti voitaisiin sanoa, että reunalaskennan käsite muodostui johtuen epäyhteensopivuuksista pilvilaskennan ja IoT:n välillä.

Pilvilaskenta (engl. cloud computing) mullisti ohjelmistopalveluiden toiminnan sovelluskeskittyneillä palvelumalleilla ja skaalautuvalla *elastiseksi* kutsutulla infrastruktuurilla. Pilvilaskennan hyötyjen keskiössä on Armbrustin ym. (2010) mukaan elastisuus ja hetken mukaisiin tarpeisiin vastaaminen huomioiden sekä tarvittavien laskentaresurssien määrän että niiden käytöstä laskuttamisen. Taloudellisiin näkökulmiin kytkeytyen eroaa pilvilaskenta perinteisestä datakeskusasetelmasta heidän mukaansa kolmella tavalla: 1) laskentaresurssit näyttäytyvät käyttäjille (so. ohjelmistotuotteen kehittäjille) äärettöminä ja välittömästi saatavina, 2) käyttäjiä ei (yleensä) ole sidottu sopimuksella vain tiettyyn määrään resursseja ja 3) resurssien käytöstä voidaan maksaa erittäin joustavasti kulutuksen mukaan. Myös Pan ja McElhannon (2018) tunnistavat nämä tekijät. Pilvilaskenta koottuna tarjoaa siis maailmanlaajuisesti ulottuvaa ja yhdenmukaisena näyttäytyvää alustaa ohjelmien suorittamiseen. Se tukee ohjelmistopalveluiden kehittäjiä ulkoistamalla laitteistoon ja saatavuuteen liittyviä hallinnointikysymyksiä pilven ylläpitäjille.

Tietoteknisen kehittymisen mukana kuitenkin nousevat sekä laitteiden lukumäärä että niiden erilaiset käyttötarkoitukset. Kuluttamisen sijaan verkon reunalla olevat laitteet nyt myös tuottavat dataa (Shi ja Dustdar 2016) ja Cisco (2015) oli arvioinut, että vuoteen 2019 men-

nessä vuosittain datakeskuksiin kulkevan liikenteen määrä olisi vain murto-osa kaikesta siitä datasta, jota IoT-ympäristön laitteet samalla tuottaisivat ($\frac{10.4}{507.4}$ tsettatavua). Pilvilaskennan käyttö ei Shin ja Dustdarin (2016) mukaan tule olemaan uskottavaa IoT:n sovelluksia varten: tietoliikenneverkon hitaammin kasvava kapasiteetti on pullonkaula, vaikka pilvessä laskentaresurssit olisivatkin "äärettömiä", eikä viive tue reaaliaikaisuutta vaativia IoT-sovelluksia vaikkapa älykkäästä liikenteestä (Shi ym. 2016). Teknisten haasteiden lisäksi Shi ja Dustdar (2016) esittävät pilvilaskennan olevan ongelmallinen siirrettävän ja varastoitavan datan yksityisyyden suhteen.

Näihin pilvilaskennalle haasteellisiksi ennustettuihin tapauksiin vastaamiseksi on muodostunut hierarkkinen ajatusmalli erityisestä teknologiasta pilven ja verkon reunan välillä niin kutsutulla pilvi-reunajatkumolla. Reunalaskennan määritelmä on sijainniltaan pilven ja datalähteiden kuten IoT-laitteiden keskivälillä tapahtuvaa laskentaa, jonka Shi ym. (2016) tarkentavat käsittämään erityisesti niitä "teknologioita", jotka tätä laskentaa tukevat. Reuna heidän mukaansa käsittää siis datalähteiden ja pilven välillä olevia laskenta- tai verkkoresursseja, kuten älypuhelimet sykemittareiden lukemiseen, yhdyskäytävät älykodin tarkkailemiseen tai pikkupilvet (engl. cloudlet) mobiililaitteiden laskennan tehostamiseen. Tullee erikseen mainita, että IoT:n kuuluessa reunalaskennan piiriin, sen mukana tulevat myös erittäin rajoittuneet laitteet ja sensorit, jotka voivat vaatia erityishuomioita sovellusten suunnittelussa.

Reunalaskentaa varten on esitetty erilaisia ratkaisuja ja arkkitehtuurimalleja, joiden pohjalta Shi ym. (2016) avaavat reunalaskennan käsitettä. Pohjustavissa esityksissä on yhteistä se, että pilvilaskennasta tuttuja ideoita laskennasta, varastoinnista ja verkkotoiminnasta tuodaan lähemmäs IoT-laitteita niitä avustamaan (Pan ja McElhannon 2018; Cao ym. 2020).

Pilvi-reunajatkumosta puhuttaessa voi sekaannusta aiheuttaa sumulaskennan käsite, joka on kokonaisvaltaisesti verkon hierarkkisen infrastruktuurin huomioonottava arkkitehtuurimalli (Bonomi ym. 2012). Sumu- ja reunalaskennan käsitteet voivat keskenään sekoittua, mutta Shi ym. (2016) täsmentävät niiden eroavan toisistaan siinä, että sumulaskenta keskittyy kookkaampaan infrastruktuuriin ja reunalaskenta aiemmin esitetyn määritelmän mukaisesti enemmän verkon reunalle kytkettyihin teknologioihin, laitteisiin tai "juttuihin" (engl. things). Tämän perusteella voitaisiin kai ajatella sumu- ja reunalaskennan eron olevan myös abstraktiotasoon perustuva ja sumulaskenta olisi siis tässä tasossa korkeammalla asteella:

sumulaskennan käyttäminen vaatisi vähemmän tietämystä pienistä teknologisista yksityiskohdista ja reunalaskennalle päinvastoin.

Reunalaskentaan Pan ja McElhannon (2018) yhdistävät lisäksi ajatukset laskennan purkamisesta (engl. offloading) pilveen tai kyberpoiminnasta (engl. cyber foraging). Nämä ajatukset vaikuttaisivat olevan muita edellä mainittuja esityksiä (Pan ja McElhannon 2018; Cao ym. 2020) vähemmän konkreettisia tai arkkitehtonisia ja enemmänkin ilmentymiä reunalle erityisominaisista toiminnoista. Täten ei vaikuta mielekkäältä lukea niitä suoranaisesti osaksi reunalaskennan määritelmää, vaan ne enemmänkin osoittanevat, millaisia tehokkuutta tarkastelevia ajatuksia heterogeeninen IoT ja tiheät laitteiden verkostot luontaisesti tuovat esiin.

Reunalaskennan määritelmään sopivina sovelluskohteina on nähty muun muassa videosisältöjen analysointi, viihde, älykoti sekä älykkäät kulkuneuvot ja liikenne (Bonomi ym. 2012; Shi ym. 2016). Esimerkiksi terveydenhuollon kontekstissa ajallisesti kriittisten sovellusten viiveen ja siten käytäntöön omaksumisen parantamiseksi sumulaskennassa älykäs automaattinen päätöksenteko on Shuklan ym. (2019) simulaation valossa näyttänyt lupaavalta. Myös Mutlag ym. (2020) näyttävät tutkineen samaa aihealuetta. Koska IoT-sovellukset luottavat useiden erilaisten laitteiden harmoniseen ja saumattomaan yhteistyöhön kauempana suuren laskentakapasiteetin ja säilytystilan pilvipalveluista, IoT-järjestelmän yhteydessä korostuu huolellinen valmistelu ja kunnossapito niin ennen toimeenpanoa kuin toiminnan aikanakin. Muutamat lähteet (Pan ja McElhannon 2018; Vaño ym. 2023; Taivalsaari ja Mikkonen 2017) mainitsevat muun muassa “orkestraatioksi” kutsutun toiminnan oleellisuudesta IoT:lle ja sitä ympäröivälle reunalaskennalle.

Datan turvallisuudesta puhuen Shi ja Dustdar (2016) antavat ymmärtää, että asia näyttäisi reunalaskennassa olevan pelkkää pilveen siirtoa ja varastointia moniulotteisempi. Moniulotteisuus heidän esittämänään käsittää muun muassa laitteiden halpuuden, lukumäärän ja heterogeenisyyden sekä liikkuvuudesta johtuvan verkon dynaamisuuden. Lieneekin itsestään selvää, että nämä tekijät luovat huomattavaa haastetta laskentaympäristöstä päättelemiseen ja siten myös sen turvaamiseen. Avoimina kysymyksinä reunalaskennalle Shi ja Dustdar (2016) esittävät muun muassa ohjelmoitavuuden kaksijakoisuuden pilven ja reunan kesken, manuaalisen säätötyön vaativuuden, sekä verkon dynaamisen topologian hallinnoinnin. Pan ja McElhannon (2018) esittävät dynaamisuuden vahvistamisen olevan avainasemassa, kun

halutaan tukea IoT-sovelluksia reunan ja pilven kautta. Dynaamisuutta edustavia teknologisia toimia ovat heidän mukaansa verkon virtualisointi, IoT-tarkoituksiin abstrahoitu ja automaattinen orkestraatio sekä dynaaminen laskennan purkaminen.

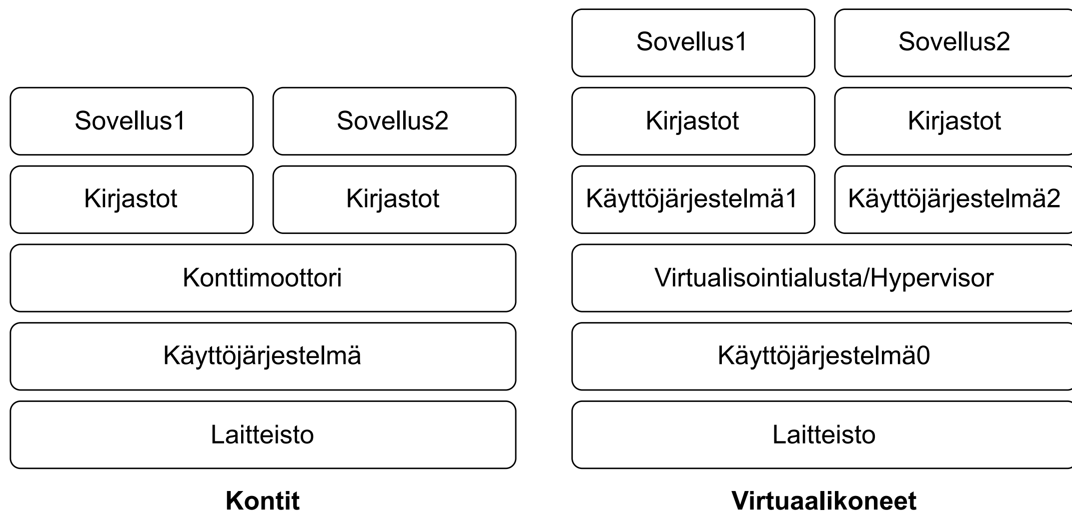
2.2 Ohjelmistokontit

Erilaiset tavat *virtualisaatioon* eli saatavilla olevien resurssien jonkinlaiseen eristykseen ja hallinnointiin on pilvilaskennan ja sen tuomien etujen kuten skaalautuvuuden ja elastisuuden keskiössä (Bernstein 2014). Erityistä *virtualisointialustaa* tai *hypervisoria*, voidaan käyttää luomaan toisistaan erillisiä virtuaalisia palvelimia, joiden kunkin käyttöastetta voidaan kontrolloida sekä paremmin piilottaa näkymä käyttöjärjestelmätoimintoihin ja fyysiseen laitteistoon (De Gelas ja ESX, n.d.). Toinen yleinen teknologia laskennan eristämiseen ja resurssien jakamiseen ovat ohjelmistokontit (engl. container) (Bernstein 2014).

Toteutuksessaan virtuaalikoneiden ja konttien voitaisiin sanoa eroavan siinä abstraktiotasoon kytkeytyvässä näkökulmassa, jossa niiden tarjoamia toimintoja on lähdetty suunnittelemaan. Virtuaalikoneissa nimensä mukaisesti tarjotaan kokonaista fyysistä tietokonetta emuloivaa alustaa, jolle käyttöjärjestelmä voidaan asettaa, kun taasen konteissa tämä alusta on käyttötarkoitukseen mukautettava ja paketoitava “kevyt” käyttöjärjestelmä. Tätä oleellista eroa konttien ja virtuaalikoneiden välillä on kuvattu Kuviossa 1. (Dua, Raja ja Kakadia 2014; Bernstein 2014)

Konttien avulla erillisten virtuaalikoneiden tai kokonaisten käyttöjärjestelmien sijaan kukin laskentayksikkö käyttää yhtä ja samaa käyttöjärjestelmää, minkä ansiosta konttien käynnistysaika ja koko ovat huomattavasti pienemmät. Tämä keveys erityisesti mahdollistaa suurten palvelinkoneiden käytännössä äärettömän skaalautumisen lukuisiin palveluinstansseihin ja siten palvelun parempaan saatavuuteen. Muita konttien etuja ovat siirrettävyys sekä erikoistuneiden ohjelmistojen tukemana hajautettujen sovellusten helpompi hallinnointi. (Casalicchio ja Iannucci 2020; Bernstein 2014; Dua, Raja ja Kakadia 2014)

Felterin ym. (2015) mukaan kontit voivat olla virtuaalikoneita myös huomattavasti tehokkaampia laskennan nopeudessa. Heidän kokeeessaan suoritusteho natiiviin verrattuna oli virtuaalikoneella noin 40% huonompaa kuin konteilla, vaikkakin konttien teho vaihteli luku- ja



Kuvio 1. Kontit verrattuna virtuaalikoneisiin (mukailtu (Bernstein 2014)). Kontit jakavat keskenään yhden (fyysisen tai virtuaalisen) alustan (kuvassa "Konttimoottori"), kun taasen virtualisointialusta toisintaa tämän jokaiselle virtuaalikoneelle.

kirjoitusasetuksista riippuen. Felter ym. (2015) esittävätkin, että CPU:n tai muistinkäytön sijaan luku- ja kirjoitusoperaatiot ovat syypäänä eroon ja kontit **oletusarvoisesti eivät** välttämättä olisikaan perinteistä virtualisointia nopeampia: virtuaalikonetta (tässä tapauksessa KVM[1]) on vain huomattavasti vaikeampi säätää ja optimoida haluttua työtä varten. Näitä näkemyksiä tukee myös Casalicchion ja Iannuccin (2020) selvitys virtuaalikoneita ja kontteja suorituskyvyn puolesta vertailevista tutkimuksista, jossa he lisäävät, että konteissa verkkotoiminnan kirjoitus- ja lukuoperaatiot jopa kärsivät verrattuna virtuaalikoneisiin. Huomattavat suorituskykyparannukset vaikuttavat siis olevan konteissa selkeästi mahdollisia, mutteivät ehkä niiden käyttämisen pääasiallinen tekijä.

Konttien kehityshistoria perustuu erilaisiin Linux-käyttöjärjestelmäperheestä löytyviin komentoihin ja käsitteisiin, jotka eristävät käyttöjärjestelmän osia prosesseilta. Tämä kokoelma lopulta kiteytyi Linux-ytimen (engl. kernel) eristystoiminnoiksi (*Linux Containers*[2], LXC), jotka sisällytettiin käyttöjärjestelmän standardijakeluun (Bernstein 2014). Dua, Raja ja Kakadia (2014) erittelevät konttien yleiskuvauksen yhteydessä seuraavat teknologian perustaan kuuluvat komennot ja käsitteet:

1. https://linux-kvm.org/page/Main_Page (haettu 9.2.2024)
2. https://linuxcontainers.org/ (haettu 9.2.2024)

- `chroot` prosessin ja virtuaalisten ympäristöjen tiedostojärjestelmäpääsyn eristämiseen,
- `cgroups` resurssien kuten CPU:n tai muistinkäytön kontrolloimiseen ja
- `kernel namespaces` konttienväliseen eristämiseen esimerkiksi verkkonimiavaruuden tai levyn osioinnin suhteen.

