Kanditutkielma

Tietojenkäsittelytieteen kandiohjelma

Järjestelmäintegraatioiden tyylit ja suunnittelumallit

Joona Halonen

14.7.2024

Matemaattis-luonnontieteellinen tiedekunta

Helsingin yliopisto

Yhteystiedot

PL 68 (Pietari Kalmin katu 5)
00014 Helsingin yliopisto

Sähkopostiosoite: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Batchelor's Programme in Computer Science

Tekijä — Författare — Author

Joona Halonen

Työn nimi — Arbetets titel — Title

Järjestelmäintegraatioiden tyylit ja suunnittelumallit

Ohjaajat — Handledare — Supervisors

Vanhenpi yliopiston lehtori Lea Kutvonen

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Bachelor's thesis	July 14, 2024	28 pages

Tiivistelmä — Referat — Abstract

Write your abstract here.

In addition, make sure that all the entries in this form are completed.

Finally, specify 1–3 ACM Computing Classification System (CCS) topics, as per https://dl.acm.org/ccs. Each topic is specified with one path, as shown in the example below, and elements of the path separated with an arrow. Emphasis of each element individually can be indicated by the use of bold face for high importance or italics for intermediate level.

ACM Computing Classification System (CCS)
Applied computing → Enterprise computing → Enterprise interoperability → Enterprise application integration
Software and its engineering → Software organization and properties → Contextual software domains → Operating systems → Communications management → Message passing

Avainsanat — Nyckelord — Keywords

järjestelmäintegraatiot, sanomat, suunnittelumallit, arkkitehtuuri

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Sisältö

1 Johdanto

Järjestelmäintegraatiot *(Enterprise Application Integration, EAI)* tarkoittaa ohjelmistoja, jotka yhdistävät eli integroivat organisaation olemassa olevia erillisiä järjestelmiä toimimaan yhdessä.

Tarve EAI:lle kehittyi kun organisaation jo olemassa olevien järjestelmien hajautunut tieto piti yhdistää tiedonkulun optimoimiseksi ja automatisoimiseksi. Esimerkiksi organisaation asiakkuudenhallinnasta vastaavan ohjelmiston pitäisi pystyä keskustelemaan myynninhallinnointiohjelmiston kanssa pitääkseen asiakastilastot ajan tasalla. Järjestelmät jäivät helposti erillisiksi saarekkeiksi eivätkä jakaneet tietoa keskenään. Syntyi niin sanottuja "savupiippujärjestelmiä", joissa järjestelmä oltiin suunniteltu pitämään data esimerkiksi firman yksittäisen osaston sisällä, eikä järjestelmän suunnitteluvaiheessa oltu huomioitu datan jakamista; järjestelmästä puuttui esimerkiksi ulospäin suuntavat ohjelmointirajapinnat.

Järjestelmäintegraatio mahdollistaa näiden eri järjestelmien keskinäisen keskustelun toisilleen. Näin voidaan rakentaa yhteinen järjestelmä jo olemassa olevista palasista [23, sivu 15]. Useissa tapauksissa organisaation sisäisiä järjestelmiä yhdistetään myös ulkoisiin järjestelmiin tai muiden organisaatioiden kanssa [22]. Organisaation olemassa olevat järjestelmät voi olla kehitetty eri käyttöjärjestelmille, hyödyntävät eri ohjelmointikieliä tai tietokantaratkaisuita. Väliin tarvitaan integraatio, joka hoitaa kommunikoinnin ohjelmistojen välillä

Vuonna 2020 EAI-markkinan arvioitiin olevan 9,26 miljardin dollarin (USD) arvoinen [26] ja koostuu ohjelmistojättien (Microsoft, Oracle, Salesforce IBM) ja pienempien yksittäisten yritysten (Workato, SnapLogic, Tibico) tarjoamista järjestelmistä.

Monet tämän päivän järjestelmäintegraatioalustat tarjoavat joukon työkaluja ja lähestymistapoja joilla, on paljon yhteisiä piirteitä keskenään. Tämän tutkielman tarkoituksena on avata, mistä nämä toisiaan muistuttavat lähestymistavat ovat peräisin, miten niihin on päädytty ja mikä on niiden asema tänä päivänä.

Luku 2 kuvaa, miten järjestelmäintegraatioita on lähestytty historiallisesti ja esittelee järjestelmäintegraatioiden merkittävintä kirjallisuutta.

Toinen luku paneutuu sanomapohjaisiin suunnittelumalleihin ja erityisesti järjestelmäintegraatioiden suunnittelumalleina *(enterprise integration patterns, EIP)* tunnettuihin lähestymistapoihin ja perehtyy niiden yläkategorioihin ja selittää itse mallien lähestymistapoja ja niiden perusteita. Suunnittelumallien tilannetta nykypäivässä käsitellään lyhyesti.

Kolmas luku käsittelee, miten järjestelmäintegraatioiden suunnittelumallit ovat asentuneet moderneja teknologioita käyttäviin järjestelmiin ja perehtyy, miten suunnittelumallien heikkouksina pidetyt piirteet ovat kehittyneet integraatioalustojen ympäristössä.

2 Järjestelmäintegraatioiden lähestymistavat

Esittelen tässä luvussa suosituimpia jäsentely ja lähestymistapoja järjestelmäintegraatioille. Eri lähestymistavat perustuvat viitatuimpaan järjestelmäintegraatioarkkitehtuurin kirjallisuuteen. Luvun tarkoituksena on antaa kattokategorioita erilaisille teknisille lähestymistavoille ja antaa historiallista kontekstia eri arkkitehtuurisuuntauksille.

2.1 Integraatiotasot

Integraatioiden lähestymistavat voidaan luokitella neljään eri tasoon [23, sivu 21-23]

1. Datatasolla dataa siirretään eri tietovarastojen välillä ja mahdollinen datan prosessointi ja muokkaus toteutetaan siirron yhteydessä. Kyseisellä menetelmällä on myös paljon yhtäläisyyksiä tyypillisen datavaraston *(data warehouse)* toteutuksen kanssa.

2. Rajapintatasolla pyritään hyödyntämään tarkoitusta varten tilattujen tai paketoitujen sovellusten, esimerkiksi SAP, PeopleSoft, tarjoamia ohjelmointirajapintoja.

3. Metoditasolla pyritään hyödyntämään jaettua sovelluslogiikka. Esimerkiksi hyödyntämällä sovelluspalvelimia *(application server)* metodienjakamiseen, käyttämällä hajautettuja objekteja tai etäkutsuja *(Remote Procedure Call, RPC)* hyödyntäviä teknologioita.

4. Käyttöliittymätasolla yhdistävä tekijä on olemassa olevan sovelluksen käyttöliittymä kun sovellus ei tarjoa muita keinoja datan jakamiseen. Tämä toteutetaan hyödyntymällä ruudun tiedonharavointi tekniikoita (screen scraping).

2.2 Tekniset lähestymistavat

Järjestelmäintegraatioiden arkkitehtuurin määrää pitkälti liiketoiminnan tarpeet ja niiden asettamat vaatimukset. Integraatioiden tekniset lähestymistavat voi luokitella neljään erilaiseen yläkategoriaan [18, sivu 64]:

1. tiedostojen siirto,
2. jaettu tietokanta,
3. etäproseduuri,
4. sanomat

Tiedostojen siirrossa integroitavat sovellukset voivat toimia itsenäisesti ja yhden sovelluksen muutokset eivät vaikuta toisen sovelluksen toimintaan kunhan sovellukset toimivat sovituilla tiedostotyypeillä ja tiedoston nimeämisstrategioilla. Integroijan vastuulla on taata, että tiedostot muutetaan toisen sovelluksen ymmärtämään muotoon. Lähestymistapana tiedostojen siirto on yksinkertainen eikä vaadi lisätyökaluja, koska yleisten tiedostotyyppien (JSON, XML, CSV) tuki löytyy käytännössä jokaisesta ohjelmointikielestä tai integraatiotyökalusta. Data-analytiikassa ja datavarastojen saralla on myös muita yleisiä tiedostomuotoja kuten Apache Parquet, jolle löytyy myös tuki useista ohjelmointikielistä ohjelmistokirjaston muodossa [5]. Sovitusta tiedostomuodosta tulee käytännössä integroitavien sovellusten välinen rajapinta.

Tiedostojen siirron heikkouksina on tiedostojärjestelmäoperaatioiden hallinnointi ja datan synkronointi. Integraatiokehittäjien vastuulle jää siis tiedostojen lukitseminen kirjoitusoperaatioiden ajaksi tai kirjoitusten ajoittaminen jotta ne eivät mene päällekkäin lukemisen kanssa, tiedostojen nimeämiskäytännöt, tiedostojen arkistointi ja poistaminen. Jos integroitavilla sovelluksilla ei ole pääsyä samoille levyille niin kehittäjien ratkaistavaksi jää myös tiedoston siirtäminen oikealle laitteelle. Datan synkronointi järjestelmien välillä tuo oman haasteensa, koska tiedostojen siirtoa tapahtuu yleensä harvakseltaan. Esimerkiksi jos asiakkuudenhallintajärjestelmä tuottaa tiedostoja datan synkronointia varten vain kerran päivässä ja laskutusjärjestelmä lähettää laskut aikaisemmin samana päivänä, niin osa laskuista on jo saattanut lähteä vanhaan osoitteeseen, jos asiakas on päivittänyt osoitetietojaan aikaisemmin saman päivän aikana.

