

Matti Turpeinen

DYNAAMISESTI TYYPITETTYJEN VERKKO-OHJELMISTOKEHysten TEHOKKUUSVERTAILU

Informaatioteknologian ja viestinnän tiedekunta
Pro gradu -tutkielma
Toukokuu 2023

TIIVISTELMÄ

Matti Turpeinen: Dynaamisesti tyypitettyjen verkko-ohjelmistokehysten tehokkuusvertailu
Pro gradu -tutkielma
Tampereen yliopisto
Tietojenkäsittelytieteiden tutkinto-ohjelma
Toukokuu 2023

Verkkoteknologioiden ja -kehysten suuren käyttömäärän vuoksi olisi hyödyllistä tarkastella niiden käytön tehokkuutta REST- ja WebSocket-konteksteissa, sillä niiden käyttömäärien perusteella pienetkin erot voivat vaikuttaa suuresti eri yhteyksissä. Tutkielmassa tarkastellaan joidenkin suosittujen dynaamisesti tyypitettyjen verkkokehysten tehokkuutta, jotta voidaan löytää niiden reaaliaikaisen tehokkuuseroja, sekä täydentää olemassa olevia tutkimuksia.

Tutkielmassa tarkastellaan ensin verkkorajapintakehysten Express, Django, FastAPI, Flask, Laravel, sekä Ruby on Rails ja niiden ohjelmointikielten tyylejä ja malleja, jotta niistä saadaan hyvä yleiskuva. Onkin huomattavaa, että edellisten kehysten tehokkuudet ovat osittain ristiriidassa keskenään aiemmissa tutkimuksissa, jota tällä tutkielmalla pyritään osaltaan paikkaamaan. Tätä on tutkittu rakentamalla uniikit, mutta naiivit verkkopalvelimet tutkituista kehyksistä, sekä asiakasohjelma, joka pyrkii rasittamaan näitä palvelimia. Mainittu asiakasohjelma mittaa samalla eri palvelinkehysten aikoja antaen kuvaa niiden tehokkuudesta asiakkaan näkökulmasta.

Tutkielmassa kerätystä aineistosta voidaan huomata, että mitattavat kehykset tulisi jakaa erikseen täysien pinokehysten kategoriaan, sekä muiden kehysten kategoriaan näiden tarjoaman erilaisen ympäristön ja käyttötarkoituksen vuoksi, jonka oletetaan olennaisesti vaikuttavan tehokkuuteen. Tutkielman ja muiden tutkimusten tuloksista voidaan päätellä, että kehyksistä ilmeisesti joko FastAPI tai Flask on tehokkain ohjelmistokehys REST-kontekstissa, sekä Ruby on Rails on tehokkain täysistä pinokehyksistä. Vastaavasti WebSocket-käytössä Rails vaikuttaisi olevan nopein kaikista kehyksistä, kun taas epätäysistä kehyksistä nopein vaikuttaisi olevan FastAPI. Vaikkakin tutkielmassa huomautetaan aiheen erittäin suuresta laajuudesta, voidaan molemmille verkkoprotokollille suositella käytetyllä palvelinympäristöllä 512 limittäisen yhteyden raja-arvoa. Vastaavasti testaamisen raja-arvoksi voidaan suositella 128 limittäistä yhteyttä testatulla arkkitehtuurilla.

Avainsanat: REST, WebSocket, verkko-ohjelmistokehys, web-api, tehokkuus

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck –ohjelmalla.

SISÄLLYS

1	Johdanto	1
1.1	Tausta	1
1.2	Ongelman asettelu ja rajaukset	2
2	Web-rajapintakehysten teknologiat.....	3
2.1	Tulkatut kielet	3
2.2	Verkkorajapinnat ja REST yleisesti	4
2.3	WebSocket yleisesti	5
2.4	JavaScript ohjelmointikielenä	7
2.5	NodeJS ja Express-ohjelmistokehys	8
2.6	Python ohjelmointikielenä	10
2.6.1	Python Django-ohjelmistokehyksellä	11
2.6.2	Python Flask-ohjelmistokehyksellä	12
2.6.3	Python FastAPI-ohjelmistokehyksellä	13
2.7	Ruby ohjelmointikielenä ja Ruby on Rails -ohjelmistokehyksellä	14
2.8	PHP ohjelmointikielenä ja Laravel-ohjelmistokehyksellä	16
3	Kehysten vertailu aiemmissä tutkimuksissa	19
3.1	Aiemmat tutkimukset	20
3.2	Tehokkuustestaustyökalujen hyödyntäminen	22
4	Verkko-ohjelmistokehysten tehokkuusvertailu.....	23
4.1	Tutkimusmenetelmä	23
4.2	Testidata	26
4.3	Datan välitys tietokannasta	27
4.4	Päivystävä WebSocket-rajapinta	28
4.5	Testausympäristön tiedot	30
5	Tulokset	31
5.1	Datan välitys tietokannasta	31
5.1.1	Django	34
5.1.2	Flask	35

5.1.3	FastAPI	36	
5.1.4	Express	37	
5.1.5	Ruby on Rails		39
5.1.6	Laravel	40	
5.1.7	Tasapainotetut tulokset	41	
5.2	Päivystävä WebSocket-rajapinta		41
5.2.1	Django	44	
5.2.2	FastAPI	44	
5.2.3	Express	45	
5.2.4	Rails	45	
6	Johtopäätökset ja jatkokehitys.....		46
6.1	Johtopäätökset		46
6.2	Puutteet ja kehitysideat		48
7	Yhteenveto.....		50
8	Viiteluettelo		51

KUVAT

Kuva 1. Käännetyn koodin reitti suoritukseen.	4
Kuva 2. Tulkattavan koodin reitti suoritukseen.....	4
Kuva 3. HTTP-protokollan toimintamalli.	5
Kuva 4. WebSocket-protokollan toimintamalli.....	6
Kuva 5. HTTP- ja WebSocket-palvelujen nopeusero teoreettisessa tekstiviestipalvelussa [Lasocha ja Badurowicz, 2021].	7
Kuva 6. V8-moottorin toimintamalli [McIlroy, 2016].	8
Kuva 7. Express-ohjelmistokehyksen latausmäärät [Potter, 2023].	10
Kuva 8. Django-pakettien latausmäärät päivittäin [PyPI Stats, 2023c].	12
Kuva 9. Flask-paketin päivittäiset latausmäärät [PyPI Stats, 2023a].	13
Kuva 10. FastAPI-paketin latausmäärät päivittäin [PyPI Stats, 2023b].	14
Kuva 11. Ruby-kielen JIT-kääntö [Ruby, 2023c].	15
Kuva 12. Laravelin latausmäärät kuukausittain [Packagist, 2023].	18
Kuva 13. Yleisimmin käytetyt verkkorajapintakehykset ja teknologiat Stack Overflowin suorittaman kyselytutkimuksen perusteella [Stack Overflow, 2022].	19
Kuva 14. Testitietokannan pöytien rakenne.	27
Kuva 15. Testiarkkitehtuuri.	30
Kuva 16. Eri ohjelmointikehysten käsittelyaika suhteessa yhtäaikaisiin pyyntöihin.	32
Kuva 17. Eri ohjelmointikehysten virhemäärä suhteessa pyyntöihin.	33
Kuva 18. Eri kehysten vasteaika suhteessa limittäisiin pyyntöihin.	33
Kuva 19. Django-kehysten pyyntöjen käsittelyaika ja virheiden määrä limittäisten pyyntöjen määrää kohti.	34
Kuva 20. Django-kehysten vasteaika suhteessa limittäisiin pyyntöihin.	34
Kuva 21. Flask-kehysten pyyntöjen käsittelyaika ja virheiden määrä limittäisten pyyntöjen määrää kohti.	35
Kuva 22. Flask-kehysten vasteaika suhteessa limittäisiin pyyntöihin.	35
Kuva 23. FastAPI-kehysten pyyntöjen käsittelyaika ja virheiden määrä limittäisten pyyntöjen määrää kohti.	36
Kuva 24. FastAPI-kehysten vasteaika suhteessa limittäisiin pyyntöihin.	36
Kuva 25. Express-ohjelmistokehyksen pyyntöjen käsittelyaika ja virheiden määrä limittäisten pyyntöjen määrää kohti.	37
Kuva 26. Express-ohjelmistokehyksen vasteaika suhteessa limittäisiin pyyntöihin.	38
Kuva 27. Ruby on Rails -kehysten pyyntöjen käsittelyaika ja virheiden määrä limittäisten pyyntöjen määrää kohti.	39
Kuva 28. Ruby on Rails -kehysten vasteaika suhteessa limittäisiin pyyntöihin.	39

Kuva 29. Laravel-kehiksen pyyntöjen käsittelyaika ja virheiden määrä limittäisten pyyntöjen määrää kohti.	40
Kuva 30. Laravel-kehiksen vasteaika suhteessa limittäisiin pyyntöihin.	40
Kuva 31. Kehysten WebSocket-pyyntöjen käsittelyaika suhteessa limittäisten yhteyksien määrään.	42
Kuva 32. Kehysten virhemäärä suhteessa limittäisiin WebSocket-yhteyksiin.	43
Kuva 33. Django-kehiksen keskimääräinen käsittelyaika suhteessa virheiden määrään.	44
Kuva 34. FastAPI-kehiksen keskimääräinen käsittelyaika suhteessa virheiden määrään.	44
Kuva 35. Express-ohjelmistokehiksen keskimääräinen käsittelyaika suhteessa virheiden määrään.	45
Kuva 36. Rails-kehiksen keskimääräinen käsittelyaika suhteessa virheiden määrään. .	45

TAULUKOT

Taulukko 1. Verkko-ohjelmistokehysten painotettu tehokkuus [TechEmpower, 2022a].	20
Taulukko 2. Erään avoimen lähdekoodin testaustuloksia [Web Frameworks Benchmark, 2023].	21
Taulukko 3. Tulkkien versiot.	25
Taulukko 4. Ohjelmistokehysten käytetyt käynnistyskomennot.	26
Taulukko 5. Eri kehysten tasapainoitettut keskimääräiset ajat välillä 2-512. Pienempi parempi.	32
Taulukko 6. Verkko-ohjelmistokehysten tasapainotettut tulokset.	41
Taulukko 7. Keskimääräinen käsittelyaika 1-512 limittäiselle WebSocket-yhteydelle. Pienempi on parempi.	42

KOODIESIMERKIT

Koodiesimerkki 1. ”Hello World” -ohjelma JavaScript-ohjelmointikielellä ja NodeJS:llä.	9
Koodiesimerkki 2. Express-ohjelmistokehiksen "Hello World" -ohjelma.	10
Koodiesimerkki 3. "Hello World" -ohjelma Python-ohjelmointikielellä.	11
Koodiesimerkki 4. "Hello World" -sovellus ja sen esimäärittelemä koodi Django-ohjelmistokehiksellä.	12
Koodiesimerkki 5. "Hello World" -sovellus Flask-ohjelmistokehiksellä.	13
Koodiesimerkki 6. "Hello World" -sovellus FastAPI-ohjelmistokehiksellä.	14

Koodiesimerkki 7. "Hello World" -ohjelma Ruby-ohjelmointikiellä.	15
Koodiesimerkki 8. "Hello World" -ohjelma Ruby on Rails -ohjelmistokehyksellä.....	16
Koodiesimerkki 9. "Hello World" -ohjelma PHP-ohjelmointikielillä.....	17
Koodiesimerkki 10. "Hello World" -ohjelma Laravel-ohjelmistokehyksellä.	17
Koodiesimerkki 11. Esimerkki asiakkaan vastaanottamasta datasta.	27

TERMIT

IoT: *Internet of Things*. Esineiden internet. Käsite, joka viittaa arkipäivän esineisiin ja laitteistoihin, jotka ovat yhteydessä verkkoon.

HTTP: *Hypertext Transfer Protocol*. Protokolla, jolla modernit verkkoselaimet siirtävät dataa ja lataavat verkkosivuja.

REST: *Representational State Transfer*. HTTP-protokollan päälle rakennettu protokolla, joka yhtenäistää verkkosivujen kommunikointitapaa.

RESTful: Sovellus, joka toteuttaa REST-tyylisen rajapinnan.

JIT: *Just in Time*. Tämän tutkielman kontekstissa tapa, jolla ohjelma käännetään konekielelle ajon aikana.

WebSocket: Protokolla, joka mahdollistaa reaaliaikaiset päivitykset verkkosivuille.

Vasteaika: Aika, joka kestää pyynnön vastauksen saamiseen.

Käsittelyaika: Tässä tutkielmassa käytetty termi, jolla tarkoitetaan keskimääräistä kaikkien pyyntöjen käsittelyyn kuluvaan aikaan.

Dynaaminen tyyppittely (engl. *Dynamic typing*): Ohjelman tyyppittelymalli, jossa koodin kääntäjälle ei välttämättä täsmällisesti kerrota muuttujien tyyppiä.

Ohjelmistokehys (engl. *programming framework* tai *framework*): Ohjelmistojen tuotantoa varten valmiiksi luotu mallipohja, jota laajentamalla voidaan luoda ohjelmistoja, rajoittaen joitain valintoja. Eri asia kuin *kirjasto*, joka ei puutu ohjelman ajamiseen ja arkkitehtuurimalliin.

Mielipiteellinen (engl. *opinionated*) ohjelmistokehys: Ohjelmistokehys, joka asettaa tiukat rajoitteet luotavan ohjelmiston rakenteelle ja arkkitehtuurille. Tämän vastakohtana on **Mielipiteetön** (engl. *unopinionated*) ohjelmistokehys, joka ei rajoita luotavan ohjelmiston arkkitehtuuria.

MVC (*model view controller*): Ohjelmiston arkkitehtuurimalli, jossa data (malli), logiikka (ohjain), sekä näkymä on erotettu toisistaan. Ohjaimen voidaan nähdä toimivan siltana mallin ja näkymän välillä.

MVT (*model view template*): Ohjelmiston arkkitehtuurimalli, jossa data, logiikka, sekä näkymä on erotettu toisistaan. Toisin kuin MVC-mallissa, MVT-mallissa malli (engl. *template*) voi sisältää logiikkaa.

1 Johdanto

Kuten Adeleye ja muut [2023] toteavat, on verkkopalvelujen käyttömäärä edellisten vuosien aikana noussut runsaasti, sekä uusia verkkopalveluita on kehitetty nostaen tarvetta tehokkaille verkkopalvelukehyksille, jotka tarjoaisivat ketterän kehityspohjan uusia sovelluksia varten. Näiden verkkokehysten suuren käyttömäärän vuoksi olisi hyvä tarkastella niiden tehokkuutta, sillä pienilläkin eroilla voi olla suuria vaikutuksia vaadittaviin palvelinresursseihin ja käyttökokemukseen.

1.1 Tausta

Monessa uudessa sovelluksessa ei käytetä vain yhtä kommunikointiprotokollaa, kuten HTTP, vaan yhdessä sovelluksessa voidaan käyttää monesti useampaa eri teknologiaa palvelimen ja asiakkaan väliseen keskusteluun. Esimerkkinä näitä protokollia käyttävistä teknologioista voidaankin mainita esimerkiksi uutena alueena kehittyvät tekoälyteknologiat, jotka vaativat paljon kommunikointia ihmisen toimesta verkkosovelluksissa, sekä kuten Soewito ja muut [2019] mainitsevatkin, IoT-teknologioiden keskustelu koneiden kesken. Klassisemmatkin sovellukset voivat käyttää näitä protokollia, kuten lukuiset sosiaalisen median applikaatiot, tai monet reaaliaikaiset kartta- ja seurantasovellukset.

Moni verkkokehittäjä tunnistaakin sanat REST ja WebSocket kuullessaan ne. Ensimmäinen tarjoaa yhtenäisen protokollan HTTP:lle, sekä jälkimmäisen tarjoaa tavan päivittää verkkosovelluksia tiheään tahtiin katkaisematta kommunikointia palvelimelle. Kumpaa-kin teknologiaa käytettäessä voidaankin saada aikaan hyvin eläviä reaaliaikaisia verkkosovelluksia, jotka tarjoavat niiden käyttäjille uusia mahdollisuuksia monella eri teollisuuden ja arkipäivän saralla.

Suuren levinneisyyden vuoksi onkin tärkeä tarkastella joidenkin yleisesti käytettyjen ohjelmistokehysten tehokkuutta; osittain sen vuoksi, että käyttäjät ovat tottuneet verkkopalvelinten ripeään vastaamiseen [Nielsen, 2010], sekä toisaalta myös sen takia, että tehottomat kehykset vievät enemmän resursseja, joka erityisesti ilmastokriisin ja resurssipulan vuoksi voi olla entistä tärkeämpää ottaa huomioon. Vaikka olisikin mahdollista vertailla tuloksia hyödyntämällä pelkästään tarjottuja ohjelmistokieliä, moni kehittäjä käyttää kuitenkin jotain ohjelmistokehystä tähän tarkoitukseen vähentääkseen tarvittavan ohjelmoinnin määrää, jonka voi nähdä erityisen hyvin esimerkiksi näiden kehysten latausmääristä. Tämän vuoksi olisikin hyvä tarkastella joitain yleisimpiä ohjelmistokehyksiä niiden tehokkuuden saralla; monessa tapauksessa ohjelman tehokkuutta ei määritä pelkästään sen kieli, vaan myös tapa, jolla se on toteutettu.

Tutkielmassa termillä *vasteaika* viitataan aikaan, joka asiakaskoneen näkökulmasta oli aika, joka kesti yhden viestin vastaukseen. Vastaavasti termillä *käsittelyaika* viitataan aikaan, joka kesti asiakkaalla ja palvelimella käsitellä kaikki pyynnöt. Olennaista tuossa on siis palvelimen mahdollinen pyyntöjen limittäiskäsittely, jonka avulla palvelin voi käsitellä kaikki pyynnöt nopeampaa, kuin vasteaika saattaisi antaa ymmärtää.

1.2 Ongelman asettelu ja rajaukset

Opinnäytetyön aiheena on selvittää vastaukset seuraaviin kysymyksiin:

1. Mikä suosituimmista REST- ja WebSocket-verkkorajapintoja tarjoavista ohjelmistokehityksistä olisi tehokkain?
2. Kuinka paljon rasitetta eri kehykset kestävät?

Koska edellä mainitut kysymykset sinänsä vaatisivat suuren määrän kerättävää dataa ja luotavaa koodia, rajoitetaan sitä sillä tavalla, että otetaan huomioon vain joitain suosituimpia ympäristöjä ja ohjelmistokehityksiä. Nämä ohjelmistokehykset on valittu valtaosin Stack Overflown [2022] aktiivisimmista tai pidetyimmistä kehyksistä. Tällä tavalla rajaamalla saadaan mukaan seuraavat ohjelmistokehykset: *Express*, *Django*, *Flask*, *FastAPI*, *Ruby on Rails*, sekä *Laravel*.

On huomioonotettavaa, että koska kaikki edellä mainitut kehykset toimivat jonkin tulkin päällä, on kaikille olemassa vaihtoehtoisia tulkkeja, joista jotkin voivat olla tehokkaampia ja paremmin palvelinympäristöihin sopivia kuin toiset. Tässä tutkimuksessa tarkastellaan kuitenkin vain edellä mainittujen kehysten kielten virallisia tulkkeja, valtaosin niiden yleisyyden vuoksi. Vastaavasti ohjelmistoihin ei toteuteta testien puolesta rinnakkaisprosessointia, vaan kaikki rinnakkaisprosessointi mitä käytetään, tulee vain ohjelmistopakettin, mahdollisen tulkin, tai suositellun ajoympäristön kautta.

Luvussa 2 selvitetään mitä tutkittavat kehykset ovat, sekä kerrotaan niiden yleisestä toiminta-arkkitehtuurista ja tavasta, jolla ne tulkkaavat annettuja ohjelmistoja. Taustaselvitysten jälkeen luvussa 3 kerrotaan edellisistä tutkimuksista, joista voidaan saada varteenotettavia huomioita tämän tutkielman toteutukseen. Neljännessä luvussa selvitetään, miten tutkimus on toteutettu, sekä minkälaisessa ympäristössä tutkimus on suoritettu. Luvussa 5 tarkastellaan rajapinnoista kerättyjä tuloksia. Seuraavaksi luvussa 6 kerrotaan johtopäätökset, sekä tämän tutkielman mahdolliset puutteet ja kehitysehdotukset. Lopuksi luvussa 7 kerrotaan koko tutkielman yhteenveto.

2 Web-rajapintakehysten teknologiat

Tutkielman testattavia kehyksiä ja teknologioita käydään läpi seuraavissa kohdissa, sekä niistä annetaan niiden kontekstiin sopiva ”Hello World” -esimerkki. Kielten tapauksessa tämä esimerkki on minimaalinen tapa saada tulostettua ”Hello World”, sekä kehyksissä tämä esimerkki toimii mallina sille, kuinka saadaan ”Hello World” tulostettua luodun rajapinnan kautta.

Jotta tutkimus voidaan määritellä rajatusti, tulee myös määritellä joidenkin termien, kuten tehokkuuden konteksti. Kuten Shaw [2000] mainitseekin julkaisussaan, tehokkuudella ei ole vain yhtä määritelmää; joillekin tehokkuus voi merkitä palvelinten sähkönkulutusta, toiselle laitteiston hyödynnystä, ja jollekin muulle hävikkidatan määrää. Kuten Shawin [2000] paperissa, tässäkin tutkimuksessa mitataan osittain käyttäjän näkökulmasta tapahtuvaa palvelimen vastausten nopeutta. Tämän lisäksi otetaan huomioon kuitenkin myös nopeus, jolla palvelin pystyy käsittelemään limittäisiä pyyntöjä, sekä missä vaiheessa testikokoonpanolla palvelin ei enää pysty käsittelemään enempää limittäisiä pyyntöjä ilman virheitä. Edellä mainitulla tavalla voidaan näin ottaa huomioon myös se, että vaikka yhden asiakkaan aika olisikin hiukan hitaampi, niin palvelin saattaa mahdollisesti palvella tehokkaammin useampaa asiakasta samanaikaisesti. Tämä on mainittu myös tavallaan Shawin [2000] julkaisussa, että vaikka suurempi määrä asiakkaita saisi palvelimen näkökulmasta saman määrän vastauksia lyhyessä ajassa, voi yksittäisen asiakkaan vasteaika olla liian hidas sulavan käyttökokemuksen takaamiseksi.

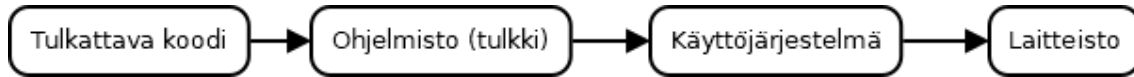
2.1 Tulkatut kielet

Tulkatut kielet ovat kieliä, jotka eivät toimi suoraan käyttöjärjestelmän tarjoamilla komennoina, vaan tulkin kautta, joka välittää kirjoitetun ohjelmiston komennot käyttöjärjestelmälle. Tämä eroakin kuvassa 1 näkyvästä konekielisen koodin reitistä laitteistolle, jota voidaan verrata kuvaan 2, jossa näkyy tulkatun kielen reitti laitteistolle. Koska tulkkaus vaatii jonkin verran tehoa, vaikuttaa se oleellisesti kirjoitetun ohjelmiston tehokkuuteen, jonka vuoksi se nähdään yleisesti konekielisiä ohjelmia tehottomampana.