Konttien käytön suosioon saattanut Docker[3] on kehittänyt LXC:n päälle ohjelmointirajapintoja (engl. Application Programming Interface, API) ja nimiavaruuksia (engl. namespace), jotka mahdollistavat yhä tarkkarajaisempaa hallintaa ja eristämistä käyttöjärjestelmästä. Lisäksi *Docker-järjestelmäkuvien*[4](lyhyesti *kuva*) muodossa konttien siirrettävyys paranee, kun kontin ominaisuudet voidaan dokumentoidusti uudelleenrakentaa perustuen kerroksittaisiin Dockerfile-komentosarjoihin tai *skripteihin*. Näissä skripteissä kerroksia kasataan jonkin valitun pohjakuvan päälle komennoilla ja argumenteilla, joilla muodostetaan sovellukselle sopiva ja yhtenäinen ympäristö paketteineen ja asetuksineen. (Bernstein 2014)

Docker kontin elinkaari voidaan kuvata kuudella vaiheella (“The 6 steps of the container lifecycle - Cloud computing news”, n.d.; Casalicchio ja Iannucci 2020):

- *Hankkiminen* - Toiminnallisuuksien ja sisällön kartoittamista ja valittujen kuvien lataamista konttivarastosta (engl. container repository).
- *Rakentaminen* - Hankittujen kuvien koostamista kokoon uudeksi haluttua sovellusta ilmentäväksi kuvaksi.
- *Toimittaminen* - Rakennetun sovelluksen tyypillisesti automatisoiduin keinoin siirtäminen järjestelmään, jossa sovelluksen on lopulta tarkoitus toimia. Tämä vaihe voi esimerkiksi sisältää kuvalle tehtäviä turvallisuustarkastuksia.
- *Toimeenpaneminen* - Olemassaolevan sovellusinstanssin korvaaminen uudella toimitetulla versiolla.
- *Ajaminen* - Kuvasta käynnistettävän kontin hallinnointistrategioiden ja ympäristön asettaminen huomioiden sovelluksen skaalaamisen, virheistä palautumisen ja kommunikaatioyhteydet.
- *Ylläpitäminen* - Sovelluksen tilan ja muutosten monitorointia, jossa määritetyt tapah-

3. https://www.docker.com/ (haettu 9.2.2024)
4. https://docs.docker.com/get-started/#what-is-an-image (haettu 9.2.2024)

tumat voivat laukaista tilannetta korjaavia reaktiivisia toimenpiteitä.

2.3 Orkestraatio

Sovelluksen koostaminen useista ja mahdollisesti hajautetuista ohjelmistopalasista kasvattaa järjestelmän monimutkaisuutta ja automatisaation merkitystä kokonaisuuden hallinnassa. Orkestraatio tarkoittaa ohjelmistopalvelun tai sovelluksen ylläpitotoimien ja työnkulkujen kokonaisvaltaista automatisointia. Sen avulla monimutkaiset ja hajautetut palvelut voidaan pitää toiminnassa vähäisellä tarpeella ihmisen väliintulolle. Ihmistä luonnollisesti kuitenkin tarvitaan palvelun halutun tilan tai toisin sanoen **sovelluksen** kuvaamiseen, joka toimeenpanon (engl. deployment) ja orkestraation kannalta voidaan tehdä erilaisin tavoin.

2.3.1 Peruspiirteet ja -ominaisuudet

Ranjan ym. (2015) määrittävät pilvilaskennan yhteydessä "resurssien orkestraatioksi" ne operaatiot, joita laskentaresurssien ja niillä suoritettavien sovellusten ylläpitäjät käyttävät laitteistojen ja ohjelmisto-osasten valintaan, toimeenpanoon, monitorointiin ja dynaamiseen asetusten hallitsemiseen. Dynaamisuuden vuoksi he nostavatkin esiin, että **manuaaliseen** konfigurointiin perustuva orkestraatio ei ole riittävää pilven autonomisuudesta ja pilvipalveluiden heterogeenisyydestä johtuen.

Red Hat, Inc. (2022) mukaan "konttiorkestraatio" on juurikin automatisaatiota, johon sisältyy, Ranjanin ym. (2015) listaamia operaatioita tarkentaen, myös verkkoyhteyksien muodostaminen. Red Hat, Inc. (2022) mukaan orkestraatio on tehokasta kun orkestroitava sovellus koostuu mikropalveluarkkitehtuurin (Namiot ja Sneps-Sneppe 2014) mukaisesti yhden vastuualueen tai tehtävän täyttävistä ja hyvin määriteltyjä rajapintoja tarjoavista palveluista.

Google on aktiivisesti kehittänyt joitakin erilaisia orkestraatiojärjestelmiä käyttöönsä (Burns ym. 2016), joiden yhteydessä tunnistettiin keskeisiksi muun muassa resurssivaatimusten ennustaminen, palveluiden etsintä, kuorman tasaaminen ja skaalaus. Nämä erilaiset konttien orkestraatioon kehitetyt järjestelmät erikoistuivat lopulta liikaa tehtäviinsä ja niiden ongelmia haluttiin lähteä korjaamaan korostaen kehittäjäystävällisyyttä. Samalla kun hyödynnettiin konttien mahdollistamaa tehokasta käyttöastetta, haluttiin paremmin tukea kehitetyille

järjestelmille ominaisten ryppäissä (engl. cluster) ja hajautetusti suoritettavien sovellusten kirjoittamista.

Khanin (2017) mukaan konttiorkestraatioalustalta vaaditaan olennaisesti järjestelmän tilan hallintaa, korkean saatavuuden ja virheensiedon takaamista, turvallisuutta, verkkotoiminnan yksinkertaistamista sekä tukea palveluiden löytämiselle, jatkuvalle ohjelmistokehitykselle ja monitoroinnille. Truyen ym. (2019) vertailevat avoimen lähdekoodin konttiorkestraatiokehyksiä pilvikontekstissa. Kehyksille tunnistetuista yhteisistä 124:stä ja ainutlaatuisista 54:stä ominaisuudesta he kokoavat yhdeksän korkean tason toiminnallista piirrettä. Näitä yhdeksää piirrettä voitaisiin ali-piirteittensä ja ominaisuuksien perusteella lyhyesti kuvata seuraavasti:

- Järjestelmän arkkitehtuuri, konfigurointi- ja asennusmenetelmät.
- Kehyksen mukauttamismahdollisuudet.
- Konttienvälisen ja ulospäin ohjautuvan verkkotoiminnan dynaamisuus ja ylläpito.
- Sovellusten paketointi-, toimitus- ja siirrettävyysmallit (kontit).
- Laskentaresurssien kiintiöiden hallinta (so. paljonko kukin kontti saa käyttöönsä).
- Konttien palvelunlaadun hallinta (so. reagointi vaihteleviin resurssitarpeisiin).
- Järjestelmään pääsyn ja kommunikaation tietoturva.
- Datan ja ohjelmistojen suojaaminen sekä konttien eristäminen.
- Järjestelmän luonti-, tarkastelu-, huolto- ja hallinnointimenetelmät.

Näiden määritelmien perusteella orkestraatiota joudutaan kovin laajapiirteisesti kuvaamaan ohjelmistopalvelun tai sovelluksen ylläpitotoimien ja tukevien työnkulkujen ohjaamisen kokonaisvaltaisena automatisointina.

2.3.2 Kubernetes

Orkestraatiota nykyisessä ohjelmistokontekstissa keskeisesti ilmentää Googlen vuosikymmenten kehitys- ja kokemustyönä muodostunut *Kubernetes*[5] (Burns ym. 2016). Truyenin ym. (2019) vertailemista orkestraatiokehyksistä juuri Kubernetes osoittautui ominaisuuksiltaan ylivoimaiseksi sekä luovuuden (eli tuottanut eniten uusia) että laajuuden (eli sisältää eniten) puolesta. Kubernetes onkin konttiorkestraation niin sanottu de facto tai standardityö-

5. https://kubernetes.io/ (haettu 18.11.2023)

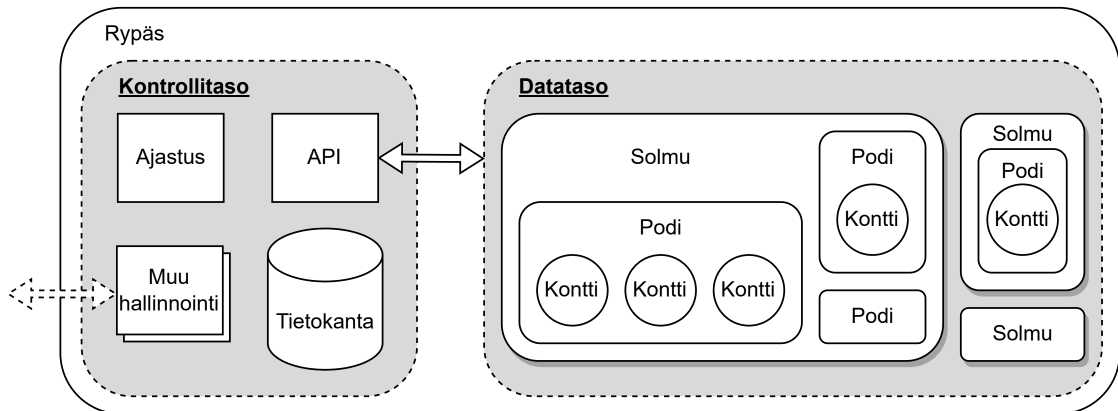
kalu (Vaño ym. 2023; Kayal 2020), ja voitaisiin sanoa Kubernetesin ilmentävän nykyisen orkestraation keskeisimpiä ominaisuuksia. Alaluvussa 2.3.1 esiteltiin orkestraatioon yhdistettäviä yleisominaisuuksia, joita konkreettisempia "Kubernetesin resurssienhallinnan" ominaisuuksia ovat Carriónin (2022) listaamana seuraavat:

- Ajastaminen (engl. scheduling) - Konttien sijoittamistavat solmuille.
- Pääsyn kontrollointi - Käyttäjän resurssienkäytön rajoittaminen.
- Kuormantasaus - Laskennan purkaminen ja tasaaminen useammille solmuille.
- Terveyskysely - Kontin pyyntöjenkäsittelyn tarkkailu.
- Virheensieto - Konttien sopivan lukumäärän määrittäminen ja ylläpito.
- Autoskaalaus - Resurssienkäytön kohdistaminen tarpeiden mukaan.

Kubernetesista kuten toisistakin orkestraatiojärjestelmistä voidaan havaita käsitteellinen jaottelu kontrollitasoon (engl. control plane) ja työläissolmuista koostuvaan datatasoon (engl. data plane) (Carrión 2022; The Kubernetes Authors 2023a). Tämä komponenttien jaottelu on esitetty Kuviossa 2. Kubernetesin (The Kubernetes Authors 2023a) oman määritelmän mukaan kontrollitasolla koordinoidaan fyysisiä laitteita edustavista *solmuista* (engl. node) ja niissä suoritettavista konttijoukoista tai *podeista* (engl. pod) koostuvaa *rypästä*. Rypäs on siis joukko laitteita, joille sijoitetaan suoritettavien konttien kokoelmia. Kontrollitaso on selkeästi orkestraatioksi nimitettävän toiminnan keskiössä, mutta se toki vaatii ennalta määritettyä yhteistoimintaa niiltä yksiköiltä, jotka orkestroitavan sovelluksen osasia lopulta suorittavat (so. ryppään laitteet). Datatasoon (Carrión 2022) kuuluvat siten nämä sovellustoimintaa ilmentävät solmut, joilla olevat ohjelmistokomponentit pitävät huolta podien ylläpidosta niille (so. podeille) asetettujen tehtävien, suorituskyvyn ja kommunikaatioyhteyksien suhteen.

Kubernetesin kontrollitasolle sijoittuva orkestraatioalgoritmi perustuu Kayalin (2020) esittämänä oleellisesti kahteen komponenttiin: *kube scheduler* -ajastin ja konttienvälinen verkkotoimintamalli. Ajastinkomponentti automaattisesti valitsee parhaiten soveltuvan solmun uusien tai vielä suorittamattomien podien sijoittamiseen, ottaen huomioon podien erilaiset vaatimukset mm. laitteiston, ohjelmistojen, käyttöehtojen (engl. policy) tai datan läheisyyden suhteen (The Kubernetes Authors 2023b). Podille parhain solmu valitaan kahdessa vaiheessa ensin suoraviivaisesti vaatimusten mukaan mahdolliset vaihtoehdot suodattamalla (engl. filtering), ja sitten tarkemmin solmuja niiden suoriutumiskykyä mittaavalla funktiolla pis-

teyttäen (engl. scoring) (Kayal 2020; The Kubernetes Authors 2023b).



Kuvio 2. Kubernetesin komponenttien jaottelu (mukailtu (Carrión 2022; The Kubernetes Authors 2023a))

2.3.3 Orkestraatio reunalaskennassa

Orkestraatioteknologioiden voisi olettaa sopivan hyvin reunalaskennan ja IoT:n piiriin, sillä esimerkiksi Kubernetes Mcluckien (2014) esittelemänä tukee erityisesti sekä fyysisten resurssien abstrahointia että konttien vahvuutta hajautetuissa järjestelmissä. Teknologian raskaus voi kuitenkin olla merkittävä haaste reunalaskennan kontekstissa ja esimerkiksi Vanõn ym. (2023) mukaan Kubernetesin soveltaminen **sellaisenaan** IoT:n rajoittuneiden laitteiden ympäristöön ei vaikuta realistiselta. He listaavatkin useita kevyempiä muunnelmia ja mukaelmia, joita Kubernetesista tai sen inspiroimana on tehty reunalle soveltuvaa orkestraattoritoteutusta varten. Vaño ym. (2023) perustavat kartoituksensa erityisesti reunalaskennalle tyypillisen laitteiston suorituskykyyn. Heidän mukaansa suosittu lähestymistapa ongelmaan on pyrkiä vähentämään Kubernetesin ja konttien suorittamisesta johtuvaa muistinkulutusta sekä ympäröivien ohjelmien kokoa, mutta samalla, kuten Docker, pysyä Kubernetes-yhteensopivana muodostaen "monirypäsarkkitehtuurin" (engl. multi-cluster architecture). Yhteensopivuuden tarkoituksena vaikuttaisi siis olevan datatason Kubernetes-komponenttien korvaaminen kevennetyillä versioilla, jolloin ei tarvittaisi kokonaisvaltaisesti uudelleentyöstettyä orkestraattoria. Tämän lähestymistavan vastapainoksi Vaño ym. (2023) tunnistavat myös Kubernetesin periaatteita mukailevia, mutta selkeämmin itsenäisiä ja reunalle erikoistettuja ratkaisuja. Reunalaskennan yhteydessä erikoistaminen vaikuttaa sopivalta, koska he-

terogeenisten laitteiden ympäristö osoitetusti eroaa pilvilaskennasta ja on vaatimuksiltaan eri.

Kayalin (2020) mukaan ongelmana Kubernetesin reunayhteensopivuudessa taasen on esimerkiksi *kube scheduler* -oletusajastinalgoritmin liiallinen kapeakatseisuus. Tätä hänen mukaansa edustaa että *kube schedulerissa* solmun valintaa ei tehdä koko ryppään huomioonottaen ja jää siten epäoptimaaliseksi, sekä että laitteiston suorituskykyä pisteyttäessä verkkotoiminnan latenssi, kaistankäyttö ja topologia jäävät huomiotta. Kokeessaan Kayal (2020) vastaa tunnistettuihin haasteisiin 1) ajastamalla (engl. scheduling) ja sijoittamalla eniten **sisäistä** kommunikaatiota käyttävät podit ensin; 2) pilkkomalla suodatuksen tuloksena mahdottomiksi merkityt podit pienemmiksi kokonaisuuksiksi, jotta niistä saadaan yhteensopivia solmuille; ja 3) pisteyttämällä solmuja myös verkkoalueen (engl. zone, region) läheisyyteen perustuen, jolloin saman palvelun podit ovat myös fyysisesti lähempänä toisiaan.

2.4 Liukkaat ohjelmistot

Tässä luvussa tarkastellaan ohjelmistojen liukkauden käsitettä sekä tätä mahdollisesti tukevia teknologisia ratkaisuja. Liukkaus johdetaan tässä sekä saumattoman laskennan käsitteestä (McCormack, Koontz ja Devaney 1999; Suh ym. 2008; Gogouvitis ym. 2020) että eräästä liukkaiden ohjelmistojen perustavanlaatuisten piirteiden julistuksesta (Taivalsaari, Mikkonen ja Systä 2014).

2.4.1 Määritelmä

Saumattoman laskennan (engl. seamless computing) voisi yleisesti sanoa tarkoittavan suoritettavan ohjelman liikkumista eri laitteiden välillä ilman, että ohjelman toiminta liikkumisesta johtuen häiritsevästi katkeaa. Tämä voi tarkoittaa muun muassa heterogeenisten supertietokoneiden yhtenäisiä käyttöliittymiä (McCormack, Koontz ja Devaney 1999) tai käyttäjän kokonaisen ohjelmatilan siirtymistä erilaisten laitteiden välillä (Suh ym. 2008). Nämä esimerkit viittaavat saumattomuuden keskittyvän erityisesti ohjelmien käyttökokemukseen.

Ohjelmien liukkaus (engl. *liquid software*, suomennos (Mikkonen 2023)) taasen näyttäisi olevan jonkinlainen korkeampi vahvuusaste saumattoman laskennan piirissä. Siinä data ja

logiikka edelleen liikkuvat saumattomasti laskentaympäristöstä tai laitteesta toiseen (Taivalsaari, Mikkonen ja Systä 2014), mutta vahvemmin korostaen tämän näkymättömyyttä ja helppoutta käyttäjälle. Ajatusta hieman raaemman käyttöesimerkin esittävät Suh ym. (2008), jossa videopelin suoritustila siirtyy USB-tikun mukana ja vaatii siis käyttäjältä yksinkertaista napin painallusta tai kosketusnäytön hipaisua huomattavasti enemmän vaivaa.