Jaetun tietokannan etuna tiedostojen siirtoon on muutosten nopea propagoituminen eri sovelluksille. Samassa tietokannassa uusin tieto on saatavilla eri sovelluksille lähes välittömästi. Nopea tiedon liikkuminen tekee virheiden havaitsemisesta ja korjaamisesta helpompaa. Etuna on myös, että tietokantojen datamallit takaavat yhtenevän datan esitysmuodon verrattua tiedostojen siirtoon. Datan synkronoinnista ja kirjoitus- ja lukuvuoroista vastaa tietokannan hallintajärjestelmä *(DBMS, Database Management System)* ja transaktioiden hallinnointijärjestelmä antaa hyvät työkalut datan eheyden takaamiselle.

Gregor Hohpen ja Bobby Woolfin teos [18, sivu 69] korostaa myös SQL-pohjaisten relaatiotietokantojen yleisyyttä. Integraatiokehittäjien ei tarvitse opetella uutta teknologiaa tai

taistella uuden tiedostoformaatin kanssa vaan kehittäjät voivat työskennellä laajalti tunnettujen relaatiotietokantojen parissa. Valtaosa ohjelmointikielistä ja kehitystyökaluista tukee SQL:n kanssa työskentelyä joten jaetun tietokannan kanssa työskentely on suoraviivaista ja adoptointi helppoa.

Saman tietokannan käyttö estää datan tulkitsemiseen liittyvien ongelmien, kuten semanttisen dissonanssin *(semantic dissonance)* pitkittymistä, missä samaa dataa voidaan tulkita ristiriitaisilla tavoilla. Koska integroitavat sovellukset käyttävät samaa datalähdettä, niin nämä tulkintakysymykset on kohdattava varhaisessa vaiheessa integraatiokehitystä, eikä vasta tuotannossa jossa data voi olla jo yhteensopimatonta tulkintaeroista johtuen.

Jaetun tietokannan suunnitteluhaasteisiin sisältyy yhtenäisen skeemaan suunnittelu, jota useampi eri sovellus pystyy tehokkaasti hyödyntämään. Usein useamman sovelluksen tuomat vaatimukset johtavat monimutkaiseen tietokantaskeemaan jonka käytön kehittäjät kokevat haastavaksi. Yhtenäisen skeeman suunnittelua voi myös vaikeuttaa "poliittiset" haasteet, koska tietokannan suunnittelu voi johtaa aikataulujen venymiseen ja kommunikaatiohaasteisiin tietokantaa hyödyntävien eri yksiköiden välillä. Lisää suunnitteluhaasteita tuo ulkoiset ohjelmistot. Lähes poikkeuksetta kaupalliset ohjelmistot tukevat vaan omaa ohjelmiston mukana tulevaa tietokantaformaattia eivätkä taivu siitä poikkeavaan tietokantaskeemaan. Vastaavia haasteita tuovat sovellukset, jotka on peritty toiselta organisaatiolta esimerkiksi yrityskaupan yhteydessä. Sovellusten jälkeenpäin tehtävä jatkokehittäminen jaettua tietokantaa hyödyntäväksi on yleensä työlästä ja kallista. Kun jaettuun tietokantaan yhdistettyjen sovellusten määrä lisääntyy, ratkaisu voi aiheuttaa suorituskykyhaasteita, varsinkin jos luku- ja kirjoitusoperaatiot kohdistuvat vaan muutamaan tietokantatauluun. Jos sovellukset on hajautettu useammalle laitteelle ja tietokanta niiden kanssa, jotta sovelluksilla on lokaali pääsy kantaan, niin hajauttaminen tuo omat haasteensa. Pääosin datan hajauttamistaktiikkojen muodossa ja lisää näin ratkaisun kompleksisuutta.

Etäproseduuri *(Remote Procedure Invocation)* eroaa edellisistä lähestymistavoista. Aikaisemmat lähestymistavat keskittyivät pääosin datan jakamiseen, mutta näissä lähestymistavoissa pienet datanmuutokset voivat johtaa eri toimintoihin useiden sovellusten taholta. Osoitteen vaihto voi olla yksinkertainen kentän muutos tai laukaista useita rekisteröinti- ja lakiprosesseja useassa eri sovelluksessa. Jaettu tietokanta ei mahdollista minkäänlaista datan kapselointia ja tämä yksi iso datalähde tekee datamuutosten havaitsemisesta ja muutosten vaatimien prosessien aktivoimisesta haastavaa. Tiedostojen siirto tarjoaa suoraviivaisen tavan reagoida datan muutoksen, mutta tämä tapahtuu yleensä viiveellä johtuen tiedostojen synkronoimisen haasteista. Jaetun tietokannan kapseloimattomuus tarkoittaa myös integraatioiden ylläpidon joustamattomuutta. Muutokset yhdessäkään integroidussa sovelluksessa vaikuttavat jaettuun tietokantaan ja tietokantamuutokset voivat aiheuttaa kauas kantautuvia

muutoksia tietokantaa käyttävien sovellusten kesken.

Etäproseduuri mahdollistaa mekanismin, jossa sovellus voi kutsua toisen sovelluksen funktiota, jakaa vain tarvittavan datan ja kutsua funktiota jonka tarkoitus on kertoa datan vastanottajalle, miten toimia jaetun datan kanssa. Jos sovellus tarvitsee toisen sovelluksen dataa se voi kysyä sitä suoraan toiselta sovellukselta. Jokainen sovellus vastaa oman datansa eheydestä ja jokainen sovellus voi tehdä muutoksia omaan dataansa, vaikuttamatta muiden sovellusten tilaan.

Etäproseduurien mahdollistavat teknologiat ovat myös yleisiä ja tuttuja kehittäjille. Etäproseduurikutsu *(Remote Procedure Call, RPC)* teknologiat ja kirjastot ovat tunnettuja ja yleisesti käytettyjä. Esimerkkeinä teoksessa [18, sivu 71] listataan teknologoina CORBA, COM, .NET Remoting ja Java RMI ja todetaan, että web-palveluiden yleistyessä http-yhteyksiä hyödyntävät lähestymistavat, kuten SOAP ja XML ovat yleistyneet kehityksessä. Teoksen julkaisun jälkeen REST ja JSON ovat pitkälti korvanneet SOAP:in ja XML:än web-palveluiden suosituimpana lähestymistapana.

Etäproseduureilla on mahdollisuus vähentää semanttista dissonanssia, koska sovellukset voivat tarjota usean erilaisen rajapinnan samalle datalle. Eri asiakasohjelmille voidaan tarjota erilainen datanesitysmalli riippuen siitä, mikä asiakasohjelma on kyseessä. Tämä antaa enemmän mahdollisuuksia esittää datan useammalla eri tavalla verrattuna pelkkään relationaaliseen malliin. Useammat eri rajapinnat tarkoittavat lisää työtä integraatiokehittäjille datan muokkaamisen parissa ja integroitavien sovellusten täytyykin neuvotella keskenään mitä rajapintoja ne toisiltansa käyttävät.

Etäproseduurien helppous kehittäjille voi luoda myös haasteita jos integraatiokehittäjät eivät tiedosta etäkutsujen suorituskyky- ja luotettavuuseroja verrattuna paikallisiin kutsuihin. Tiheä etäkutsujen käyttö voi pinota näitä ongelmia ja johtaa hitaaseen ja epäluotettavaan järjestelmään.

Vaikka etäproseduurin mahdollistama datan kapselointi vähentää sovelluksen kiinteitä kytköksiä karsimalla suuren yhteisen datalähteen, niin se voi silti aiheuttaa solmukohtia. Kytkösten solmukohdat korostuvat kun kyse on jaksossa - tietyssä järjestyksessä tehtävistä- operaatioista. Integraatiojärjestelmistä näistä muodostuu helposti ongelma, koska vastaavat kytkökset eivät välttämättä aiheuttaisi ongelmia yksittäisessä sovelluksessa, mutta useamman sovelluksen integraatiossa lisäkytkökset tarkoittavat lisäviivettä ja ylimääräisiä verkkokutsuja.

Sanomat*(Messaging)* eroavat muista lähestymistavoista. Aikaisemmat integraatioiden lähestymistavat keskittyivät joko datan tai toiminnallisuuden jakamiseen. Gregor Hohpen ja Bobby Woolfin mukaan yleinen integraationkehityksen haaste on saada eri järjestelmät toi-

mimaan yhdessä mahdollisimman viipeettä ilman. Samalla välttäen kytköksiä järjestelmien välillä, jotka tekevät järjestelmästä epäluotettavan joko sovelluksen suorittamisen tai kehittämisen kannalta [18, sivu 72]. Tiedostojen siirrossa datan siirtyminen ei ole tarpeeksi viipeetöntä ja sovellusten välinen toiminta tarpeeksi sujuvaa vaikka lähestymistapa estääkin rajoittavien kytkösten muodostamisen. Datan siirtyminen vaatii kalliita levy- ja tiedosto-operaatioita ja tiedosto-operaatioiden tilanhallinta ja synkronointi muodostuu nopeasti monimutkaiseksi. Jaetussa tietokannassa datan jakaminen on implisiittisesti helppoa ja datan muutokset ovat responsiivisia, mutta kaikki sovellukset ovat kytköksissä samaan tietokantaan ja lähestymistapa ei mahdollista sovellusten yhteistä toimintaa. Etäproseduurien heikkoudet olivat yleiset hajautettujen järjestelmien riskit, kuten verkkoviiveet ja verkon luetettavuus, varsinkin jos etäkutsuja käytetään samalla lailla kuin paikallisia kutsuja ja lähestymistavassa sovellusten pitää jakaa tietoa toistensa rajapinnoista, mikä lisää kehitystä vaikeuttavien kytkösten määrää.