Tulkkauksen lisäksi on myös olemassa JIT-malli (*just-in-time*), joka pohjautuu tulkkaamiseen. JIT-mallilla toimiva koodi käännetään ajonaikaisesti tavu- tai laitekoodiksi, jota tulkki tai laite pystyy ajamaan, käytännössä näin ollen käännetyn ja tulkatun kielen väli-muoto. Tällä tavalla saadaan vähennettyä tulkille tai laitteistolle käännöksestä koituvaa rasitetta, nopeuttaen ohjelmiston ajoa. [McIlroy, 2016]



Kuva 1. Käännetyin koodin reitti suoritukseen.



Kuva 2. Tulkattavan koodin reitti suoritukseen.

Yleisesti voidaan nähdä monien modernien verkkosovellusten pohjautuvan *MVC*-malliin (model-view-controller), jossa malli (engl. *model*) on sovelluksen data, jota voidaan muokata ohjaimella (engl. *controller*), sekä näyttää käyttäjälle näkymällä (engl. *view*). Käytännössä ohjaimen voidaan nähdä olevan mallin ja näkymän välissä, jakaen sovelluksen kolmeen karkeaan osa-alueeseen.

2.2 Verkkorajapinnat ja REST yleisesti

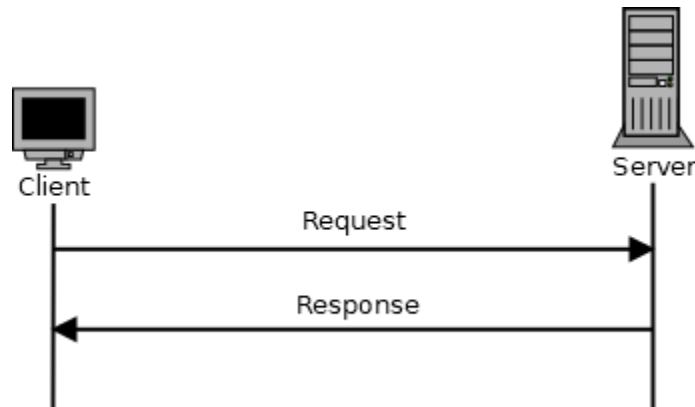
REST (REpresentational State Transfer) on RESTful-sovellusten HTTP-protokollaa hyödyntävä keskustelumalli, jossa Fieldingin [2000] mukaan palvelimen puolella ei nimensä mukaisesti säilytetä käyttäjän applikaation tilaa, kuten esimerkiksi sisäänkirjautumistilaa. REST on Roy Fieldingin [2000] esittelemä vaihtoehtoinen verkkorajapintamalli aiempien teknologioiden tilalle, jotka saattoivat olla palvelimelle raskaampia juuri käyttäjän tilan säilyttämiskaavan vuoksi. Sovelluksia, jotka toteuttavat REST-mallista rajapintaa kutsutaankin RESTful-sovelluksiksi.

RESTful-sovelluksissa palvelimen käyttäjältä vastaanottamat pyynnöt (engl. *request*) käsitellään palvelimen ymmärtämällä tavalla, josta se käsittelyn jälkeen lähettää yksittäisen vastauksen (engl. *response*) käyttäjälle. Koska jokaiseen pakettiin sisältyy kaikki informaatio mitä palvelin tarvitsee ymmärtääkseen ja vastataksaan pyyntöön, ei palvelinpuolella tarvitse säilyttää applikaatioiden tilaa. [Fielding, 2000]

Koska mainittu käyttäjä-palvelin-malli (engl. *client-server*) käsittelee dataa edellä mainitulla tavalla, voidaan myös olettaa, että moni palvelimen vastaus ei muutu sitä uudelleen pyydettyä. Tämän vuoksi RESTful-sovellukset voivat sisällyttää vastaukseensa tietoa siitä voidaanko vastauksista pitää välimuistissa, sekä kuinka pitkään. Tällä tavalla saadaan rajoitettua verkkoliikenteessä kulkevan datan määrää, sekä sitä hyödynnetäänkin nykyisellään lukuisissa verkkosovelluksissa.

Lopulta REST-arkkitehtuuri käytännössä vain asettaa rajoitteita HTTP-protokollalle, jossa kuvaillaan, kuinka pyyntöjen tietoa käsitellään, sekä missä sitä hallinnoidaan. Tästä syystä kuten HTTP-protokollassa, myöskään REST:ssä ei ylläpidetä jatkuvaa yhteyttä

palvelimeen. Näin toimiessaan jokainen pyyntö palvelimelle vaatii uuden yhteyden, sekä vastauksen. Koska RESTful-palvelut toimivat HTTP-protokollan päällä, voidaan niiden toimintamallin kuvaukseen käyttää HTTP:n toimintamallia, joka on esitelty kuvassa 3. Tästä voidaan olennaisesti huomata kuinka edellä mainitun tavalla palvelin vastaa asiakkaan yksittäiseen pyyntöön yksittäisellä vastauksella, ilman muuta kommunikointia.



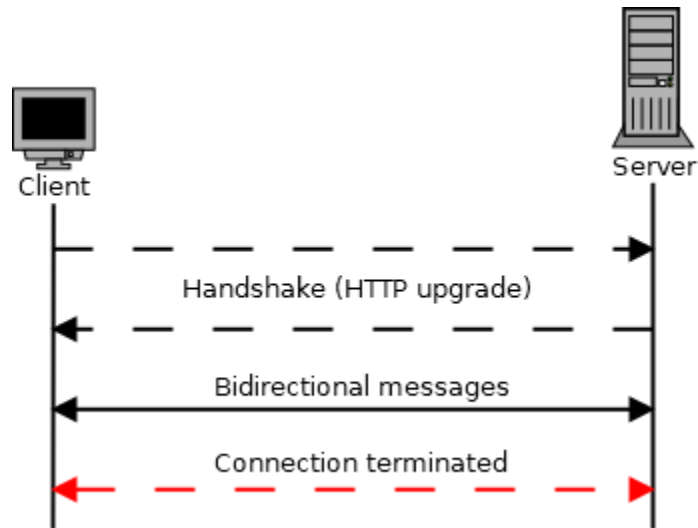
Kuva 3. HTTP-protokollan toimintamalli.

2.3 WebSocket yleisesti

Monissa verkkosovelluksissa hyödynnetään laajasti WebSocket-teknologiaa [Soewito *et al.*, 2019], jossa asiakkaan voidaan sanoa olevan jatkuvassa yhteydessä palvelimeen. Tällaisia sovelluksia ovat esimerkiksi erilaiset viestintäalustat, sekä reaaliaikaiset karttasovellukset. WebSocket-teknologiaa hyödynnetäänkin runsaasti tiheästi päivittyvissä sovelluksissa sen tarjoaman HTTP-protokollaa kevyemmän rasiuksen vuoksi [Lasocha ja Badurowicz, 2021].

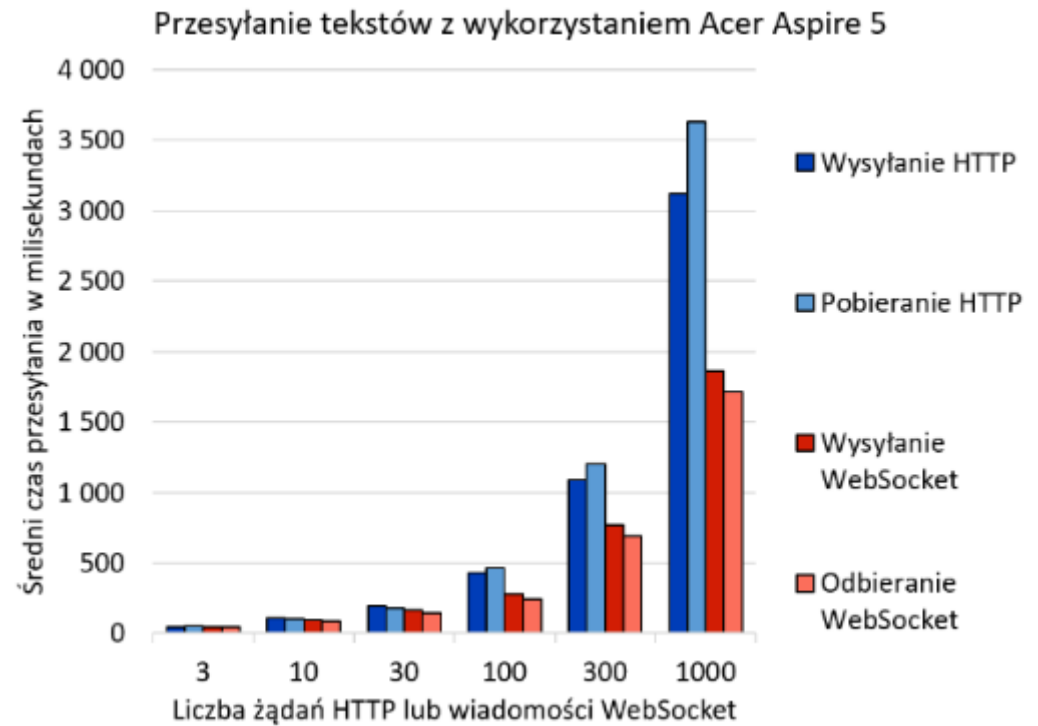
WebSocket-yhteys luodaan asiakkaan ja palvelimen välille aluksi HTTP:llä, joka päivitetään (engl. *HTTP-upgrade*) WebSocket-yhteydeksi [Fette ja Melnikov, 2011; Soewito *et al.* 2019]. Tätä proseduuria kutsutaan kättelyksi (engl. *handshake*). WebSocket on aplikaatiotason verkkoarkkitehtuuri kuin myös HTTP, mutta siitä poiketen WebSocket-yhteyksissä hyödynnetään *kaksisuuntaista* (engl. *bi-directional, full-duplex*) keskusteluyhteyttä kättelyn jälkeen, jota ei sammuteta kuin asiakkaan tai palvelimen toimesta [Fette ja Melnikov, 2011]. WebSocket-protokollan toimintamallia on kuvattu myös kuvassa 4, jossa yhteyden sammuttaminen on kuvattu tekstillä *connection terminated*. Kuten Fette ja Melnikovin [2011] kirjoittamassa WebSocketin dokumentaatiossa kerrotaan, WebSocket-viestit lähetetään TCP-yhteydellä laitteiden välillä. Edellä mainituilla ominaisuuksilla WebSocket-yhteydellä voidaan keskustella laitteiden välillä tavalla, joka ei väärinkäytä muita protokollia keskustelun keventämiseen. Ennen WebSocketsia käytettiin

muun muassa long polling -taktiikkaa, jossa yhteyden sulku-aikaa siirrettiin myöhemmäksi, näin sallien suuremman datamäärän lähetyksen, sekä Comet-taktiikkaa, joka saatoi tietyissä toteutustavoissa pohjautua long polling -taktiikkaan [Carbou, 2011].



Kuva 4. WebSocket-protokollan toimintamalli.

WebSocketin keveys tulee osittain siitä, että yhteyttä palvelimelle ei tarvitse luoda uudelleen, näin säästyen uuden salauksen luonnilta ja muusta yhteyden luontiin kuuluvasta tietojen käsittelystä, kuin myös siitä, että normaaleihin HTTP-pyyntöihin verrattuna WebSocket ei vaadi *otsikoiden* (engl. *header*) sisällyttämistä pyyntöihin [Fette ja Melnikov, 2011]. Toisin kuin HTTP:ta käyttävissä RESTful-palveluissa, WebSocket-protokollassa jokaiseen pyyntöön ei välttämättä tarvitse myöskään sisällyttää kaikkea pyyntöön liittyvää dataa, näin säästäen verkkokaistaa ja pyynnön lukemiseen käytettävää aikaa [Lasocha ja Badurowicz, 2021]. Lasochan ja Badurowiczin tutkimuksesta lainatusta kuvasta 5 voidaan vertailla näiden kahden eri teknologian nopeuseroja tekstiviestikäytössä. Kuvassa sinisillä palkeilla on kuvattu HTTP-pyyntöjen viivettä, sekä punaisella WebSocket-pyyntöjen viivettä. Tummemmat palkit kuvaavat lähetysaikaa, kun taas vaaleammat vastaanottoaikaa, sekä palkit on ryhmitelty pyyntöjen määrän perusteella.

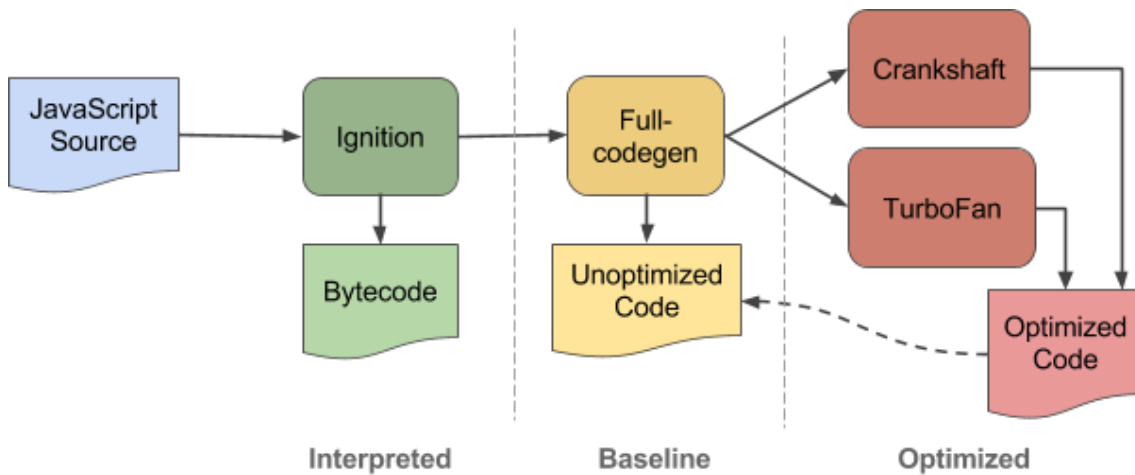


Kuva 5. HTTP- ja WebSocket-palvelujen nopeusero teoreettisessa tekstiviestipalvelussa [Lasocha ja Badurowicz, 2021].

2.4 JavaScript ohjelmointikielenä

JavaScript on nykyisellään ECMAScript-standardiin pohjautuva dynaamisesti ja heikosti tyypitetty oliopohjainen ohjelmointikieli, jota käytetään valtaosin verkkosovelluksissa [Mozilla Corporation, 2023]. Vuonna 1995 julkaistu JavaScript [Netscape, 1995] onkin päätoimisesti toiminut verkkosivujen toimintojen mahdollistajana, erityisesti moderneissa yksisivuisissa sovelluksissa (engl. *SPA, single-page-application*). JavaScriptin kehitti Brendan Eich Sun Microsystemsin Netscapelle ja sen kehitys siirrettiin myöhemmin ECMAScript standards committeeelle [Wirfs-Brock ja Eich, 2020]. JavaScriptin nimi itsessään on rekisteröity [Wirfs-Brock ja Eich, 2020], jonka vuoksi moni ECMAScript-standardin toteuttava tai siihen pohjautuva kieli ei itsessään käytä nimeä JavaScript, näistä esimerkkinä TypeScript, joka lisää ohjelmointikielen vahvan tyypityksen.

Tulkkauksesta johtuvan hitauden vuoksi monissa JavaScriptiä ajavissa ohjelmistoissa koodi toimii nykyisellään JIT-mallilla, jossa tulkattavasta kielestä luodaan joko tavukoodia tai laitteistokoodia, joita voidaan ajaa laitteistolla tehokkaammin. Esimerkkinä tästä on JavaScript-moottorin V8 Ignition-tulkki, jonka toimintamalli näkyy kuvassa 6.



Kuva 6. V8-moottorin toimintamalli [Mcllroy, 2016].

Koska JavaScript perustuu ECMAScript-standardiin, jolla ei ole virallista tulkkiä, ei JavaScriptillä itselläänkään ole varsinaisesti omaa virallista tulkkiä. Yleisimmäksi nähtävää Alphabetin JavaScript-moottoria V8 käytetään esimerkiksi Alphabetin tuotteissa, kuten Chromium, jonka päälle kirjoitushetkellä valtaosa nykyisistä verkkoselaimista on kehitetty [StatCounter, 2023]. NodeJS toimii oletuksena V8-moottorilla, joka kirjoitushetkellä asennettaessa asentuu aina NodeJS-ympäristön mukana [OpenJS Foundation, 2023a].

V8-moottori toimii kuvan 6 osoittamalla osittaisella JIT-mallilla, jossa JavaScript lähdekoodi käännetään Ignitionilla tavukoodiksi, sekä TurboFan voi kääntää koodia laitteistokoodiksi, sekä tarpeen mukaan optimoida lisää [Mcllroy, 2016]. Optimoitua koodia voidaan tämän jälkeen ajaa ohjelmistoa ajavalla laitteistolla. Valittu moottori ei itsessään kuitenkaan käännä kaikkea koodia laitteistokoodiksi, vaan valitsee tähän kääntämiseen ainoastaan eniten käytetyt koodin osat [Mcllroy, 2016]. Nykyisellään kuvassa näkyvää Crankshaftia ei käytetä osana V8-moottoria, koska kehitystiimi ei uskonut sen kehityksen pysyvän JavaScriptin kehityksen perässä [V8, 2017], näin korvaten sen TurboFanilla.

2.5 NodeJS ja Express-ohjelmistokehys

NodeJS on palvelinpuolen JavaScript-ympäristö, jolla ajetaan JavaScriptiä asynkronisesti. NodeJS:n verkkosivujen mukaan se on kehitetty toimimaan pääasiallisesti palvelintarkoituksessa. Tätä varten NodeJS:ssä ei valtaosin käytetä I/O-operaatioita paljoa, koska ne voivat estää muita säikeitä etenemästä. NodeJS itsessään toimii tapahtumasilmukoiden avulla, jossa jokainen uusi pyyntö laitetaan silmukan käsiteltäväksi asynkronisena JavaScript-funktiona. Vaikka pääosin tämän perusteella säikeitä ei NodeJS:ssä käytetä estäen useamman laitteiston ytimen hyödyntämisen, voidaan uusia prosesseja luoda kirjastoon luoduilla metodeilla. Käytettäessä NodeJS-palvelinta itse prosessi nukkuu, kunnes

se saa pyynnön käsitellä dataa, jonka vuoksi sen voidaan katsoa olevan tehokas silloin, kun prosessoitavaa dataa ei ole saatavilla. [OpenJS Foundation, 2023b]

Pohjimmiltaan NodeJS on käyttänyt Alfabetin V8-moottoria koodin ajamiseen, mutta myöhemmin siitä on julkaistu muitakin versioita (esim. SpiderNode ja ChakraCore), jotka ovat kirjoitushetkellä hylättynä. Jotta edellä mainitut vaihtoehtoiset JavaScript-moottorit olisivat olleet helpompia kehittää, on NodeJS julkaissut rajapinnan Node-API, jonka avulla C/C++-koodilla voidaan käsitellä JavaScript-olioita riippumatta siitä, mikä JavaScript-moottori toimii taustalla [OpenJS Foundation, 2023a]. Koodiesimerkissä 1 näytetään yksinkertainen koodiesimerkki NodeJS- ja JavaScript-ympäristössä.

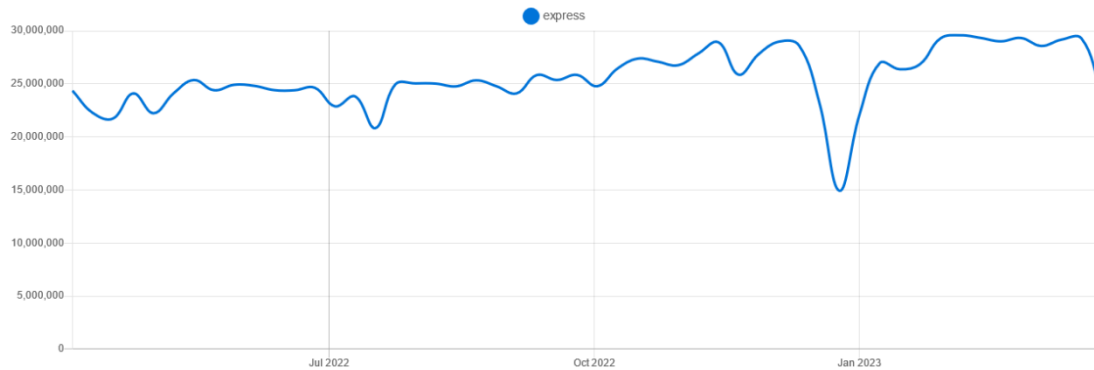
```
console.log("Hello World");
```

Koodiesimerkki 1. ”Hello World” -ohjelma JavaScript-ohjelmointikielellä ja NodeJS:llä.

Express on NodeJS-ympäristölle kehitetty avoimen lähdekoodin ohjelmistokehys, jota käytetään yleisesti RESTful-verkkorajapintojen toteutukseen. Express-kehys oli Holowaychukin vuonna 2010 julkaisema ohjelmistokehys [Holowaychuk, 2010], joka on nykyisellään NodeJS:n hallitsemana [Krill, 2016]. Tämä kehys on ottanut kehittäjän alkuperäisen julkaisun [Holowaychuk, 2010] mukaan mallia Sinatra:sta, joka on Rubylla oleva verkko-ohjelmistokehys.

Express-ohjelmistokehys oli vuoden suosituin verkkorajapintaohjelmistokehys Stack Overflown [2022] käyttäjäkyselyn perusteella, sekä se itsessään on *mielipiteetön* (engl. *unopinionated*) rajapinta, näin mahdollistaen vapaasti valittavan arkkitehtuurin ja tyyli-rakenteen. Kuvasta 7 voidaan myös nähdä, että mainitun kehysten latausmäärät ovat suuria.

Kuten Expressin julkaisussa mainitaankin [Holowaychuck, 2010], ohjelmoija on kohtuullisen vapaa päättämään miten käyttää Express-kehystä. Yleisesti sitä voidaan käyttää esimerkiksi luomaan verkkosivuja PHP:n tyyliä muistuttavasti, kuin myös puhtaana verkkorajapintana. Koodiesimerkissä 2 näytetään, kuinka Express-kehyksellä voidaan luoda yksinkertainen ohjelma.



Kuva 7. Express-ohjelmistokehyksen latausmäärät [Potter, 2023].

```
const express = require('express');
const app = express();
app.get('/', (req, res) => {
    res.send("Hello World");
});
app.listen(3000);
```

Koodiesimerkki 2. Express-ohjelmistokehyksen "Hello World" -ohjelma.