Taivalsaari, Mikkonen ja Systä (2014) listaavat liukkaiden ohjelmistojen vaatimukset seuraavasti:

1. Käyttäjien tulee voida vapaasti liikkua (engl. roam) laitteidensa käyttämisen välillä.
2. Laitteiden huoltamisen ja hallinnoinnin kitkaisuus tulee minimoida tai piilottaa käyttäjältä.
3. Käyttäjän sovellusten ja datan tulee synkronoitua laitteiden kesken huomioiden laitekohtaiset yhteensopivuudet.
4. Kun se on tarpeen, tulee liikkuminen laitteiden välillä sisältää sovelluksen koko tilan saumattoman siirtymisen, jolloin työn alla olevaa aktiviteettia voidaan suoraan jatkaa laitteella jonne siirrytään.
5. Laitteiden fyysisten rajoitteiden (esim. näyttö, syötelaitteet, tietoliikenneyhteydet yms.) puitteissa laitteiden välillä liikkumisen ei tule olla sidottu tiettyyn laitetarjoajaan tai ekosysteemiin.
6. Sovellusten ja datan liikkuvuus tulisi olla käyttäjän kontrollissa siten, ettei ole epäselvää mikä laite saa käyttää mitäkin toiminnallisuutta tai käsitellä mitäkin dataa.

Siispä laitteiden fyysiset ominaisuudet luonnollisestikin rajoittavat liikkuvuutta ja laitteet vaatisivat virtualisoidun ohjelmistoalustan, joka toimisi yhtenäistävänä abstraktiona niiden fyysisille ominaisuuksille sekä suorittaisi tilan (luultavasti jatkuvasti tapahtuvan) synkronoinnin näkymättömästi. Samaan aikaan heterogeenisyyden mukana tulevasta käyttötapauksiin erikoistumista ja tehokkuudesta ei tulisi tinkiä, koska se voisi vaikuttaa käyttökokemuksen sulavuuteen. Kun laite erikoistetaan sille tarkoitettuja toimenpiteitä varten, se toiminee paremmin sekä käyttökokemuksen että taloudellisuuden kannalta. Tämä näkyy esimerkiksi IoT-laitteissa joista yhdet ovat koneoppimistapauksia varten laitekiihdytettyjä [6], ja toiset

6. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/ (haettu 7.12.2023)

matalalla virrankäytöllä ja itsenäisiksi tarkoitettuja pitkäikäisiä sensoreita [7]. Liukkauden lisäksi myös yleisen saumattoman laskennan vaatimuksiin Gogouvitisin ym. (2020) mukaan kuuluu laitteidenvälisen heterogeenisyyden tukeminen. Kotilaisen ym. (2022) ehdotuksessa *isomorfista koodia* käytetään keinona IoT-laitteiden heterogeenisyyden häivyttämiseen.

2.4.2 Teknologisia lähestymistapoja - isomorfisuus ja WebAssembly

Ohjelmien liukkauden mahdollistajana ajatellaan olevan koodin isomorfisuus. Sen sijaan että ohjelmakoodin tasolla pitäisi huomioida suoritusalustan erityisominaisuudet tai -piirteet, Mikkosen, Pautasson ja Taivalsaaren (2021) mukaan isomorfisuus ja tarkemmin "dynaaminen isomorfisuus" tarkoittaa ajoympäristön yhteneväisyyttä tai että fyysinen alusta on virtualisoitu ohjelmakoodia varten. Mikkonen, Pautasso ja Taivalsaari (2021) esittävät olemassaolevina esimerkkeinä isomorfisuudesta Javan (Arnold, Gosling ja Holmes 2005) ja Squeak Smalltalkin (Ingalls ym. 1997) virtuaalikoneet sekä WWW-selaimista tutun ja palvelinpuolellekin siirtyneen Javascriptin[8]. Isomorfisuus parantaa luonnollisesti koodin uudelleenkäytettävyyttä mistä juuri Javascript kirjastojen käyttö sekä selaimessa että palvelimella esimerkiksi Node.js:n[9] muodossa on selvä esimerkki. Isomorfisuuden ominaisuudet ovat siten selkeästi **kehittäjäystävällisiä** ja lisäarvoa tuo liukkaus, joka taasen näyttäisi lupaavalta myös uusien ja innovatiivisten **käyttäjäkokemusten**, ohjelmistosovellusten ja -paradigman näkökulmasta (Taivalsaari, Mikkonen ja Systä 2014).

WebAssembly (W3C 2022) on WWW-selaimia varten kehitetty ja edelleen aktiivisessa kehityksessä oleva binääriformaatti, joka alhaisen abstraktiotasonsa ansiosta sopii kiihdyttämään muutoin hitaammilla tulkituilla kielillä kirjoitettuja ohjelmia. Lisäksi ohjelmistokehittäjän näkökulmasta kiinnostavasti WebAssembly toimii käännöskohteena useille eri ohjelmointikielille[10] ja myös siis rajapintana kielten välille. WebAssembly voidaan nähdä myös eräänlaisena ohjelmointikielen ajoympäristöön perustuvana kevyenä konttiteknologiana (Mäkitalo ym. 2021). WebAssemblyn ja konttien vastaavuutta edustavan näkemyksen näyttäisi tunnustavan myös konttiteknologian suosiosta nauttinut Docker julistaessaan beta-version integraa-

7. https://store.arduino.cc/products/arduino-mkr-env-shield-rev2 (haettu 7.12.2023)
8. https://developer.mozilla.org/en-US/docs/Web/javascript (haettu 7.12.2023)
9. https://nodejs.org/en (haettu 7.12.2023)
10. https://www.fermyon.com/wasm-languages/webassembly-language-support (haettu 7.12.2023)

tiosta WebAssemblyn kanssa (Irwin 2022).

Mäkitalo ym. (2021) tarkastelevat WebAssemblya erityisemmin IoT:n näkökulmasta. Heidän nähdäkseen WebAssembly pienen koon, turvallisuuden, modulaarisuuden ja miltei natiivin suorituskyvyn vuoksi vaikuttaa kiinnostavalta WWW-teknologian ja IoT:n yhdistämiseen. Myös Mikkonen, Pautasso ja Taivalsaari (2021) katsovat, että WebAssembly tehokuudellaan olisi varteenotettava vaihtoehto isomorfisuuden tuomiseksi IoT-laitteille. WebAssemblyn moduulien hiekkalaatikkomaisuuden ja alustan ominaisuuksien poisrajaamisen pohjalta Mäkitalo ym. (2021) vetävät yhteyksiä kontteihin ja dynaamisten ohjelmistomuutoksien helpottamiseen.

Konttien eristyksen ominaisuuksia WebAssemblyssä ilmentävät *WebAssembly-moduulit* ja ajoympäristöjen (engl. runtime) resurssipääsyn kykenevyyssuuntautuneisuus (engl. capability-oriented) WebAssembly System Interface (WASI) määrittelyyn (Bytecode Alliance contributors 2019) perustuen. Kykenevyyssuuntautuneisuus vaikuttaa olevan samankaltainen valkolistaamisen (engl. whitelisting) kanssa, eli WebAssembly-moduulit eivät esimerkiksi voi lukea tai kirjoittaa tiedostoja ilman, että niitä ajoympäristön käynnistyksessä erikseen määritellään. WebAssemblyn kanssa kehitystyötä kuitenkin hankaloittaa sen osoittimiin perustuvaa muistinkäsittelyä vaativa ohjelmointimalli. Tämän peittämiseksi on kehityksessä olevan *WebAssembly komponenttimallin* tarkoitus mahdollistaa pelkkiä osoittimia ihmisystävällisempien tietotyyppien ja rajapintojen käyttö, jolloin voidaan helpommin hyödyntää korkeamman tason ohjelmointikielten käytäntöjä (WebAssembly Community Group 2023). Kuitenkin manuaaliset muistioperaatiot näyttävät edelleen olevan säännöllisin tapa kommunikoida WebAssemblyn kanssa ja esimerkiksi mikropalveluita kapseloivien moduulien välillä kommunikointi on toteutettava paljolti WebAssemblyn hiekkalaatikon ulkopuolella (Kotilainen ym. 2023).

Kokoavasti WebAssemblyssa vaikuttaisivat yhdistyvän niin suoritusteho, modulaarisuus ja koostaminen kuin myös konttien parhaat puolet eristyksestä ja siirrettävyydestä. Tämä tekee teknologiasta houkuttelevan heterogeenisille alustoille ja hajautettuun laskentaan, joiden myötä se sopinee luonnollisesti myös liukkaisiin ohjelmiin.

3 Tutkimusasetelma

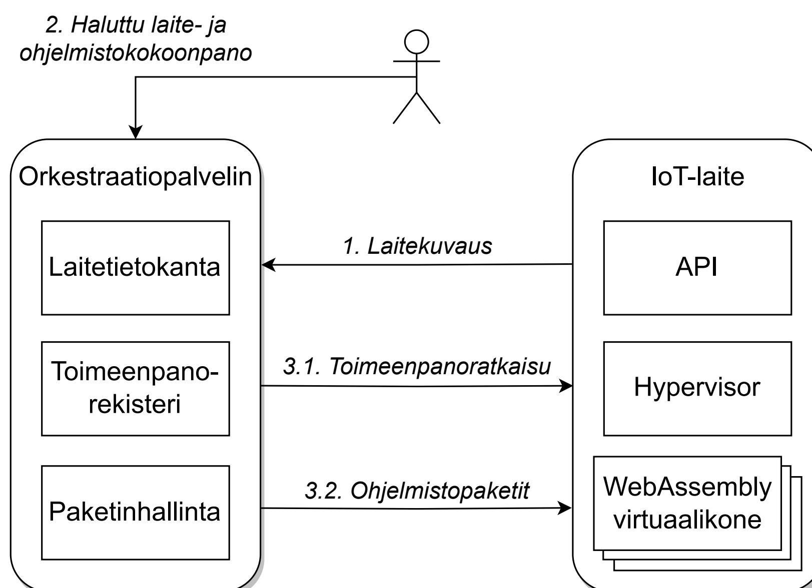
Taivalsaaren ja Mikkosen (2017) esityksen mukaan IoT:n ja ohjelmoitavan maailman toteutuminen näyttää nojaavan pilven tarjoamiin hyötyihin sekä tietoturvallisesta että käytännöllisestä datamassojen varastoinnista, analysoinnista ja laitteiden ohjaamisesta. Yhtenä IoT-pilvialustojen haasteena he näkevät lukemattomien laitteiden keskenään muodostaman verkoston hallinnoinnin ja "orkestraation". Vaikka on mahdollista että he ovat käyttäneet orkestraatio-sanaa sen tavanomaisessa ja yksinkertaisesti **järjestelyn** merkityksessä, konttien ja pilven kontekstissa käsite on kuitenkin jo esitettyjen tietojen mukaisesti määritellympi ja sitä keskeisesti ilmentää Kubernetes -järjestelmä.

Reunalaskenta- ja IoT-sovellusten käyttöönottoa ja kehittämistä sotkee huomattavasti se, että hajautetuille ja heterogeenisille laitteille ohjelmointi on vaivalloista. Laitteiden toiminnalliset ja laadulliset vaihtelut heijastunevat luonnollisesti myös niiden tarjoamia resursseja soveltaviin ohjelmointikieliin ja -tapoihin. Sovelluskehityksen näkökulmasta potentiaalinen helpotus voisi olla useiden ohjelmointikielten tukeminen samalla alustalla antaen kehittäjille valinnanvapautta sekä uuden että olemassaolevan koodin hyödyntämisessä. (Mikkonen, Pautasso ja Taivalsaari 2021)

3.1 Tutkimuskysymykset

Aiemmassa tutkimuksessa Kotilainen ym. (2022) ehdottavat ideapaperimuodossa arkkitehtuurin ja teknologisia valintoja isomorfiselle IoT-mikropalveluarkkitehtuurille. Heidän suunnitelmassaan huomioidaan sekä isomorfisuus että IoT olennaisesti WebAssemblyllä, jonka lähes natiivia suorituskykyä vastaava virtualisointi olisi ihanteellinen heterogeenisten laitealustojen yhtenäistämiseksi. Keskeinen komponentti, sekä merkitykseltään että verkkotopologisesti, on Kotilaisen ym. (2022) ehdottamassa arkkitehtuurissa "orkestraatiopalvelin", joka toimii IoT-laitteita toisiinsa nitovana käsky-ja-kontrolliyksikkönä (engl. command and control center). Orkestraatiopalvelin suunnitelmassa olennaisesti: 1. etsii yhteensopivia laitteita verkosta; 2. vastaanottaa käyttäjän WebAssembly-moduuleista koostuvia toimeenpanopyyntöjä (engl. deployment request); ja 3. ylläpitää toimeenpanoja dynaamisesti (so. suoritus-

kyvyn vaihtelua tarkkaillen) hajautettujen laitteiden verkossa. Arkkitehtuuri on nähtävissä Kuviossa 3.



Kuvio 3. Ehdotettu isomorfinen IoT-arkkitehtuuri (mukailtu (Kotilainen ym. 2022)). Orkestraatiopalvelin toimii keskitetysti etsien ja ohjaten verkosta löytyviä laitteita asentamaan sopivat sovelluspalaset ja kommunikaatioyhteydet.

Tämän tutkielman tavoitteena on toteuttaa yksinomaan Kotilaisen ym. (2022) ehdottama orkestraatiopalvelin, eli rajataan pois suunnitelmassa oleva paketinhallinta ja laitteiden ohjelmisto (Kuviossa 3 “Paketinhallinta” sekä “IoT-laitteen” sisältämät komponentit). Tarkoituksena on esittää, että arkkitehtuuria ja sille suunniteltuja teknologisia valintoja (so. mDNS, WebAssembly ym.) voidaan todellakin soveltaa käytännössä. Lisäksi koska suunnitelma on kuvaukseltaan melko korkealla abstraktiotasolla, on tarve tarkentaa esimerkiksi laitteiden järjestelyyn ja niiden väliseen viestintään tarvittavaa informaatiota ja sen kuvailutapoja.

Näistä tavoitteista johtaen tämän tutkielman tutkimuskysymykset ovat Thuan, Drechsler ja Antunes (2019) ehdottamien konstruktiivisten tutkimuskysymyspohjien päälle muotoillusti seuraavat:

- Kuinka IoT:lle suunnatun isomorfisen koodin arkkitehtuuri voidaan toteuttaa?
- Kuinka tähän toteutukseen voidaan soveltaa WebAssemblyä?
- Miten tätä toteutusta voidaan käyttää?

3.2 Tutkimusmenetelmä

Tutkimusmenetelmänä on suunnittelutiede niin kuin Peffers ym. (2007) kuvaavat. Menetelmä on valittu tutkimuksen taustana olevan aiemman ja erillisen WebAssemblyyn ja liukkaisiin ohjelmistoihin perustuvan tutkimusprojektin (Kotilainen ym. 2022) ohessa kehitetyn suunnitelman toteuttamiseksi. Toteutuksen avulla voidaan sekä todeta ehdotetun suunnitelman toiminta että järjestelmällisemmin vastata tutkimuskysymyksiin erittelemällä artefakti osiinsa.

Peffers ym. (2007) kuvailevat kuudesta “aktiviteetista” koostuvan prosessimallin, joita seuraten suunnittelutieteen menetelmää voidaan soveltaa tutkimukseen. Tässä tutkielmassa ei kuitenkaan seurata Peffersin ym. (2007) prosessimallia järjestyksessä ensimmäisestä ja toisesta aktiviteetista alkaen, sillä niihin kuuluva artefaktin motivointi ja tavoitteiden määrittely on tehty jo aiemmin Kotilaisen ym. (2022) puolesta. Sen sijaan aloitetaan Peffersin ym. (2007) ehdottamasti ratkaisukeskeisestä kolmannesta aktiviteetista, *suunnittelusta ja kehittämisestä*, mikä sopii tämän tutkielman tutkimuskysymyksiin paremmin.

Huomioitavaa lisäksi on, että tutkimusmenetelmästä muodollisesti poiketaan tutkielman aikana iteraatioiden suhteen, joihin sisällytettäviä arviointikierroksia suoritetaan vähemmän järjestelmällisesti. Suunnittelutieteen menetelmässä artefaktin arviointi voi Peffersin ym. (2007) mukaan vaihdella yksinkertaisesta demonstraatiosta järjestelmällisempään ja määrälliseen esimerkiksi artefaktin suorituskykyä sille tarkoitetussa tehtävässä mittaavaan arviointiin. Artefaktin määrällisestä arvioinnista saatavaa dataa voitaisiin esimerkiksi verrata toiseen vastaavanlaiseen järjestelmään. Tässä tutkielmassa katsotaan kuitenkin yksinkertaisen demonstraation riittävän, sillä taustakirjallisuuden perusteella on selvää että ko. orkestraatiopalvelimen roolia vastaavia järjestelmiä on lukuisia ja niitä tukevat vaikuttavat järjestöt kuten Google (2014) ja Cloud Native Computing Foundation[1]. Näin ollen määrällinen vertailu tutkielman puitteissa kehitetyn artefaktin ja jonkin tuotantokäytössä kypsytellyn järjestelmän välillä katsotaan merkityksettömäksi; tutkielman artefaktin ala on sille mahdollisia verrokkeja huomattavasti pienimuotoisempi. Arviointiin käytetään toiminnallista demonstraatiota, jonka avulla voidaan osoittaa tutkielman taustasta johdetun käyttötapauksen olevan mahdollista laadullisista ominaisuuksista niinkään piittaamatta.