Sanomien käyttö erityisesti asynkroniseen viestintään on aikaisemmin esiteltyjen lähestymistapojen parhaiden puolien yhdistelmä [18, sivu 73]. Sanomien käyttö mahdollistaa pienien tiheästi kulkevien datapakettien lähetyksen ja tallettamisen ja tiedosto-operaatioiden yksityiskohtien abstraktoinnin. Tämä mahdollistaa nopeat skeeman muutokset vastaten yrityksen tarpeisiin. Sovellukset pystyvät jakamaan toiminnallisuuksiaan lähettämällä sanoman toinen toiselleen joka herättää esimerkiksi datanmuokkausproseduurin. Asynkroninen viestintä ei taas vaadi vastaanottajan olemaan saatavilla lähesty hetkellä ja asynkroninen viestintä ohjaa kehittäjiä ymmärtämään, että etäyhteyksien käyttäminen on hitaampaa ja suunnittelemaan korkeamman koheesion komponentteja, jolloin etänä tehtävien operaatioiden käyttö on harkitumpaa. Sanomapohjaiset järjestelmät myös mahdollistavat tiedostojen siirron kaltaisten löyhien kytkösten käytön. Sanomia voidaan muokata kesken lähetyksen ilman, että lähettäjä- tai vastaanottajasovelluksen tarvitsee olla tietoinen muokkausoperaation yksityiskohdista. Tämä mahdollistaa integroijien yleislähettävän *(broadcast)* sanomia useammalle vastaanottajalle, valitsevan yhden vastaanottajan useamman joukosta tai valita useasta muunlaisesta topologiasta jotka sallivat integraation irrottamisen sovelluksen kehitysprosessista. Sanomien tiheä lähettäminen mahdollistaa säännöllisen datan jakamisen lisäksi toiminnallisuutta. Käsittelyprosessi voidaan käynnistää heti kun yksittäinen sovellus saapuu ja asynkronisten kutsujen avulla lähettävän sovelluksen suoritus ei keskeydy odottamaan vastausta.

Sanomien lähetyksen tiheä datanvaihto ei kuitenkaan estä semanttisen dissonanssin syntymistä, varsinkin kuin datan esitysmuoto voi vaihtua useasti sanomia muokatessa. Sanomien lähetyksen tiheys ei myöskään täysin poista samoja datan synkronointihaasteita, joita tiedostojen siirrossa ilmeni. Sanomien siirrossa on jonkin verran viiveitä ja niiden ajoituksella tulee edelleen olemaan merkitystä. Asynkronisuus tuo myös lisähaasteensa integraatioiden

kehitys vaiheessa. Testaus ja sovellusten virheenpaikannus tulee olemaan monimutkaisempaa sanomien lähetyksen rinnakkaisuuden takia ja vaati integraatiokehittäjiltä jonkin verran lisätotuttelua. Sanomien käytön löyhät kytkennät lisäävät integroitavien sovellusten koheesioita, mutta tarkoittavat kuitenkin vaikeammin ylläpidettävän "liimakoodin"tarvetta jotta integraatiot saadaan toimimaan yhdessä.

Edellä mainitut haasteet tarkoittavat sanomapohjaisille järjestelmille suunniteltuja lähestymistapoja ja arkkitehtuureja mitkä toistuvat järjestelmissä niiden yksittäisistä eroista huolimatta.

3 Sanomapohjaiset suunnittelumallit

3.1 Suunnittelumallien asema ja kategorisointi

Edellisessä luvussa esitellyistä neljästä kategoriasta Hohpe ja Woolf suosivat sanomien käyttöä integraatioratkaisuissa ja suurin osa teoksesta keskittyy sanomapohjaisen suunnittelumallien esittelyyn [18, sivu 76]. Nämä järjestelmäintegraatioiden suunnittelumalleina *(enterprise integration patterns, EIP)* tunnetut kehitysohjeet ovat vaikuttaneet useamman eri integraatioalustan arkkitehtuuriin.

Vuonna 2018 viimeisintä tekniikka edustavista 48: sta integraatioalustasta yksitoista tuki EIP-malleja [7]. 12 vuotta kirjan julkaisun jälkeen pidetyssä haastattelussa kirjailijat sanovat, että useimmat avoimen lähdekoodin liikepalveluväylät *(enterprise service bus, ESB)* ovat omaksuneet teoksessa esitellyn mallikielen *(pattern language)* [36]. Samassa haastattelussa Hohpe ja Woolf toteavatkin, että kirjan pysyminen relevanttina yli 12 vuotta julkaisun jälkeen on harvinaista tietokoneaiheisille kirjoille [36]. Woolf selittää kirjan pitkän vaikutuksen johtuvan kirjan keskittymisestä suunnittelumalleihin eikä spesifisiin teknologioihin [36].

Hohpe ja Woof esittelevät 67 sanomapohjaista suunnittelumallia ja nämä mallit on jaettu seitsemään eri kategoriaan [18]:

1. sanomajärjestelmät *(messaging systems)*,
2. sanomakanavat *(messaging channels)*,
3. sanomien rakentaminen *(message construction)*,
4. sanomien reititys *(message routing)*,
5. sanomien muokkaus *(message transformation)*,
6. sanomien päätepisteet *(messaging endpoints)* ja
7. järjestelmän hallinta *(systems management)*.

Tässä tutkielmassa perehdytään vain sanomajärjestelmiin, koska muissa kategorioissa esitetyt EIP:eet sisältyvät sanomajärjestelmissä esiteltyihin malleihin.

3.2 Sanomajärjestelmät

Sanomajärjestelmät kategoria toimii kattokategoriana muille kategorioille ja tarjoaa konseptit ja termit muille suunnittelumalleille. Kategoria esittelee lähtökohtaiset pohjasuunnittelumallit minkä päälle muut suunnittelumallit rakentavat.

Sanomajärjestelmät *(messaging systems)* on kattokategorian nimi EIP:lle, mutta viittaa myös järjestelmään joka jakelee sanomia. Vaihtoehtoinen termi sanomajärjestelmille on sanomapohjainen väliohjelmisto *(message-oriented middleware)*. Hohpe ja Woof käyttävät termiä vastaavasti teoksessaan [18]. Jos tekstissä viitataan sanomajärjestelmiin kattokategoriana se tarkennetaan erikseen, muussa tapauksessa sanomajärjestelmä viitta edellä mainittuun ohjelmistojärjestelmään joka kuljettaa sanomia.

Sanomakanava *(Message Channel)*: Kanavia käytetään sanomien organisointiin ja jaotteluun sanomajärjestelmän sisällä. Kanava toimii eräänlaisena virtuaalisena putkena lähettäjän ja vastaanottajan välillä. Kehittäjän vastuulle jää kanavien luonti ja organisointi. Sovelluksen lähettäessä dataa, dataa ei lähetetä satunnaisesti mille tahansa kanavalle vaan lähettävä sovellus tietää juuri oikean kanavan jolle data on tarkoitettu. Vastaavasti sama dynamiikka kuuluu myös vastaanottavalle sovellukselle; sovellus tietää juuri oikean kanavan mistä odottaa oikeanlaatuista dataa. Kanavat toimivat loogisina osoitteina sanomien kululle. Kanavan toteutusyksityiskohdat vaihtelevat eri sanomia käyttävien järjestelmien välillä [10] [18]. Kanavien käyttö on laskentakapasiteetin osalta halpaa, mutta ei täysin ilmaista; jokainen kanava tarvitsee muistia sanomien datan esittämiseen ja pysyvien sanomien tapauksessa – sanomat jotka säilyvät levyllä esimerkiksi virhetilanteessa – myös levytilaa. Kanavia kannattaa siis luoda useita, mutta lukumäärän kasvaessa tuhansiin, alkaa sanomien hallintajärjestelmän skaalautuvuudesta tulla haaste [18].

Sanoma *(Message)*: Jos kanavaa voi ajatella putkena niin sanomaa voi ajatella putken vetenä, paitsi virran sijaan putkessa data kulkee yksittäisinä datayksiköinä. Kuten monen muunkin protokollan kohdalla, niin myös sanomakin sisältää otsakkeen *(header)* ja hyötykuorman *(payload)*. Sanomajärjestelmät eivät ota kantaa hyötykuorman sisältöön, mutta vastaanottava sovellus voi reagoida eri tavoila hyötykuorman sisältöön. Saapuva sanoma voi aktivoida proseduurin, välittää dataa, ilmoittaa sovellusta tilamuutoksesta tai vaatia sovelluksesta vastausta [18]. Sanomat voi myös lähettää sarjana jos lähetettävä data ei mahdu yhteen sanoman hyötykuormaan. Jos datalla on rajallinen voimassaoloaika, voi sanoma tarkentaa lähetetyn datan voimassa olo ajan [18]. Hohpen ja Woofin teos sisältää suunnittelumallit näille erilaisille sanoman käyttötavoille.