2.6 Python ohjelmointikielenä

Pythonin dokumentaation mukaan se on Python Software Foundationin hallitsema yksityylinen tulkattu oliopohjainen ohjelmointikieli, jolla on mahdollista toteuttaa muun muassa proseduraalista ja funktionaalista ohjelmointia. Kuten NodeJS, myös Python-kieltä on mahdollista jatkaa C/C++-ohjelmointikielillä, mikäli sille on tarvetta. Python itsessään on määritelty vahvasti tyytellyksi dynaamiseksi ohjelmointikieleksi, jota tulkataan monesti CPython-tulkilla, vaikkakin vaihtoehtoisia tulkkeja on olemassa. [Python Software Foundation, 2023]

Tutkielmassa käytetty virallinen CPython-tulkki itsessään ei sisällä JIT-kääntäjää, mutta joillakin epävirallisilla tulkeilla, kuten PyPy, voidaan ohjelmistokoodi kääntää JIT-tyylisesti, sallien mahdollisia parannuksia tehokkuuteen. Koodiesimerkissä 3 on näytetty, kuinka näillä tulkeilla voidaan tulostaa "Hello World".

Pythonin kehittäjä Guido van Rossum aloitti kehittämään Python-kieltä vuoden 1989 lopulla, sekä julkaisi sen vuoden 1991 alkupuolella noin vuoden kehityksen jälkeen. Kieli kehitettiin korvaamaan ABC-ohjelmointikieltä Amoeba-käyttöjärjestelmässä. Nykyisellään Python Software Foundation (PSF) omistaa oikeudet Python-ohjelmointikieleen versiosta 2.1 eteenpäin. Kirjoitushetkellä Pythonista on olemassa muitakin tulkkeja PSF:n

oman CPython-tulkin lisäksi, kuten esimerkiksi PyPy, Jython, sekä IronPython. [Python Software Foundation, 2023]

```
print('Hello World')
```

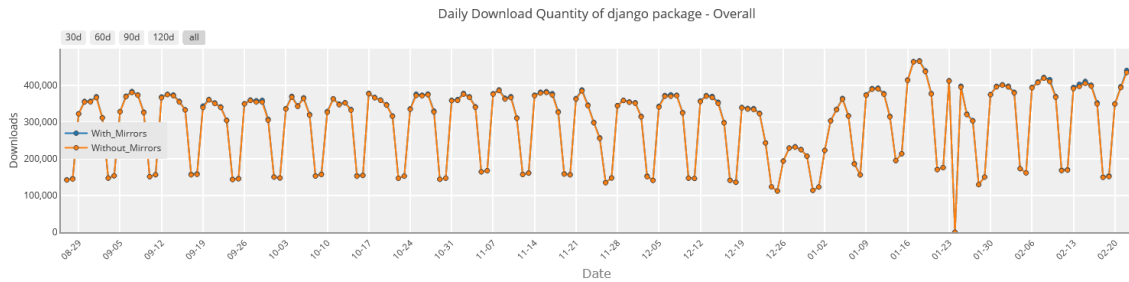
Koodiesimerkki 3. "Hello World" -ohjelma Python-ohjelmointikielellä.

2.6.1 Python Django-ohjelmistokehyksellä

Django on Python-kielelle rakennettu mielipiteetön verkko-ohjelmistokehys, josta tehtiin avoin vuonna 2005 [Django, 2022a]. Alun perin World Onlinen kehittämä ohjelmistokehys käyttää omien sanojensa mukaan ”shared-nothing” arkkitehtuuria, johon voi heidän sanojensa mukaan lisätä palvelinarkkitehtuuria millä tahansa palvelun tasolla [Django, 2022a]. Django Software Foundation [2022a] mainitsee Djangon erotteluvan toisistaan eri palvelujen tasot, kuten esimerkiksi tietokantoihin liittyvät koodit ja itse palveluun liittyvät koodit. Djangon latausmäärät on esitelty kuvassa 8, sekä ”Hello World” -malli esimäärittelyineen koodeineen koodiesimerkissä 4.

Koska Django on PHP-kielen palvelumallin tyylinen, käytetään tässä opinnäytteessä lisäksi kirjastoa nimeltä ”Django REST framework”, jolla helpotetaan RESTful-toimintojen toteutusta. Django mainitsee toimivansa *MVT*-mallilla (model-view-template), jolla voi olla yhteys tähän PHP:n mallilla toimimiseen kehittäjien näkökulmasta. *MVT*-mallin voidaan katsoa olevan *MVC*-mallin alahaara, jossa *template* vastaa kehyksen näkökulmasta itse pyyntöjen ja vastausten käsittelytasoa, sekä *view*, eli näkymä on käyttäjän vastaanottama data. [Django, 2022a]

Django toteuttaa myös Python-standardiin kuuluvan *WSGI*-verkkorajapintamallin (Web Server Gateway Interface) [Django, 2022b]. Tiivistettynä tällä määritellään vain säännöt sille, kuinka verkkorajapintaan tulisi päästä käsiksi ohjelmistotasolla, jotta pohjalla toimivan verkkorajapinnan vaihtaminen toiseen olisi helpompaa [Eby, 2022]. Nykyisellään Django toteuttaa myös *ASGI*-verkkorajapintamallin, mahdollistaen asynkronisen pyyntöjen käsittelyn [Django, 2022c], joka voi joissain tapauksissa helpottaa WebSocket-rajapintojen luontia. On huomattavaa, että Djangoa ajettaessa *WSGI*-muodossa se jää odottamaan vastausta tietokannalta ennen kuin jatkaa pyyntöjen käsittelyä, jonka vuoksi joissain tapauksissa onkin suositeltavaa käyttää *ASGI*-ajomuotoa.



Kuva 8. Django-pakettien latausmäärät päivittäin [PyPI Stats, 2023c].

```
# views.py
from django.http import HttpResponse

def hello_world(request):
    return HttpResponse('Hello World')

# urls.py
from django.contrib import admin
from django.urls import path
from . import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', views.hello_world, name='hello_world')
]
```

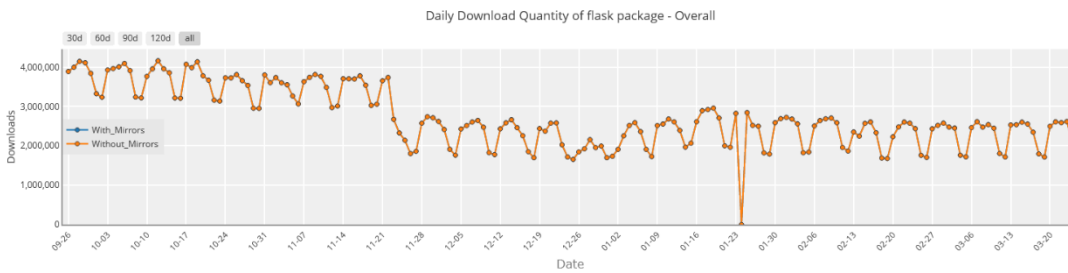
Koodiesimerkki 4. "Hello World" -sovellus ja sen esimäärittelemä koodi Django-ohjelmistokehyksellä.

2.6.2 Python Flask-ohjelmistokehyksellä

Flaskin kehittäjä Armin Ronacher [GitHub, 2023a; Ronacher, 2023] julkaisi Flaskin vuonna 2010 [GitHub, 2023a]. Se on kirjoitushetkellä tunnettu verkkorajapintaohjelmistokehys, joka toimii Pythonin WSGI-standardilla, ja joka sai alkunsa toisten ohjelmistokehysten kääreinä [Pallets Projects, 2023c]. Flask itsessään sallii vapaamuotoisen ohjelmistorakenteen toteuttamisen, näin ollen mielipiteetön ohjelmistokehys. Dokumentaationsa mukaan Flask toimiessaan WSGI-pohjalla ei toimi tapahtumapohjaisesti pääsäikeellä, vaan ajaa pyynnöt asynkronisesti omalla säikeellään, jonka sivusto itsekin myöntää johtavan jotenkin kohonneisiin tehokkuusvaatimuksiin [Pallets Projects, 2023c].

Koska Flask on pohjimmiltaan luotu kääreeksi Jinjalle ja Werkzeugille [Pallets Projects, 2023c], toimii Flask niiden mukaisesti. Djangosta mallia ottaen, myös Flask pystyy Jinjalla toteuttamaan RESTful-applikaatioiden ohella näkymiä [Pallets Projects, 2023a]. Flaskin latausmäärät löytyvät kuvasta 9, sekä yksinkertainen esimerkki rajapinnan toteutuksesta on annettu koodiesimerkissä 5.

Koska Flask on luotu käsittelemään pyynnöt WSGI-standardilla, se ei toimi ASGI-standardin mukaisesti, vaikka käsitteleekin pyynnöt asynkronisesti kohonneella käsittelykustannuksella [Pallets Projects, 2023c]. ASGI-standardin toteutusta varten Flaskista on olemassa vaihtoehtoisia toteutushaaroja, kuten Quart, joka on toteutettu hyödyntämään ASGI-mallia [Pallets Projects, 2023b], josta voi olla mahdollista saada erilaisia tehokkuustuloksia.



Kuva 9. Flask-paketin päivittäiset latausmäärät [PyPI Stats, 2023a].

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello_world():
    return "Hello World"

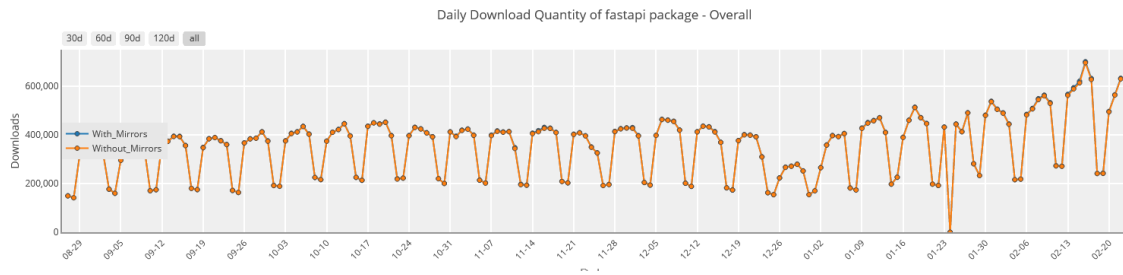
if __name__ == "__main__":
    app.run()
```

Koodiesimerkki 5. "Hello World" -sovellus Flask-ohjelmistokehyksellä.

2.6.3 Python FastAPI-ohjelmistokehyksellä

FastAPI on uudempaa ASGI-rakennetta noudattava mielipiteetön Sebastián Ramírezin julkaisema Python-verkkosovelluskehys, joka on luotu vuonna 2018 [GitHub, 2023b] Unicorn- ja Starlette-kehysten päälle [Ramírez, 2023a]. Kuvassa 10 näkyy tämän kehyksen latausmäärät, sekä "Hello World" -esimerkki on koodiesimerkissä 6.

Toisin kuin Flask, FastAPI itsessään ei sisällä varsinaisia työkaluja verkkosivujen rakentamiseen Django-tyylillä [Ramírez, 2023b], vaan tämä ohjelmistokehys antaa kehittäjän valita itse käyttämänsä lisäosat. FastAPI:n voidaan kuitenkin mahdollisesti huomata muistuttavan hyvin paljon Flask-kehysten ohjelmointityyliä, antaen käyttäjän määrittellä polut, jotka palauttavat vastauksia, kuitenkin oletuksena tarjoten enemmän valmiiksi toteutettuja työkaluja ohjelmistojen tekemiseen.



Kuva 10. FastAPI-paketin latausmäärät päivittäin [PyPI Stats, 2023b].

```
from fastapi import FastAPI
app = FastAPI()

@app.get("/")
async def root():
    return {"message": "Hello World"}
```

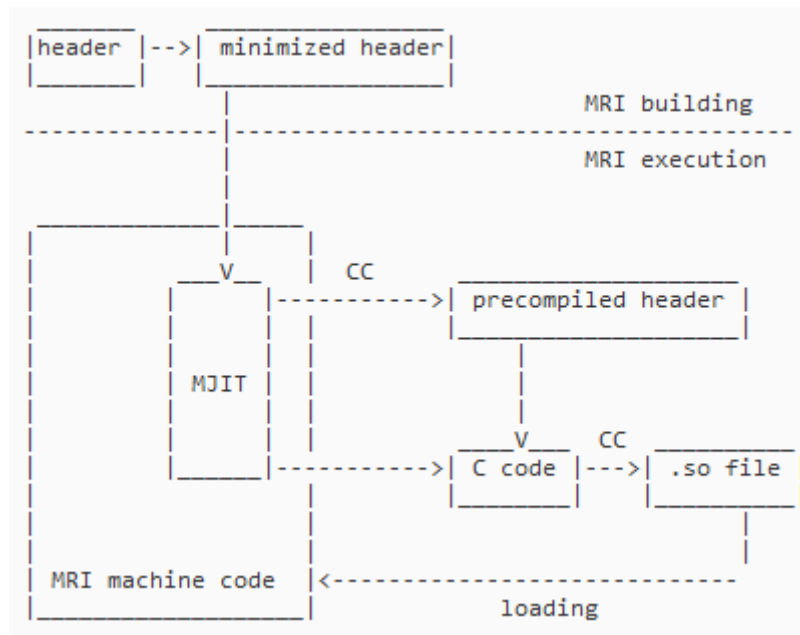
Koodiesimerkki 6. "Hello World" -sovellus FastAPI-ohjelmistokehyksellä.

2.7 Ruby ohjelmointikielenä ja Ruby on Rails -ohjelmistokehyksellä

Ruby on Yukihiro Matsumoton kehittämä ja vuonna 1995 julkaisema ohjelmointikieli, jonka toteutukseen on otettu mallia muista hänen suosimistansa ohjelmointikielistä [Ruby, 2023a]. Rubyn sivuston kirjoitushetkellä sen väitetään olleen yksi eniten kasva-neita ohjelmointikieliä, pääasiassa sen päälle rakennetun Ruby on Rails -verkko-ohjel-mointikehyksen vuoksi [Ruby, 2023a]. Esimerkki yksinkertaisesta Ruby-ohjelmasta löy-tyy koodista 7.

Ruby itsessään on vahvasti tyyppitetty oliopohjainen ohjelmointikieli, jossa kaikki käsitel-tävät datatyypit ovat olioita [Ruby, 2023b]. Monista muista ohjelmointikielistä poiketen (esimerkiksi Java ja C++) Rubyssa oliot eivät voi periä toisia olioita, vaan ennemminkin käyttää niitä lisättävinä moduuleina [Ruby, 2023a]. Kuten muillakin tulkatuilla kielillä, myös Rubylla on useampia tulkkeja, jotka voivat ajaa sillä toteutettuja ohjelmia. Yksi näistä on Rubyn vanhempi tulkki Ruby, toiselta nimeltään MRI tai CRuby, joka on sa-malla myös Rubyn tulkkien referenssitoteutus, sekä Sasadan [2005] toteuttama YARV, joka on korvannut edellä mainitun tulkin Rubyn virallisena tulkkina.

Kuten edellä on mainittu ja monen muunkin dynaamisen ohjelmointikielen tapauksessa, myös Ruby on tulkattu kieli, joka saattaa vaikuttaa sen tehokkuuteen valitun tulkin toteu-tuksen perusteella. Uudemmassa YARV-toteutuksessa Rubyssa voidaan kuitenkin hyö-dyntää JIT-mallista kääntämistä, näin mahdollisesti tehostaen kielen itsensä tehokkuutta [Ruby, 2023c], joka löytyy kuvasta 11.



Kuva 11. Ruby-kielen JIT-kääntö [Ruby, 2023c].

Rubyn YARV-tulkki toimii sisäisesti pinokoneen avulla, jossa konekäskyt on kirjoitettu peräkkäiseen järjestykseen. Tällä tulkilla on myös mahdollista kääntää koodi ennen sen suoritusta YARV:n kirjastoja hyödyntäväksi C-ohjelmaksi, mahdollisesti nopeuttaen sen suoritusaikaa verrattuna ajon aikana käännettäviin kieliin [Sasada, 2005]. Tällä JIT-mallilla tuotettu C-koodi käännetään laitteistokoodiksi, joka voisi mahdollisesti olla muita kieliä nopeampaa.

```
puts "Hello World"
```

Koodiesimerkki 7. "Hello World" -ohjelma Ruby-ohjelmointikiellä.

Ruby on Rails, lyhyemmältä nimeltään Rails, on Ruby-ohjelmointikielen päälle rakennettu avoimen lähdekoodin verkko-ohjelmistokehys, joka noudattaa MVC-mallia, näin tehden siitä mielipiteellisen, ohjaten kehittäjiä käyttämään ennalta määrättyä arkkitehtuuria. Esimerkkinä tästä voisi olla esimerkiksi ohjelmarakenteen jaottelu kanaviin (engl. *channel*), ohjaajiin (engl. *controller*), polkuihin (engl. *route*), sekä muihin vastaavanlaisiin osiin. Yksinkertaistettu malli tästä löytyy koodiesimerkistä 8.

Rails oli David Heinemeier Hanssonin vuonna 2003 tekemä ja vuonna 2004 julkaisema kehys, jonka hän kehitti työskennellessään 37signals-yritykselle, jonka hän omistaa [Hansson, 2023a]. Kirjoitushetkellä Railsilla oli päälle 420 miljoonaa latauskertaa [Hansson, 2023b]. Rails itsessään jakaa pyynnöt omille asynkronisille säikeilleen, jossa jokaisella säikeellä on oma versionsa pyyntöä käsittelevistä luokista [Hansson, 2023c].

Sisäisesti verkkopyynnön saadessaan Ruby on Rails välittää saadun pyynnön eteenpäin sitä vastaavalle ohjaajalle, joka noutaa datan sitä kuvaavalta mallilta, käsittelee sen, sekä lähettää sen asiakkaalle. Ruby on Rails on rakennettu PHP:n tavoin lähettämään vastaukset yleensä HTML-sivuna, mutta se tarjoaa myös JSON-vastaukset, näin mahdollistaen RESTful-rajapintojen toteutuksen.

```
# config/routes.rb
Rails.application.routes.draw do
  get 'hello_world/index'
end

# app/controllers/hello_world_controller.rb
class HelloWorldController < ApplicationController
  def index
  end
end

# index.html.erb
Hello World
```

Koodiesimerkki 8. "Hello World" -ohjelma Ruby on Rails -ohjelmistokehyksellä.

2.8 PHP ohjelmointikielenä ja Laravel-ohjelmistokehyksellä

PHP (PHP: Hypertext Preprocessor) on laajasti käytetty heikosti tyyhitetty avoimen lähdekoodin ohjelmointikieli, jonka avulla HTML-verkkosivuja voidaan luoda palvelimen puolella [The PHP Group, 2023a]. Kieli on Rasmus Lerdorfin vuonna 1994 kehittämä, jonka hän loi alun perin omia henkilökohtaisia ansioluettelosivuja varten [The PHP Group, 2023b]. Tyylinsä mukaisesti PHP-koodin sisälle on upotettu luotava HTML, näin mahdollistaen PHP-tulosteen lisäämisen sivustoon missä tahansa ohjelmiston ajovaiheessa.

Myös PHP on tulkattu kieli, jossa kirjoitettua koodia pystyy ajamaan eri järjestelmillä muokkaamatta alkuperäistä verkkosovellusohjelmistokoodia [The PHP Group, 2023c]. PHP-ohjelmia tulkataan yleisesti kirjoitushetkellä JIT-mallilla, jossa tavukoodiksi tulkattu koodi pidetään välimuistissa tallessa, näin nopeuttaen suoritusta vähentämällä koodin uudelleentulkkauksista [Krill, 2020]. Nykyisellään PHP:n voidaan nähdä toimivan jotakuinkin oliopohjaisesti, näin tehden siitä ainakin osittain olio-ohjelmointikielen. "Hello World" -esimerkki PHP:n toimintamallista näkyy koodissa 9.

```
<?php  
echo 'Hello World';  
?>
```

Koodiesimerkki 9. "Hello World" -ohjelma PHP-ohjelmointikielellä.

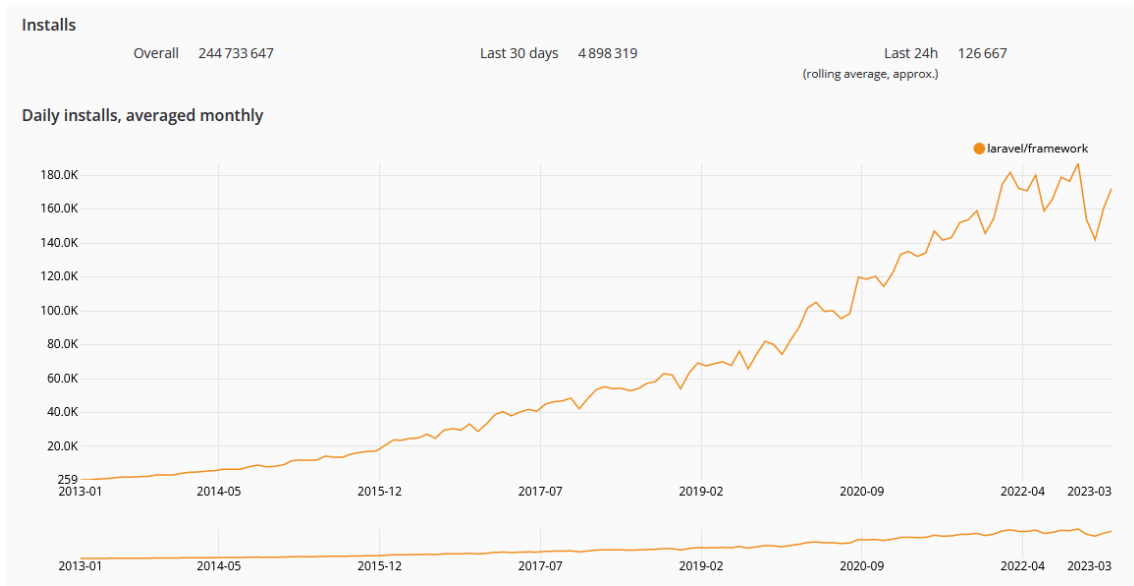
Laravel on PHP:lla kirjoitettu täydellinen pinokehys, joka on tarkoitettu verkkosivujen luontiin. Se toimii näennäisesti mielipiteellisellä mallilla, jossa tämä kehys rajoittaa käyttäjän ohjelmoinnin toteutusarkkitehtuuria. Laravelin mukana on mahdollista saada monia muitakin ohjelmointikehyksiä, joilla ohjelmointia voidaan nopeuttaa, kuten esimerkiksi Echo, jolla voidaan toteuttaa PHP-pohjaiset WebSocket-rajapinnat. [Laravel, 2023]

Laravel, kuin myös PHP, on itsessään normaalisti synkronisesti ajettu, mutta Laravel itsessään käynnistetään useasti monella pyyntöjen käsittelijällä, eli työntekijällä, jotka voivat käsitellä pyyntöjä rinnakkain. On kuitenkin huomattavaa, että tämä riippuu käytetystä työkalusta, olkoon se Nginx, Apache, tai jokin muu vastaava. Laravelin latausmäärät näkyvät kuvassa 12, sekä yksinkertaistettu "Hello World" on annettu koodiesimerkissä 10.

Laravelin kehitti Taylor Otwell, sekä se julkaistiin vuonna 2011 [Barnes, 2016] avoimen lähdekoodin pakettina. Sisäisesti Laravel käyttää ilmeisesti MVC-mallia. Toisin kuin moni muu tämän tutkielman ohjelmistokehys, Laravelin kehittäjät tarjoavat Composerin kautta useita ohjelmistopaketteja täydentämään kyseistä ohjelmistokehystä (esim. Echo, Breeze, ja Vapor).