1. “Orkestraatio” toistuu sivulla https://www.cncf.io/projects/ (haettu 15.2.2024) esitellyissä projekteissa

4 Artefakti

Tässä luvussa kuvataan taustakirjallisuudella motivoidun ja johdetun vaatimusmäärittelyn pohjalta kehitetty artefakti. Vaatimusmäärittelynä toimivat aiemmin tunnistetut yleiset piirteet reunalaskennasta, orkestraatiojärjestelmistä sekä ohjelmien liukkaudesta ja isomorfisuudesta. Luvussa siirrytään abstraktiotasossa ylhäältä alas, alkaen ominaisuuksien kuvauksista ja kulkien kohti järjestelmän arkkitehtuuria sekä joitakin huomionarvoisia alemman tason toteutusyksityiskohtia. Artefaktia arvioidaan perustuen vastaavuuteen taustakirjallisuuden kanssa sekä käyttöesimerkkeihin, jotka kertovat järjestelmän sovellusalueesta ja osoittavat sen mielekkään toiminnan.

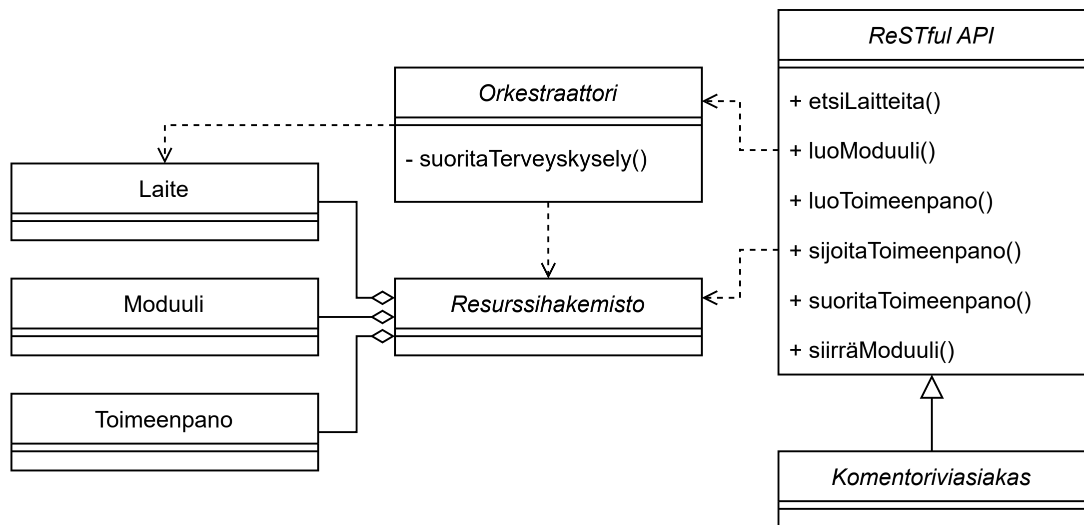
Artefaktin kuvauksessa käytetään apuna Kruchtenin (1995) 4+1 arkkitehtuurin näkymämalleja. Neljään näkymämalliin pohjautuen kuvaus on jaoteltu alalukuihin järjestyksessä alkaen *loogisesta*, *prosessi-*, *kehitys-* ja lopulta päätyen *fyysiseen näkymään*. Aivan viimeisenä alalukuna artefaktin arviointiin liittyvä demonstraatio toimii Kruchtenin (1995) kuvaukseen kuuluvana *skenaariona*.

4.1 Ominaisuudet

Tässä alaluvussa kuvaillaan kehitetyn artefaktin käyttäjälleen tarjoamat ominaisuudet artefaktia ympäröivän muun järjestelmän yhteydessä. Alaluku toimii siten eräänlaisena yleiskatsauksena järjestelmään ja sen käyttämiseen. Lyhyesti kuvailtuna artefakti on orkestraatioalusta WebAssemblyllä ohjautuvien pienten ja hajautettujen WWW-palvelimien tai *mikropalveluiden* toimeenpanoon, joka artefaktin käyttäjän kuvailemasti voi koostaa jonkin suuremman sovelluskokonaisuuden lähiverkossa. Kuviossa 4 on esitetty artefaktin korkean tason kuvaus, jonka tunteminen on käyttämisen kannalta keskeistä.

4.1.1 Ohjelmistoyksiköt ja toimeenpano

Kuten Docker ja Kubernetes käyttävät ohjelmistokontteja mikropalveluiden paketoimiseen, artefakti hyödyntää WebAssembly-moduulien virtualisointia siirrelläkseen samanmuotoisia, isomorfisia ohjelmia eri laitteiden tai alustojen välillä. Periaatteessa WebAssembly-ohjelmien



Kuvio 4. Artefaktin looginen näkymä luokkakaaviona. Kruchtenin (1995) loogisessa näkymässä kuvataan järjestelmän toiminnallisia ominaisuuksia, jotka paljastetaan käyttäjälle.

suorittamista rajoittaa vain sen ajoympäristön, eräänlaisen virtuaalikoneen sopivuus alustalle. Mielekästä ohjelmoitavuutta varten artefaktiin yhteensopivan laiteohjelmiston tulee tarjota joitakin ohjelmointirajapintoja käyttöjärjestelmän sekä oheislaitteiden käyttämiseen WebAssembly-moduuleista käsin ja moduulit toimiakseen voivat olla riippuvaisia vain näistä rajapinnoista. WASI -rajapinnan avulla voidaan käyttää kullekin WebAssembly-moduulille rajattua tiedostojärjestelmän hakemistoa rakenteellisen datan syöttämiseen ja lukemiseen WebAssemblyn yksinkertaisten numeeristen tietotyyppien lisäksi. Tutkielman *Wasm-IoT*-oheisprojektissa kehitetty `wasm3_api` rajapinta taasen mahdollistaa niin laitekohtaisten oheislaitteiden kuten kameran, lämpötila- ja kosteussensorien, kosketinnastojen kuin myös kellon kirjoittamisen ja lukemisen.

Toimitettaessa haluttu WebAssembly-moduuli artefaktille, tulee sen oheen liittää myös kuvaus moduulin itsensä rajapinnasta. Moduulin rajapinta artefaktissa tarkoittaa WebAssembly-funktioiden syötteitä ja tulosteita, joita tarvitaan datan sarjallistamista (engl. serialization) ja purkamista varten laitteidenvälisessä kommunikaatiossa. Moduulin rajapinnassa tulee kuvailla ensinnäkin mitä ja minkä tyyppisiä parametreja ja tulosteita moduulissa paljastetut funktiot käyttävät. Koska primitiivityyppien lisäksi voidaan käyttää mielivaltaista rakenteellista dataa syötteinä ja tulosteina, tulee näitä tarkoituksia varten listata moduulissa käy-

tettävät tiedostonimet mediatyyppeineen. Esimerkiksi jos moduuli tuottaa JSON-muotoon sarjallistettavaa dataa tiedostoon `data.json`, tämän tiedoston tai Docker-konttien käsitteistöstä lainatun *mountin* kuvauksena annetaan poluksi `./data.json` ja mediatyypiksi `application/json`. Liitteessä A on nähtävissä artefaktille yhteensopiva kuvausdokumentti moduulista, joka käyttää tiedostoja rakenteellisen datan käsittelemiseen. Käyttäjän toimitettua moduulin ja sille oikeanlaisen kuvauksen, artefakti tallettaa tiedot tietokantaan ja WebAssembly-moduulin datatiedostoineen tiedostojärjestelmään tulevia muokkauksia ja toimeenpanoja varten.

Luotuja moduuleja voidaan käyttää halutunlaisen sovelluksen koostamiseen toimittamalla artefaktille *toimeenpanon manifesti*. Tässä dokumentissa kuvaillaan mitä yhdistelmiä resursseista (so. laitteet, moduulit ja funktiot) halutaan sovelluksessa käytettävän. Artefakti lukee manifestin ja muodostaa siitä saatavilla oleviin resursseihin verraten toimivan ratkaisun, joka on sillä hetkellä valmis laitteille käyttöön sijoitettavaksi. Esimerkki toimeenpanon manifestin dokumentista on nähtävissä liitteessä B. Toimeenpanoa varten tehdyn ratkaisun valmistuttua voidaan laitteet käskeä asentamaan niiltä vaaditut resurssit toimintavalmiiksi ja käynnistää toimeenpanon suoritus.

Toimeenpanon suoritukseen on artefaktissa tarjolla kahdenlaisia suoritusmalleja: *putkitus* ja *pääohjelma*. Putkitusta voidaan käyttää luomaan suoraviivainen reitti funktiosta toiseen syöttämällä edellisen funktion tuloste sitä seuraavalle syötteenä. Toimintaperiaate on samanlainen kuin funktioiden koostaminen (engl. function composition) funktio-ohjelmoinnissa tai yhdistetyt funktiot matematiikassa. Pääohjelma taasen mahdollistaa hieman perinteisemmän sovellusohjelmoinnin, jossa toimeenpanon resursseja voidaan käyttää niiden muodostamiin nimiavaruuksiin perustuvilla moduulien koodista kutsuttavilla ohjelmointirajapinnoilla. Pääohjelman on nimensä mukaisesti tarkoitus olla hajautetun sovelluksen suoritusta ohjaava kehittäjänsä käyttämällä ohjelmointikielellä toteutettu skripti, joka laitteiden keskuudessa tarjotun API:n kautta toimii asiakkaana toimeenpanossa saataville palvelimille.

4.1.2 Monitorointi ja ylläpito

Lähiverkossa saatavilla olevien laitteiden jatkuvan etsimisen ansiosta artefaktin käyttäjälle tarjotaan ajantasaista tietoa saatavilla olevasta laiteresurssien valikoimasta toimeenpanon suunnittelua varten. Lisäksi jos jo löydetyt laitteet eivät vastaa artefaktin ajoittain lähettämiin terveyskyselyihin, kyseiset laitteet myös poistetaan valikoimasta. Toimeenpanon suorituksen aikana jokaisesta yksittäiselle laitteelle menevästä kutsusta talletetaan laitteen kutsuhistoriaan tietoja kuten aikaleima, käytetty polku, HTTP-metodi ja laskennan tulos. Kutsutiedot ovat saatavilla suoraan laitteilta itseltään hakemalla.

4.1.3 Dynaamisuus ja liukkaus

WebAssembly-pohjaisten moduulien isomorfisuuden mahdollistamaa liukkautta edustaa artefaktissa moduulien siirtäminen laitteiden välillä toimeenpanon aikana. Toimeenpanon ratkaisemisen yhteydessä artefakti pitää yllä kullekin laitteelle *vertaisia laitteita*, jonka myötä ne voivat kutsua muille kuin itselleen sijoitettuja moduuleja mikropalveluarkkitehtuurin mukaisesti. Täten jos halutaan siirtää tietty moduuli pois tietyltä laitteelta, artefakti ratkaisee toimeenpanon uudelleen alkuperäistä manifestia muuntaen ja lähettää päivitetyt tiedot kullekin asianomaiselle laitteelle. Huomioitavana rajoituksena on, että **siirrettävän moduulin** tulee olla sovelluksen toiminnan kannalta **tilaton**, eli kutsu ko. moduulin funktioihin ei voi riippua moduuliin aiempien kutsujen myötä luodusta tai muutetusta tilasta.

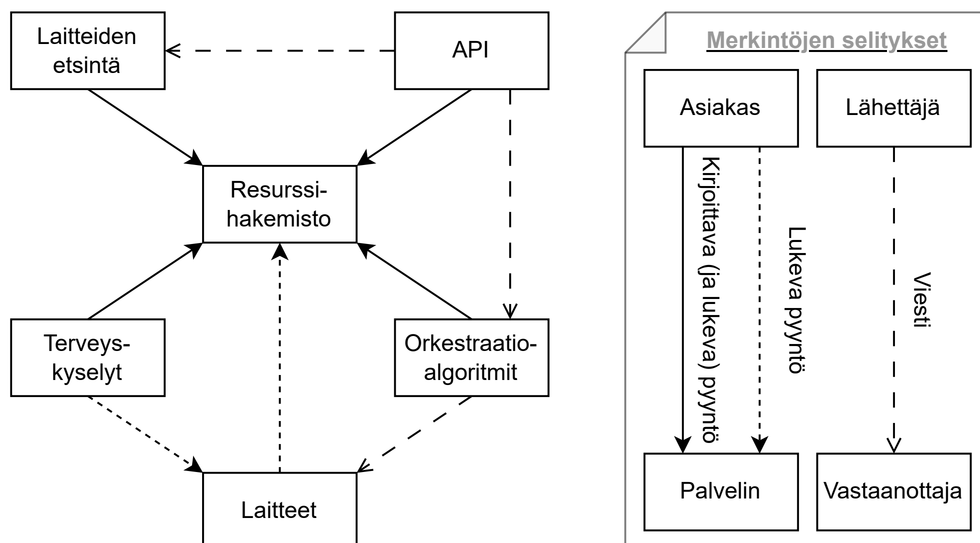
4.1.4 Käyttöliittymä ja rajapinnat

Artefakti toteuttaa ReSTful API:n keräämiensä resurssien ja ominaisuuksien hallinnointia varten. Rajapinta sisältää oleellisesti resurssien luontiin, kuvailuun, tarkasteluun ja poistoon tarvittavia toimintoja. API:n käyttäjän on luomansa toimeenpanon suorittamiseksi erikseen kutsuttava tähän tarkoitettua polkua. Artefakti ei erottele käyttäjiä toisistaan tai rajoita pääsyä resursseihin. Määritelty API mahdollistaa artefaktin käyttämiseksi vapaamuotoisen asiakasohjelmiston kehittämisen. Artefaktin yhteydessä on komentorivityökalu, jota käyttämällä keskustellaan API:n kanssa ja mahdollistetaan myös sovelluskokoonpanojen alustamisen automatisointi esimerkiksi *bash-skriptien* muodossa.

4.2 Samanaikaisuus

Suoritusaikana kommunikaatio tapahtuu artefaktin sisäisten osien välillä tietovaraston kautta ja artefaktia ympäröivässä järjestelmässä asiakas/palvelin -mallilla. Samanaikaisesti toimivia komponentteja artefaktissa voidaan katsoa olevan: laitteiden etsintä, terveyskyselyt, HTTP-palvelimen API, resurssihakemisto ja orkestraation algoritmit. Lisäksi artefaktia ympäröivässä järjestelmässä paketinhallinta ja laitteiden ohjelmisto ovat oleellisesti samanaikaisia, mutta koska ovat artefaktille tarkkaanottaen ulkopuolisia jätetään ne tässä kuvauksessa vähemmälle huomiolle.

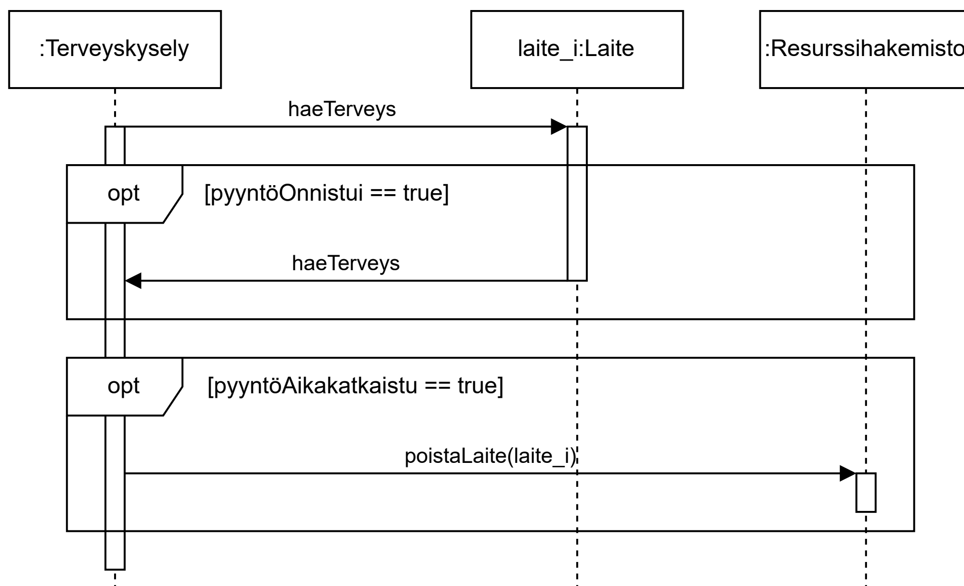
Samanaikaisuuteen kytkeytyvässä Kruchtenin (1995) mukaisessa prosessinäkymässä komponentit ovat *tehtäviä* (engl. tasks) ja tarkoituksena on osittain kuvata järjestelmän ei-toiminnallisia piirteitä kuten suorituskykyä tai saatavuutta (engl. availability). Kuviossa 5 on esitetty tehtäviksi lukeutuvien artefaktin komponenttien ja laitteiden ohjelmiston kommunikointi järjestelmässä. Kuviosta nähdään oleellisesti, että resurssihakemisto toimii yhteisenä kommunikointivälineenä, millä on vaikutuksia tiedon ajantasaisuuteen ja siis siihen, millaisessa järjestyksessä ja tiheydessä operaatioita tulee suorittaa kilpailutilanteiden (engl. race condition) estämiseksi. Esimerkiksi toimeenpano voidaan ratkaista, mutta siinä valittu laite hävitä verkosta ennen ko. laitteelle sijoitusta.