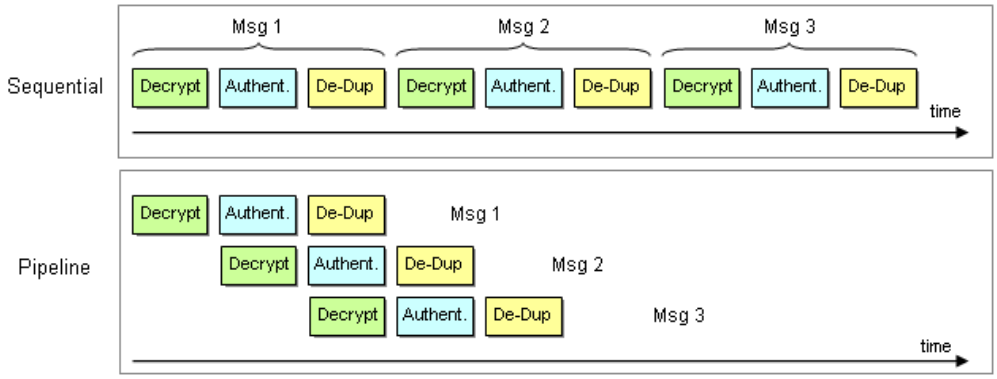
Putket ja suodattimet *(Pipes and filters)*: Putket ja suodattimet suunnittelumallissa putkena toimivat edellä mainitut kanavat ja suodattimet ovat pieniä itsenäisiä prosessointiaskeleita joilla ei ole kytköksiä muihin prosessointiaskeleihin. Kanavat yhdistävät nämä suodattimet toisiinsa ja suodattimet voivat toimia yksin tai olla yhdistettynä toisiinsa. Vaihtoehtoisesti putkena voi toimia muistissa oleva jono. Suodattimien rajapinta pitää olla tarpeeksi yksinkertainen, että putkien implementaatio on vaihdettavissa ja kiinteitä kytköksiä muihin suodattimiin tai putkiin ei synny. Asynkroninen viestintä komponenttien välillä mahdollistaa suodattimien samanaikaisen käytön. Suodattimien löyhät kytkökset helpottavat testaamista kun suodattimia voi testata itsenäisinä komponentteina jonka lopputuloksen oikeudellisuutta voi helposti verrata siihen sisälle syötettyyn dataan.

Putket ja suodattimet lähestymistapana on myös saanut suosiota integraatioalustoiden arkkitehtuureissa; neljästäkymmenestäkahdeksasta modernista integraatioalusta kymmenen hyödyntää putket ja suodattimet arkkitehtuurityyliä [7]. Putket ja suodattimet arkkitehtuurille löytyy myös implementaatio-ohjeita monelle alustalle ja teknologialle. Internet haulla putket ja suodattimet suunnittelumallille löytyy ohjeet seitsemän eri teknologian virallisilta sivuilta, Amazon Web Services, Red Hat Fuse, Spring Integration, Microsoft Azure, Mulesoft Runtime ja Apache Camel [2] [9] [33] [24] [27] [6].

Suodattamien käyttö ei ole täysin ilmaista. Suodattamien skaalautuvuuden haasteena on se, että jokainen suodatin tarvitsee oman putkensa datan kuljettamiseen. Jos putken toteuttaa sanomakanavana niin näiden käyttö ei myöskään ole täysin ilmaista kuten aikaisemmassa suunnittelumallissa on mainittu; vastaavasti muistissa olevat jonot kuluttavat luonnollisesti muistiresursseja. Myös sanomien kulku putkissa ja suodattimista tuo omat haasteensa, koska viestit pitää käsitellä kanavan formaatista sovellukseen käyttämään formaattiin ja sama prosessi pitää tehdä takaperin sanoman lähtiessä, syöden operaatiosyklejä. Pitkät suodatinketjut tarjoavat arkkitehtuurista joustavuutta löyhien kytköstensä ansioista, mutta maksavat potentiaalisen suoritustehon laskemisen muodossa, johtuen toistuvista datan esitystavan muutoksista.

Asynkronisten sanomakanavien hyödyntäminen mahdollistaa jokaisen suodatinyksikön käsitellä sanoman itsenäisesti omassa säikeessään tai prosessissaan. Tämä johtaa huomattavasti suurempaan suoritustehoon kun yksikään suodatinyksikkö ei ole odottamassa sanoman käsittelyn päättymistä vaan seuraavaa sanomaa voidaan alkaa käsittelemään enne kuin edellinen on valmistunut. Hohpe ja Woolf kutsuvat tätä prosessointiputkeksi *(processing pipeline)* jota selventää kuva 3.1 [18] [10] Kuvassa 3.1 sanoman salaus puretaan *(decrypt)*, autentikoidaan *(authent.)* ja lopuksi tarkistetaan duplikaateilta *(de-dup)*.

Peräkkäisessä käsittelyssä, eli sanomat odottavat edellisen sanoman käsittely loppumista,

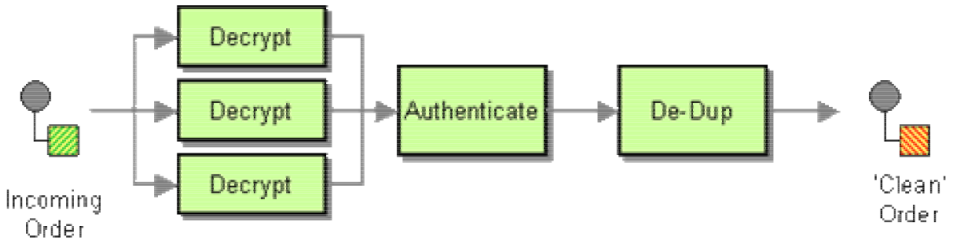


Kuva 3.1: Kuva sanomien peräkkäisen käsittelyn ja prosessointiputken eroista [18]

sanomat joutuvat odottamaan toisten sanomien käsittelyä. Alemmassa prosessointiputkessa asynkronisuus mahdollistaa suodattimien operaatiot heti kun seuraava sanoma saapuu.

Suodatinketjun suorituskykyä voidaan parantaa tästä entisestään. Edellisessä 3.1 prosessointiputken esimerkissä heikkoutena on se, että kaikesta hitain suodatinprosessi määrää koko järjestelmän sanomien käsittelytahdin. Tässä tapauksessa hitain suodatin voidaan rinnakkaistaa. Haasteena on varmistaa, että jokaisen sanoman voi käsitellä vain yksi rinnakkaisista suodatinprosesseista. Toinen huomioon otettava aspekti on, että rinnakkaistamminen ei takaa sanomien suoritusjärjestyksen säilymistä. Jos sanomien käsittelyjärjestyksellä on väliä, suodatinprosesseja voi olla vain yksi millä tahansa hetkellä. Kolmas haaste on suodattimien tilanhallinta. Tilanhallinta rinnakkaisissa operaatioissa lisää tunnetusti kompleksisuutta, niin suodattimen tilanpalautuminen lähtöpisteeseen operaation valmistuttua yksinkertaistaa huomattavasti suodattimien rinnakkaistamista.

Osittainen rinnakkaistaimisen esimerkissä 3.2 Hohpe ja Woof näyttävät miten salauksen purkamisen *(decrypt)* käytetty suodatin voidaan rinnakkaistaa kolmeksi rinnakkaiseksi prosessiksi, koska purkamisoperaatio on työläin suodattimista [18]. Esimerkissä duplikaattien tar-



Kuva 3.2: Kuva sanomien rinnakkaisesta käsittelystä kun järjestyksellä on väliä [18]

kastus *(de-dup)* on pidetty peräkkäisenä prosessina, koska duplikaattien havaitseminen tar-

vitsee sanomahistorian hyödyntämistä eli operaatio ei ole tilaton ja sen rinnakkaistaminen on haastavaa.

Sanomareititin *(Message Router)*: Putkien ja suodattimien löyhät kytkennät mahdollistavat uuden prosessointiaskeleen lisäämisen olemassa olevien väliin. Sanomareitittimen tapauksessa reititin vastaa putkien ja suodattamien suodatinta siinä suhteessa, että se tekee suorittaa prosessointia, mutta suodattimien sijaan reititin on kytketty useampaan ulostulo kanavaan. Reititin ei myöskään muokkaa sanomaa, vain on ainoastaan keskittynyt sanomien määränpään valitsemiseen. Reitittimen etuna on löyhä kytkentä muihin komponentteihin; ympärillä olevien suodattimien ei tarvitse olla tietoisia reitittimen olemassaolosta ja kaikki reitityspäätöksien logiikka elää yhdessä sijainnissa. Jos uusia sanomatyyppejä tai reititysmuutoksia toteutetaan, niin ainoastaan sanomareitittimen logiikka tarvitsee muokata. Koska saapuvan kanavan kaikki viestit siirtyvät reitittimen läpi yksitellen, on saapuvien viestien käsittelyjärjestys myös taattu. Löyhän kytkennän ylläpitohaasteet korostuvat sanomareititintä hyödyntäessä. Jos suurin osa järjestelmästä on löyhästi kytketty toiseen niin järjestelmän kokonaiskuvan muodostaminen käy hankalaksi. Hohpe ja Woofin mukaan näitä haasteita voi lieventää hyödyntämällä sanomien kulkuhistoriaa [18, sivu 93].