```
<?php /* hello.php */ ?>  
<?php  
Route::get('/', function () { return view('hello'); });  
?>  
  
<?php // resources/views/hello.blade.php ?>  
Hello World
```

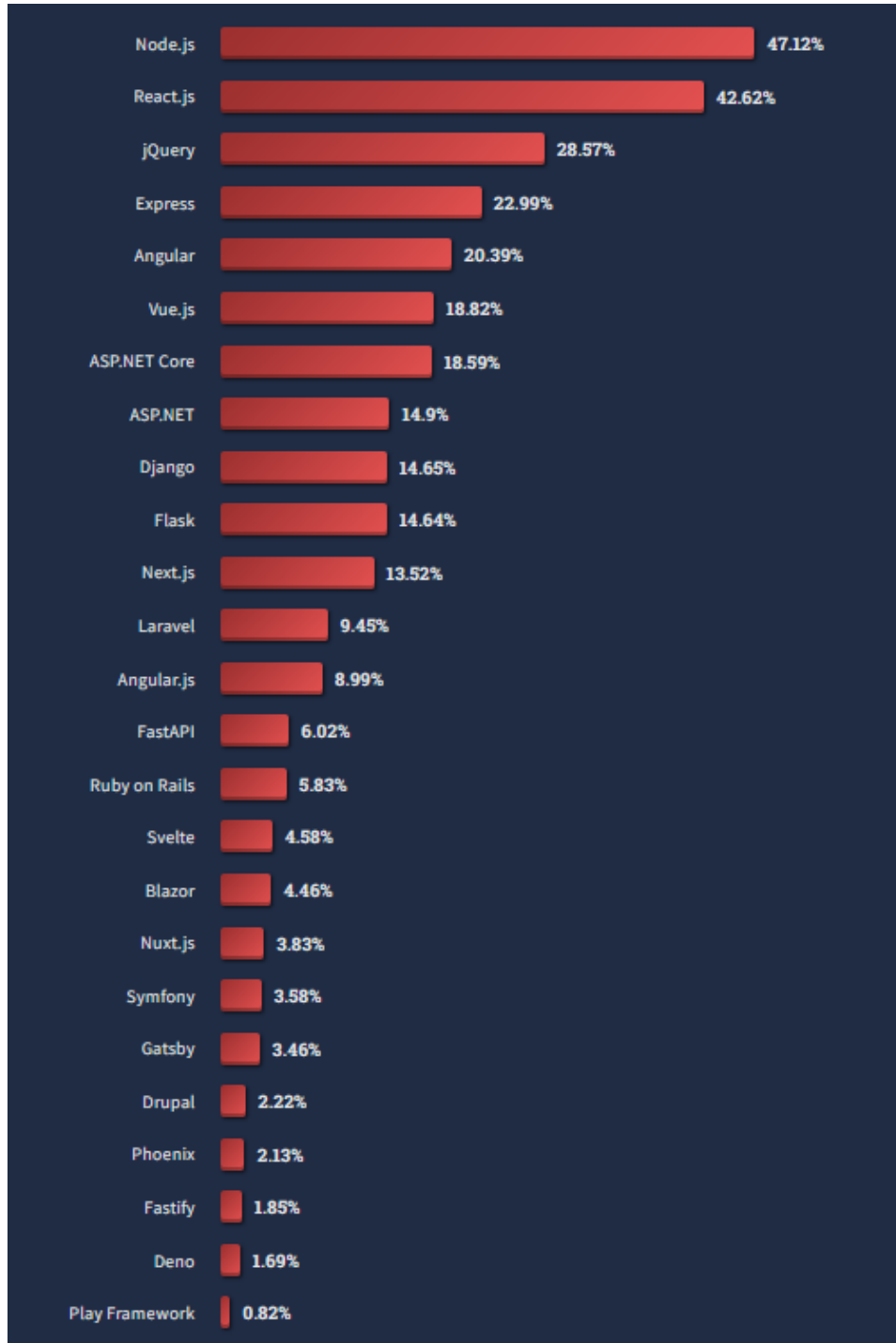
Koodiesimerkki 10. "Hello World" -ohjelma Laravel-ohjelmistokehyksellä.



Kuva 12. Laravelin latausmäärät kuukausittain [Packagist, 2023].

3 Kehysten vertailu aiemmissä tutkimuksissa

Tutkielmaan valittiin mitattavat kehykset joidenkin suosituimpien verkko-ohjelmistokehysten listalta, jotka näkyvät kuvassa 13.



Kuva 13. Yleisimmin käytetyt verkkorajapintakehykset ja teknologiat Stack Overflowin suorittaman kyselytutkimuksen perusteella [Stack Overflow, 2022].

Tutkielmassa tullaan ottamaan huomioon aiempia aiheeseen liittyviä tutkimuksia, sekä niiden tuloksia verrataan näihin tässä tutkielmassa mitattuihin tuloksiin.

3.1 Aiemmat tutkimukset

Koska monella eri tutkimuksella tulokset mitataan joissain tapauksissa pisteillä, joissain aikayksiköillä, sekä joissain muilla tavoilla, tulee nämä normalisoida, jotta eri tutkimustavoilla saadut tehokkuusluvut olisivat vertailukelpoisia keskenään yleisellä tasolla, vaikkeivat ne välttämättä mittaisikaan samaa osa-aluetta; onhan tämä tehokkuusvertailu kuitenkin vain yleisen tason vertailua. Tämä normalisointi toteutetaan tavalla, jossa jollekin rajapinnalle asetetaan arvoksi 1, sekä muiden verkkokehysten tulokset normalisoidaan tasolle, jossa kehysten keskinäinen suhde pysyy samana. Näistä esimerkkinä voidaan nostaa myöhemmin mainitut TechEmpower-sivuston vertailutulokset taulukossa 1, jonka viimeiseen sarakkeeseen tulokset on listattu normalisoidusti. Myöhemmissä tuloksissa kaikki arvot ovat normalisoituja.

TechEmpower-sivusto on mittauttanut laajan määrän erilaisia verkkokehyksiä, joista tämän tutkielman kehykset on poimittu taulukkoon 1. Onkin huomattavaa, että samalla ohjelmointikielellä toteutettuja kehyksiä löytyy sekä listan parhaimmasta että huonoimmasta päästä, näin varmentaan väitettä siitä, että pelkästään valittu kieli ei kerro kaikkea lopullisista mittaustuloksista.

Ohjelmistokehys	Painotettu tulos (isompi parempi)	Painotettu tulos (normalisoitu)
Flask	1229	1.00
FastAPI	1184	0.96
Express	615	0.50
Laravel	371	0.30
Ruby on Rails	366	0.30
Django	274	0.22

Taulukko 1. Verkkohjelmistokehysten painotettu tehokkuus [TechEmpower, 2022a].

Pimentel ja Nickerson [2012] ovat myös suorittaneet testejä, tässä tapauksessa testaten WebSocket-protokollan nopeutta mittaamalla yksittäisen WebSocket-asiakkaan keskustelunopeutta palvelimen kanssa mitaten keskimääräistä keskustelu-aikaa. Näitä tuloksia he ovat verranneet alussa mainittuihin vaihtoehtoisiin reaaliaikaisiin keskusteluyhteyksiin. Toisin kuin heidän testeissään, tässä tutkimuksessa tullaan testaamaan enemmänkin liittäisiä nopeuksia monella eri asiakkaalla, joilla vastaanotetaan pieni määrä päivityksiä.

Toisessa avoimen lähdekoodin projektissa [Web Frameworks Benchmark, 2023] on myös tarkasteltu joitain verkkorajapintakehyksiä, sekä kyseisen tutkimuksen tuloksista on poimittu tämän tutkielman kannalta tärkeimmät tulokset taulukkoon 2. Tuloksista onkin huomattavaa, että sen tulokset ovat osittain päinvastaisia TechEmpower-sivuston tulosten kanssa sijoittaen Djangon yhdeksi nopeimmista täysistä pinokehysistä, mitä se ei TechEmpower-sivuston tuloksilla ole. Taulukkoon on valittu tuloksista lähteen ensimmäinen tulossarake, jossa pyyntöjä lähetettiin 64 kappaletta sekunnissa. Jokainen tulos on kaikkien näiden pyyntöjen keskimääräinen vasteaika.

Ohjelmistokehys	Keskimääräinen vasteaika (ms) 64:llä pyynnöllä sekunnissa	Keskimääräinen vasteaika (normalisoitu) 64:llä pyynnöllä sekunnissa
Flask	3.35	0.14
FastAPI	1.84	0.08
Express	2.99	0.13
Laravel	23.55	1.00
Ruby on Rails	23.51	1.00
Django	5.2	0.22

Taulukko 2. Erään avoimen lähdekoodin testaustuloksia [Web Frameworks Benchmark, 2023].

Abbate ja muut [2020] ovat myös testanneet joitain ohjelmistokehyksiä IoT-kontekstissa, mutta tutkimuksessa on huomattavaa se, että he kertovat keskittyneensä enemmän valitun

ohjelmointikielen tehokkuuteen. Edellä mainitussa tutkimuksessa mainitaan myös se, että jotkin aiemmat tutkimukset mittaavat joitain näiden kehysten osa-alueita, mutta he ovat itse käsitelleet tietokannan käsittelyn tehokkuutta, sekä limittäisten verkkopyyntöjen käsittelyä, kuten tässäkin tutkielmassa tullaan tekemään [Abbade *et al.*, 2020]. Tutkimuksessa [Abbade *et al.*, 2020] huomattiin JavaScriptin olevan Pythonia tehokkaampi varsinkin huomioon ottaen virheiden määrän, joka voi antaa viitteitä lopullisista tuloksista. Tämän tutkimuksen voisi väittää olevan ristiriidassa aiemmin mainittujen tutkimusten kanssa, mutta täytyy ottaa huomioon se, että aiemmin mainitut tutkimukset tarkastelivat verkkorajapintoja, kun taas edellisessä tutkimuksessa vain kieliä itsessään. Tämän perusteella voisi olla mahdollista päätellä, että joidenkin kehysten toteutukset itsessään ovat laskeneet niiden tehokkuutta verrattuna tämän tutkimuksen parhaimpiin mahdollisiin tuloksiin. Tutkimuksen tuloksia tarkastellessa voidaankin huomata eri kielten onnistuneiden vastausten määrän vaihtelevan suuresti eri ajankohtina, jonka vuoksi olisi tärkeää mitata tässä tutkielmassa testit useampaan kertaan, jotta satunnaiset heikot tulokset eivät vaikuta lopullisiin mittaustuloksiin vahvasti.

3.2 Tehokkuustestaustyökalujen hyödyntäminen

Kuten Shaw [2000] mainitseekin julkaisussaan, on rajapintojen tehokkuustestausta varten julkaistu joitain työkaluja, joilla voidaan testausta automatisoida, mahdollisesti saaden tätä tutkielmaa tarkempia tuloksia. Näitä ovat esimerkiksi JMeter, Tsung, Vegeta, http_load, Thor, sekä wrk.

Näistä olemassa olevista testaustyökaluista huolimatta tähän tutkielmaan on toteutettu näivillä tavalla toimiva testaustyökalu [Turpeinen, 2023], jonka voi osittain sanoa matkivan monia JavaScriptin fetch-kutsua hyödyntäviä ohjelmistoja. Hyötynä isetoteutusta työkalusta tässä testauksessa on myös se, että tässä tullaan käyttämään Alfabetin V8-mootoria, jota käytetään myös kirjoitushetkellä yleisimmissä selaimissa [StatCounter, 2023]. Näin toimiessa voidaan replikoida mahdollisimman paljon normaalin verkkokäyttäjän mahdollista lopullista RESTful- ja WebSocket-rajapintojen ohjelmistokokoonpanoa.

4 Verkko-ohjelmistokehysten tehokkuusvertailu

Aiempien rajoitteiden ja tutkimusten perusteella voidaankin hyvin luoda tätä tehokkuusvertailua. Tämän tutkielman tehokkuusvertailua varten asetetaan tiukat määritelmät käytetyille arkkitehtuurille ja pyyntöjen käsittelylle. Näin voidaankin myös varmentaa tai yleistää joitain aiempien tutkimusten tuloksia. On myös huomattavaa, että verratessa edellä mainittujen sivustojen tuloksia, moni tyyppitön kieli sijaitsee tehokkuudessa huomattavasti joitain tyyppiteltyjä kieliä alempana tehokkuusvertailuissa. Näiden erojen vuoksi erilaisten kehysten ja kielten vertailut tulisi tehdä harkiten.

Kaikki tutkielmassa olevat ohjelmistokehykset eivät ole muutoinkaan täysin tasavertaisia keskenään johtuen niiden kokonaisvaltaisesta rakenteesta. Voidaan olettaa, että täydet ohjelmistokehykset Django, Laravel, sekä Ruby on Rails tulevat toimimaan jonkin verran muita kehyksiä hitaammin niiden mukaan valmiiksi sisällytetyn ja esimääritellyn ohjelmistokoodin määrän vuoksi, joka tulisi ottaa huomioon tuloksia vertaillessa.

Kohdassa 4.1 kerrotaan kuinka tutkimus tullaan tekemään ja ajamaan, sekä miten eri ohjelmistot on käynnistetty. Seuraavassa kohdassa 4.2 esitellään testeissä käytettävä tietokannan rakenne, sekä kuinka testidata on sinne luotu. Tämän jälkeen kohdissa 4.3 ja 4.4 kerrotaan yleinen kuvaus ajettavista testeistä, sekä lopuksi kohdassa 4.5 kerrotaan tutkimuksessa käytetyn testausympäristön tiedot.

4.1 Tutkimusmenetelmä

Testattavat ohjelmistokehykset testataan valtaosin oletusasetuksilla, sammuttaen joitain yleisiä suoritusnopeutta vähentäviä osia, kuten esimerkiksi kirjainpitoa. Vaikka joitain yleisiä tehokkuuteen vaikuttavia osia voidaankin sulkea, tässä tutkielmassa ei tulla poistamaan täysistä verkkokehyksistä niiden tarjoamia osia, jotka saattavat tehdä niistä hitaampia muihin kehyksiin verrattuna, joissa näitä ominaisuuksia ei ole toteutettuna. Jokainen kehys suoritetaan *tuotantoasetuksilla* (engl. *production*), jotta eri kielten ja kehysten virheenkorjaustyökalut eivät hidasta ohjelmiston suoritusta.

Testien tulokset mitataan asiakaskoneella, näin ollen vähentäen testauksesta aiheutuvaa raskautta palvelinkoneelle. Tästä tavasta saadaan myös selville se, että kuinka moni pyyntö kulkeutuu onnistuneesti läpi kohtuullisen aidonlaisessa palvelinympäristössä, jossa palvelin on erillisellä koneella kuin asiakas, toisin kuin joissakin muissa tutkimuksissa. Tällä testauksella voidaan saada selville myös sitä, että kuinka monta pyyntöä on mahdollista pyytää palvelinohjelmistolta ennen kuin sen resurssit ja rajoitteet eivät salli suurempaa raskautta. Jotta ohjelmistojen mahdolliset viat eivät vaikuttaisi testien tuloksiin,

palvelinkone käynnistetään jokaisen testin välissä uudelleen. Tarkemmat replikointiohjeet löytyvät seuraavasta listasta:

- Palvelin käynnistetään testattavana oleva kehys ja ohjelmisto asennettuna.
- Palvelinkoneen pääsy julkiseen verkkoon on estetty, jotta mahdolliset automaattisten verkkohakujen ja diagnostiikkaraportointien vaikutus tuloksiin voidaan minimoida.
- Palvelinkone on yhdistetty reitittimellä asiakaskoneeseen, jonka kautta otetaan suora yhteys. Näin voidaan minimoida mahdollisen satunnaisen verkkoviiveen vaikutus testin tuloksiin.
- Testien ajon ja tallennuksen jälkeen palvelinkone uudelleenkäynnistetään.
- REST-rajapinnoilta mitataan keskimääräinen käsittely- ja vasteaika, kun taas WebSocket-rajapinnoilta vain keskimääräinen vasteaika.

On huomattavaa, että kuten myös TechEmpower-sivuston testeissä, näissäkin testeissä testit ajetaan useampaan kertaan, jotta tuloksista saatavat keskiarvot vähentävät riskiä siitä, että jokin ulkopuolinen vaikuttaja olisi vaikuttanut testien tuloksiin. Koska testajalla ei ole vastaavanlaista vapaata pääsyä toisiin palvelintason laitteistoihin, joilla testit voitaisiin näin suorittaa, voidaan nämä testit ajaa vain saatavilla olevilla laitteistoilla, jotka löytyvät kohdasta 4.5.

Jotta kokonaisvaltainen kuva käytetyistä kehyksistä voidaan saada, on tärkeää ottaa huomioon myös käytettyjen tulkkien versiot, jotka löytyvät listattuna taulukosta 3. Laravelin tapauksessa on myös ajettu seuraavat optimointikomennot ennen kehysten ajoa:

- `composer install --optimize-autoloader --no-dev`
- `php artisan config:cache`
- `php artisan route:cache`
- `php artisan view:cache`

Tulkki	Tulkin versio
Python (Django, FastAPI, Flask)	3.8.10 (4.1.7, 0.94.1, 2.2.3)
PHP (Laravel)	7.4.3-4ubuntu2.18 Zend Engine v3.4.0 (7.30.6)
NodeJS (Express)	9.5.0 (4.18.2)
Ruby (Ruby on Rails)	3.1.3p185 (2022-11-24 revision 1a6b16756e) [x86_64-linux] (7.0.4.2)

Taulukko 3. Tulkkien versiot.

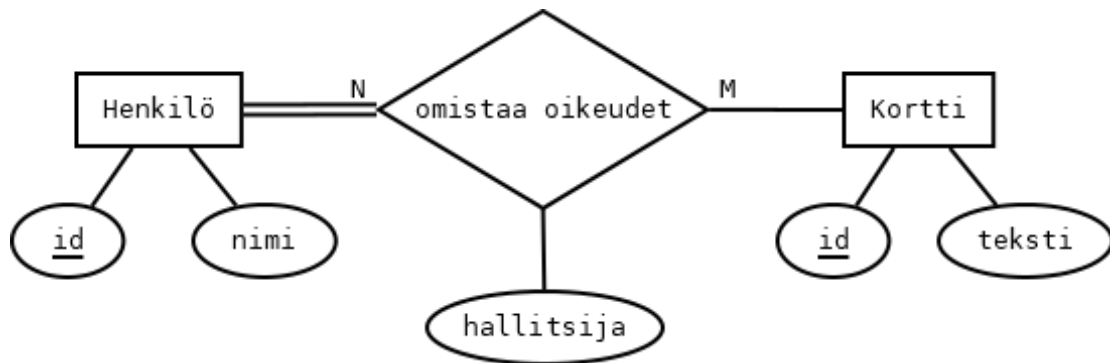
Mikäli jokin ohjelmistokehys tai ohjelmointikieli sallii optimointiasetusten käyttämisen, niitä tullaan käyttämään testatessa, mikäli se on yksinkertaisesti tehtävissä. Kielikohtaiset käynnistyskomennot parametreineen onkin listattu taulukkoon 4. Voidaankin huomata, että valtaosin komennoissa vain määritellään IP-osoite, jossa palvelin toimii, portti, jossa palvelin tullaan ajamaan, funktio tai sovellus, joka ajetaan, sekä mahdollinen työntekijän tyyppi. Voidaan myös huomata, että joillakin käynnistyskomentojen syötteillä vaihdetaan kirjanpidon määrää ja ohjelmiston ajotilaa, vaihtaen sen testaus-tilasta tuotantotilaan. Kehykset, joilla tätä asetusta ei tarjottu, on niille asetettu tuotantotila päälle asetuksista, mikäli tällainen on mahdollista.

Ohjelmistokehys	Käynnistyskomento
Django	<code>gunicorn stresstest.asgi:application -k uvicorn.workers.UvicornWorker --bind 0.0.0.0:3000</code>
FastAPI	<code>gunicorn main:app -k uvicorn.workers.UvicornWorker --bind 0.0.0.0:3000 --log-level critical</code>
Flask	<code>gunicorn -k flask_sockets.worker main:app --bind 0.0.0.0:3000</code>
Laravel	<i>Ei erikseen tehtävää käynnistystä, sillä tämä käynnistyy automaattisesti Apache2:n mukana.</i>
Express	<code>node index.js</code>
Ruby on Rails	<code>rails server -e production -b 0.0.0.0</code>

Taulukko 4. Ohjelmistokehysten käytetyt käynnistyskomennot.

4.2 Testidata

Testien data koostuu henkilöistä sekä korteista, joita jotkin määritellyt henkilöt voivat muokata, näin ollen kuvitteellinen korttitaulu, jonka ainoa toteutus löytyy tietokannasta. Tällä voidaan tässä tutkielmassa testata yleisellä tasolla sitä, kuinka nopeaa kehys onnistuu noutamaan dataa palvelimelta, muokkaamaan sitä ja järjestelemään sen, sekä vastaamaan asiakkaalle sen lähettämään pyyntöön. Tällaisiin palveluihin monesti liittyvää salausta ja todennusta ei tähän palveluun oteta mukaan, jotta tutkimuksen laajuus ei nouse liian suureksi. Tietueet ovat tallennettuna PostgreSQL-tietokantaan, johon on tallennettu ja luotu testikäyttäjiä tutkielman datanluontikoodin [Turpeinen, 2023] avulla. Tietokantarakenne löytyy visualisoituna kuvasta 14. Tietokantaan syötetään koodin avulla 5 000 käyttäjää, sekä jokaisella käyttäjällä on omistajuus viiteen (5) korttiin, sekä muokkausoikeus toiseen viiteen (5) korttiin. Näillä asetuksilla tietokantaan luodaan yhteensä 50 000 korttia. Testauksen kätevyys vuoksi kuitenkin vain ensimmäisen 4 096 käyttäjän kortteja tullaan pyytämään tietokannasta.



Kuva 14. Testitietokannan pöytien rakenne.

4.3 Datavälitys tietokannasta

REST-testauksessa käytetään TechEmpowerin [TechEmpower, 2022b] ”Fortunes” kaltaista testausmallia, jossa palvelin pyytää tietueita taustalla toimivasta tietokannasta, lisää siihen yhden tietueen lisää, sekä järjestee nämä aakkosjärjestykseen. Edellä mainitun testimallin hiukan hämmäntävä nimi tulee aiemman tutkimuksen tietokantataulun nimestä [TechEmpower, 2022b], eikä se tässä tutkielmassa tarkoita satunnaisdataa. Toisin kuin TechEmpower-sivuston testeissä, tässä testauksessa vastaus palautetaan REST-mallin mukaisesti JSON-muotoiltuna pakettina. Muut rajapinnan palauttamien arvot jätetään oletusarvoihin. Esimerkki vastaanotettavasta vastauksesta on luettavissa koodiesimerkistä 11. Testaus tullaan suorittamaan empiirisellä mallilla, jossa testit suoritetaan useaan kertaan, sekä niistä mitataan keskiarvo.