Kuvio 5. Yleisnäkymä artefaktin keskenään kommunikoivista tehtävistä

Muchandi (2007) ehdottaa Kruchtenin (1995) prosessinäkymän kuvaamiseen Unified Mode-

ling Language 2 (UML 2) sekvenssi- ja kommunikaatiokaavioita, joista jälkimmäinen keskittyy vuorovaikutustapahtumassa osanottajien välisiin yhteyksiin (engl. link). Kuviossa 6 esitetään laitteille ajoittain tehtävä terveyskysely ja kuinka siihen vastaamaton laite poistetaan resurssihakemistosta. Kuviossa 7 taasen on esitetty toimeenpanoon liittyvät tapaukset, jossa API:n kautta toimitettu manifesti ratkaistaan ja talletetun ratkaisun perusteella tehdään toimeenpanon sijoittaminen laitteille.

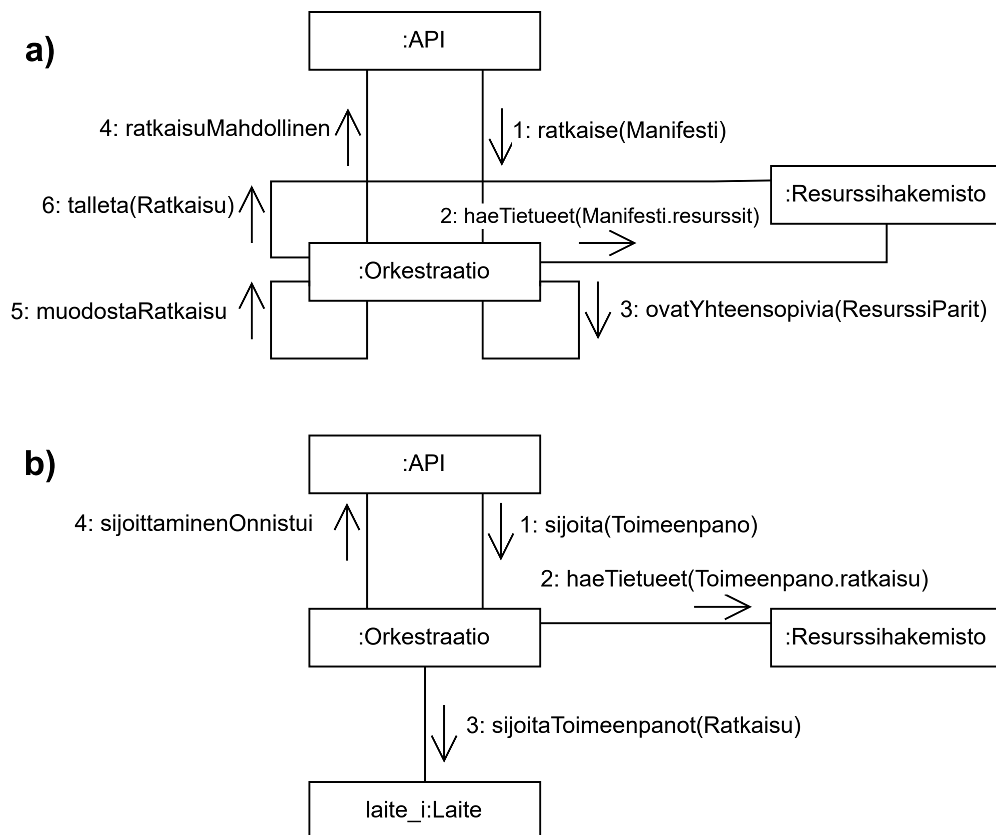


Kuvio 6. Sekvenssikaavio laitteiden terveyskyselyiden tekemisestä

Laitteidenvälistä kommunikointia sekä moduulien ajonaikaista siirtämistäkin varten laitteet saavat toimeenpanon sijoittamisen yhteydessä URL-linkkejä osoitteisiin, joista ne voivat kutsua toisilla laitteilla olevien moduulien funktioita. Näiden linkkien oletetaan toimivan koko toimeenpanoon liittyvän sovelluksen suorituksen ajan.

4.3 Ohjelmistomoduulit

Tässä alaluvussa kuvataan yksityiskohtaisemmin, kuinka artefakti toteutuksessaan rakentuu tutkielmassa ja sen rinnalla kehitetyistä sekä valmiista ohjelmistopalasista ja teknologioista. Alaluvussa mukaillaan Kruchtenin (1995) kehitysnäkymän kuvausta, joka keskittyy artefaktin todellisten ohjelmistomoduulien järjestelyyn, eli millaisiin palasiin ja palasten välisiin rajapintoihin ohjelmisto kehitystoimintaa varten jakautuu. Kruchtenin (1995) kehi-



Kuvio 7. Kommunikaatiokaavio toimeenpanon tapahtumista. Toimeenpanon manifesti **a)** ratkaistaan ja **b)** sijoitetaan laitteille.

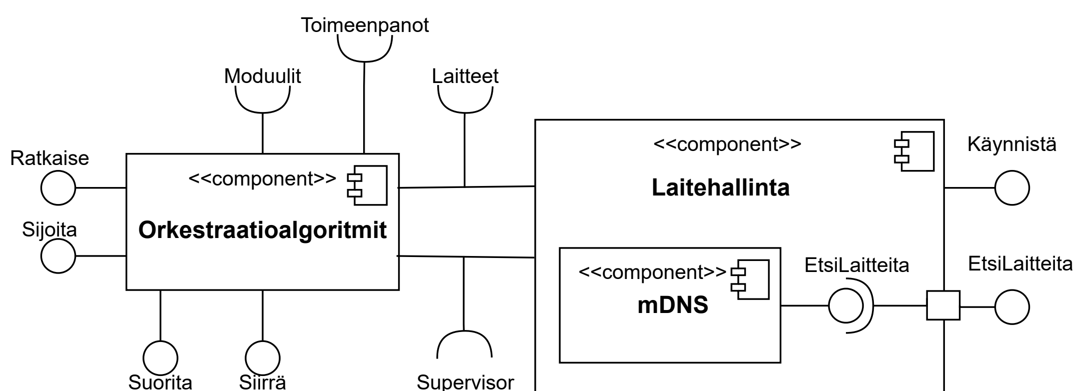
tysnäkymän esitystapana voidaan Muchandin (2007) suosituksia mukaillen käyttää UML-komponenttikaavioita.

Artefaktin palastelun pohjana on toimivaa käyttää käsitteellisempää esitystä, jollainen sen looginen näkymä Kuviossa 4 on. Tästä voidaan tunnistaa moduuli- tai komponenttimaisia kokonaisuuksia, joita ovat edelleen pienemmistä kokonaisuuksista koostuvat luokat *Orkestraattori*, *ReSTful API* ja *Resurssihakemisto*. Muut keskeiset mutta tutkielmalle ulkoiset komponentit ja teknologiat selityksineen on lyhyemmin taulukoitu Liitteessä C.

4.3.1 Orkestraattori

Orkestraattori vastaa niin laitteiden etsimisestä ja hallinnoinnista, kuin myös toimeenpanojen määrittelystä ja suorittamisesta. Nämä operaatiot ovat läheisesti riippuvaisia etenkin re-

surssihakemistosta ja laitteista. *Orkestraattorin* sisältämiä osia on esitetty Kuviossa 8. Laitteiden etsintää ja niille tehtäviä terveyskyselyitä suorittaa **Laitehallinta**, joka loputtomasti ajastetuin väliajoin seuraa lähiverkon mDNS-mainoksia sekä pyytää laitteilta terveysraportteja. Tällä tavoin artefaktin resurssihakemistossa pidetään yllä ajantasaista tietoa käytössä olevista laitteista. **Laitehallinta** tarjoaa rajapintanaan sekä itsenäisesti ajastimella toimivan laitteiden etsinnän käynnistyksen että välittömän etsinnän suorittamisen. Skannauksessa mDNS-protokollan toteutus on peräisin tutkielman ulkopuolisesta `bonjour-service` -kirjastosta. Toimeenpanoon tarvittavista ratkaisualgoritmeista sekä tulosteena saatavista tietuemuodoista tai *skeemoista* ja hallinnointioperaatioista vastaa **Orkestraatioalgoritmit**. Perustuen saatavilla oleviin resursseihin, **Orkestraatioalgoritmit** tarjoaa rajapintanaan toimeenpanon manifestien ratkaisemisen, ratkaisujen sijoittamisen sekä sijoitusten suorittamisen ja ajonaikaisen siirtämisen.



Kuvio 8. *Orkestraattorin* kehitysnäkymäkuvaus komponenttikaaviona

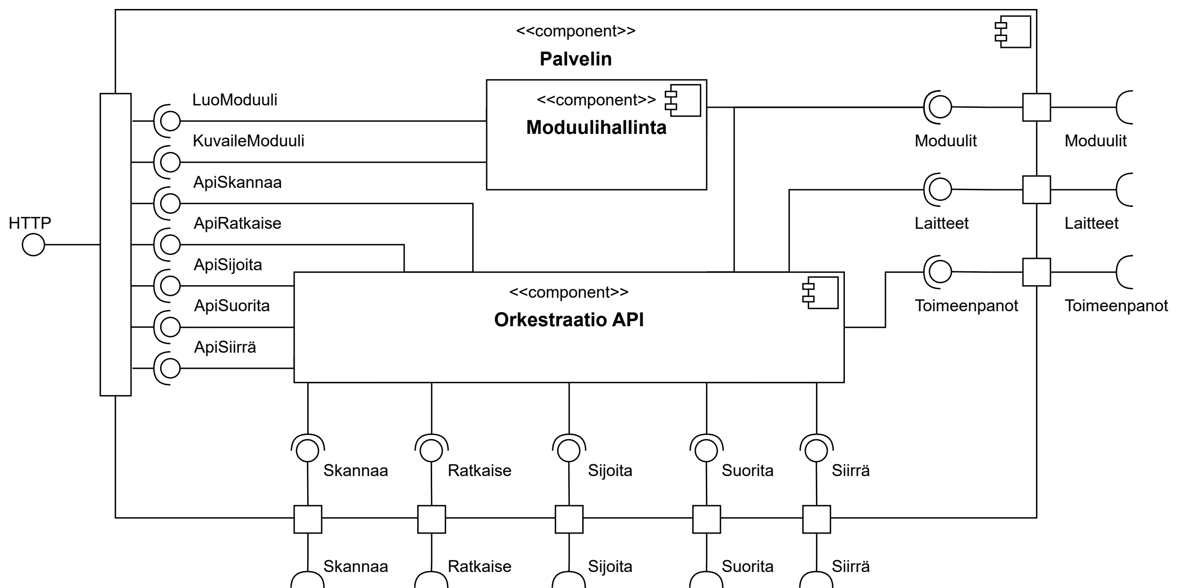
Orkestraatioalgoritmeille oleellinen riippuvuus on mielivaltaista sovelluslogiikkaa sisältävien WebAssembly-binäärien ja niiden metatietojen yhteensopivuuksia ratkaiseva paketinhallinta, joka kuitenkin on tutkimusasetelmassa määritellyn mukaisesti tämän tutkielman ulkopuolella. Tämän paketinhallinnan kaltaisen kokonaisvaltaisen ja hienojakoisemman ratkaisun käyttämisen sijaan **Orkestraatioalgoritmit** valitsevat yksinkertaisesti ensimmäisen sopivalta näyttävän laitteiden ja moduulien yhdistelmän.

Koska *Orkestraattoria* on tarkoitus käyttää *ReSTful API*:n kautta, myös sen sisäisessä toteutuksessa on pyritty seuraamaan ReST (engl. Representational State Transfer)-arkkitehtuuri-

tyyliä ja paljastettu rajapinta on kytketty API-palvelimen polkuihin vain vähäisellä välikäsittelyllä.

4.3.2 ReSTful API

Artefaktin käyttäminen tapahtuu sille määritellyn ReSTful API:n kautta, joka on toteutettu HTTP **Palvelimena**. *ReSTful API*:n kautta voidaan tarkastella artefaktin saatavilla olevia resursseja, hallinnoida moduuleja ja toimeenpanoja sekä käynnistää *Orkestraattorin* operaatioita kuten laitteiden skannauksen, toimeenpanon sijoittamisen tai moduulin siirtämisen. Kuviossa 9 nähdään, kuinka **Palvelin** jakautuu komponentteihin. **Orkestraatio API** muuntaa riippuvaiset rajapintansa *Orkestraattorista* ja *Resurssihakemistosta* HTTP-päätepisteiksi. **Palvelin** sisältää myös tutkielman ulkopuolelle jätetyn paketinhallinnan osittaisiin vastuisiin kuuluvan **Moduulihallinnan**, jota käytetään moduulien määrittelyyn ja tarjoilemiseen. **Palvelinta** käyttää myös artefaktiin kuuluva *Komentoriviasiakas*, joka käärii API:n operaatiot komentoriville käyttäjäystävällisempiin muotoihin ja tekee HTTP-pyyntöjä *ReSTful API*:n OpenAPI[1]-kuvaukseen perustuen.



Kuvio 9. *ReSTful API*:n kehitysnäkymäkuvaus komponenttikaaviona

1. https://swagger.io/specification/ (haettu 13.2.2024)

4.3.3 Resurssihakemisto

Kuten prosessinäkymäkuvauksessa 4.2 on kerrottu, artefaktin ja sitä ympäröivän järjestelmäkokonaisuuden kommunikaatio tapahtuu paljolti perustuen *Resurssihakemistoksi* nimitettyyn tietovarastoon. *Resurssihakemisto* on yksinkertaisesti tietokanta, jossa ylläpidetään kolmea erilaista tietuekokoelmaa: *laitteet*, *moduulit* ja *toimeenpanot*. Tietokanta on perinteisen relaatiotietokannan sijaan dokumenttitietokanta, johon voidaan käytännössä suoraan sijoittaa *ReSTful API*:n JSON-muotoisena käsittelemää dataa. Tiukan muodon tai skeeman määrittelyn sijaan dokumenttitietokannan dynaamisuus tuo joustavuutta artefaktin kehittämisessä, koska uusien käyttötapauksien ilmentyessä resursseihin tarvittavia kenttiä voidaan vaihtaa ohjelmakoodissa ja keskittyä alusta loppuun suunnittelemisen sijaan nopeaan prototypointiin. Toisaalta koska resurssihakemisto on jaettu artefaktin komponenttien kesken, muutosten tekeminen luonnollisesti heijastuu laajalti koko järjestelmän toimintaan.

4.4 Käyttöönotto

Artefakti koostuu kahdesta Docker-kontista, orkestraatiopalvelin ja MongoDB[2]-tietokanta, jotka ovat yhteydessä toisiinsa Dockerin sisäisessä verkossa. Koko järjestelmän rakenne vastaa kuitenkin edelleen Kuviota 3, koska tietokanta ei lisää tähän graafiin uutta solmua. Artefaktin kanssa keskustelevia laitteita voidaan liittää sekä Docker- että LAN-verkkoon, kunhan mDNS-mainokset ja HTTP-pyynnöt kulkevat laitteiden välillä. Orkestraatiopalvelimen verkko-osoitteen tulee olla ennen käynnistystä asetettu sille varattuun ympäristömuuttujaan, koska toimeenpanon yhteydessä laitteet saavat *ReSTful API*:iin osoittavia URL:eja hakeakseen asennettavat moduulit tai tehdäkseen moduulin siirtäviä pyyntöjä. Vastavuoroisesti myös laitteiden verkko-osoitteiden tulee olla orkestraatiopalvelimen sekä muiden laitteiden tavoitettavissa toimeenpanojen sijoitusta, terveyskyselyitä ja laitteidenvälistä kommunikaatiota varten. Artefaktin lähdekoodin Git-varasto on saatavilla osoitteessa https://github.com/trkks/wasmiot-orchestrator.git ja se voidaan rakentaa ja käynnistää varaston juuresta komennolla `docker-compose up` samasta hakemistosta löytyvään `docker-compose.yml` tiedostoon perustuen.