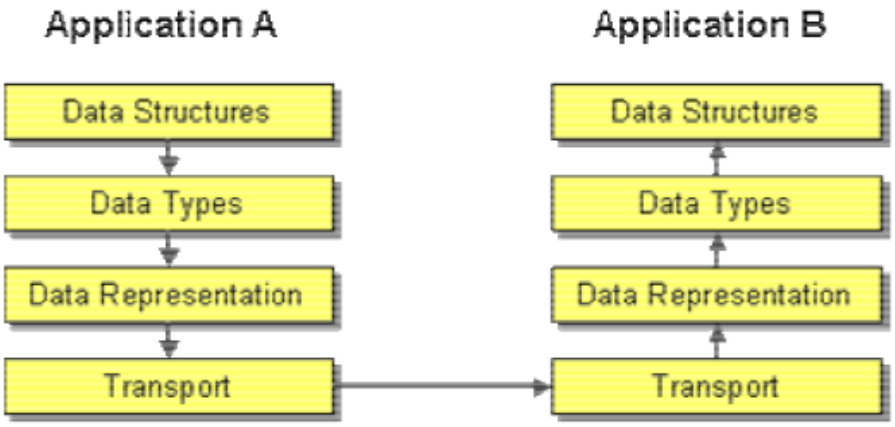
Reitittimen haasteina ovat erilaiset pullonkaulat. Reitittimestä voi syntyä ylläpidollinen pullonkaula, koska reitittimen pitää olla tietoinen kaikista vastaanottavista kanavista ja tästä voi syntyä ylläpidollinen ongelma varsinkin jos vastaanottokanavien lista vaihtuu tiheään. Toinen mahdollinen pullonkaula on suorituskyky. Reititin luonteensa vuoksi lisää ylimääräisen prosessointiaskeleen ja monissa sanomajärjestelmissä sanoman siirtäminen toiselle kanavalle vaati sanomien uudelleen koodausta. Useamman reitittimen rinnakkainen käyttö lieventää ongelmaa, mutta joka tapauksessa reititin tulee vaikuttamaan negatiivisesti sanomien viiveeseen vaikka sanomien käsittelytahti pysyisikin samana [18, sivu 93].

Reitittimen käyttö voidaan kategorioida useampaan eri tyyppiin. Yksinkertaisin reititin on kiinteä, jolloin sanoma kulkee yhdestä kanavasta yhteen vastaanottaja kanavaan. Kiinteiden reitittimen tarkoitus on irrottaa kiinteitä kytkentöjä erilaisista alijärjestelmistä tai välittää sanomia eri integraationratkaisujen välillä. Reititin voi olla sisältöpohjainen *(content-based)* tai ympäristöpohjainen *(context-based)*. Sisältöpohjainen tekee reitityspäätöksiä sanoman sisällön perusteella. Ympäristöpohjainen ottaa huomioon ympäristön tilan ja tekee esimerkiksi kuormituksen tasaamista tai uudelleenreititystä ympäröivän systeemin virhetilanteessa. Reitittimet voi jakaa tilallisiin ja tilattomiin; tilalliset reitittimet ottavat huomioon aikaisemmat sanomat ne voivat esimerkiksi estää duplikaattisanomien saapumisen tallentamalla jo saapuneet sanomat. Reitin voi olla myös dynaaminen *dynamic router* ja vastaanottaa konfiguraatiomuutoksia ohjausväylän kautta. Tällä tavalla reititin voi muutta reitityslogiikkaansa ilman koodimuutoksia [18, sivu 94].

Sanomankääntäjä *(Message Translator)*: Sanomankääntäjä on suodatintyyppi, joka mahdollistaa eri dataformaatteja käyttävät sovellukset keskustelemaan keskenään sanomien avulla. Hohpe ja Woof vertaavat sanomankääntäjä EIP:tä sanomajärjestelmille tarkoitetuksi versioksi tunnetusta sovitinsuunnittelumallista [18, sivu 97], joka sai alkunsa Gang of Four teoksesta [8]. Kertauksena aikaisempiin esiteltyihin EIP: hin verrattuna sanomakääntäjän tarkoituksena on vähentää tiukkoja riippuvuuksia arkkitehtuurissa; sanomakanava mahdollisti, että sovellusten ei tarvitse tietää toistensa sijaintia, sanomareititin mahdollisti löyhät kytkennät sovellusten viestien reitittämisessä, joten sanomakääntäjä estää järjestelmien tiukat riippuvuudet toistensa dataformaateista. Sanomakääntäjä ratkaisee jaetun tietokannan haasteita jota käsiteltiin luvussa 2.2. Erityisesti yhtenäisen skeemaan löytäminen ei muodostu ongelmaksi ja järjestelmät on kytketty löyhemmin toisiinsa kääntäjiä hyödyntämällä. Vaihtoehtoinen lähestymistapa sanomakääntäjän sijaan olisi käyttää yhteistä datatyyppiä koko sanomajärjestelmässä, joka jakaa myös yhtäläisyyksiä jaetun tietokannan kanssa 2.2, toteuttamalla sanoman datatyypin muunnokset sanomien päätepisteisiin 3.2, mutta valmiissa kaupallisissa sanomajärjestlemiä tukevissa sovelluksissa päätepisteen koodi ei ole muokattavissa. Lisäksi sanomien kääntämislogiikan eläessä päätepisteessä, koodin uudelleenkäyttäminen on haastavaa.

Hohpe ja Woof [18] jakavat sanomakääntäjän operaatiot eri abstraktiotasoihin, ottaen löyhästi vaikutteita OSI-mallista [21], jakaen kääntämisoperaatiot kuljetus-, datan esitys-, datatyyppi- ja tietorakennekerroksiin (sovelluskerros). Kuljetuskerrokseen kuuluvat protokollat kuten http, mqtt, JMS. Datan esityskerros on kiinnostunut datan muodosta kuten JSON, XML ja enkoodauksesta kuten UTF-8. Datatyyppi viittaa soveluksen muistinsisäisiin esitystapoihin esim. ovatko postinumerot merkkijonoja vai kokonaisnumeroita ja ovatko esim. aikaleimat aina UTC-ajassa vai sisältävätkö aikavyöhykeinformaation. Tietorakennekerros (tai sovelluskerros) ottaa kantaa loogisiin kokonaisuuksiin, kuten eri dataentiteettien assosiaatioihin ja niiden kardinaliteetin esim. onko dataentiteettien välillä moni-moneen-yhteyksiä. Jakamalla kääntämisoperaatiot useampaan kerrokseen voidaan operaatioita käsitellä itsenäisesti omassa abstraktiotasossaan, kuten kuvassa 3.3, ilman riippuvuutta muihin kääntäjiin ja kääntämisoperaation koodin uudelleenkäyttäminen on suoraviivaista. Myös eri abstraktiotasojen operaatiot ovat vaihdettavissa toisiin. Esimerkiksi datan esityskerroksessa *(Data Representation)* voidaankin päättää, että data esitetäänkin JSON:ina CSV:een sijaan ilman, että vaihto aiheuttaa muutoksia muiden tasojen operaatioissa.

Sanomien päätepiste *(Message Endpoint)*: Verrattuna aikaisempiin esiteltyihin kattokategorian EIP:in, sanoman päätepiste on puuttuva palanen, joka mahdollistaa integroitavan sovelluksen liittämisen osaksi sanomaviestintää hyödyntävää kokonaisuutta; sanomien päätepiste on siis asiakasohjelmisto, joka yhdistää sovelluksen sanomakanavaan ja mahdollistaa



Kuva 3.3: Sanomakääntäjän operaatiot voivat kohdistua kaikkiin abstraktiotasoihin [18].

sanomien lähettämisen tai vastaanottamisen. Sanomajärjestelmät ovat palvelinohjelmia ja sanomien päätepiste toimii niiden asiakasohjelmana. Sanomien päätepiste vastaanottaa sovelluksen komennon tai datan ja muuttaa sen sanomaksi ja lähettää sanoman sanomakanavalle ja päinvastoin sanomia vastaanottavalle päätepisteelle. Hohpe ja Woof rajaavat yhden päätepisteinstanssin vain vastaanottamaan tai lähettämään, eikä toteuttamaan molempia eli sovellus hyödyntäisikin useampaa päätepistettä jos sen pitäisi kommunikoida useammalle sanomakanalle [18, sivu 106]. Sovellus voi kuitenkin hyödyntää useampaa päätepisteinstanssia vaikka sanomat lähtisivätkin vai yhdelle kanavalle, mahdollistaakseen instanssien skaalautumisen useammalle samanaikaiselle säikeelle.

Päätepisteen koodi on räätälöity kyseiselle sovellukselle ja sanomajärjestelmän asiakasrajapinnalle sopivaksi. Päätepiste kapseloi sanomajärjestelmän sovellukselta joten sovellus ei ole tietoinen sanomajärjestelmästä. Esimerkiksi sanomajärjestelmän asiakasrajapinnan muutokset vaativat vain muutoksia päätepisteen koodiin ja integroitava sovellus on kapseloitu niin etteivät nämä muutokset vaikuta sen toimintaan.

Päätepiste EIP:estä on muokattuja versioita riippuen siitä haluaako sanomia vastaanottava päätepiste käyttää erilaista vastaanottostrategiaa kuten kiertokyselyä tai tapahtumapohjaista reagointia. Useammat vastaanottavat päätepisteet voivat kilpailla sanomien käsittelystä *(computing consumer)* tai jakaa viestin lähettämisen eri päätepisteelle sisäisen logiikan mukaan *(message dispatcher)*. Vastaanottava päätepiste voi vastaavasti olla idempotenssi niin toistuivat sanomat eivät aiheuta järjestelmässä virhetiloja [18, sivu 106].