```
[
  {
    "id": 1,
    "teksti": "Kortin 1 teksti",
    "hallitsija": true
  },
  {
    "id": 10000,
    "teksti": "Kortin 10000 teksti",
    "hallitsija": false
  },
  ...
]
```

Koodiesimerkki 11. Esimerkki asiakkaan vastaanottamasta datasta.

REST-rajapintojen päätepisteet sijoitetaan osoitteeseen ”/users/{id}/cards” josta voidaan pyytää edellä mainitun tavan mukaisesti kaikki käyttäjän kortit, joita tyhjän mallikortin kanssa on yhteensä 11 kappaletta. Mikäli kortteja palautetaan virheellinen määrä taikka yhteys hylätään jollakin tavalla, merkitään kyseinen operaatio epäonnistuneeksi.

Tällä testauksella on mahdollista saada selville ohjelmistorajapinnan tietokantojen hyödyntämisen, datan JSON-muotoilun, joidenkin yleisten algoritmien, muistinhallinnan, sekä yleisen verkkoliikenteen hyödyntämisen tehokkuutta yhdistettynä yleisellä tasolla. Datan noutopyyntöjä lähetetään palvelimelle eksponentiaalisesti kasvavassa määrin, jotta voidaan nähdä järjestelmän ja kehityksen yhteistoiminnan toimivuus ja tehokkuus pienestä rasituksesta suureen. Näissä pyynnöissä pyydetään 4 096 käyttäjän tiedot rinnakkain tietyn kokoisissa määrissä.

Yhden yhteyden suurin aika on rajoitettu 30:een sekuntiin, jonka jälkeen se suljetaan. Mikäli testaustuloksissa vasteaika on noin 30:n sekunnin tasolla, voi se realistisessa ympäristössä olla suurempi.

Virheiden määrien mittauksessa ei erotella palvelinten antamia virheellisiä vastauksia aikakatkaistuihin yhteyksiin, koska käytännössä molemmilla tuloksilla on samankaltainen lopputulos käyttäjän näkökulmasta. Lopullisissa tuloksissa on mukana asiakkaan ja palvelimen yhdistämiseen kuluva aika RESTful- ja HTTP-pyyntöjen arkkitehtuurin vuoksi.

4.4 Päivystävä WebSocket-rajapinta

Koska moni verkkorajapinta hyödyntää WebSocket-rajapintaa nopeasti ja usein tapahtuviin muutoksiin, myös tätä testataan erilaisista rajapinnoista. Testattavassa järjestelmässä asiakasohjelma pyytää palvelimelta sen kellonaikaa. Vaikka WebSocket-rajapinnassa palvelin tyypillisesti työntää datan asiakkaille, myös asiakkaat voivat työntää dataa palvelimille. Tämän testin ratkaisussa molempia tapahtumia suoritetaan yhtäaikaisesti.

Testeissä testien määrät on määritelty kahdella parametrilla, asiakkaiden yhtäaikaisella määrällä, sekä asiakkaiden lähettämien pyyntöjen määrällä. Tämän tutkielman testeissä limittäisten asiakkaiden määrä on ilmoitettu tuloksissa, mutta asiakkaille lähetettyjen viestien määrä on rajoitettu 20 viestiin, joka toimii suurimpana mahdollisena määränä mitä rajapinta voi sillä hetkellä tuottaa virheellisiä tuloksia. Mikäli yhdellä pyynnöllä on yksikin virhe, on ne kaikki merkattu virheellisiksi. Voidaankin sanoa, kuten edellisessä testissä, että tämä testaus tullaan suorittamaan empiirisesti.

Koska verkkoyhteyksissä on riskinä puolikkaan yhteyden syntyminen johtuen WebSocket-protokollan toteutustavasta [Melnikov ja Fette, 2011], on jokainen pyyntö rajoitettu 30:n sekunnin aikarajaan, jonka jälkeen verkkoyhteys pakotetaan sammumaan. Vaikka aikaraja voisi olla suurempikin, on epätodennäköistä, että verkkokontekstissa rationaalinen ihmiskäyttäjä jäisi odottamaan tätä pidempään jonkin resurssin latautumista

[Nielsen, 2010]. Tämä 30:n sekunnin odotus ja yhteyden pakollinen sulkeminen on kriittinen osa testausta, sillä päälle jäävä kuollut yhteys jää varaamaan kaistaa sekä palvelimelta että asiakkaalta, näin vaikuttaen kaikkien seuraavien testien tuloksiin.

Mikäli näitä testejä ajetaan, täytyy palvelinkoneelta sallia prosesseille suuri yhdenaikaisen tiedostojen lukumäärä, sillä muutoin oletusarvolla oleva järjestelmä (testiympäristössä 1 024) ei pysty vastaamaan kaikkiin pyyntöihin käyttöjärjestelmän toimesta. Mikäli tätä rajamäärää ei nosta, on hyvinkin mahdollista, että lopullisissa tuloksissa on hyvin paljon virheitä kaikilla kehyksillä, rajoittaen testausmäärää. Tässä kontekstissa tiedostojen sallittu avausmäärä on asetettu suurimpaan laitteiston sallimaan määrään.

4.5 Testausympäristön tiedot

Asiakasympäristö ajetaan kirjoitushetkellä modernilla tietokoneella, jonka olennaisimmat komponentit on listattu seuraavassa listassa:

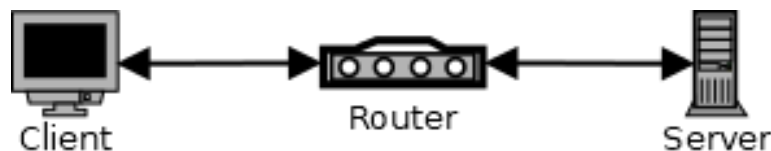
- Käyttöjärjestelmä: Windows 10 Pro 64-bit
- Prosessori: AMD Ryzen 5 3600X 6-Core, 3793 Mhz
- Keskusmuisti: 16 GB
- Fyysisen muistin tyyppi: SSD

Palvelinkone vastaavasti ajetaan huomattavasti vanhemmalla palvelinkoneella, jonka olennaisimmat tiedot ovat seuraavat:

- Käyttöjärjestelmä: Linux Mint 20.3 Cinnamon
- Prosessori: Intel Xeon W3520 4-Core, 2.67 GHz
- Keskusmuisti: 6 GB
- Fyysisen muistin tyyppi: HDD

Palvelinkoneen tietokantana toimii PostgreSQL-tietokanta, joka saattaa osaltaan vaikuttaa mitattuihin tuloksiin. Tietokanta itsessään on asennettu samalle koneelle kuin testattava kehys, mutta reaali maailman järjestelmissä tämän sijoittaminen eri järjestelmään olisi mahdollista.

Nämä kaksi järjestelmää on yhdistetty toisiinsa reitittimen (engl. *router*) avulla kuvan 15 mallisesti. Kuvasta voidaankin kertoa, kuten edellisessä kappaleessa, että palvelin ei ole yhteydessä muihin järjestelmiin kuin reitittimen kautta asiakaskoneeseen sitä palvellesaan.



Kuva 15. Testiarkkitehtuuri.

5 Tulokset

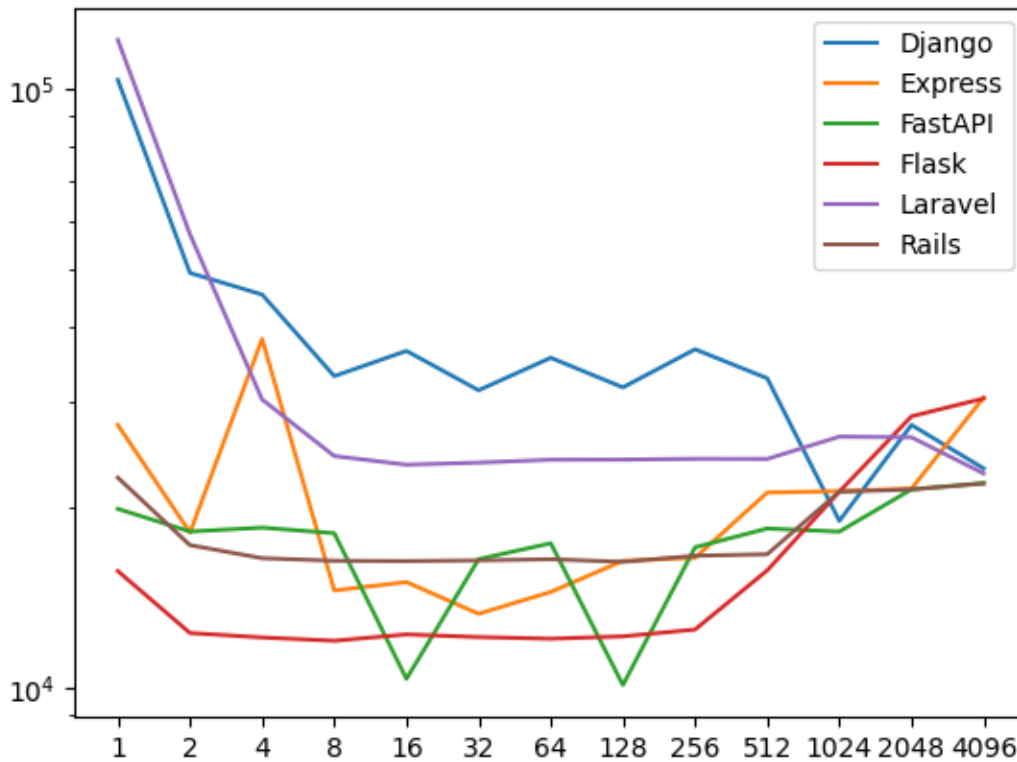
Tämän luvun tulokset saatiin ajamalla jokainen testi viisi kertaa läpi ja keräämällä saatujen tulosten keskiarvo. Kohdassa 5.1 käydään lävitse REST-rajapintojen tuloksia, sekä kohdassa 5.2 vastaavasti WebSocket-rajapintojen tuloksia.

5.1 Datan välitys tietokannasta

Tutkielmassa saatiin kerättyä runsaasti mittausaineistoa, jota voidaan samalla verrata yleisesti käytettyihin Nielsenin [2010] määrittelemiin aikarajoihin. Aikakuviin on lisätty kuhunkin kolme viivaa, jotka kuvaavat näitä aikojen raja-arvoja. Vihreä viiva kuvastaa aikaa, joka näyttäisi asiakkaalle välittömältä vastaukselta (500 ms), oranssi viiva kuvastaa aikaa, jossa asiakkaan toiminta pysyisi sujuvana (1 000 ms), sekä ylin punainen viiva kuvastaa aikaa, jonka jälkeen on todennäköistä, että asiakas lopettaa pyynnön, tai siirtyy tekemään toisia toimia (10 000 ms). Mittaustulokset eri kehyksistä näkyvät yhdistettynä kuvissa 16, 17, sekä 18. Vastaavasti numeeriset keskiarvot löytyvät taulukosta 5.

Ohjelmistojen tulosten kuvien ja taulukon perusteella voidaankin todeta, että niiden tehokkuus on verrattain lähellä TechEmpower-sivuston tuloksia REST-rajapintakäytössä. Yleisellä tasolla kaikissa kehyksissä voidaan huomata, että mikäli vastaukset lähetetään yksittäin palvelimelle, on niiden käsittelyaika seuraavia testejä hitaampi.

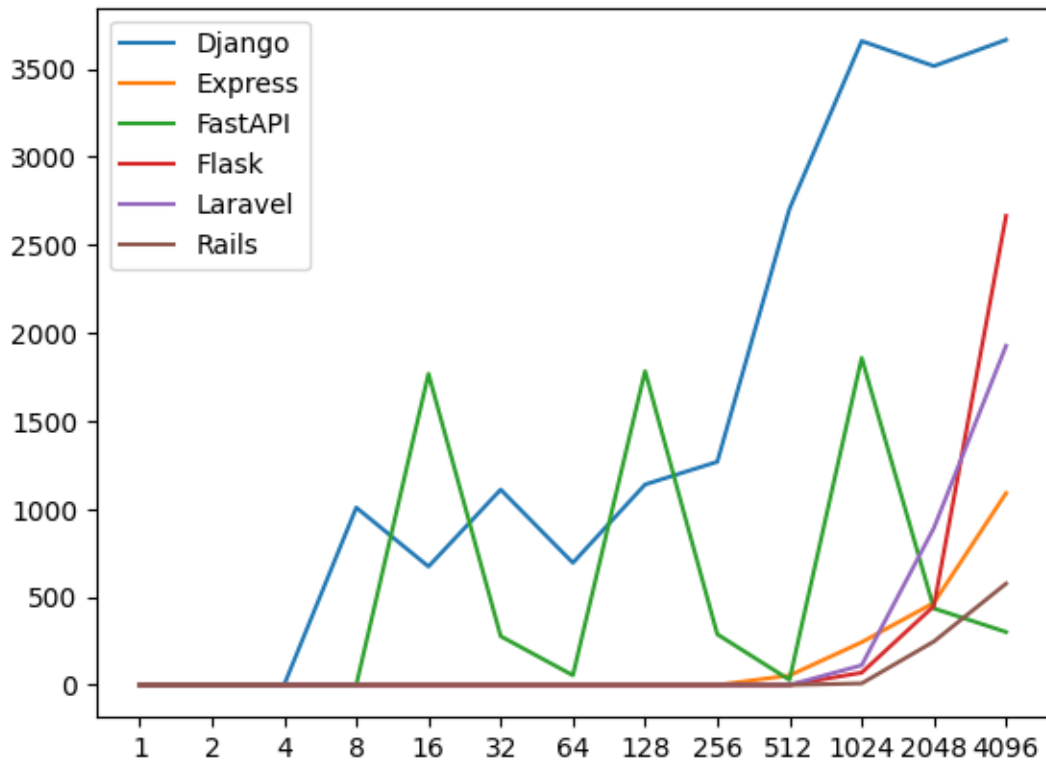
Tuloksia tarkastellaan alakohdissa pääasiassa välillä, jossa limittäisiä pyyntöjä oli 2–512. Tämä rajamäärä on otettu sen vuoksi, koska arvolla 512 voidaan huomata olevan jo joillakin rajapinnoilla haasteita vastata kaikkiin pyyntöihin, sekä yleisesti lähes kaikilla on jo ongelmia siinä vaiheessa, kun limittäisten palvelinpyyntöjen määrä on 1 024. Keskiarvosta on jätetty myös pois yksittäiset pyynnöt sen vuoksi, koska sen huomaa jo visuaalisesti olevan runsaasti erilainen kuin muut yllä mainitulla välillä olevista ajoista kaikilla kehyksillä.



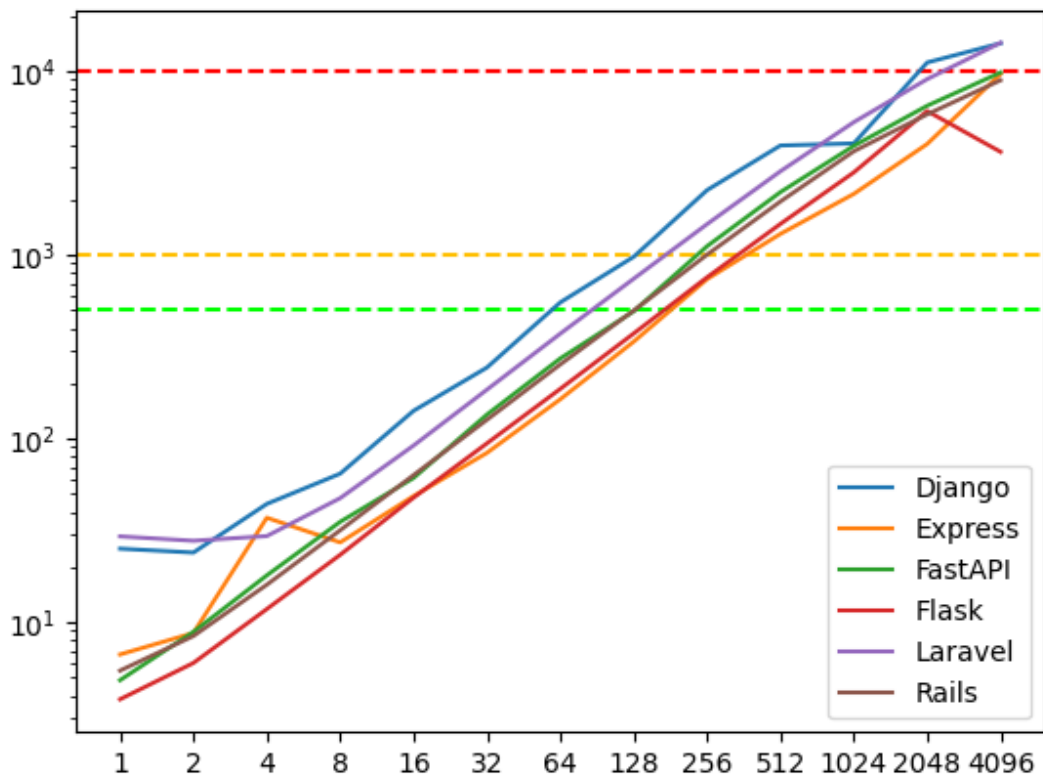
Kuva 16. Eri ohjelmointikehysten käsittelyaika suhteessa yhtäaikaisiin pyyntöihin.

Kehys	FastAPI	Express	Flask	Laravel	Rails	Django
Käsittely-aika	0.44 16 042	0.50 18 579	0.34 12 565	0.77 28 303	0.45 16 468	1.0 36 848
Vasteaika	0.53 482	0.33 305	0.36 331	0.71 647	0.48 440	1.0 917

Taulukko 5. Eri kehysten tasapainoitettut keskimääräiset ajat välillä 2-512. Pienempi parempi.

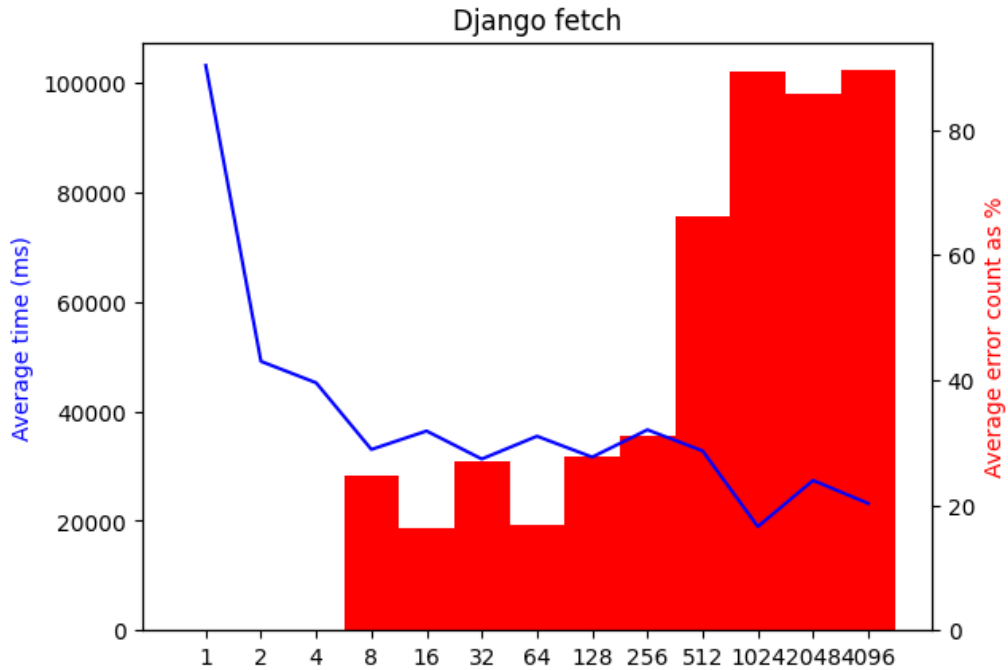


Kuva 17. Eri ohjelmointikehysten virhemäärä suhteessa pyyntöihin.

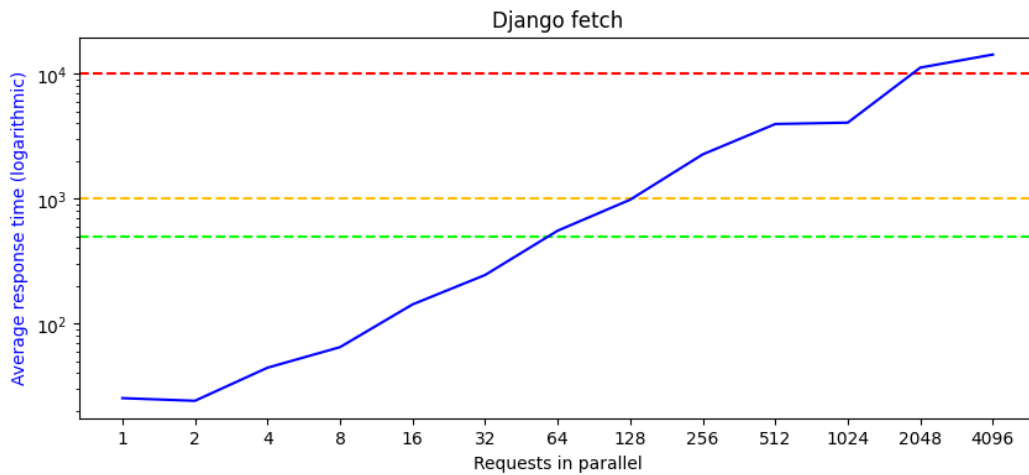


Kuva 18. Eri kehysten vasteaika suhteessa limittäisiin pyyntöihin.

5.1.1 Django



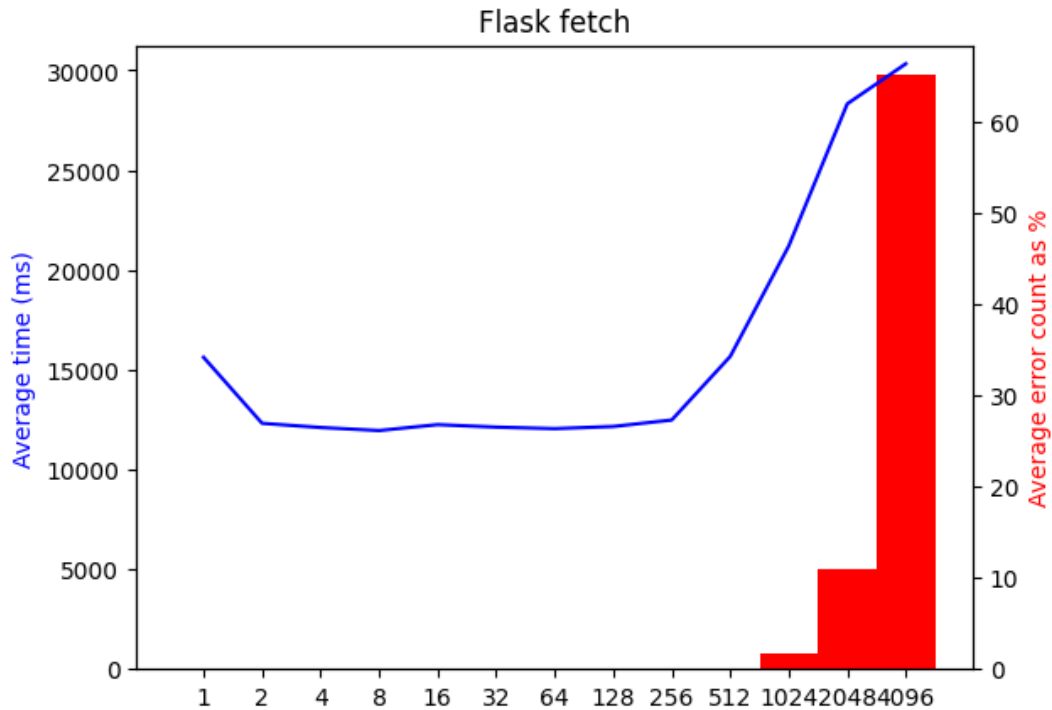
Kuva 19. Django-kehiksen pyyntöjen käsittelyaika ja virheiden määrä limittäisten pyyntöjen määrää kohti.



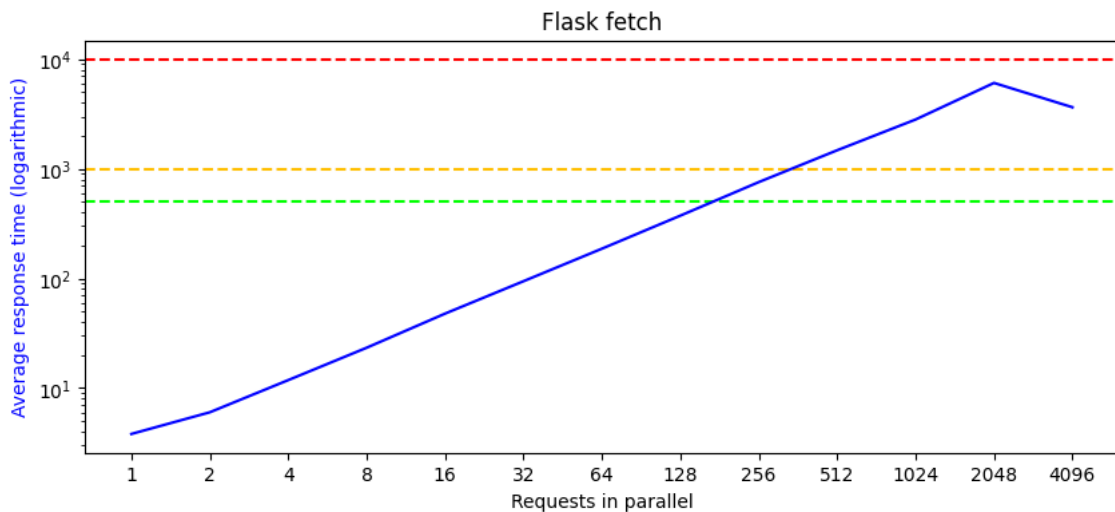
Kuva 20. Django-kehiksen vasteaika suhteessa limittäisiin pyyntöihin.

Django-kehiksen kuvissa 19 ja 20 voidaan huomata sen luovan virheellisiä vastauksia jo 8:n limittäisen pyynnön rajalla, sekä ylittävän suurimman suositellun vasteajan 4 096:lla pyynnöllä. Django-vasteaika oli keskimäärin 917 ms, sekä käsittelyaika 36 848 ms tarkastellessa väliä 2–512 limittäistä yhteyttä.

5.1.2 Flask



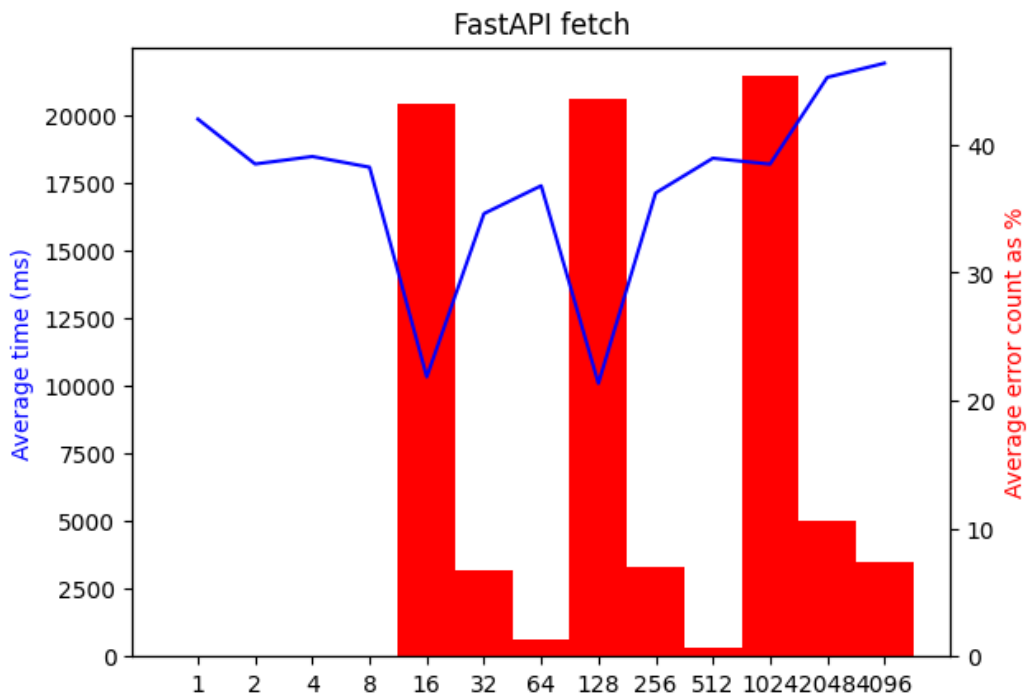
Kuva 21. Flask-kehiksen pyyntöjen käsittelyaika ja virheiden määrä limittäisten pyyntöjen määrää kohti.



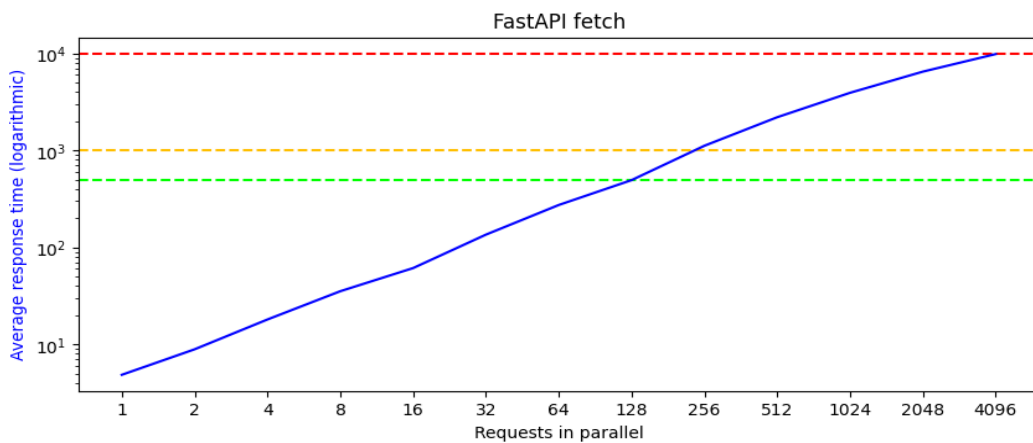
Kuva 22. Flask-kehiksen vasteaika suhteessa limittäisiin pyyntöihin.

Flask-kehiksen tuloksista kuvissa 21 ja 22 voidaan huomata, että kuten muillakin kehiksellä, yksittäisiin pyyntöihin vastaaminen oli hitainta. Keskimääräinen vasteaika yhteen pyyntöön oli noin 331 ms välillä 2–512 limittäistä pyyntöä. Samalla välillä voidaan huomata ympäristön vastaavan suurin piirtein yhtä tehokkaasti pyyntöihin käsittelyajalla noin 12 565 ms.

5.1.3 FastAPI



Kuva 23. FastAPI-kehiksen pyyntöjen käsittelyaika ja virheiden määrä limittäisten pyyntöjen määrää kohti.



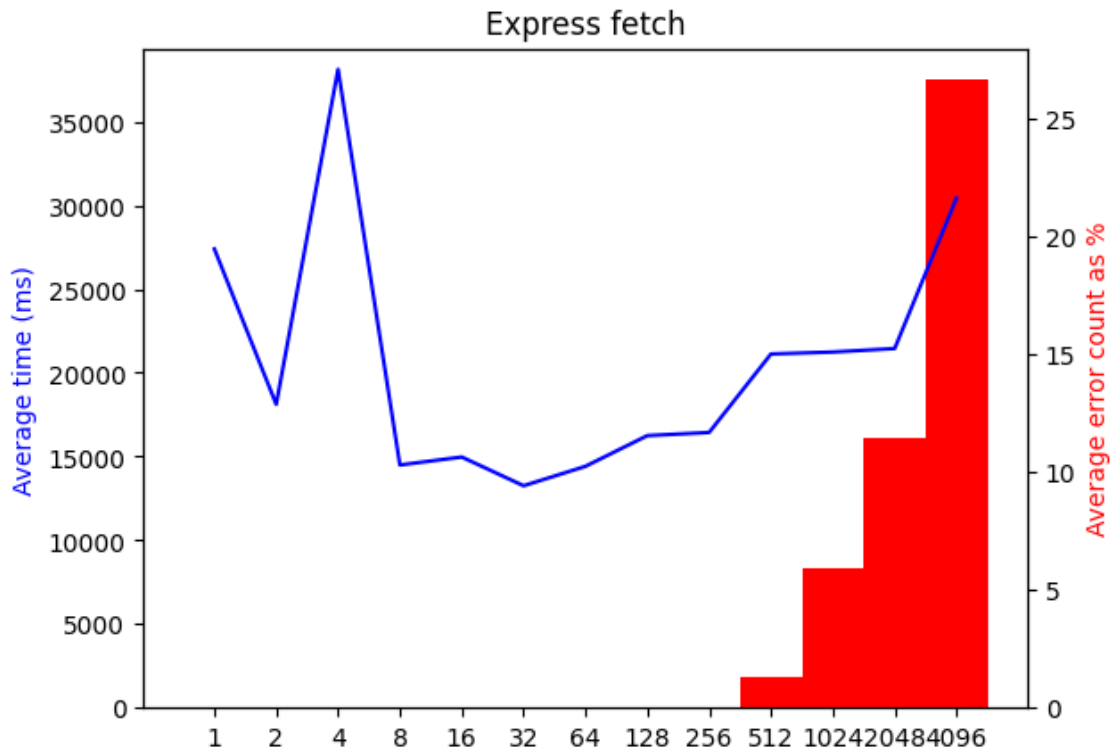