2. https://www.mongodb.com/ (haettu 13.2.2024)

4.5 Demonstraatio

Tässä alaluvussa demonstroidaan artefaktin toimintaa sen arviointia varten. Kruchtenin (1995) 4+1 mallin mukaisesti tämä demonstraatio vastaa *skenaariota*, jossa esitetään kuinka artefaktin arkkitehtuurinäkymät vuorovaikuttavat keskenään ja osoittaa, että muodostettu prototyyppi toimii. Käytettävä sovellustapaus perustuu *pilvipelaamiseen* (engl. cloud-gaming), joka Shean ym. (2013) mukaan yksinkertaisimmassa muodossa tarkoittaa ohjelmaa, joka esitetään käyttäjälle vuorovaikutteisena videovirtana kuten mikä tahansa videopeli, mutta video muodostetaan käyttäjän vähäpätöisemmän laitteen sijasta pilven massiivista laskentakapasiteettia hyödyntäen. Tapauksen valinta on perusteltua, sillä reunalaskennan visioon ja haasteisiin katsotaan kuuluvan myös reaaliaikaisten sovellusten kuten pelien viiveen parannukset (Shi ym. 2016).

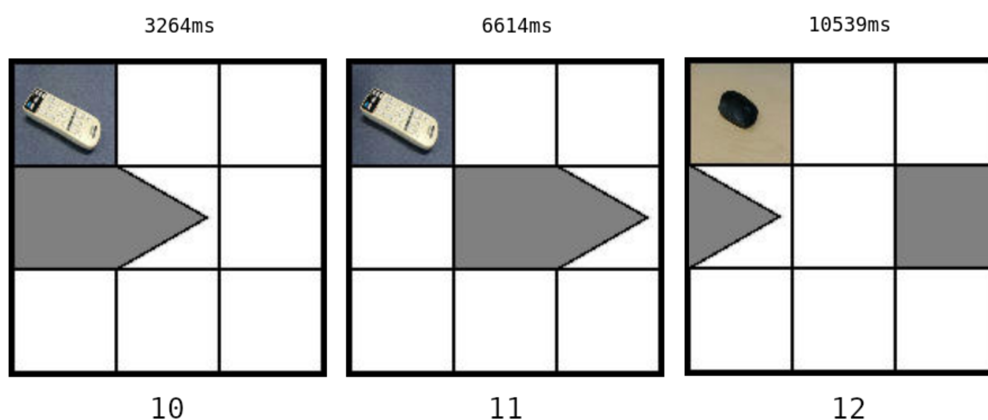
Sovellustapauksessa loppukäyttäjä pelaa WWW-selaimen kautta nk. *matopeliä*, jossa pelialue on kaksiulotteinen ruudukko. Ruudukolla on pelin aikana satunnaisiin sijainteihin ilmestyvä *ruoka* sekä pelihahmo, joka liikkuu tasaisin aikavälein ruudukon solu kerrallaan eteenpäin. Yhdellä laitteella muodostetaan pelin videonäkymä, johon liitetään samalta laitteelta peräisin olevaa kamerakuvaa. Lopullinen koostettu näkymä virtautetaan selaimeen, jonka kautta kerätään myös käyttäjän syötettä. Käyttäjä voi pelaamisen aikana siirtää kamerakuvaa keräävän moduulin toiselle laitteelle, minkä seurauksena tämä osa pelinäkymästä muuttuu kameran vaihtumisen myötä. Arviointia varten muodostettu artefaktin pääominaisuudet kattava käyttöskenaario kuvataan tapahtumien sarjana seuraavanlaisesti:

1. Käynnistetään saataville kaksi laitetta L_{K1} ja L_{K2}, joista kumpikin tarjoaa WASI-rajapinnan sekä pääsyn kameraan.
2. Käynnistetään artefakti O, joka etsii käynnistetyt laitteet.
3. Luodaan WebAssembly-moduuli M_M, joka sisältää matopelin logiikan.
4. Luodaan WebAssembly-moduuli M_K, joka tallettaa kuvia kamerasta.
5. Luodaan WebAssembly-moduuli M_A, joka sisältää pelinäkymän muodostavan logiikan sekä lisäksi HTML- ja Javascript-tiedostoista koostuvan asiakasohjelman *index*.
6. Toimitetaan ja kuvaillaan moduulit komentoriviasiakkaan avulla O:lle.
7. Toimitetaan O:lle toimeenpanon T manifesti, joka määrittelee:
   - Pääohjelmaksi M_A:n ja aloitusfunktioksi *index*:n tarjoavan funktion.

- Saatavilla kaikki funktiot moduuleista M_M, M_K, M_A.
- Moduulit sijoitettaviksi laitekokoonpanolla $(L_{K1}, \{M_A, M_K\}), (L_{K2}, \{M_M\})$.

8. Komennetaan O:ta sijoittamaan luotu T laitteille.
9. Komennetaan O:ta suorittamaan T:n pääohjelma.
10. Avataan vastauksena saatu linkki WWW-selaimessa ja pelataan matopeliä, jossa *ruoan* sijainnissa nähdään L_{K1}:n taltioima kuva.
11. Pyydetään *index*:n avulla L_{K1}:tä häätämään M_K. L_{K1} tekee siirtopyynnön O:lle, joka uudelleenratkaisee toimeenpanon ja sijoittaa M_K:n laitteelle L_{K2}.
12. TULOS: *Ruoan* sijainnissa nähdään nyt L_{K2}:n taltioima kuva.

Kuviossa 10 nähdään kuvankaappauksia käyttöskenaarion kohdasta 10. lähtien. Aikaleimojen perusteella esimerkiksi moduulin siirto laitteelta toiselle ja siten kameralla taltioidun kuvan vaihtuminen on kestänyt 10539ms − 6614ms = 3925ms tai noin 4 sekuntia. Koko järjestelmä laitteineen toimii tässä demonstraatiossa lähiverkon sijasta simuloidusti Dockerin sisäisessä verkossa.



Kuvio 10. Sarja kuvankaappauksia artefaktin toimintaa demonstroivasta sovelluksesta. Kunkin kuvankaappauksen yllä on aikaleima millisekunteina pelin aloituksesta päivitetyn pelinäkymän vastaanottamiseen ja alapuolella vastaava käyttöskenaarion kohdan numero.

5 Pohdinta

Tässä luvussa esitetään vastaukset tutkielman tutkimuskysymyksiin, arvioidaan artefaktin toimivuutta sekä ehdotetaan jatkotutkimuskohteita. Tutkimuskysymyksiä käsitellään yksitellen perustellen päätelmiä artefaktista tehdyllä teknisellä kuvauksella ja saaduilla kokemuksilla. Kritiikissä sekä lopulta jatkotutkimusehdotuksissa nojaudutaan enemmän aihepiirin taustakirjallisuudesta haettuihin tietoihin sekä artefaktin demonstraatiosta tehtyihin havaintoihin.

5.1 Tutkimuskysymyksiin vastaaminen

Artefaktin toteutus pyrkii vastaamaan seuraaviin kolmeen tutkimuskysymykseen, jotka perustuvat aiempiin tutkimuksiin (Kotilainen ym. 2022; Kotilainen ym. 2023) isomorfisen koodin arkkitehtuurin ympärillä.

- **Kuinka IoT:lle suunnatun isomorfisen koodin arkkitehtuuri voidaan toteuttaa?**

Artefaktin perusteella arkkitehtuuri voidaan toteuttaa monilta osin itsenäisesti ja pro gradun kokoisen tutkielman puitteissa sekä käyttämällä nykyaikaisia ja laajalti tuotantokäytössä olevia teknologioita kuten Javascriptiä, Node.js:ää, dokumenttitietokantaa ja Dockeria. Valmista orkestraatioalustaa kuten Kubernetes ei siis välttämättä tarvita pohjaksi. Artefaktin moduuleihin ja verkkolaitteisiin jakautumisen perusteella toteutus myös vastaa melko suoraan suunniteltua arkkitehtuuria, johon kuuluvat orkestraatiopalvelin ja ajonaikaisesti etsittävät erilliset IoT-laitteet.

Isomorfisuus on kapseloitu moduuleihin, joita varten artefakti tarvitsee metatietoja niiden tarjoamista toiminnoista eli funktioiden rajapinnoista tietotyyppeineen. Artefakti tallettaa moduulit yksilöidyin tunnistein, joiden kautta nämä paketoidut ohjelmistoyksiköt ovat tallessa ja myöhemmin saatavissa. Talletus vastaa ohjelmistokonttien yhteydessä käytettävien konttivarastojen toimintaa. Tämän tutkielman ulkopuolelle suunnitelmasta jätetyn paketinhallinnan on kuitenkin tarkoitus olla kattavammin vastuussa moduulien päivityksiin, yhteensopivuuksiin ja riippuvuuksiin liittyvästä ratkaisutoiminnasta ja jakelusta.

URL-linkkejä käyttäen pyritään luomaan laitteista käsitteellinen graafi, jonka sisällä laitteet voivat suoraan kommunikoida (ns. *laitteelta laitteelle*, engl. Machine-to-machine, M2M), eikä tällöin orkestraatiopalvelimen tai muunkaan erillisen verkon solmun tarvitse toimia viestinvälittäjänä laitteiden välillä. Alkuperäisen suunnitelman (Kotilainen ym. 2022) vastaisesti IoT-laitteiden kanssa ja niiden välillä kommunikointiin käytetään CoAP-protokollan sijasta HTTP-protokollaa. Protokollalla voitaisiin katsoa olevan **orkestraation** kannalta merkitystä etenkin, jos laitteille jatkuvasti suoritettavat terveyskyselyt olisivat keskeinen osa artefaktia, mutta näin ei ole.

Orkestraation yleisiä piirteitä seuraten, on artefaktissa alustavasti toteutettu toimeenpanon ylläpitotoimia. Terveyskyselyiden perusteella tehtävät automaattiset resurssihakemiston päivitykset liittyvät oleellisesti *sovellusten ja ryppäiden luontimenetelmiin*: moduuleita ei voi lähettää ja sovellusta suorittaa jos valittuja laitteita ei sillä hetkellä löydy tai ne ovat viallisia. Toinen tunnistettava orkestraation piirre on, että laitteiden kutsuhistorian aikaleimoja vertailemalla olisi mahdollista tarkastella kuinka kauan kutsujen suoritus kestää ja reagoida muuttamalla moduulienvälistä topologiaa (so. siirtää moduuleja eri laitteille) halutulle sovellukselle tehokkaammaksi. Tämä vastaa selkeästi *palvelunlaadun hallinnan* piirrettä. Toiminnan voitaisiin lisäksi sanoa liittyvän myös *laskentaresurssien kiintiöiden hallintaan*, sillä artefaktissa ei ole konttien tapaan esimerkiksi `cgroups`-tyyppistä menetelmää CPU:n tai muistin käytön rajoittamiseen WebAssembly-moduuleissa.

- **Kuinka tähän toteutukseen voidaan soveltaa WebAssemblyä?**

WebAssembly-kohtaista toiminnallisuutta ei tarvitse artefaktissa paljoakaan käyttää, jotta voidaan soveltaa teknologian isomorfisuutta. Artefakti ainoastaan käyttäjäystävällisyyden vuoksi lukee WebAssembly-binääristä sen paljastamat funktionimet, mutta on muutoin ottamatta kantaa suoraan moduulien sovelluskoodiin. Artefaktin sijaan laitteilla käytetty `wasmiot-supervisor` taasen on WebAssemblyyn paljon kytkeytyneempi.

Toisaalta WebAssemblyn **primitiivisyys** oleellisesti vaikuttaa siihen, miten artefakti valmistelee moduulit ja toimeenpanot. Rakenteellista dataa kuten kuvia tai JSON-dokumentteja varten käytetään tiedostoja, jotka liitetään moduulien yhteyteen. Näitä liitettyjä tiedostoja nimitetään artefaktissa Dockerin samantapaisen käsitteen mukaisesti *mounteiksi*. Myöhem-

min kun moduuleita suoritetaan laitteilla, oletetaan tiedostojen olevan eristettyjä ja luettavissa WebAssembly-ajoympäristön ja WASI-rajapinnan avulla. Samanlaista periaatetta tiedostojen yhdistämisestä rajattuun joukkoon sallittuja tiedostopolkuja käyttää *Extism* (Dylibso, Inc. 2024), joka on ohjelmistokehys WebAssembly-pohjaisten liitännäisten käyttämiseen.

Vaatimus toimeenpanon suoritusaikana siirrettävän moduulin tilattomuudesta johtuu siitä, ettei WebAssembly-moduulin muistin sarjallistamista ole artefaktin järjestelmään kuuluvassa laitteilla olevassa ohjelmistossa (so. `wasmiot-supervisor`) toteutettu. Tutkielman ja sen rinnalla olevien tutkimusprojektien aikana oli myös epäselvää, kuinka determinististä WebAssembly-muistin siirtäminen moduuli-instanssien välillä oikeastaan on. Erilaiset WebAssembly ajoympäristöt tai etenkin virtuaalikoneisiin pohjautuvat lähtökielet kuten C# ja Java voivat mahdollisesti käsitellä muistipaikkoja arvaamattomasti moduulien instantioimisen välillä. Toisaalta WebAssemblyn muistin lukeminen näyttää kuuluvan teknologian perustoimintoihin, eikä sitä siinä mielessä tulisi täysin kaihtaa vaan yksinkertaisesti harkita toteutusta tarkkaan. WebAssemblyn käyttämiseen ja kehittäjäystävällisyyteen liittyen tutkielma toistaa vahvasti aiemman tutkimuksen (Kotilainen ym. 2023) näkökantaa, jonka mukaan WebAssembly teknologiana on vielä niin alkutekijöissään, että mielekkäästi kokonaisvaltaisen sovelluksen toteuttaminen on sillä takeltelevaa.

- **Miten tätä toteutusta voidaan käyttää?**

Esitettyyn demonstraatioon perustuen artefaktin avulla voidaan koostaa ohjelmapalasista suurempi reunalaskennan käyttötapauksista johdettu sovellus, joka pilkottuna sijoitetaan verkosta löydettäville yhteensopiville laitteille. Laitteille sijoitetut moduulit luovat kullekin oman toimeenpanokohtaisen HTTP-API:nsa muiden kutsuttavaksi, jolloin artefaktille kehitettäessä voidaan noudattaa mikropalvelumallia. Tämän tutkielman demonstraation lisäksi artefaktin erästä versiota (Git-commit -tiiviste 227ad60bc8fea34e1014560046c04a543c4ecc21) on käytetty tieteellisen konferenssiesityksen yhteydessä liittyen aiempaan tutkimukseen (Kotilainen ym. 2023). Esityksessä artefaktin toisena suoritusmallina olevaa WebAssembly-funktioiden putkitusta käytettiin käyttötapaukseen, jossa kameran ottama kuva ohjattiin hahmontunnistavalle koneoppimismallille, joka sitten palautti tunnistuksen tulosta vastaavan kokonaisluvun.

Orkestraation yleispiirteistä artefaktin API sopii *sovellusten luontiin ja hallinnointiin*; resursseja voidaan luoda, päivittää, suorittaa ja poistaa API:n kautta. Kuitenkin samanaikaisuuden suhteen esimerkiksi Kuvioista 6 ja 7 yhdessä nähdään, että laitteen poistuminen resurssihakemistosta voi vaikuttaa siihen, onko toimeenpanon ratkaiseminen ja myöhempi sijoittaminen mahdollista. Artefaktin mielekkäältä käyttämiseltä vaaditaan siis jonkinasteista tietämystä siitä, milloin ja missä tilanteissa resurssihakemistoon talletettu tieto on ajankohtaista. Lisäksi voidaan tulkita terveyskyselyillä olevan vaikutuksia monilaitteisten toimeenpanojen suorituskykyyn, verraten kyselyiden tiheyttä ja siirrettävän datan määrää laitteiden ja verkon suorituskykyyn.

Sovelluksen käyttämisen aikana voidaan ohjelmapalasia siirtää laitteelta toiselle ilman, että sovellus pysähtyy tai kontrolli siihen täysin keskeytyy. Vertaislaitteiden listan avulla toimeenpano voi tässä tapauksessa jatkua ilman häiritseviä keskeytyksiä, sillä rajapinta laitteidenväliseen kommunikointiin on kutsujalle aina samanmuotoinen eikä moduulia itseään siis tarvitse päivittää kutsumuotojen vaihtamisen vuoksi.

Artefaktin komentorivipohjainen asiakastoteutus tukee pelkkää graafista käyttöliittymää paremmin orkestraatioon yhdistettäviä automatisaatiotarpeita mahdollistamalla toistuvien työkulkujen skriptaamisen manuaalisten komentojen antamisen lisäksi. Muilta osin orkestraatioon liitettävä palvelunlaadun ylläpito tai vaihteleviin tilanteisiin reagointi on artefaktissa olematonta johtuen erityisesti ajastusalgoritmien ja terveyskyselyiden käyttämisen naiviudesta.