4 Järjestelmäintegraatioiden suunnittelumallien kehitystä

4.1 Suunnittelumallien rooli nykypäivän ympäristössä

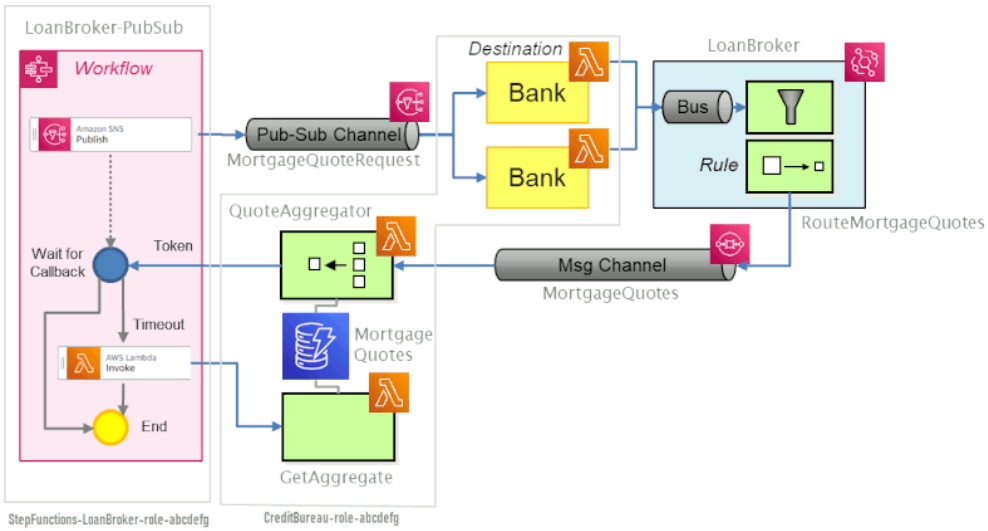
EIP-teoksen [18] julkaisun jälkeen kirjailijat (pääasiassa Gregor Hohpe) ovat pyrkineet pitämään sisällön relevanttina. Vaikka kirjan arkkitehtuurisisältö itsessään on pysynyt relevanttina ja vaikuttaa selkeästi nykypäivän järjestelmäintegraatioratkaisuissa ja asynkronisten sanomien käytössä, on kirjan koodiesimerkit ja teknologiamaininnat vanhentuneet; näitä on myös tässä tutkielmassa pyritty korjaamaan tuoreemmilla esimerkeillä, jotka ovat tämän päivän opiskelijalle tuttuja.

Hohpen henkilökohtainen blogi sisältää useamman päivitetyn esimerkin tunnetuista EIP-suunnittelumallista, mutta nyt toteutettuna käyttäen pilviteknologioita:

- Sisältöpohjainen reititin, EIP:een kategoriassa sanomien reititys, modernisoitiin käyttäen Googlen pilvialustan *(GCP)* Google Cloud Functions palvelitonta arkkitehtuuria vuonna 2017 [14]. Alkuperäisen teoksen C# Microsoft Message Queuing pohjainen toteutus vaihtui Node.js pohjaiseen, Google Cloud Pub/sub kirjastoa hyödyntävään toteutukseen.

- EIP-kirjassa esitelty monimutkaisempi esimerkki lainanvälittäjästä *(Loan Broker)* [18, sivu 317] toteutettuun kolmella eri tavalla: Java / Apache Axis web-palvelimena, C# / Microsoft Message Queuing sanomajonona ja Tibco ActiveEnterprise julkaise ja tilaa kanavana. Vuonna 2021 Hohpe aloitti sarjan blogikirjoituksia, joissa lainanvälittäjä esimerkki toteutetaan uudelleen käyttäen Amazon Web Services *(AWS)* pilviteknologioita [11]. Lainanvälittäjä esimerkki toteutettiin pilvituotteilla AWS Step Functions, AWS Lambda, AWS Simple Notification Service, AWS Simple Queuing Service ja AWS DynamoDB. Katso kuva 4.1.

  Myöhemmissä sarjan osissa Hohpe keskittyy ohjelmiston julkaisemiseen AWS:ää hyödyntäen [15] [16]. Hohpe myös argumentoi, että EIP-suunnittelumallit ovat itse pilviarkkitehtuuriratkaisun lisäksi hyödyllisiä abstraktioita koodin julkaisua automatisoidessa [16]. Julkaisussa käytettävät teknologiat ovat AWS CloudFormation, AWS Simple Storage Service, AWS Serverless Application Model ja AWS Cloud Development Kit.



Kuva 4.1: Kuva EIP:een lainanvälittäjä esimerkistä AWS:än palveluita hyödyntämällä [15].

- Hohpen esimerkkien modernisointi jatkuu samana vuonna, lainanvälittäjä esimerkin siirtämisenä AWS:ltä Google Cloudille [17]. Ohjelman siirto ja kuvan 4.1 AWS palvelut kuvaantuivat yksi yhteen GCP:een palveluille. Esimerkissä hyödynnettiin GCP Workflows, GCP Cloud Functions, GCP PubSub, GCP DataStore teknologoioita. Julkaisun automaatiossa huomattavaa oli, että AWS CloudFormationille ei löytynyt suoraa vastaavuutta GCP:een tarjonnasta ja vastaava toiminnallisuus pitää löytää kolmannen osapuolen ratkaisuista [17].

EIP-teoksessa esiteltyjen suunnittelumallien esimerkkejä on myös vähitellen uudistettu EIP:een kotisivulle [13]. Tällä hetkellä 67 suuniittelymallista 14 on modernisoitu. Modernisoidut on koottu yhteen paikkaan Hohpen blogiin [12]. Aiemmin mainittujen pilvialustaesimerkkien lisäksi EIP-suunnittelumalleja on kyseisissä esimerkeissä toteutettua Golangilla, Apache Kafkalla, Apache Camelilla, Mule ESB:llä, RabbitMQ:lla ja Microsost Azurella.

Edellä mainitut esimerkit, mukaan lukien listatut pilvialustaesimerkit, löytyvät julkisesta Githubissa isännöidystä tietovarastosta [19].

4.2 Tilalliset protokollat

EIP-suunnittelumallien heikkous on tilalliset *(stateful)* protokollat. Tarve tilallisille protokollille on todettu 2017 integraatiotrendien analyysissä [29] Tämän lisäksi alkuperäisien EIP-kirjan kirjailijat ilmaisivat vuonna 2016, että tilallisien protokollien suunnittelumalleille on tarvetta, mikä voisi oikeuttaa toisen EIP-voluumin luomiseen [36]. Hohpe on alkanut ke-

räämään tilallisia suunnittelumalleja sivulle [13], mutta toistaiseksi suunnittelumallit eivät ole poikenneet uutta kirjaa. Verrattuna EIP-suunnittelumalleihin, Hohpen tilalliset mallit käyttävät sanomien sijaan keskusteluja *(conversations)* jonka ympärille suunnittelumallien abstraktiot on rakennettu.

Integraatiotrendien ja kaupallisten sekä avoimen lähdekoodin alustojen analyysissä todettiin, että lähes kaikki alustat tarjoavat omat tilalliset "keskustelunsa" ja mahdollisuudet tallennustilan käyttöön [29]. Tosin tämä alustojen tarjoama tuki todetaan alkeelliseksi. Palvelukeskeisen arkkitehtuurin *(Service Orieted Architecture, SOA)* kirjallisuudesta löytyy kuitenkin keskustelusuunnittelumalleja mitä ei löydy integraatioalustojen toteutuksista [29]; julkaisussa [1] suunnittelumallit yhdestä-moneksi *(one-from-many)* ja yhdestä-moneen *(one-to-many)* ovat monenvälisiä keskustelusuunnittelumalleja jotka todetaan puuttuvan olemassa olevista toteutuksista [29].

Koska nykypäivän integraatiot keskittyvät enemmän pilvipalveluhin, integraatiovaatimukset ovat alkaneet kallistua tilallisiin tallennustilaa vaativiin skenaarioihin mihin EIP ei vastaa ja kerätyt keskustelusuunnittelumallit ovat myös keskeneräisiä vastatakseen [29]. Paremmin muodostetuille keskustelusuunnittelumalleilla löytyisi tarvetta.

Huomioitavaa on, että keskustelusuunnittelumalleja ei voi kuitenkaan verrata palvelukeskeisen arkkitehtuurin koreografia- *(choreography)* tai vuorovaikutussuunnittelumalleille *(interaction patterns)*, koska keskustelumallit kuvaavat monimutkaisempia tehtäviä kuin datan lähettämisen ja vastaanottamisen [29].

4.3 Datavirtaprotokollat

Synkronisia ja datavirtaprotokollia *(streaming protocol)* ei ole huomioitu järjestelmäintegraatioissa eikä niiden suunnittelumalleissa, minkä Hohpe ja Woof myöntävät haastattelussa [36]. Erityisesti virtausprotokollien puute aiheuttaa haasteita esimerkiksi big data järjestelmien integraatioissa [29]. Samassa haastattelussa Hohpe ja Woof toteavatkin, että virtausprotokollien suunnittelumallien dokumentointi parantaisi sanomien käytön ja virtauskäyttötapausten yhtäläisyyksien ymmärtämistä ja johtaisi EIP:een kaltaisten suunnittelumallien löytämiseen [29]. Kirjailivat toteavat, että koko EAI ekosysteemistä puuttuu datavirtaprotokollien suunnittelumallit ja yhteiset protokollat [36], mutta virtaprosessointia hyödynnettiin vuosi myöhemmin ainakin Jitterbitin ja Apache Camelin EAI-ratkaisuissa [29].

Virtausdatan käsittelemistä EIP-malleilla on kokeiltu laitteistokiihdyttämisen kanssa hyödyntämällä ohjelmoitavia porttimatriiseja [28]. Tutkimuksessa virtausdatan käsittelyä varten EIP-malleja laajennettiin kahdella uudella suunnittelumallilla: kuormituksen tasaaja *(load*

balancer) ja liittäjä reititin *(join router)*. Kuormituksen tasaaja lähettää sanomaan yhteen useasta kanavasta hyödyntämällä yleisiä kuormituksen tasaus mekanismeja. Liittäjä reititin liittää useasta eri kanavasta saapuvat sanomat yhdelle kanavalle ohjelmoidun logiikan mukaisesti [28].