Kuva 24. FastAPI-kehiksen vasteaika suhteessa limittäisiin pyyntöihin.

FastAPI-kehiksen keskimääräinen vasteaika yhteen pyyntöön oli 482 ms. Vastaavasti tämän rajapinnan keskimääräinen käsittelyaika oli 16 042 ms. Täytyy kuitenkin ottaa huomioon, että kuten kuvasta 23 näkyikin, ei tämä aika ole tasainen. FastAPI-kehiksen suorituksessa on huomattavissa epätasaisuutta, sillä tietyillä pyyntömäärillä huomattiin runsaasti virheitä melkein puolessa kaikista pyynnöistä. Mainitut virheet laskevatkin annettua keskimääräistä aikaa, sillä virheellisillä määrillä on huomattavissa huomattavasti no-

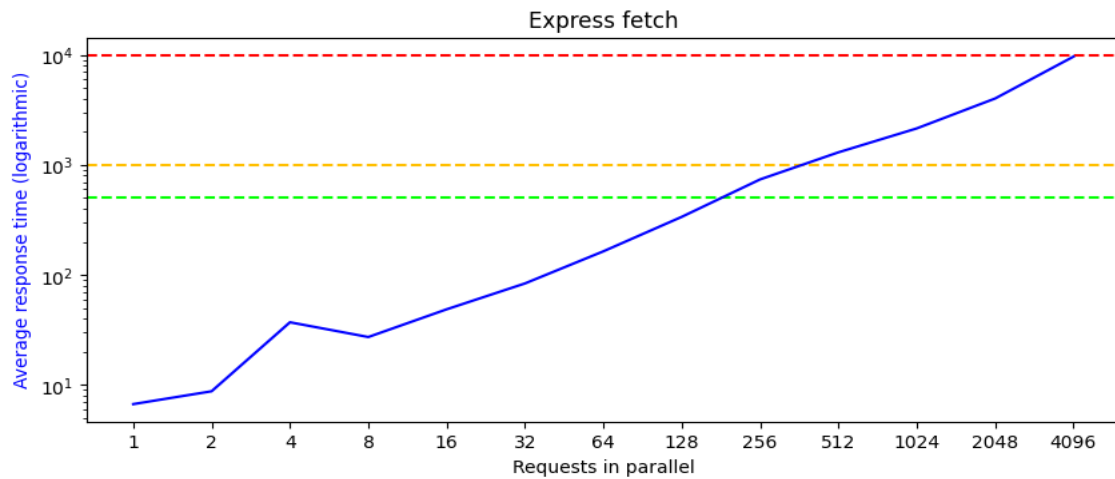
peammat suoritusajat suhteessa kehyksen normaalisti toimiviin suoritusaikoihin. Mielenkiintoisesti voidaan kuitenkin huomata, että vaikka keskimääräinen käsittelyaika laskee virheiden sattuessa, se ei ole visuaalisesti näin vahvasti huomattavissa kuvan 24 vasteajan tuloksissa.

Kehyksen virheiden syytä ei tässä kontekstissa selvitetty, mutta voidaan epäillä, että FastAPI käsittelee pyyntöjen sammuttamisen erilaisella tavalla kuin muut kehykset, mutta tämä on vain arvailua ilman syyn tarkempaa selvitystä.

5.1.4 Express



Kuva 25. Express-ohjelmistokehyksen pyyntöjen käsittelyaika ja virheiden määrä limittäisten pyyntöjen määrää kohti.

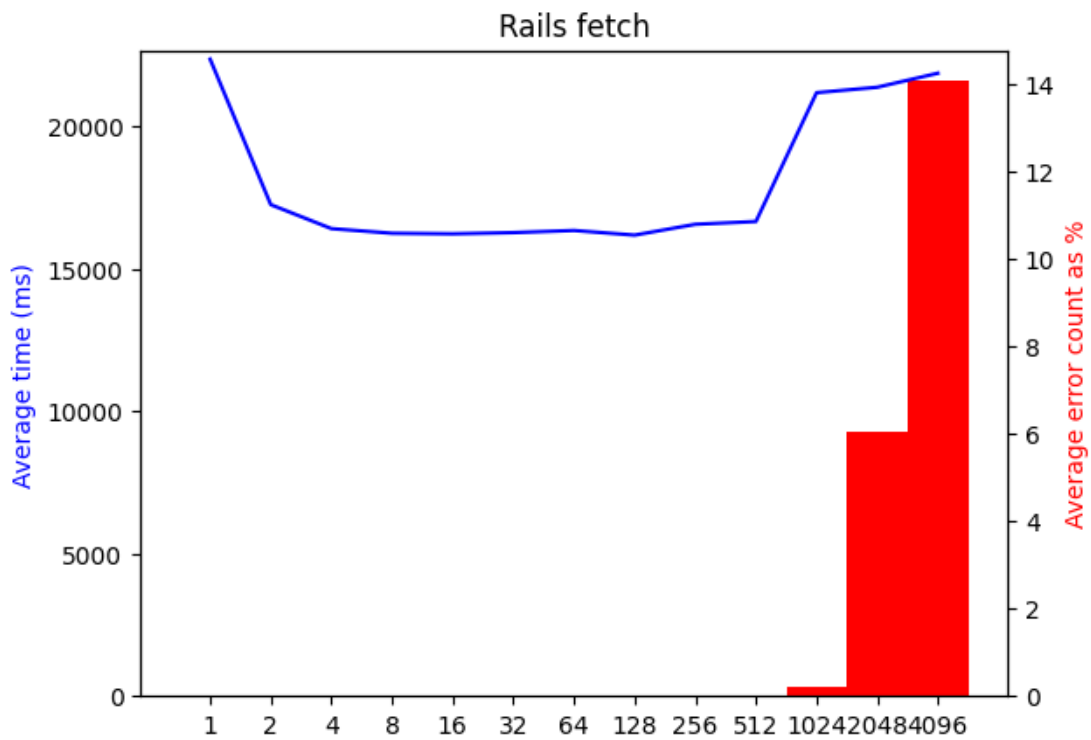


Kuva 26. Express-ohjelmistokehyksen vasteaika suhteessa limittäisiin pyyntöihin.

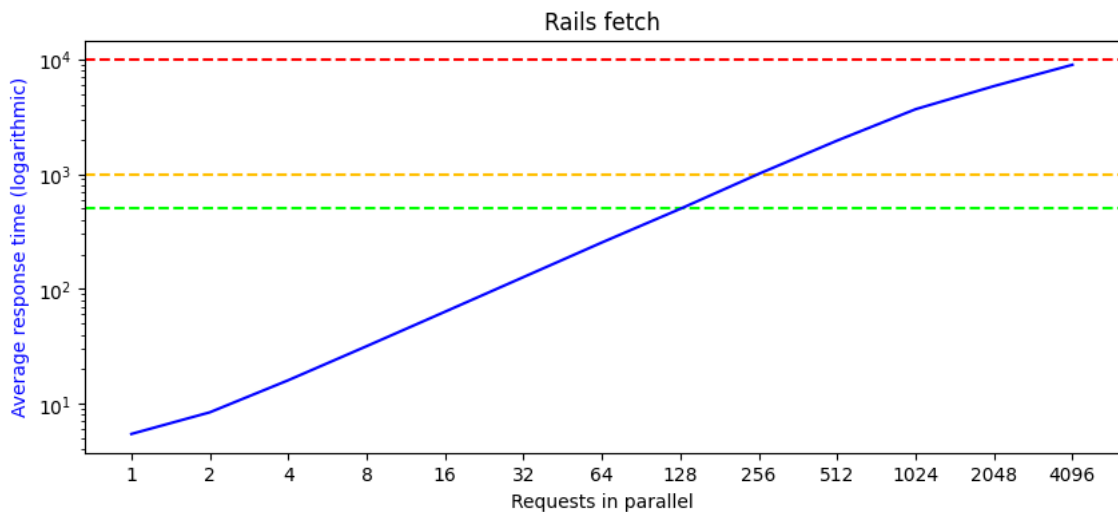
Express-kehysten tuloksista kuvissa 25 ja 26 nähdään niiden olevan kohtuullisen lähellä toisten verkkokehysten tuloksia, käsittelyajan ollessa 18 579 ms ja vasteajan ollessa 305 ms välillä 2–512. On kuitenkin huomattavaa, että neljällä rinnakkaisella pyynnöllä Express-kehys oli hitaampi kuin muilla pyyntömäärillä, sekä vasteajalla että käsittelyajalla. Tämän tuloksen varmentamisen vuoksi Express-kehysten testit ajettiin uudelleen, saaden samankaltaisia tuloksia, jotka on tästä jätetty pois. On erityisen huomattavaa, että hitaampi käsittelyaika tapahtui samalla määrällä kuin mitä järjestelmällä oli prosessorin ytimiä käytettävissä, joka saattaa mahdollisesti antaa viitteitä hidastumisen syystä.

Kuvasta 18 voidaan huomata mielenkiintoisesti, että alkuvaiheessa Express toimii hitaammin kuin Rails, FastAPI, sekä Flask, erityisesti neljän limittäisen pyynnön kohdalla. Huomattavasti kuitenkin tämän jälkeen vasteaika vaikuttaa laskevan samanlaiselle tasolle kuin nopeimmat kuvan kehukset.

5.1.5 Ruby on Rails



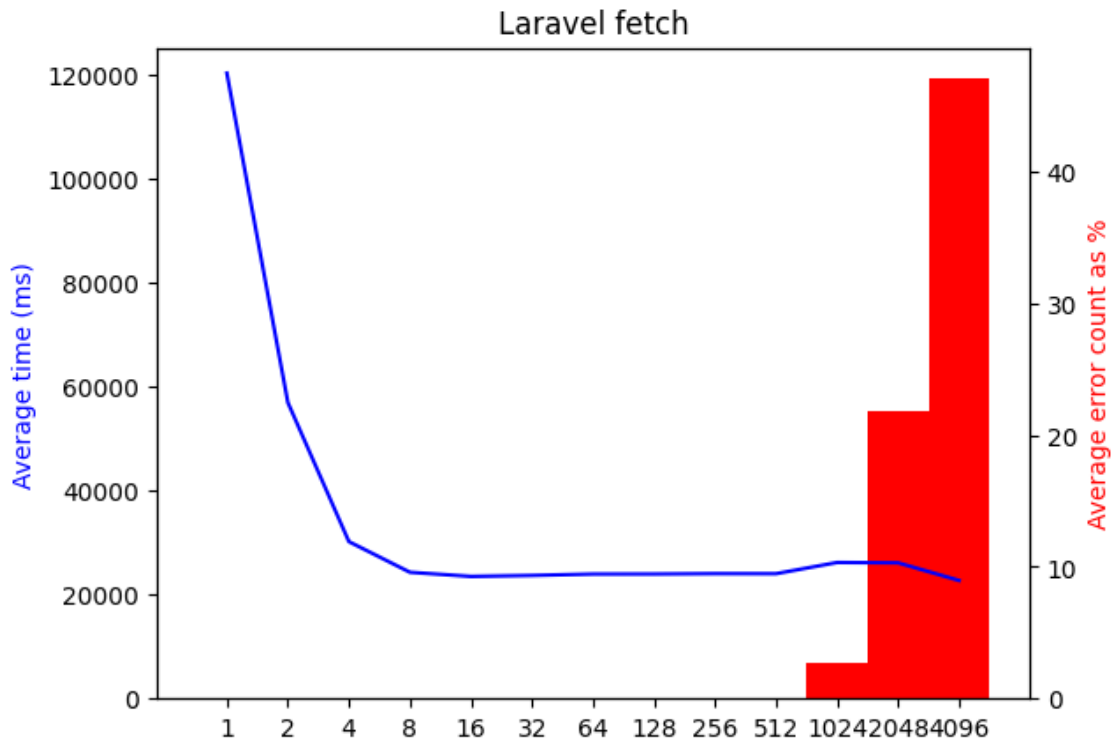
Kuva 27. Ruby on Rails -kehiksen pyyntöjen käsittelyaika ja virheiden määrä limittäisten pyyntöjen määrää kohti.



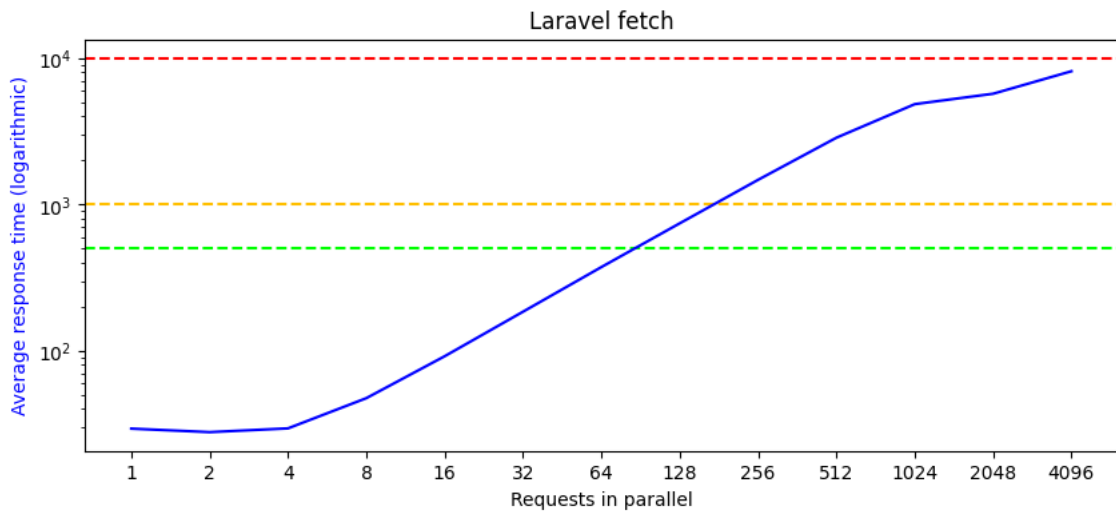
Kuva 28. Ruby on Rails -kehiksen vasteaika suhteessa limittäisiin pyyntöihin.

Ruby on Rails -kehiksen käsittelyaika oli 16 468 ms, sekä vasteaika oli 440 ms välillä 2–512 limittäistä pyyntöä. Kuten kuvista 27 ja 28 voidaan huomata, pysyvät ajat tasaisina.

5.1.6 Laravel



Kuva 29. Laravel-kehiksen pyyntöjen käsittelyaika ja virheiden määrä limittäisten pyyntöjen määrää kohti.



Kuva 30. Laravel-kehiksen vasteaika suhteessa limittäisiin pyyntöihin.

Laravel-kehiksen käsittelyaika oli 28 303 ms, sekä vasteaika oli 647 ms välillä 2–512 limittäistä pyyntöä. Voidaan huomata, että tavalla, jolla kehystä ajetaan, pystyy se hyödyntämään saatavilla olevia resursseja, kuten näkyy kuvissa 29 ja 30. Kuten monessa aiemmassakin kehiksessä, ei Laravel vaikuttaisi testin puitteissa saavuttavan kriittistä 10 sekunnin vasteajan raja-arvoa.

5.1.7 Tasapainotetut tulokset

Alla olevaan taulukkoon 6 on listattu eri tutkimusten tasapainotetut tulokset. Käsittelyaika ja vasteaika ovat tämän tutkielman testeistä, sekä väri on sitä vihreämpi, mitä parempi normalisoitu tulos on kyseisessä tutkimuksessa. On huomattavaa, että yleisellä tasolla Laravel ja Django ovat muita kehyksiä hitaampia, mutta tämän tutkielman puitteissa Rails vaikuttaisi olevan muiden tutkimusten tuloksia nopeampi.

Kehys	Django	Express	FastAPI	Flask	Laravel	Rails
Käsittelyaika	1	0,5	0,44	0,34	0,77	0,45
Vasteaika	1	0,33	0,53	0,36	0,71	0,48
TechEmpower (käänteinen)	1	0,45	0,23	0,22	0,74	0,75
web-frameworks-benchmark	0,22	0,13	0,08	0,14	1	1

Taulukko 6. Verkkohjelmistokehysten tasapainotetut tulokset.

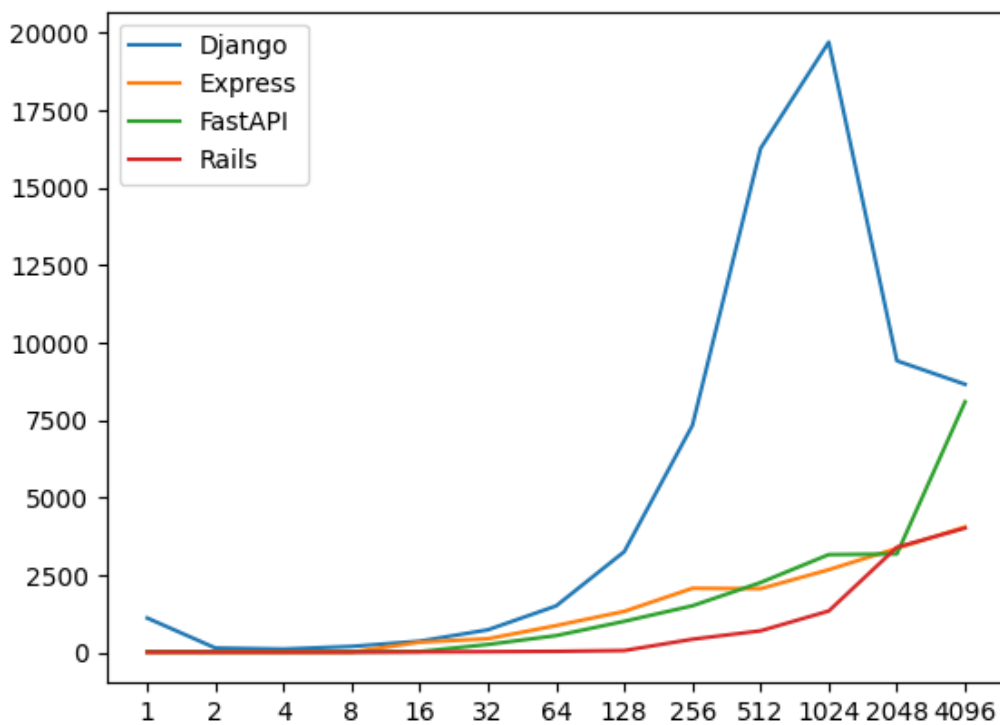
5.2 Päivystävä WebSocket-rajapinta

WebSocket-rajapintatestauksessa on listattu taulukkoon limittäisen välin 1–512 pyyntöjen keskiarvo käsittelyajalle. Tämä väli on valittu johtuen sen pienestä virhemäärästä, jonka jälkeen virheiden määrä alkoi kasvamaan, erityisesti Djangoilla, kuten kuvasta 32 voidaankin nähdä. Saaduissa keskiarvoissa onkin huomioitava se, että jokainen uusi yhteys lisää 20 uutta pyyntöä palvelimelle, sekä olisi oletettavaa, että tämä lisäisi käsittelyaikaa lineaarisesti. Mitatuista kehyksistä voidaankin huomata, että Djangoa lukuun ottamatta kehysten käsittelyaika pysyi kohtuullisen tasaisena.

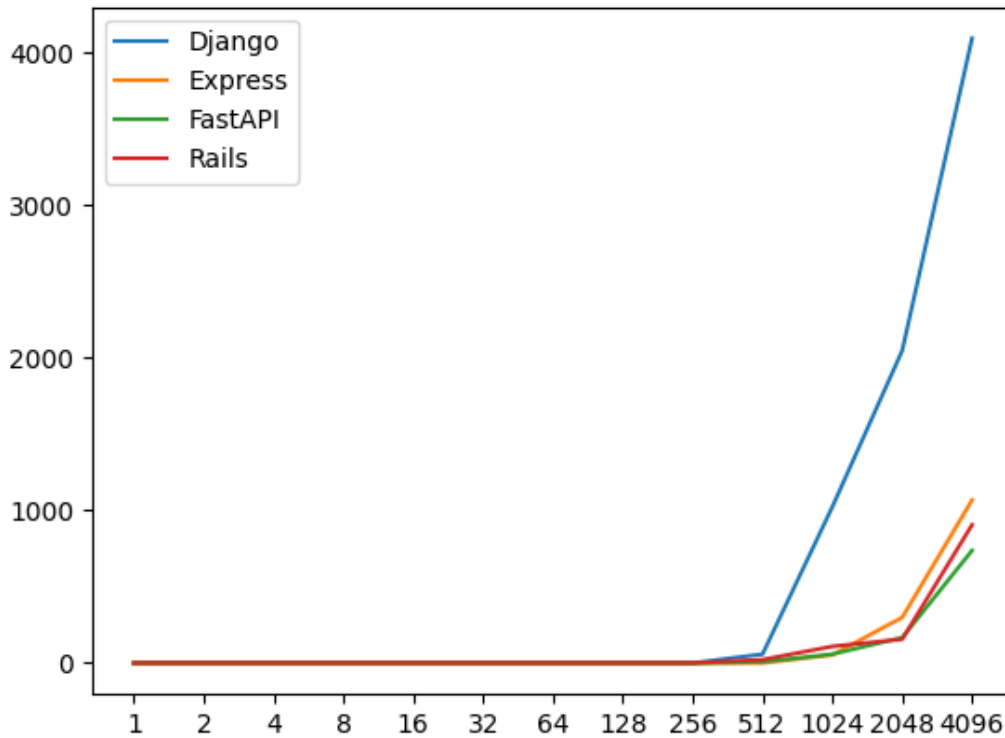
Näiden testien lasketut tulokset löytyvät taulukosta 7, sekä visualisoituina kuviin 31 ja 32, joissa kuvaillaan käsittelyaikaa ja virheiden määrää suhteessa pyyntöjen määrään.

Kehys	Django	Express	FastAPI	Rails
Käsittelyaika (ms)	1.0 3 101	0.23 713	0.18 568	0.04 132

Taulukko 7. Keskimääräinen käsittelyaika 1-512 limittäiselle WebSocket-yhteydelle. Pienempi on parempi.



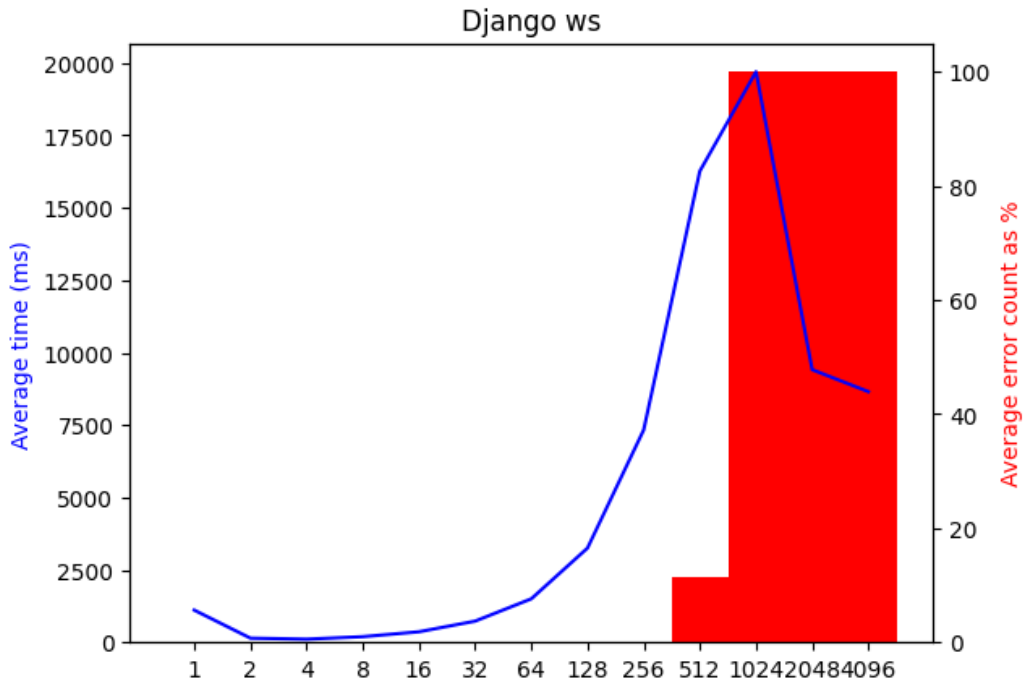
Kuva 31. Kehysten WebSocket-pyyntöjen käsittelyaika suhteessa limittäisten yhteyksien määrään.



Kuva 32. Kehysten virhemäärä suhteessa limittäisiin WebSocket-yhteyksiin.

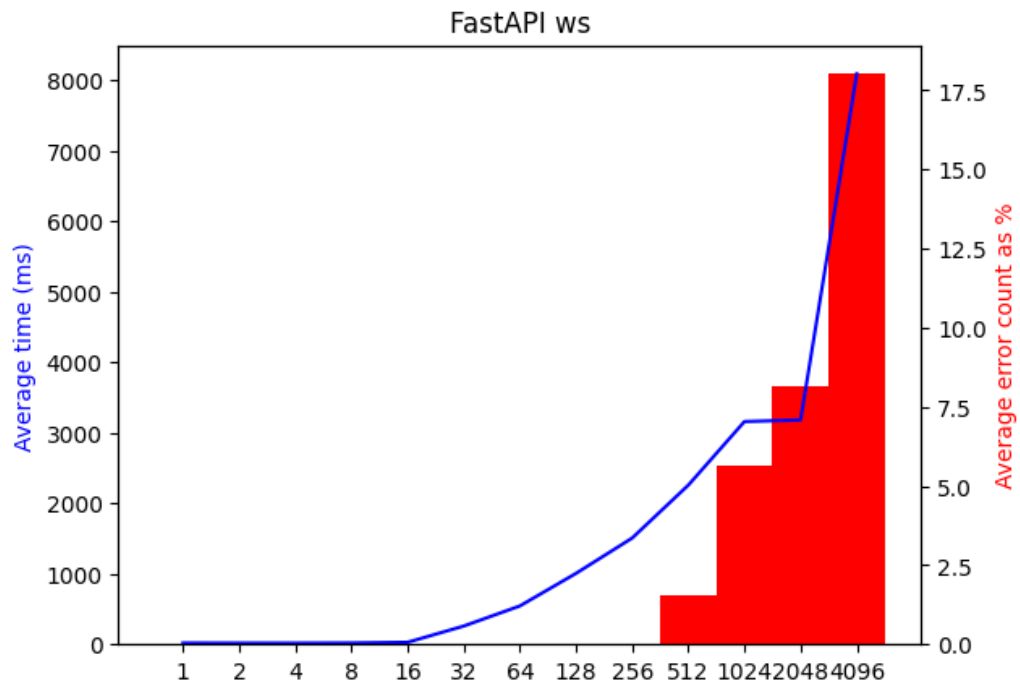
Kuvista 33, 34, 35, sekä 36 voidaan nähdä näiden kehysten WebSocket-käsittelyajat ja virheet suhteessa pyyntöjen määrään. Näistä tuloksista voidaan huomata, että kaikilla rajapinnoilla alkoi tulemaan ongelmia 512 limittäisen yhteyden kohdalla Express-kehystä lukuun ottamatta, joka toimi virheettömästi 1 024:n pyynnön rajaan asti. Muita kehyksiä heikompana voidaan kuitenkin nähdä Djangon suoritus, jossa sen käsittelyaika oli moninkertainen verrattuna muihin ohjelmistokehyksiin.

5.2.1 Django



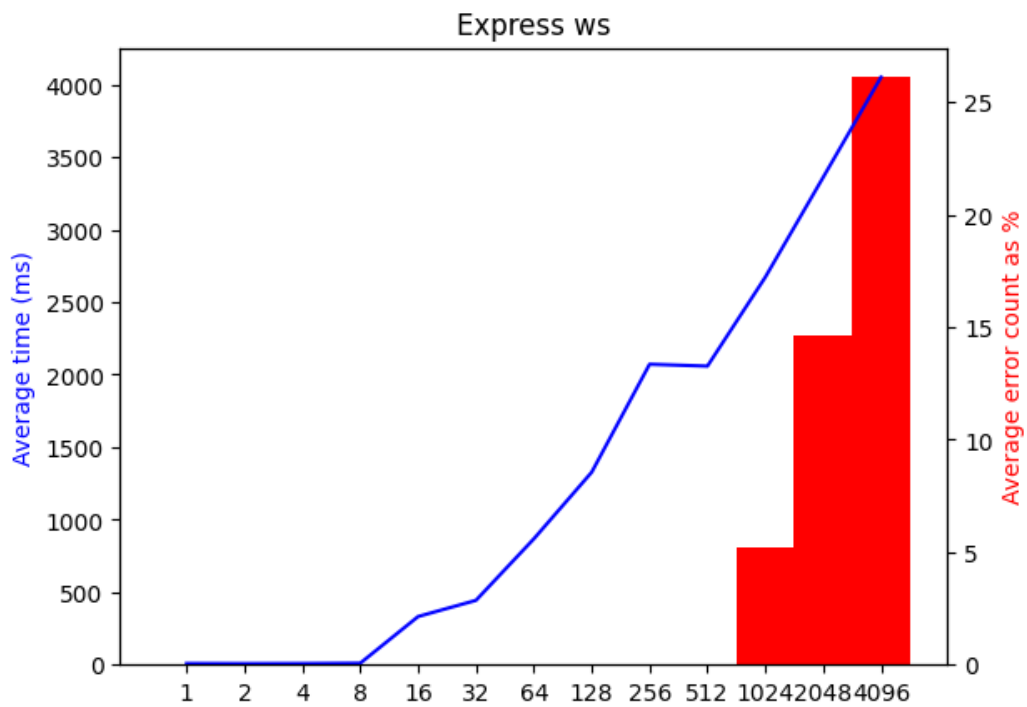
Kuva 33. Django-kehiksen keskimääräinen käsittelyaika suhteessa virheiden määrään.

5.2.2 FastAPI



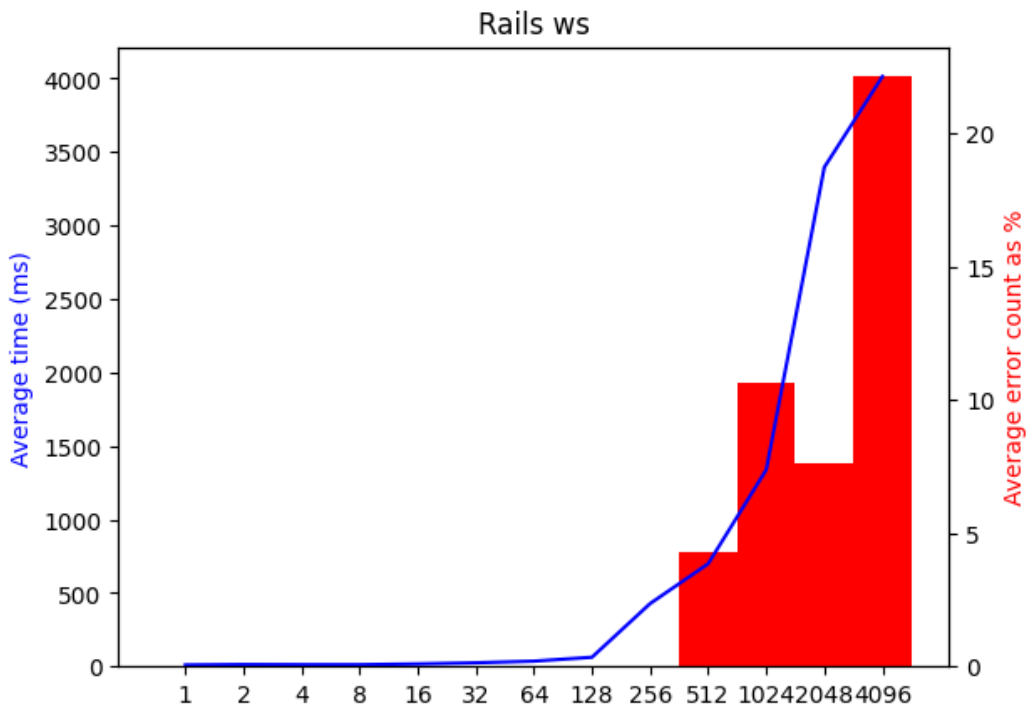
Kuva 34. FastAPI-kehiksen keskimääräinen käsittelyaika suhteessa virheiden määrään.

5.2.3 Express



Kuva 35. Express-ohjelmistokehityksen keskimääräinen käsittelyaika suhteessa virheiden määrään.

5.2.4 Rails



Kuva 36. Rails-kehityksen keskimääräinen käsittelyaika suhteessa virheiden määrään.

6 Johtopäätökset ja jatkokehitys

Tämän tutkielman tarkoituksena oli löytää tietoa siitä, miten eri verkkorajapintakehykset toimivat eri rasiusten alla, sekä kuinka tehokkaasti ne vastaavat asiakaskoneen pyyntöihin. Tutkimusaiheen laajuuden vuoksi sitä oli rajattu runsaasti, joten tutkimuksista saatavia tuloksia tulee käyttää harkiten, sekä reaali maailman sovelluksia varten tulisikin tutustua tämän tutkielman puutteisiin. Tutkielman rajauksiin kuuluikin käytetty testausarkkitehtuuri, testien ajotapa aikarajoituksineen, yhteyksien pakottaminen kiinni, testien välissä tapahtuva palvelimen uudelleenkäynnistys, valitut teknologiat, asiakasohjelmiston tyyppi, sekä valitut testattavat kehykset.

Tuloksia tarkastellessa on myös huomioitava, että tutkielmassa ei ole tallennettu rajapintojen virheellisten tulosten syitä. Tämän vuoksi oikeaa syytä virheille ei ole mahdollista selvittää tuloksista, sekä ne syyt voivat vaihdella käyttöjärjestelmän rajoituksista tietokannan ruuhkautumiseen.

Kuten Shaw [2000] mainitseekin, tehokas kone ei välttämättä varmista nopeaa vastausta, vaan on tärkeämpää tarkastella toteutetun järjestelmän rakennetta. Edellistä jatkaen osia tähän ovatkin kehitystapa, arkkitehtuuri, käyttötapa, sekä käyttöympäristö [Shaw, 2000], joiden voisi ajatella olevan testattujen kehysten osia.

6.1 Johtopäätökset

Vaikka mitattuja tuloksia voidaankin käyttää viitearvoina jotain verkkokehystä valittaessa, palvelinarkkitehtuurin rakentamista ei suositella tehtävän tämän tutkielman tulosten pohjalta johtuen siitä, että mitatut tulokset ovat riippuvaisia jo itsessään käytetystä palvelimesta. Näin onkin suositeltavampaa tarkastella mitattuja eroja rajapintojen välillä vain pinnallisesti, sekä löytää hyviä testausrajoja.

Tuloksia tarkastellessa voidaankin huomata, että monessa kehyksessä WebSocket-pyyntöt toimivat sulavasti valitulla järjestelmäkokoopanolla 8:aan tai 16:een asiakkaaseen asti, jonka jälkeen viive alkaa näennäisesti lisääntymään. Kohtuullisen pienellä virhemäärällä päästään kuitenkin noin 512 limittäiseen yhteyteen asti, ennen kuin virheiden määrä kasvaa suureksi. Toisin kuin Soewiton ja muiden [2019] tutkimuksessa, tässä tutkimuksessa ei tarkasteltu WebSocket-rajapinnasta saatavia vasteaikoja, joiden he suosittelivat pysyvän 300 ms alapuolella.

Vastaavasti RESTful-aplikaatioilla järjestelmä vaikuttaa toimivan valtaosin sulavasti 128 limittäiseen pyyntöön asti, jonka rajoille testaus voisi olla hyvä järjestelmien vertailussa sijoittaa. On kuitenkin huomioitavaa, että tässä täytyy ottaa mukaan huomioon myös palvelimen tyyppi ja konteksti, sillä esimerkiksi ripeästi päivittyvän sosiaalisen median rajapinta on hyvin erilaisen rasituksen ja liikenteen alla verrattuna esimerkiksi mahdolliseen kotona toimivaan IoT-laitteeseen. Tällä perusteella tämän tutkielman tulokset eivät suoranaisesti ole yleisesti käytettävissä, vaan on hyödyllisempää tehdä päätökset vastaavien kohdistettujen jatkotutkimusten perusteella.