5.2 Kritiikki ja jatkotutkimuskohteita

Tutkielmassa kehitetty artefakti on vain osittain alkuperäisen suunnitelman mukainen ja puutteellinen monelta keskeiseltä kantilta niin IoT-orkestraation kuin yleisenkin ohjelmistokehityksen periaatteiden näkökulmasta. Lisäksi pääsynhallinta ja tietoturva ovat artefaktissa olemattomia.

Muistin ja laskentatehon suhteen rajoittuneemmat laitteet tulisi järjestelmässä huomioida paremmin. Artefakti voitaisiinkin ensitöiksi muuttaa käyttämään tähän ympäristöön paremmin sopivaa CoAP protokollaa, kuten Kotilainen ym. (2022) ovat alkuperäisessä suunnitelmas-

sa ehdottaneet. Myöskin Node.js/Javascriptin käyttäminen artefaktin toteutuskielenä voi olla heikko valinta, vaikka tämän palvelimen sijoituspaikka olisikin pienten sensorilaitteiden sijaan hieman tehokkaampi laite kuten pikkupilvi. Artefaktista laitettiinkin tutkielman aikana alulle Rust-versio, jota varten kutakuinkin kartoitettiin kaikki tarvittavat kirjastot, mutta tämä ohjelmointikielen vaihtamistyö keskeytettiin uudelleenkirjoituksen vaatiman virhepotentiaalin ja aikavaatimusten vuoksi. Node.js:n ja Javascriptin dynaamisuuden omaksuminen toisaalta vaikutti sujuvoittavan kokeellisuuteen suuntautuvaa kehitystä ja lisäksi kielen osaajia on nykypäivänä paljon (Stack Overflow 2023), mikä vahvistanee artefaktin jatkokehityksen mahdollisuuksia.

Artefaktille yhteensopivat kuvailudokumentit ovat myös pelkistettyjä, eli kenttiä on lisätty sitä mukaa kuin on kehityksen yhteydessä katsottu tarpeelliseksi. Laitteiden moduuleille tarjoamilla rajapinnoilla ei esimerkiksi ole versiointia, mikä olisi tarpeellista riippuvuuksien kokonaisvaltaisempaan hallinnointiin (so. laitteella oleva ohjelmisto voi vanhentua, jolloin sille ei saisi lähettää uudemmasta versiosta riippuvaista moduulia). Tähän liittyvät aiheet ovat toisaalta jo alusta alkaen rajattu tutkielman ulkopuolelle. Tähän tarkoitukseen liittyen Kotilainen ym. (2023) tutkivat *Web of Things*[1]-dokumenttimuodon käyttöä, mutta tutkielman artefaktin kehityksessä tätä ei edelleenkään saatu tarpeeksi selkeästi sovitettua toteutukseen. Epäselvyyttä oli esimerkiksi siinä, kuvailtaisiinko tällaisella standardoidulla dokumentilla laitteiden, moduulien, vaiko laitteiden ja moduulien yhdessä muodostamia ominaisuuksia. Artefaktissa onkin käytössä moduulien kuvaileminen OpenAPI-dokumenteilla, jotka generoidaan artefaktin sisäiseen ja huomattavasti yksinkertaisempaan moduulien kuvausdokumenttimuotoon perustuen. OpenAPI ei kuitenkaan vaikuttanut täysin taipuvan esimerkiksi laitteille eri toimeenpanon vaiheissa (so. *sijoitus* (deployment), *suoritus* (execution) ja *tuloste* (output)) odotettavien tiedostojen ilmaisemiseen.

Reunalaskennan olemukseen kuuluu integraatio pilven kanssa. Pilven ja artefaktin yhteensopivuutta erityisesti lisää se, että artefakti on paketoitu Docker-kontiksi ja luultavasti myös Node.js:n käyttö, jota tuetaan laajalti pilvipalveluiden yhteydessä. Artefaktia ei kuitenkaan tietoisesti kehitetty pilvipalveluihin integrointia varten, joten yhteensopivuuden varmistaminen vaatisi tarkempaa aiheeseen tutustumista. Erityisesti lähiverkkoon perustuva laitteiden

1. https://www.w3.org/WoT/ (haettu 16.2.2024)

mDNS-skannaus tuottanee haasteita, mikäli artefaktia koetetaan suorittaa suoraan pilvessä.

IoT:n erityisiin tietoturvahaasteisiin varautuminen ei artefaktissa juurikaan näy, sillä taso on tätäkin alhaisempi. Selkein vajavuus on palvelimelle pääsyssä, joka on mahdollista yksinkertaisesti Docker-kontin paljastaman IP-osoitteen ja portin tietämällä. Toisekseen kommunikaatio palvelimen ja laitteiden välillä on selkokielistä. Nämä aukot voitaisiin kuitenkin melko suoraviivaisesti paikata käyttäjänhallinnalla (so. API:n käyttämiseksi tulisi tunnistautua) sekä järjestämällä Public Key Infrastructuren (PKI) salattua HTTPS-kommunikaatiota varten. Tällaisia yleisiä korjauksia varten voitaisiin esimerkiksi seurata ohjeita OWASP:n listaamista yleisimmistä WWW-sovellusten tietoturvariskeistä [2]. Artefaktille näitä ohjelmistokehityksen peruspalasia hieman ominaislaatuisempi tietoturvahaaste taasen on WebAssemblyn käyttäminen. Koska artefaktin tarkoituksena on mahdollistaa käyttäjiensä toimittamien mielivaltaisten ohjelmien suorittaminen, WebAssemblyn haavoittuvaisuus on keskeinen riskitekijä, joka tulisi järjestelmässä ottaa hyvin vakavasti. Esimerkiksi Lehmann, Kinder ja Pradel (2020) osoittavat että ei-selaimessa suoritettavien WebAssembly-ajoympäristöjen yhteydessä WebAssemblyn muistinhallinnointia voidaan käyttää mielivaltaiseen tiedostojen manipulointiin. Tämä on selvä riski artefaktin järjestelmän suhteen, sillä laitteet oleellisesti hyödyntävät tiedostojärjestelmää rakenteellisen datan syöttämiseen ja tulostamiseen. Toisekseen artefakti itsessään instantioi vastaanottamansa WebAssembly-binäärin lukeakseen siitä paljastetut funktiorajapinnat, missä voitaisiin potentiaalisesti hyödyntää Node.js:n globaalista WebAssembly-oliosta paljastuvia haavoittuvaisuuksia palvelimen haltuunottamiseksi.

Orkestraatioon liitettävien ominaisuuksien kannalta artefaktissa voidaan nähdä toteutuvan: sovellusta varten tarvittavan laitteiston valinta; toimeenpano; sovellusten paketointimalli moduulien muodossa; konttien eristäminen; ja sovellusten luonti- ja tarkastelumenetelmät. Artefakti keskittyy siis paljolti valmisteluun **ennen** sovelluksen suorittamista. Merkittävä osuus orkestraatiolle yleisesti tunnistetuista piirteistä kuitenkin vaikuttaa olevan ajonaikaista hallintaa: ryppäiden monitorointia; palvelunlaadun takaamista; sekä dynaamisesti muuttuvassa ympäristössä kommunikaatioyhteyksien ylläpitoa ja suojaamista. Kehitetty artefakti ei kuitenkaan konkreettisesti suorita tällaisia ajonaikaisia hallintatoimia. Sen sijaan se on vain osittain varustettu niiden toteuttamiseen esimerkiksi laitteiden kutsuhistoriaa käyttämällä. Mie-

2. https://owasp.org/www-project-top-ten/ (haettu 6.2.2024)

lekkäästi käynnissä olevan toimeenpanon mukauttaminen suorituksesta kerättyyn dataan perustuen vaatisi jonkinlaisen optimointialgoritmin liittämistä artefaktiin, mikä jätettiin tutkielman puitteissa pois.

Liukkauden tavoittelussa artefakti toisaalta osoittaa orkestraation dynaamisuutta siinä, kuinka ohjelmakoodia voidaan sovelluksen ajonaikana siirtää laitteelta toiselle. Liukkaus, jota tämä isomorfisen WebAssembly-ohjelmakoodin siirtäminen tai *migraatio* edustaa, onkin ehkä tutkielman uutuusarvoisin puoli. WebAssembly on kovin uutta teknologiaa ja sen soveltaminen reunalaskentaan ja IoT:hen mielenkiintoista (Kotilainen ym. 2022; Kotilainen ym. 2023; Vaño ym. 2023; Mäkitalo ym. 2021). Tutkielman kontekstissa liukkauden haasteena kuitenkin on, että sille ominaisten käyttötapausten ja sovellusten keksiminen on vaikeaa. Esimerkiksi liukkauden visioon (Taivalsaari, Mikkonen ja Systä 2014) liitetty “tulevaisuuden lasiteknologia” on vaikeaa sovittaa tämän tutkielman kokoiseen työhön. Yleisesti liukkauden ja IoT:n tutkimuksen uutuusarvoisuutta varten voisikin olla mielekästä etsiä ja määritellä pienempiä käyttötapauksia yksittäisiä kokeilullisia tutkimuksia varten.

6 Yhteenveto

Tässä tutkielmassa kuvailtiin toteutus palvelimesta ja tarkemmin HTTP-palvelimesta, jota voidaan käyttää alustana isomorfisten mikropalveluiden koostamiseen, sijoittamiseen ja suorittamiseen lähiverkosta löytyvillä yhteensopivilla laitteilla. Tutkielman keskeisenä tuloksena on, että WebAssembly teknologiana on käytettävissä isomorfisen koodin käyttötapauksissa.

Tutkielmassa kehitetty artefakti edustaa joitakin orkestraation ominaisuuksia, mutta on vajanainen erityisesti suoritusaikaisten toimien kuten laitekokoonpanon monitoroinnin ja palvelunlaadun takaamisen sekä tietoturvan suhteen. Artefaktin toimintaa on demonstroitu sekä erään tieteellisen konferenssiesityksen yhteydessä, että tätä tutkielmaa varten pilvi- ja reunalaskennan ja liukkaiden ohjelmien yhteydestä johdetun sovellusskenaarion muodossa. Lopputuloksena voidaan havaita, että vaikka tutkielman artefaktia voidaan monelta osin kuvailla keskeneräiseksi, se toteuttaa suunnitelmansa määrittelemät toiminnallisuudet niiltä osin kuin tutkielman tutkimusasetelmassa on rajattu.

Tulevaisuudessa artefaktia voidaan yleisiä ohjelmistoteknologioita käyttäen jatkokehittää tutkimaan isomorfisuuden, liukkauden ja IoT:n aiheita muillakin käyttötapauksilla, kuin mitä tässä tutkielmassa käytettiin vain yksinkertaisiin demonstraatiotarkoituksiin. Toisaalta jonkin toisen olemassaolevan orkestraatiokehyksen kuten Kubernetesin omaksuminen ja mukauttaminen valittuihin tutkimustavoitteisiin tuottaisi mitä luultavimmin tämän artefaktin käyttämistä luotettavamman ratkaisun ja siten myös uskottavampia tuloksia. Artefaktin Git-varasto on saatavilla osoitteessa https://github.com/trkks/wasmiot-orchestrator ja tutkielman ajankohtana sen viimeisin versio Git-commit -tiivisteellä tunnistettuna on 6ef66e9a5dc40f7978f 0655fe34f1a4eac500aae.

Lähteet

Armbrust, Michael, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee ym. 2010. “A view of cloud computing”. *Communications of the ACM* 53, numero 4 (huhtikuu): 50–58. ISSN: 0001-0782, 1557-7317, viitattu 6. lokakuuta 2023. https://doi.org/10.1145/1721654.1721672. https://dl.acm.org/doi/10.1145/1721654.1721672.

Arnold, Ken, James Gosling ja David Holmes. 2005. *The Java programming language.* Addison Wesley Professional.

Bernstein, David. 2014. “Containers and Cloud: From LXC to Docker to Kubernetes”. Conference Name: IEEE Cloud Computing, *IEEE Cloud Computing* 1, numero 3 (syyskuu): 81–84. ISSN: 2325-6095, viitattu 3. lokakuuta 2023. https://doi.org/10.1109/MCC.2014.51. https://ieeexplore.ieee.org/abstract/document/7036275.

Bonomi, Flavio, Rodolfo Milito, Jiang Zhu ja Sateesh Addepalli. 2012. “Fog computing and its role in the internet of things”. Teoksessa *Proceedings of the first edition of the MCC workshop on Mobile cloud computing,* 13–16. MCC ’12. New York, NY, USA: Association for Computing Machinery, 17. elokuuta 2012. ISBN: 978-1-4503-1519-7, viitattu 5. helmikuuta 2024. https://doi.org/10.1145/2342509.2342513. https://dl.acm.org/doi/10.1145/2342509.2342513.

Burns, Brendan, Brian Grant, David Oppenheimer, Eric Brewer ja John Wilkes. 2016. “Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade”. *Queue* 14, numero 1 (1. tammikuuta 2016): 70–93. ISSN: 1542-7730, viitattu 14. elokuuta 2023. https://doi.org/10.1145/2898442.2898444. https://dl.acm.org/doi/10.1145/2898442.2898444.

Bytecode Alliance contributors. 2019. “wasmtime/docs/WASI-overview.md at main · bytecodealliance/wasmtime”. GitHub, 12. marraskuuta 2019. Viitattu 15. helmikuuta 2024. https://github.com/bytecodealliance/wasmtime/blob/main/docs/WASI-overview.md.

Cao, Keyan, Yefan Liu, Gongjie Meng ja Qimeng Sun. 2020. "An Overview on Edge Computing Research". Conference Name: IEEE Access, *IEEE Access* 8:85714–85728. ISSN: 2169-3536, viitattu 20. lokakuuta 2023. https://doi.org/10.1109/ACCESS.2020.2991734. https://ieeexplore.ieee.org/abstract/document/9083958.

Carrión, Carmen. 2022. "Kubernetes Scheduling: Taxonomy, Ongoing Issues and Challenges". *ACM Computing Surveys* 55, numero 7 (15. joulukuuta 2022): 138:1–138:37. ISSN: 0360-0300, viitattu 20. heinäkuuta 2023. https://doi.org/10.1145/3539606. https://dl.acm.org/doi/10.1145/3539606.

Casalicchio, Emiliano, ja Stefano Iannucci. 2020. "The state-of-the-art in container technologies: Application, orchestration and security". *Concurrency and Computation: Practice and Experience* 32 (17): e5668. ISSN: 1532-0634, viitattu 2. lokakuuta 2023. https://doi.org/10.1002/cpe.5668. https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5668.

Cisco. 2015. *Cisco Global Cloud Index: Forecast and Methodology, 2014–2019.* https://community.cisco.com/kxiwq67737/attachments/kxiwq67737/5451-discussions-cisco-bug-discussions/3432/1/cloud_index_white_paper.pdf.

De Gelas, Johan, ja Intel ESX. n.d. "Hardware Virtualization: the Nuts and Bolts".

Dua, Rajdeep, A Reddy Raja ja Dharmesh Kakadia. 2014. "Virtualization vs Containerization to Support PaaS". Teoksessa *2014 IEEE International Conference on Cloud Engineering,* 610–614. 2014 IEEE International Conference on Cloud Engineering. Maaliskuu. https://doi.org/10.1109/IC2E.2014.41.

Dylibso, Inc. 2024. "The Manifest | Extism - make all software programmable. Extend from within.", 2. helmikuuta 2024. Viitattu 2. helmikuuta 2024. https://extism.org/docs/concepts/manifest.

Felter, Wes, Alexandre Ferreira, Ram Rajamony ja Juan Rubio. 2015. "An updated performance comparison of virtual machines and Linux containers". Teoksessa *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS),* 171–172. 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). Maaliskuu. https://doi.org/10.1109/ISPASS.2015.7095802.

Gogouvitis, Spyridon V., Harald Mueller, Sreenath Premnadh, Andreas Seitz ja Bernd Bruegge. 2020. “Seamless computing in industrial systems using container orchestration”. *Future Generation Computer Systems* 109 (1. elokuuta 2020): 678–688. ISSN: 0167-739X, viitattu 29. elokuuta 2023. https://doi.org/10.1016/j.future.2018.07.033. https://www.sciencedirect.com/science/article/pii/S0167739X17330236.

Ingalls, Dan, Ted Kaehler, John Maloney, Scott Wallace ja Alan Kay. 1997. “Back to the future: the story of Squeak, a practical Smalltalk written in itself”. Teoksessa *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications,* 318–326. OOPSLA ’97. New York, NY, USA: Association for Computing Machinery, 9. lokakuuta 1997. ISBN: 978-0-89791-908-1, viitattu 31. elokuuta 2023. https://doi.org/10.1145/263698.263754. https://dl.acm.org/doi/10.1145/263698.263754.

Irwin, Michael. 2022. “Introducing the Docker+Wasm Technical Preview | Docker”. Section: Products, 24. lokakuuta 2022. Viitattu 15. helmikuuta 2024. https://www.docker.com/blog/docker-wasm-technical-preview/.