Akateeminen ja tekninen kirjallisuus integraatioista ja datavirroista jää vähälle vuoden 2017 jälkeen. Datavirtojen hallinnoinnista EAI-kontekstissa löytyy kuitenkin joku tuoreempi esimerkki [34], mutta kirja ei pyri suunnittelumallien kaltaiseen arkkitehtuuristandardisointiin. Tuoreita kaupallisten integraatioalustojen kirjoituksia datavirtojen hyödyntämisestä on julkaistu ja yksi suuri datavirtaus integraatioiden suosion ajaja näyttää olleen Apache Kafka [35] [20]; samanlaisia trendejä on myös nähtävissä dataintegraatioiden tutkimuspapereissa [3].

4.4 Virheenhallinta suunnittelumalleissa

EIP-lähdeteos käsitteli niukasti virheenhallintaa. Teos sisälsi virheenhallinnan suunnittelumalleina kelvottomien sanomien kanavan *(Invalid Message Channel)* johon sanomat, joiden käsittely päättyi virheeseen lähetetään, sekä toimittamattomien kirjeiden kanavan *(Dead Letter Channel)* johon päätyneiden sanomien lähetys on epäonnistunut eikä sanoma pystynyt jatkamaan käsittelyään esimerkiksi järjestelmähäiriön takia [18]. Kirjailijat ovat myöhemmin avanneet virheenhallinnan sisällyttämisestä teokseen [18] sillä, että virheenhallintastrategioiden käsittely vaatii kirjalta laajempaa sanastoa erityisesti tilanhallinnasta, mikä olisi paisuttanut teosta [36].

Kaupallisissa ja avoimen lähdekoodin EAI-järjestelmissä virheenhallinta ja erityisesti poikkeuksien hallinnointi on kehittynyt pidemmälle kuin EIP-kirjallisuus [29]. Järjestelmät sisältävät yleensä systeemin kaikkien virheiden nappaamisen ja kaupalliset palvelut kuten Dell Boomi, IBM, SAP Cloud Integration ja Tibco sisältävät hienovaraisempia työkaluja virheiden vaikutusalan hallinnoimiseen [29].

Tutkimukset [32] [31] ovat kartoittaneet EAI-järjestelmistä käytettyjä virheenhallintastrategioita ja pyrkineet mallintamaan niitä suunnittelumalleina. Virheenhallinnasta on siis tehty myös varsin kattavaa kartoitusta suunnittelumallien perspektiivistä. Tapahtuman uudelleen yritys virheen tapahtuessa *(retry pattern)* on yleinen suunnittelumalli joka ilmenee sanomakanavan tai yksittäisen tapahtuman tasolla ja yrittää operaatiota annetun määrän kertoja [32]. Virheitä hallinnoidaan vikatilanne reitittimellä *(failover router)* jossa sanoma reitietään vaihtoehtoisella kanavalle virheen tapahtuessa. Tämä suunnittelumalli eroaa alkuperäisen teoksen [18] kiertotie EIP:stä *(detour)* tarkastamalla jatkokäsittelyn EIP:tä kutsua ja

reitittää vaihtoehtoiselle reitille ilman konfiguraatiosanomia. Kompensointialue *(compensation sphere)* on suunnittelumalli jossa prosessi tai aktiviteetti sisältää joukon korjaavia operaatioita (kompensointeja) jotka aktivoituvat eri virhetilanteista. Kompensoinnit voivat olla vain tiettyyn tilanteeseen reagoivia tai koko määriteltyyn alueeseen reagoivia [32].

Tämän jälkeen EAI-järjestelmien virheenhallinta strategioita on tarkemmin lajiteltu erilaisiin suunnittelumalleihin [31]. Järjestelmien virheenhallintatavat kategorisointiin yhteentoista samankaltaiseen lähestymistapaan, jotka on esitetty taulukossa 4.2.

Table 2. Exception handling strategies and patterns overview showing "known uses" (i.e. covered: √, partially covered: (√), not covered: –).

Category	Pattern Name	Known Uses				Example
		Flume[4]	Nifi[5]	Camel[3]	HCI[51]	
Tolerance	Message Redelivery on Exception[47]	√	(√)	√	(√)	Try again on condition
	Delayed Retry	√	–	√	(√)	Delay processing
	Delayed Channel	√	√	√	√	Delay operations connection channel
	Failover Router[47,52]	√	–	√	√	Fallback to alternative route
	Skip Step[47]	(√)	–	√	√	Ignore one operation
	Pause Operation	–	√	(√)	(√)	Delay one operation until error fixed
Escalation	Stop Local[47]	–	(√)	√	√	Stop one operation
	Stop All[47]	–	–	√	√	Stop context
	(Re-) Throw Exception	(√)	–	√	√	Message cannot be processed further
	Raise Indicator	–	√	(√)	√	Send e-mail, monitor
	Timeout[47]	(√)	(√)	(√)	√	Unblocking operation
	Back-Pressure	(√)	(√)	(√)	(√)	Ask endpoint to stop sending
Handling	Catch Local[47,52]	√	(√)	√	(√)	Handle within one operation
	Catch-all[47,52]	√	(√)	√	√	Global process handler
	Invalid Message Channel[26]	√	–	(√)	(√)	Separate invalid messages
	Dead Letter Channel[26]	√	–	√	(√)	Separate situationally
	Exception Sphere	√	(√)	√	(√)	Define exception scope
Compensation	Compensation Sphere[47,52]	(√)	–	(√)	(√)	Cleanup on failure
Prevention	Message Validator	(√)	√	(√)	√	Validate schema, required fields
	Message Throttler	(√)	–	√	–	Max. number of message per time
	Message Sampler	(√)	(√)	√	(√)	Skip any nth message, period
	Load Balancer	(√)	(√)	√	(√)	Distribute load across resources

Kuva 4.2: Taulukko EAI-systeemien virheenhallinnan suunnittelumalleista ja niiden kategorioista [31].

Tutkimuksessa arvioitiin Apache Flume, Apache Cameli, Apache Nifi ja SAP HCI EAI-järjestelmien virheenhallintatapoja. Jos virheenhallintatapa esiintyi useammin kuin kerran,

virheenhallintatapa sisällytettiin suunnittelumalliksi [31].

4.5 Suunnittelumallien formalisointi

EIP-suunnittelumallien formalisointi on kiinnostava aspekti erityisesti akateemisen kirjallisuuden kontekstissa. Formalisointi mahdollistaa suunnittelumallien matemaattisen analyysin ja varmentamisen ja takaisi varman pohjan järjestelmäintegraatioiden rakentamiselle suunnittelun ja ajon aikana.

Vuoden 2017 kartoitus [29] löysi yhden yrityksen formalisoida EIP:tä [4] jonka metodi formalisoinnille on hyödyntää värjättyjä Petri-verkkoja jossa värit kuvaavat datatyyppejä ja hallintasäikeet *(control threads)* mallintavat kontrollirakenteita verkossa. Tämän jälkeen formalisointityötä on laajennettu vuonna 2021 [30], jonka formalisointimetodi perustuu DB-verkoille *(DB-nets)* [25]; tutkimuksessa on pyritty rakentamaan vastuullinen formalisointitapa EIP-suunnittelumalleille, mikä täyttää EAI:in vaatimukset ohjausvuon, datan, ajan ja transaktioiden suhteen [30]. Jatkotutkimukseen tarpeen tutkimuksen tekijät löytävät EIP-yhdistelmien formalisoinnista, koska nykyinen tutkimus keskittyi yksittäisten EIP-suunnittelumallien formalisointiin [30].

Formalisoinnin tutkimus on tuoretta ja keskenräistä verrattuna EIP-suunnittelumallien olemassaolon pituuteen. Suhteutettuna siihen kuinka paljon EIP:tä käytetään kaupallisissa ja avoimissa EAI-alustoissa, on nykyisen formalisoinnin keskeneräisyys harmillista, mutta viime vuosien uusiutunut kiinnostus aiheesta on lupaavaa. Ylipäätään EAI-teknologioiden formalisointi EIP-mallien ulkopuolella on keskittynyt palvelukeskeisen arkkitehtuurien ratkaisujen formalisointiin [29].

5 Yhteenveto

Suunnittelumalli	Tuki	Tilallinen/Tilaton	Modernisoitu	Formalisointi
Jakaja *(Splitter)*	• 12/15 alustassa täysi tuki [29]. • 1/15 alustassa osittainen tuki [29].	• Tilaton. Jakaja jakaa vain yhden sanoman sisällön eikä säilytä tilaa.	• Ei	• DB-net formalisointi esimerkki [30].
Sanomakääntäjä *(Message Translator)*	• 8/15 alustassa täysi tuki [29]. • 5/15 alustassa osittainen tuki [29].	• Tilaton. Kääntäjä kääntää vain yhden sanoman sisällön eikä säilytä tilaa.	• Ei	• DB-net formalisointi esimerkki [30]. • Värjätty Petri-verkko formalisointi esimerkki [4].
Sisältöpohjainen reititin *(Content Based Router)*	• 11/15 alustassa täysi tuki [29]. • 1/15 alustassa osittainen tuki [29].	• Tilaton. Reititin ei muista aikaisempi sanomia.	• Kyllä. Modernisoitu esimerkki Apache Camelille [12].	
Sisällön rikastaja *(Content Enricher)*	• 6/15 alustassa täysi tuki [29]. • 6/15 alus-	• Tilaton. Rikastaja ei muista aikaisempia	• Kyllä. Modernisoitu esi-	

Lähteet

[1] A. Barros, M. Dumas ja A. H. T. Hofstede. ”Service Interaction Patterns”. *Lecture Notes in Computer Science* 3649 (2005), s. 302–318. ISSN: 1611-3349. DOI: 10.1007/11538394_20.