Vastaavasti verrattaessa aiempia tutkimuksia ja tämän tutkielman tuloksia keskenään, voidaan huomata tulosten vaihtelevan kohtuullisesti. Tästä voidaan päätellä mahdollisesti sitä, että joissakin tutkimuksissa mahdollinen kehyksen ajotapa tai ohjelmointityyli voivat vaikuttaa tuloksiin, jota tulisi tarkastella. Esimerkiksi Python-kieleen pohjautuvilla kehyksillä on huomattavaa, että niillä on useita erilaisia kirjastoja, joilla niitä voidaan ajaa, kuten esimerkiksi *uvicorn*, *gunicorn*, ja *daphne*. Tulosten ollessa jonkin verran ristiriitaisia ei voidakaan sanoa, että tämä tutkielma varsinaisesti todistaisi jonkin toisen tutkimuksen tuloksia, näin ollen yksi hiukan ristiriitainen tulos muiden joukossa, vaikkakin tuloksista voi mahdollisesti huomata joitain suuntauksia. Keräämällä suurempi määrä tutkimuksia ja vertaamalla näiden tuloksia keskenään, voisi olla mahdollista nähdä yleisiä ja tarkempia suuntauksia kehyksien tehokkuuksista.

REST-tuloksista voidaan huomata myös se, että Rails, Laravel, sekä Django ovat muita tämän tutkielman kehyksiä hitaampia. Tämä voi osaltaan johtua siitä, että ne ovat mielpiteellisiä täysiä pinokehyksiä, jotka sisältävä muita runsaammin toiminnallisuuksia ja osia. Jotta todellisen kokonaiskuvan saanti olisi mahdollista, tulisi tarkastella miten näiden ominaisuuksia toteutus vaikuttaisi näiden kevyempien kehysten tehokkuuteen, mikäli niitä haluttaisiin toteuttaa. Kevyemmällä kirjastoilla ohjelmoija voi itse päättää mitä haluaa toteuttaa, kun taas näillä täysillä pinokehyksillä ohjelmoija joutuu todennäköisemmin estämään epähaluttuja ominaisuuksia, joita tässä testauksessa ei suoranaisesti paljoa estetty. Mainitut raskaammat kehykset ovatkin luotu täysiksi pinokehyksiksi, joiden voidaan väittää huolehtivan käytännössä lähes kaikista verkkorajapintoihin liittyvistä käsittelyn osa-alueista.

Eri REST-vertailujen kesken sekä FastAPI, että Flask vaikuttavat antavan keskimäärin nopeimpia tuloksia, sekä täyden pinokehyksen tarjoavista toteutuksista Ruby on Rails vaikuttaisi olevan tehokkain. Kaikilta osa-alueiltaan heikoimmaksi jää Django, antaen runsaasti virheitä tämän tutkielman puitteissa. Vastaavasti heikoimmaksi kehykseksi epä-

täysistä pinokehyksistä tarkastelluissa tutkimuksissa vaikuttaisi jäävän Express. On kuitenkin huomattavaa, että tämän tutkielman omien tuloksien perusteella Expressillä oli keskimäärin paras vasteaika, vaikka käsittelyaika itsessään olikin Flask- ja FastAPI-kehyskiä hitaampi. Tämän vuoksi olisikin tulevaisuudessa hyvä tarkastella näiden vastaajien jakaumaa, jotta voitaisiin saada kokonaislaajuisempi kuva.

Mitattuja tuloksia olisikin hyvä verrata aiemmin mainittuihin Nielsenin [2010] aikarajoihin tuloksia hyödyntäessä; toisin sanottuna vaikka käsittelyaika olisi nopea, tulisi vasteaika olla mieluiten alle sekunti, mutta enintään alle 10 sekuntia. Parhaimmat limittäisten yhteyksien raja-arvot tulisivat tutkia laitteistokohtaisesti testaamalla erilaisia rajapintoja vasten. Onkin huomioitava, että todellisuudessa tietyissä konteksteissa mahdollisen käyttäjän näkemä aika on todennäköisesti suurempi, johtuen esimerkiksi mahdollisesta verkkosivun rakenteen päivittämisestä.

Toisin kuin aiemmissa mittauksissa, WebSocket-tuloksissa Rails on nopein tämän tutkielman mittauksista, jota seuraavana on FastAPI yli nelinkertaisella ajalla. Kuten REST-mittauksissa, tässäkin mittauksessa epätäyisistä kehyksistä FastAPI toimi nopeimmin.

Kuten tuloksia lukiessa voidaankin mahdollisesti tulkita, voi tehokkaimman kehyksen valinta olla haastavaa; tämän vuoksi voikin olla parempi tarkastella tulevaisuudessa tätä käsitettä käänteisesti: mitkä ovat tehottomimpia verkko-ohjelmistokehyksiä?

6.2 Puutteet ja kehitysideat

Olenaisiin puute voitaisiin sanoa olevan tutkielman laatijan puuttuva syvälinen tietämys testatuista ohjelmistokehyksistä; ilman syvempää tietoa ja kokemusta on hyvinkin mahdollista, että jokin kehys olisi voinut toimia mitattuja tuloksia tehokkaammin. Tässä on erityisen huomioonotettavaa, että kehyksiä testattiin ilman niiden mahdollisesti tarjoamia tehostustapoja, joilla voi olla hyvinkin suuria vaikutuksia eri kehysten nopeuteen.

Toinen iso puute tutkielmassa oli palvelinjärjestelmän kokoonpano, joka ei itsessään välttämättä vastaa monia moderneja palvelinkehityksissä käytettyjä koneita. On mahdollista, että modernimmat koneet sallivat helpommin prosessorin usean ytimen hyödyntämisen, näin parantaen joidenkin kehysten toimintatehokkuutta, mahdollisesti moninkertaisten. Toisaalta on myös otettava huomioon se, että palvelimia ei ole vain yhdenlaisia, sekä jokaisella palvelimella on eri kyvyt vastata pyyntöihin. Tulisikin ottaa huomioon myös erilaisten palvelinten skaala, joka voi vaihdella pienistä IoT-laitteista suurempiin datahallien laitteistoihin.

Kolmas puute tässä tutkielmassa on oikeanlaisen palvelinarkkitehtuurin puute; monessa laajemmin käytetyssä järjestelmässä käyttäjillä ei ole suoraa pääsyä palvelinohjelmiston tarjoamaan rajapintaan, vaan liikenne reititetään monesti joidenkin reitittimien ja palomuurien lävitse, joista esimerkkinä voidaan antaa esimerkiksi Nginx ja Apache.

Neljäntenä puutteena on huomioitu tutkielmassa poisjätetty osa-alue, jossa olisi voitu tarkastella palvelinjärjestelmän omien resurssien käyttöä. Mikäli jokin kehys olisi käyttänyt samalla palvelutasolla huomattavasti vähemmän resursseja kuin toinen kehys, saattaisi tässä tutkimuksessa raskaalla liikenteellä hitaammalta vaikuttavan kehysten saada toimimaan toista tehokkaammin jakamalla se useammalle pienemmälle palvelimelle. Näitä klustereita rinnakkain ajamalla voisi olla mahdollista sallia huomattavasti suurempi määrä asiakasyhteyksiä, vaikka yhden asiakkaan käsittely veisikin enemmän aikaa.

Viidentenä puutteena huomioidaan jo alussa pois jätetty eri tulkkien vertailu. On hyvinkin mahdollista, että jokin tulkki voisi toimia testissä käytettyjä virallisia tulkkeja tehokkaammin palvelinympäristössä. Näistä esimerkkinä aiemmin mainittu PyPy-tulkki, joka mahdollistaisi JIT-kääntämisen, jota normaalissa CPython-tulkissa ei ole.

Kuudentena puutteena voidaan huomata pieni ajettujen testien määrä, jota tutkielman aikarajoitteiden vuoksi ei ollut mahdollista kasvattaa suuremmaksi. Tämän puutteen vuoksi onkin vaikea luottaa keskimääräisiin käsittelyaikoihin. Tämän lisäksi jatkotutkimuksissa voisi olla hyvä tallettaa kaikki vasteajat talteen, jotta voitaisiin tarkastella mitattujen vasteaikojen vakautta ja jakaumaa. Edellisestä voidaankin sanoa se, että vaikka rajapinta vaikuttaisi taulukoissa olevan ripeästi päivittyvä, riittää siihen se, että puolet päivittyy ripeästi, kun taas puolet liian hitaasti, näin antaen tasa-arvossa näiden välin, joka olisi juuri ja juuri ripeästi päivittyvä. Vastaavasti keskiarvoissa voi käydä myös niin, että murto-osa erittäin hitaista vastauksista saisi suuremman määrän nopeita vastauksia vaikuttamaan hietaalta lopullisissa tuloksissa.

Puutelistauksesta voidaan hyvin huomata, että tutkielman aihe ei ole niin yksikäsitteinen kuin voisi ensivilkaisemalta olettaa. Mikäli laajempaa testausta halutaan suorittaa, tulee olla varautunut erittäin laajaan tutkimukseen, jossa jo itsessään testien läpiajo voi viedä useampia vuorokausia tai viikkoja riippuen valitun tutkimuksen laajuudesta. Vertauskuvana tämän tutkielman onnistuneiden mittauksien suoritusajaksi voidaan mainita noin viikko, ottaen huomioon, että se sisälsi runsaasti manuaalista käsittelyä testien ajamiseksi. Automaatiolla tämän tutkielman testit olisi mahdollista lyhentää arviolta noin vuorokautteen, joka sallisi suuremman datamäärän keräämisen samalla aikavälillä.

7 Yhteenveto

Tutkielman alussa tutustuttiin viime vuosina runsaasti kehittyneisiin verkkoteknologioiden osa-alueisiin, sekä mainittiin tutkielman merkitevyys sekä ympäristön että palvelin-kontekstin mahdollisten puutteellisten resurssien puitteissa. Tämän jälkeen tarkasteltiin joitain dynaamisesti tyypitettyjä kieliä, sekä niillä yleisesti käytettyjä verkkorajapintakehyksiä, jotka olivat kirjoitushetkellä suosittuja joissain palvelinympäristöissä. Jokaisesta näistä annettiin lyhyehkö kuvaus, sekä pieni pala koodia, joilla voi saada esimakua käytettävästä kielestä ja kehyksestä.

Tämän jälkeen tarkasteltiin joitain aiemmin tehtyjä tutkimuksia ja niistä saatuja tuloksia, sekä tarkasteltiin, kuinka niitä voitaisiin tässä tutkielmassa hyödyntää. Joitain aiempia tutkimuksia huomioon ottaen luotiin pohja sille, miten testataan ja mitä dataa testauksessa tullaan käyttämään, sekä kuinka lopulliset tulokset tullaan mittaamaan.