Kayal, Paridhika. 2020. “Kubernetes in Fog Computing: Feasibility Demonstration, Limitations and Improvement Scope : Invited Paper”. Teoksessa *2020 IEEE 6th World Forum on Internet of Things (WF-IoT),* 1–6. 2020 IEEE 6th World Forum on Internet of Things (WF-IoT). Kesäkuu. Viitattu 8. marraskuuta 2023. https://doi.org/10.1109/WF-IoT48130.2020.9221340. https://ieeexplore.ieee.org/abstract/document/9221340.

Khan, Asif. 2017. “Key Characteristics of a Container Orchestration Platform to Enable a Modern Application”. Conference Name: IEEE Cloud Computing, *IEEE Cloud Computing* 4, numero 5 (syyskuu): 42–48. ISSN: 2325-6095, viitattu 3. lokakuuta 2023. https://doi.org/10.1109/MCC.2017.4250933. https://ieeexplore.ieee.org/abstract/document/8125559.

Kotilainen, Pyry, Teemu Autto, Viljami Järvinen, Teerath Das ja Juho Tarkkanen. 2022. “Proposing Isomorphic Microservices Based Architecture for Heterogeneous IoT Environments”. Teoksessa *Product-Focused Software Process Improvement,* toimittanut Davide Taibi, Marco Kuhrmann, Tommi Mikkonen, Jil Klünder ja Pekka Abrahamsson, 621–627. Lecture Notes in Computer Science. Cham: Springer International Publishing. ISBN: 978-3-031-21388-5. https://doi.org/10.1007/978-3-031-21388-5_47.

Kotilainen, Pyry, Viljami Järvinen, Juho Tarkkanen, Teemu Autto, Teerath Das, Muhammad Waseem ja Tommi Mikkonen. 2023. "WebAssembly in IoT: Beyond Toy Examples". Teoksessa *International Conference on Web Engineering,* 93–100. Springer.

Kruchten, P.B. 1995. "The 4+1 View Model of architecture". Conference Name: IEEE Software, *IEEE Software* 12, numero 6 (marraskuu): 42–50. ISSN: 1937-4194, viitattu 19. joulukuuta 2023. https://doi.org/10.1109/52.469759. https://ieeexplore.ieee.org/abstract/document/469759.

Lehmann, Daniel, Johannes Kinder ja Michael Pradel. 2020. "Everything Old is New Again: Binary Security of {WebAssembly}", 217–234. 29th USENIX Security Symposium (USENIX Security 20). ISBN: 978-1-939133-17-5, viitattu 26. helmikuuta 2024. https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann>.

McCormack, Ryan P., John E. Koontz ja Judith Devaney. 1999. "Seamless computing with WebSubmit". *Concurrency: Practice and Experience* 11 (15): 949–963. ISSN: 1096-9128, viitattu 19. lokakuuta 2023. https://doi.org/10.1002/(SICI)1096-9128(19991225)11:15<949::AID-CPE462>3.0.CO;2-Y. https://onlinelibrary.wiley.com/doi/abs/10.1002/%28SICI%291096-9128%2819991225%2911%3A15%3C949%3A%3AAID-CPE462%3E3.0.CO%3B2-Y.

Mcluckie, Craig. 2014. "Containers, VMs, Kubernetes and VMware". Google Cloud Platform Blog, 25. elokuuta 2014. Viitattu 3. lokakuuta 2023. https://cloudplatform.googleblog.com/2014/08/containers-vms-kubernetes-and-vmware.html.

Mikkonen, Tommi. 2023. Suullinen tiedonanto, 6. syyskuuta 2023.

Mikkonen, Tommi, Cesare Pautasso ja Antero Taivalsaari. 2021. "Isomorphic Internet of Things Architectures With Web Technologies". Conference Name: Computer, *Computer* 54, numero 7 (heinäkuu): 69–78. ISSN: 1558-0814. https://doi.org/10.1109/MC.2021.3074258.

Muchandi, Veer. 2007. *Applying 4+1 View Architecture with UML 2.* White paper. FCGSS. Viitattu 29. tammikuuta 2024. https://www.sparxsystems.com/downloads/whitepapers/FCGSS_US_WP_Applying_4+1_w_UML2.pdf.

Mutlag, Ammar Awad, Mohd Khanapi Abd Ghani, Mazin Abed Mohammed, Mashael S. Maashi, Othman Mohd, Salama A. Mostafa, Karrar Hameed Abdulkareem, Gonçalo Marques ja Isabel de la Torre Díez. 2020. "MAFC: Multi-Agent Fog Computing Model for Healthcare Critical Tasks Management". Number: 7 Publisher: Multidisciplinary Digital Publishing Institute, *Sensors* 20, numero 7 (tammikuu): 1853. ISSN: 1424-8220, viitattu 14. syyskuuta 2023. https://doi.org/10.3390/s20071853. https://www.mdpi.com/1424-8220/20/7/1853.

Mäkitalo, Niko, Tommi Mikkonen, Cesare Pautasso, Victor Bankowski, Paulius Daubaris, Risto Mikkola ja Oleg Beletski. 2021. "WebAssembly Modules as Lightweight Containers for Liquid IoT Applications" [kielellä en]. Teoksessa *Web Engineering,* toimittanut Marco Brambilla, Richard Chbeir, Flavius Frasincar ja Ioana Manolescu, 328–336. Lecture Notes in Computer Science. Cham: Springer International Publishing. ISBN: 978-3-030-74296-6. https://doi.org/10.1007/978-3-030-74296-6_25.

Namiot, Dmitry, ja Manfred Sneps-Sneppe. 2014. "On Micro-services Architecture". 2 (9).

Pan, Jianli, ja James McElhannon. 2018. "Future Edge Cloud and Edge Computing for Internet of Things Applications". Conference Name: IEEE Internet of Things Journal, *IEEE Internet of Things Journal* 5, numero 1 (helmikuu): 439–449. ISSN: 2327-4662, viitattu 3. lokakuuta 2023. https://doi.org/10.1109/JIOT.2017.2767608. https://ieeexplore.ieee.org/abstract/document/8089336.

Peffers, Ken, Tuure Tuunanen, Marcus A. Rothenberger ja Samir Chatterjee. 2007. "A Design Science Research Methodology for Information Systems Research". Publisher: Routledge _eprint: https://doi.org/10.2753/MIS0742-1222240302, *Journal of Management Information Systems* 24, numero 3 (1. joulukuuta 2007): 45–77. ISSN: 0742-1222, viitattu 12. syyskuuta 2023. https://doi.org/10.2753/MIS0742-1222240302. https://doi.org/10.2753/MIS0742-1222240302.

Ranjan, Rajiv, Boualem Benatallah, Schahram Dustdar ja Michael P. Papazoglou. 2015. "Cloud Resource Orchestration Programming: Overview, Issues, and Directions". Conference Name: IEEE Internet Computing, *IEEE Internet Computing* 19, numero 5 (syyskuu): 46–56. ISSN: 1941-0131. https://doi.org/10.1109/MIC.2015.20.

Red Hat, Inc. 2022. “What is container orchestration?”, 10. toukokuuta 2022. Viitattu 4. syyskuuta 2023. https://www.redhat.com/en/topics/containers/what-is-container-orchestration.

Shea, Ryan, Jiangchuan Liu, Edith C.-H. Ngai ja Yong Cui. 2013. “Cloud gaming: architecture and performance”. Conference Name: IEEE Network, *IEEE Network* 27, numero 4 (heinäkuu): 16–21. ISSN: 1558-156X, viitattu 25. tammikuuta 2024. https://doi.org/10.1109/MNET.2013.6574660. https://ieeexplore.ieee.org/abstract/document/6574660.

Shi, Weisong, Jie Cao, Quan Zhang, Youhuizi Li ja Lanyu Xu. 2016. “Edge Computing: Vision and Challenges”. Conference Name: IEEE Internet of Things Journal, *IEEE Internet of Things Journal* 3, numero 5 (lokakuu): 637–646. ISSN: 2327-4662. https://doi.org/10.1109/JIOT.2016.2579198.

Shi, Weisong, ja Schahram Dustdar. 2016. “The Promise of Edge Computing”. Conference Name: Computer, *Computer* 49, numero 5 (toukokuu): 78–81. ISSN: 1558-0814, viitattu 13. lokakuuta 2023. https://doi.org/10.1109/MC.2016.145. https://ieeexplore.ieee.org/abstract/document/7469991.

Shukla, Saurabh, Mohd Fadzil Hassan, Muhammad Khalid Khan, Low Tang Jung ja Azlan Awang. 2019. “An analytical model to minimize the latency in healthcare internet-of-things in fog computing environment”. Publisher: Public Library of Science, *PLOS ONE* 14, numero 11 (13. marraskuuta 2019): e0224934. ISSN: 1932-6203, viitattu 18. elokuuta 2023. https://doi.org/10.1371/journal.pone.0224934. https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0224934.

Stack Overflow. 2023. “Most used languages among software developers globally 2023”. Statista, 2. kesäkuuta 2023. Viitattu 26. helmikuuta 2024. https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/.

Suh, Sang-Bum, Joo-Young Hwang, Joon-Young Shim, JaeMin Ryu, Sungkwan Heo, Chan-Ju Park, ChulRyun Kim, Jae-Ra Lee, Ilpyoung Park ja Hosoo Lee. 2008. “Computing State Migration Between Mobile Platforms for Seamless Computing Environments”. Teoksessa *2008 5th IEEE Consumer Communications and Networking Conference,* 1216–1217. ISSN: 2331-9860, 2008 5th IEEE Consumer Communications and Networking Conference. Tam-

mikuu. Viitattu 19. lokakuuta 2023. https://doi.org/10.1109/ccnc08.2007.274. https://ieeexplore.ieee.org/document/4446572.

Taivalsaari, Antero, ja Tommi Mikkonen. 2017. “A Roadmap to the Programmable World: Software Challenges in the IoT Era”. Conference Name: IEEE Software, *IEEE Software* 34, numero 1 (tammikuu): 72–80. ISSN: 1937-4194. https://doi.org/10.1109/MS.2017.26.

Taivalsaari, Antero, Tommi Mikkonen ja Kari Systä. 2014. “Liquid Software Manifesto: The Era of Multiple Device Ownership and Its Implications for Software Architecture”. Teoksessa *2014 IEEE 38th Annual Computer Software and Applications Conference,* 338–343. ISSN: 0730-3157, 2014 IEEE 38th Annual Computer Software and Applications Conference. Heinäkuu. https://doi.org/10.1109/COMPSAC.2014.56.

“The 6 steps of the container lifecycle - Cloud computing news”. n.d. Viitattu 24. lokakuuta 2023. http://web.archive.org/web/20201012051642/https://www.ibm.com/blogs/cloud-computing/2016/02/08/the-6-steps-of-the-container-lifecycle/.

The Kubernetes Authors. 2023a. “Kubernetes Components”. Kubernetes. Section: docs, 18. marraskuuta 2023. Viitattu 18. marraskuuta 2023. https://kubernetes.io/docs/concepts/overview/components/.

———. 2023b. “Kubernetes Scheduler”. Section: docs, 18. marraskuuta 2023. Viitattu 18. marraskuuta 2023. https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/.

Thuan, Nguyen Hoang, Andreas Drechsler ja Pedro Antunes. 2019. *Construction of Design Science Research Questions.* 1. tammikuuta 2019.

Truyen, Eddy, Dimitri Van Landuyt, Davy Preuveneers, Bert Lagaisse ja Wouter Joosen. 2019. “A Comprehensive Feature Comparison Study of Open-Source Container Orchestration Frameworks”. Number: 5 Publisher: Multidisciplinary Digital Publishing Institute, *Applied Sciences* 9, numero 5 (tammikuu): 931. ISSN: 2076-3417, viitattu 17. lokakuuta 2023. https://doi.org/10.3390/app9050931. https://www.mdpi.com/2076-3417/9/5/931.

W3C. 2022. “WebAssembly Core Specification”, 19. huhtikuuta 2022. Viitattu 7. joulukuuta 2023. https://www.w3.org/TR/wasm-core-2/.

Vaño, Rafael, Ignacio Lacalle, Piotr Sowiński, Raúl S-Julián ja Carlos E. Palau. 2023. "Cloud-Native Workload Orchestration at the Edge: A Deployment Review and Future Directions" [kielellä en]. Number: 4 Publisher: Multidisciplinary Digital Publishing Institute, *Sensors* 23, numero 4 (tammikuu): 2215. ISSN: 1424-8220, viitattu 10. elokuuta 2023. https://doi.org/10.3390/s23042215. https://www.mdpi.com/1424-8220/23/4/2215.

WebAssembly Community Group. 2023. "component-model/design/high-level/Goals.md at main · WebAssembly/component-model". GitHub, 6. syyskuuta 2023. Viitattu 15. helmikuuta 2024. https://github.com/WebAssembly/component-model/blob/main/design/high-level/Goals.md.

Liitteet

A Moduulikuvasdokumentin esimerkki

Listaus 6.1. Esimerkki artefaktille yhteensopivasta moduulikuvausdokumentista

```
{
    "scale_image": {
        "method": "GET",
        "parameters": [
            {
                "name": "width",
                "type": "integer"
            },
            {
                "name": "height",
                "type": "integer"
            }
        ],
        "mounts": [
            {
                "name": "image.jpeg",
                "mediaType": "image/jpeg",
                "stage": "execution"
            },
            {
                "name": "scaled_image.jpeg",
                "mediaType": "image/jpeg",
                "stage": "output"
            }
        ]
    },
    "classify_image": {
        "parameters": [],
        "method": "POST",
        "output": "integer",
        "mounts": [
```

```
            {
                "name": "model",
                "mediaType": "application/octet-stream",
                "stage": "deployment"
            },
            {
                "name": "image.jpeg",
                "mediaType": "image/jpeg",
                "stage": "execution"
            }
        ]
    }
}
```

B Toimeenpanon manifestin esimerkki

Listaus 6.2. Esimerkki artefaktille yhteensopivasta toimeenpanon manifestin dokumentista

```
{
  "sequence": [
    {
      "device": "webcam",
      "module": "camera",
      "func": "take_image"
    },
    {
      "device": "compute-box",
      "module": "ml_inference",
      "func": "classify_image"
    }
  ]
}
```

C Tutkielman ulkoiset teknologiat

Taulukoissa 1 ja 2 on nähtävissä, mitä kolmannen osapuolen ja tälle tutkielmalle tarkkaanottaen ulkoisia kirjastoja artefaktin komponenteissa ja kehityksessä on käytetty sekä niiden lyhyesti kuvatut tehtävät.

	Teknologia[, versio]	**Rooli tai käyttökohde**
Ohjelmointikieli	Node.js, 18.16.0	Artefaktin pääohjelmointikieli
	Rust, 1.74.0	Testauksessa käytettävien WebAssembly-moduulien lähtökieli
	TypeScript, 5.3.2	Komentoriviasiakkaan ohjelmointikieli
Kirjasto	`bonjour-service`, 1.1.1	Laitteiden etsintä mDNS:llä
	`commander`, 11.1.0	Komentoriviasiakkaan käyttöliittymä
	`dotenv`, 16.0.3	Asetusten kuten verkkonimen tai etsittävän laitetyypin asettaminen ympäristömuuttujista
	Express, 4.18.2	HTTP-API palvelin
	`mongodb`, 4.13.0	Tietokannan käyttäminen
	`multer`, 1.4.5-lts.1	Tiedostojen vastaanottaminen API:n kautta
	`openapi-typescript-codegen`, 0.25.0	Komentoriviasiakkaan käyttämien HTTP-kutsujen generoiminen OpenAPI kuvauksesta
	`wasmiot-supervisor`, eb89e1a3f8d69bb65159 29c046ce8f94ae4b3774	Asiakasohjelmisto ja `wasm3_api` laitteilla
	`@types/node`, 20.10.1	Komentoriviasiakkaan Typescript-käännökseen tarvittavia tyyppimäärittelyjä

Taulukko 1. Artefaktissa käytetyt teknologiat (1/2)

	Teknologia[, versio]	Rooli tai käyttökohde
Tiedostomuoto	OpenAPI, 3.0.3	ReSTful API:n ja resurssien määrittely
	JSON	Resurssien sarjallistettu muoto
	WebAssembly, `wasm32-wasi`	Moduulit
Kehitystyökalu	Git, 2.34.1	Versionhallinta
	GitHub	Projektin hallinta ja varmuuskopiointi
	VSCode, 1.86.2	Integroitu kehitysympäristö
Muut	Docker, 24.0.5	Artefaktin käynnistys ja testausympäristö
	MongoDB, mongo@ sha256:d371c8693a1ce3974 c7bda0141fe38c063998e806 b9c35eaba28290af9d5973c (Docker Hub)	Resurssihakemiston tietokanta (konttikuva)

Taulukko 2. Artefaktissa käytetyt teknologiat (2/2)