[2] J. Beswick. *Application integration patterns for microservices: Orchestration and coordination, AWS Compute Blog*. 2024. URL: https://aws.amazon.com/blogs/compute/application-integration-patterns-for-microservices-orchestration-and-coordination/ (viitattu 02. 02. 2024).

[3] A. Bousdekis ja G. Mentzas. ”Enterprise Integration and Interoperability for Big Data-Driven Processes in the Frame of Industry 4.0”. *Frontiers in Big Data* 4 (2021), s. 644651. ISSN: 2624909X. DOI: 10.3389/FDATA.2021.644651/BIBTEX.

[4] D. Fahland ja C. Gierds. ”Analyzing and Completing Middleware Designs for Enterprise Integration Using Coloured Petri Nets”. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 7908 LNCS (2013), s. 400–416. ISSN: 1611-3349. DOI: 10.1007/978-3-642-38709-8_26.

[5] A. S. Foundation. *Apache Parquet*. 2023. URL: https://parquet.apache.org/ (viitattu 11. 08. 2023).

[6] A. S. Foundation. *Pipeline :: Apache Camel*. 2024. URL: https://camel.apache.org/components/4.0.x/eips/pipeline-eip.html (viitattu 02. 02. 2024).

[7] D. L. Freire, R. Z. Frantz, F. Roos-Frantz ja S. Sawicki. ”Survey on the run-time systems of enterprise application integration platforms focusing on performance”. *Software: Practice and Experience* 49 (3 2019), s. 341–360. ISSN: 1097-024X. DOI: 10.1002/SPE.2670.

[8] E. Gamma, R. Helm, R. Johnson ja J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994, 139ff. ISBN: 0-201-63361-2. (Viitattu 20. 04. 2024).

[9] R. Hat. *4.4. Pipes and Filters Red Hat JBoss Fuse 6.0, Red Hat Customer Portal*. 2024. URL: https://access.redhat.com/documentation/en-us/red_hat_jboss_fuse/6.0/html/implementing_enterprise_integration_patterns/msgsys-pipes (viitattu 02. 02. 2024).

[10] G. Hohpe. *Enterprise Integration Patterns*. 2024. URL: https://www.enterpriseintegrationpatterns.com (viitattu 26. 01. 2024).

[11] G. Hohpe. *Loan Broker Implementation with AWS Step Functions - Enterprise Integration Patterns*. 2021. URL: https://www.enterpriseintegrationpatterns.com/ramblings/loanbroker_stepfunctions.html (viitattu 03. 05. 2024).

[12] G. Hohpe. *Modern Examples for Enterprise Integration Patterns - Enterprise Integration Patterns*. 2017. URL: https://www.enterpriseintegrationpatterns.com/ramblings/eip1_examples_updated.html (viitattu 03. 05. 2024).

[13] G. Hohpe. *Overview - Enterprise Integration Patterns 2*. 2017. URL: https://www.enterpriseintegrationpatterns.com/patterns/conversation/index.html (viitattu 02. 03. 2024).

[14] G. Hohpe. *Serverless Integration Patterns on Google Cloud Functions - Enterprise Integration Patterns*. 2017. URL: https://www.enterpriseintegrationpatterns.com/ramblings/google_cloud_functions.html (viitattu 03. 05. 2024).

[15] G. Hohpe. *Serverless Loan Broker @ AWS, Part 4: Automation - Enterprise Integration Patterns*. 2021. URL: https://www.enterpriseintegrationpatterns.com/ramblings/loanbroker_automation.html (viitattu 03. 05. 2024).

[16] G. Hohpe. *Serverless Loan Broker @ AWS, Part 5: Integration Patterns with CDK - Enterprise Integration Patterns*. 2022. URL: https://www.enterpriseintegrationpatterns.com/ramblings/loanbroker_cdk.html (viitattu 03. 05. 2024).

[17] G. Hohpe. *Serverless Loan Broker @ GCP - Enterprise Integration Patterns*. 2022. URL: https://www.enterpriseintegrationpatterns.com/ramblings/loanbroker_gcp_workflows.html (viitattu 03. 05. 2024).

[18] G. Hohpe ja B. Woolf. "Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions: Hohpe, Gregor, Woolf, Bobby: 9780321200686: Amazon.com: Books". *Addison-Wesley Professional; 1 edition* (2004), s. 736. URL: https://books.google.com.cu/books?hl=es&lr=&id=bUlsAQAAQBAJ&oi=fnd&pg=PR7&dq=HOHPE,+G.+AND+WOOLF,+B.+Enterprise+Integration+Patterns:+Designing,+Building,+and+Deploying+Messaging+Solutions.+edited+by+I.+PEARSON+EDUCATION.+Edtion+ed.+Boston,+MA,+USA:+Addis.

[19] G. Hohpe ja B. Woolf. *GitHub - spac3lord/eip: Code for modern EIP examples*. 2023. URL: https://github.com/spac3lord/eip (viitattu 04. 05. 2024).

[20] IBM. *Enterprise Integration: What It Is and Why It's Important - IBM Blog*. 2021. URL: https://www.ibm.com/blog/enterprise-integration/ (viitattu 18. 05. 2024).

[21] ISO. *ISO/IEC 7498-1:1994 - Information technology — Open Systems Interconnection — Basic Reference Model: The Basic Model*. URL: https://www.iso.org/standard/20269.html.

[22] P. Johannesson ja E. Perjons. "Design principles for process modelling in enterprise application integration". *Information Systems* 26 (3 2001). ISSN: 03064379. DOI: 10.1016/S0306-4379(01)00015-1.

[23] D. S. Linthicum. *Enterprise application integration*. 2000.

[24] Microsoft. *Pipes and Filters pattern - Azure Architecture Center, Microsoft Learn*. 2024. URL: https://learn.microsoft.com/en-us/azure/architecture/patterns/pipes-and-filters (viitattu 02. 02. 2024).

[25] M. Montali ja A. Rivkin. "DB-nets: On the marriage of colored petri nets and relational databases". *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 10470 LNCS (2017), s. 91–118. ISSN: 16113349. DOI: 10.1007/978-3-662-55862-1_5/FIGURES/5.

[26] Mordorintelligence. *Enterprise Application Integration Market: Growth, Trends, Forecasts (2020 - 2025)*. 2020. URL: https://www.mordorintelligence.com/industry-reports/enterprise-application-integration-market (viitattu 11. 05. 2020).

[27] Mulesoft. *Enterprise Integration Patterns Using Mule, MuleSoft Documentation*. 2024. URL: https://docs.mulesoft.com/mule-runtime/latest/understanding-enterprise-integration-patterns-using-mule (viitattu 02. 02. 2024).

[28] D. Ritter, J. Dann, N. May ja S. Rinderle-Ma. "Industry paper: Hardware accelerated application integration processing". *DEBS 2017 - Proceedings of the 11th ACM International Conference on Distributed Event-Based Systems* 12 (2017), s. 215–226. DOI: 10.1145/3093742.3093911.

[29] D. Ritter, N. May ja S. Rinderle-Ma. "Patterns for emerging application integration scenarios: A survey". *Information Systems* 67 (2017), s. 36–57. ISSN: 0306-4379. DOI: 10.1016/J.IS.2017.03.003.

[30] D. Ritter, S. Rinderle-Ma, M. Montali ja A. Rivkin. "Formal foundations for responsible application integration". *Information Systems* 101 (2021), s. 101439. ISSN: 0306-4379. DOI: 10.1016/J.IS.2019.101439.

[31] D. Ritter ja J. Sosulski. "Exception Handling in Message-Based Integration Systems and Modeling Using BPMN". 25 (2 2016). ISSN: 02188430. DOI: 10.1142/S0218843016500040.

[32] D. Ritter ja J. Sosulski. ”Modeling Exception Flows in Integration Systems”. *Proceedings . IEEE 18th international Enterprise Distributed object computing conference* 2014-December (December 2014), s. 12–21. ISSN: 15417719. DOI: 10.1109/EDOC.2014.13.

[33] Spring. *3. Spring Integration Overview*. 2024. URL: https://docs.spring.io/spring-integration/docs/4.2.5.RELEASE/reference/html/overview.html (viitattu 02. 02. 2024).

[34] S. Wilkes ja A. Pareek. *Streaming Integration [Book]*. O’Reilly Media, Inc, 2020. ISBN: 9781492045816. URL: https://www.oreilly.com/library/view/streaming-integration/9781492045823/.

[35] WSO2. *Streaming Integration Overview - WSO2 Enterprise Integrator Documentation*. URL: https://ei.docs.wso2.com/en/7.0.0/streaming-integrator/quick-start-guide/getting-started/getting-started-guide-overview/ (viitattu 18. 05. 2024).

[36] O. Zimmermann, C. Pautasso, G. Hohpe ja B. Woolf. ”A decade of enterprise integration patterns: A conversation with the authors”. *IEEE Software* 33 (1 2016), s. 13–19. ISSN: 07407459. DOI: 10.1109/MS.2016.11.