Lopullisissa mitatuissa tuloksissa tarkasteltiin käyttäjän näkemää pyynnön käsittelyaikaa sekä REST:n että WebSocketin konteksteissa, joista molemmista saatiin ulos hyödyllisiä raja-arvoja, joihin testausta voidaan kohdistaa tulevissa järjestelmissä, mikäli ne ovat samankaltaisia kuin testauksessa käytetty järjestelmä. Testauksessa huomattiin, että suuria aikaeroja oli kehysten ja täysien pinokehysten välillä, joiden tulokset tulisi asetella erilleen. Täysistä pinokehyksistä yhtä tutkimusta lukuun ottamatta Rails vaikuttaisi olevan nopein REST-kontekstissa, kun taas muista kehyksistä nopein olisi joko FastAPI tai Flask muut tutkimukset huomioon ottaen. WebSocketia käytettäessä vastaavasti mitatuista kehyksistä nopeimmat olivat Rails ja FastAPI, jossa Rails oli täytenä pinokehystenä muitakin nopeampi. Testausympäristössä REST-käytössä kehykset kestivät Djangoa lukuun ottamatta 512 limittäiseen pyyntöön asti, kun taas WebSocketia käytettäessä kohtuullisen virheettömästi 512 yhtäaikaiseen yhteyteen asti.

Kuten puutteissa mainittiinkin, tässä tutkielmassa oli myös runsaasti puutteita sen laajuuden vuoksi; jatkotutkimuksia varten onkin otettava huomioon jokin rajatumpi osa-alue tai mahdollisesti kehitettävä palvelu, jotta tarkennettua testausta voitaisiin suorittaa, saaden enemmän reaali maailmassa hyödynnettäviä arvoja. Käytännössä tätä tutkielmaa voidaan siis käyttää tiivistämään tulevia tutkimuksia, näin tarjoten yleisen pohjan teknologioiden ja testien suorittamiseen.

8 Viiteluettelo

- Lucas R. Abbade, Mauro A. A. da Cruz, Joel J. P. C. Rodrigues, Pascal Lorenz, Ricardo A. L. Rabelo, and Jalal Al-Muhtadi. 2020. Performance comparison of programming languages for Internet of Things middleware. *Transactions on Emerging Telecommunications Technologies* 31(12), e3891. DOI: 10.1002/ett.3891
- Olayinka Adeleye, Jian Yu, Guiling Wang and Sira Yongchareon. 2021. Constructing and Evaluating Evolving Web-API Networks - A Complex Network Perspective. *IEEE Transactions on Services Computing*. 16(1), 177–190. DOI: 10.1109/tsc.2021.3114709
- Eric L. Barnes. 2016. Laravel Release Process. *Laravel News*. Retrieved 2023-04-04 from <https://laravel-news.com/laravel-release-process>
- Mathieu Carbou. 2011. Introduction to Comet. Retrieved 2023-03-22 from <https://web.archive.org/web/20201203115253/http://www.ibm.com/developerworks/web/library/wa-reverseajax1/index.html>
- Django. 2022a. FAQ: General. Retrieved 2023-03-25 from <https://docs.djangoproject.com/en/4.1/faq/general/#why-does-this-project-exist>
- Django. 2022b. How to deploy with WSGI. Retrieved 2023-03-25 from <https://docs.djangoproject.com/en/4.1/howto/deployment/wsgi/>
- Django. 2022c. How to deploy with ASGI. Retrieved 2023-03-25 from <https://docs.djangoproject.com/en/4.1/howto/deployment/asgi/>
- Phillip J. Eby. 2022. PEP 3333 - Python Web Server Gateway Interface v1.0.1. Retrieved 2023-03-25 from <https://peps.python.org/pep-3333/>
- Roy Fielding. 2000. *Architectural Styles and the Design of Network-based Software Architectures*. Ph.D. Dissertation. University of California, Irvine, CA. Retrieved undefined from https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation_2up.pdf
- GitHub. 2023a. pallets/flask. Retrieved 2023-03-25 from <https://github.com/pallets/flask/releases/tag/0.1>

- GitHub. 2023b. tiangolo/fastapi. Retrieved 2023-03-25 from <https://github.com/tiangolo/fastapi/tags?after>
- David Heinemeier Hansson. 2023a. David Heinemeier Hansson. Retrieved 2023-03-25 from <https://dhh.dk/>
- David Heinemeier Hansson. 2023b. Rails 7.0.4.3. Retrieved 2023-03-25 from <https://rubygems.org/gems/rails>
- David Heinemeier Hansson. 2023c. Threading and Code Execution in Rails. Retrieved 2023-03-25 from https://guides.rubyonrails.org/threading_and_code_execution.html
- TJ Holowaychuk. 2010. Express 1.0beta. Retrieved 2023-03-25 from <https://web.archive.org/web/20150706050636/https://tjholowaychuk.tumblr.com/post/820103177/express-1-0beta>
- Paul Krill. 2016. Node.js Foundation to shepherd Express Web framework. Retrieved 2023-03-25 from <https://www.infoworld.com/article/3031686/nodejs-foundation-to-shepherd-express-web-framework.html>
- Paul Krill. 2020. PHP 8.0 arrives with union types, JIT compilation. Retrieved 2023-04-21 from <https://www.infoworld.com/article/3599161/php-80-arrives-with-union-types-jit-compilation.html>
- Laravel LLC. 2023. The PHP Framework for Web Artisans. Retrieved 2023-04-03 from <https://laravel.com/>
- Wojciech Łasocha and Marcin Badurowicz. 2021. Comparison of WebSocket and HTTP protocol performance. *Journal of Computer Sciences Institute* 19, 67–74. DOI: 10.35784/jcsi.2452
- Ross McIlroy. 2016. Firing up the Ignition interpreter. Retrieved 2023-03-25 from <https://v8.dev/blog/ignition-interpreter>
- Alexey Melnikov and Ian Fette. 2011. The WebSocket Protocol. *RFC 6455*. 1–71. DOI: 10.17487/RFC6455

- Mozilla Corporation. 2023. JavaScript. Retrieved 2023-04-03 from <https://developer.mozilla.org/en-US/docs/Web/javascript>
- Netscape. 1995. Netscape and Sun announce JavaScript, the open, cross-platform object scripting language for enterprise networks and the internet. Retrieved 2023-03-25 from <https://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/news-release67.html>
- Jakob Nielsen. 2010. Website Response Times. Retrieved 2023-03-25 from <https://www.nngroup.com/articles/website-response-times/>
- OpenJS Foundation. 2023a. What is Node-API? Retrieved 2023-04-03 from <https://nodejs.github.io/node-addon-examples/about/what/>
- OpenJS Foundation. 2023b. About Node.js®. Retrieved 2023-04-03 from <https://nodejs.org/en/about>
- Packagist. 2023. laravel/framework statistics. Retrieved 2023-03-25 from <https://packagist.org/packages/laravel/framework/stats>
- Pallets Projects. 2023a. Jinja. Retrieved 2023-03-25 from <https://palletsprojects.com/p/jinja/>
- Pallets Projects. 2023b. Welcome to the Pallets Projects. Retrieved 2023-03-25 from <https://palletsprojects.com/>
- Pallets Projects. 2023c. Design Decisions in Flask. Retrieved 2023-04-17 from <https://flask.palletsprojects.com/en/2.2.x/design/>
- Victoria Pimentel and Bradford Nickerson. 2012. Communicating and Displaying Real-Time Data with WebSocket. *IEEE Internet Computing* 16(4), 45–53. DOI: 10.1109/mic.2012.64
- John Potter. 2023. express. Retrieved 2023-03-03 from <https://npmtrends.com/express>
- PyPI Stats. 2023a. flask. Retrieved 2023-03-25 from <https://pypistats.org/packages/flask>

- PyPI Stats. 2023b. fastapi. Retrieved 2023-03-25 from <https://pypistats.org/packages/fastapi>
- PyPI Stats. 2023c. django. Retrieved 2023-03-25 from <https://pypistats.org/packages/django>
- Python Software Foundation. 2023. General Python FAQ. Retrieved 2023-03-25 from <https://docs.python.org/3/faq/general.html>
- Sebastián Ramírez. 2023a. FastAPI. Retrieved 2023-03-25 from <https://fastapi.tiangolo.com/>
- Sebastián Ramírez. 2023b. Templates. Retrieved 2023-03-25 from <https://fastapi.tiangolo.com/advanced/templates/?h>
- Armin Ronacher. 2023. Armin Ronacher. Retrieved 2023-03-25 from <http://armin.ronacher.eu/>
- Ruby. 2023a. About Ruby. Retrieved 2023-03-03 from <https://www.ruby-lang.org/en/about/>
- Ruby. 2023b. To Ruby from C and CPP. Retrieved 2023-03-03 from <https://www.ruby-lang.org/en/documentation/ruby-from-other-languages/to-ruby-from-c-and-cpp/>
- Ruby. 2023c. How to use Ruby's JIT compiler. Retrieved 2023-03-03 from <https://bugs.ruby-lang.org/projects/ruby/wiki/MJIT>
- Koichi Sasada. 2005. YARV: yet another RubyVM: innovating the ruby interpreter. In: *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2005, ACM. 158–159. DOI: 10.1145/1094855.1094912
- Jonathan Shaw. 2000. Web application performance testing - a case study of an on-line learning application. *BT technology journal* 18(2), 79–86.
- Benfano Soewito, Christian, Fergyanto E. Gunawan, Diana and I Gede Putra Kusuma. 2019. Websocket to Support Real Time Smart Home Applications. *Procedia Computer Science*. 157, 560-566. DOI: 10.1016/j.procs.2019.09.014

- Stack Overflow. 2022. 2022 Developer Survey. Retrieved 2023-03-25 from <https://survey.stackoverflow.co/2022/#section-most-popular-technologies-web-frameworks-and-technologies>
- StatCounter. 02.2023. Browser Market Share Worldwide. Retrieved 2023-03-25 from <https://gs.statcounter.com/browser-market-share>
- TechEmpower. 2022a. Web Framework Benchmarks. Retrieved 2023-03-25 from <https://www.techempower.com/benchmarks/#section=data-r21&test=composite>
- TechEmpower. 2022b. Web Framework Benchmarks. Retrieved 2023-03-25 from <https://www.techempower.com/benchmarks/#section>
- The PHP Group. 2023a. What is PHP? Retrieved 2023-04-03 from <https://www.php.net/manual/en/intro-whatis.php>
- The PHP Group. 2023b. History of PHP. Retrieved 2023-03-03 from <https://www.php.net/manual/en/history.php.php>
- The PHP Group. 2023c. Your first PHP-enabled page. Retrieved 2023-04-03 from <https://www.php.net/manual/en/tutorial.firstpage.php>
- Matti Turpeinen. 2023. web-api-stresstest. Retrieved 2023-04-20 from <https://github.com/Nuubles/web-api-stresstest>
- V8. 2017. Launching Ignition and TurboFan. Retrieved 2023-04-21 from <https://v8.dev/blog/launching-ignition-and-turbofan>
- Web Frameworks Benchmark. 2023. Web Frameworks Benchmark. Retrieved 2023-03-31 from <https://web-frameworks-benchmark.netlify.app/result>
- Allen Wirfs-Brock and Brendan Eich. 2020. JavaScript: the first 20 years. *Proceedings of the ACM on Programming Languages*. 4, 1–189. DOI: 10.1145/3386327