Lauri Pimiä

Alustariippumattomien tekniikoiden haasteet mobiilikehityksessä

Tietotekniikan pro gradu -tutkielma

22. huhtikuuta 2021

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Lauri Pimiä

Yhteystiedot: lauri.i.pimia@student.jyu.fi

Ohjaaja: Vesa Lappalainen ja Ari Viinikainen

Työn nimi: Alustariippumattomien tekniikoiden haasteet mobiilikehityksessä

Title in English: The challenges of cross-platform technologies in mobile development

Työ: Pro gradu -tutkielma

Suuntautumisvaihtoehto: Ohjelmistotekniikka

Sivumäärä: 112+8

Tiivistelmä: Älypuhelimien jatkuva yleistyminen ohjaa yhä enemmän resursseja erilaisten mobiilisovellusten kehittämiseen. Mobiilisovelluksia kehitetään perinteisesti kohdealustan mukaisilla natiiveilla tekniikoilla, mutta tämä voi aiheuttaa merkittävää epätehokkuutta, jos sovelluksen kohteena on useampi alusta. Tämän epätehokkuuden ratkaisemiseksi on kehitetty niin sanottuja alustariippumattomia tekniikoita, joiden tarkoituksena on mahdollistaa saman koodin hyödyntäminen useammalla kohdealustalla. Eri alustariippumattomia tekniikoita on kuitenkin valtava määrä, jonka lisäksi ne tarjoavat hyvin eri tasoisia lopputuloksia. Tässä tutkielmassa tutkitaan kahden uuden alustariippumattoman tekniikan: Flutterin ja React-Nativen soveltuvuutta yleiseen mobiilikehitykseen. Erityistä tarkkuutta kiinnitetään siihen, kuinka edellä mainitut tekniikat toimivat natiiveja tekniikoita vastaan ja mitä kompromisseja ne vaativat. Tutkimus toteutettiin vertailututkimuksena, johon kuului kokeellisena tutkimuksena eri tekniikoilla tuotettu testisovellus. Tämän testisovelluksen tuloksia ja kirjallisuuskatsauksessa esiin tullutta tietämystä hyödynnettiin sitten viitekehyksen luomiseen, joka antaa selkeän kuvan Flutterin ja React-Nativen heikkouksista ja vahvuuksista sekä sovelluskehittäjän että loppukäyttäjän näkökulmasta. Tutkielmassa kehitetyt testisovellukset tarjoavat myös lisäarvoa mahdollistamalla tutkielman tulosten uusimisen, mikäli jokin tutkielmassa tutkituista tekniikoista kokisi merkittäviä uudistuksia.

Avainsanat: Alustariippumaton, Mobiilikehitys, Mobiilisovellukset, Haasteet, React-Native, Flutter, Kotlin, Swift, Android, iOS

Abstract: The ever-increasing proliferation of smartphones is directing more and more resources to the development of various mobile applications. Mobile applications are traditionally developed with native technologies based on the target platform, but this can cause significant inefficiencies if the application is targeting multiple different platforms. To address this inefficiency, so-called cross-platform techniques have been developed to enable the utilization of same code on multiple target platforms. However, there are a huge number of different cross-platform techniques, in addition to which they offer very different levels of results. This thesis investigates the applicability of two new cross-platform technologies: Flutter and React-Native to general mobile application development. Particular attention is paid to how the above techniques work compared to native techniques and what trade-offs they require. The study was carried out as a comparative study, which also included a test application produced with the different native and cross platform techniques as an experimental study. The results of this test application and the knowledge gathered in the literature review were then utilized to create a framework that provides a clear picture of the weaknesses and strengths of Flutter and React-Native from the perspective of both the application developer and the end user. The test applications developed in this thesis also provide added value by enabling the renewal of the results in the future, should any of the studied techniques experience significant innovations.

Keywords: Cross-platform, Mobile development, Mobile applications, Challenges, React-Native, Flutter, Kotlin, Swift, Android, iOS

Kuviot

Kuvio 1. Mobiilikäyttöjärjestelmien maailmanlaajuinen jakautuminen vuonna 2020 lokakuussa (StatCounter 2020c). 5
Kuvio 2. Android-versioiden jakautuminen vuonna 2020 lokakuussa (StatCounter 2020a). 6
Kuvio 3. iOS-versioiden jakautuminen vuonna 2020 lokakuussa (StatCounter 2020b). 7
Kuvio 4. Web-sovellusten rakenne (Rahul Raj ja Seshu Babu Tolety 2012). 9
Kuvio 5. Hybridisovelluksen rakenne (Rahul Raj ja Seshu Babu Tolety 2012). 11
Kuvio 6. Tulkillisen sovelluksen rakenne (Rahul Raj ja Seshu Babu Tolety 2012). 13
Kuvio 7. Ristiinkääntyvän sovelluksen rakenne (Rahul Raj ja Seshu Babu Tolety 2012). 15
Kuvio 8. Esimerkki malliperusteisen MD2-sovelluksen rakenteesta (Rieger 2018). 18
Kuvio 9. Hello World -projekti SwiftUI:lla ja Xcodella. 21
Kuvio 10. Hello World -projekti Kotlinilla ja Android Studiolla. 24
Kuvio 11. Hello World -projekti React-Nativella. 26
Kuvio 12. Hello World -projekti Flutterilla. 31
Kuvio 13. Eri intensiteettien animaatiotestit Flutterilla kehitetyssä sovelluksessa ennen testien käynnistämistä. 43
Kuvio 14. Animaatiotestin kulkukaavio. 45
Kuvio 15. GPS-testi ja sen tulokset Flutterilla toteutetussa testisovelluksessa iOS:llä. 46
Kuvio 16. GPS-testin kulkukaavio. 48
Kuvio 17. Kryptografisen testin kulkukaavio. 49
Kuvio 18. Kryptografisen ja JSON-testin ajalliset tulokset Kotlinilla tuotetussa sovelluksessa. 50
Kuvio 19. JSON-testin kulkukaavio. 51
Kuvio 20. Android Studion suorituskyvyn profilointityökalu. Yläreunan punaiset pisteet kuvaavat kosketustapahtumia, joiden perusteella testien aloitus voidaan havaita. 54
Kuvio 21. Osa gfxinfo-työkalun tulostamasta datasta. Huomioitavia kohtia varsinkin kokonaiskuvien (*engl.* total frames) ja pudotettujen kuvien (*engl.* janky frames) määrä. 55
Kuvio 22. Esimerkki Xcoden profilointityökalujen tulosteesta. 56
Kuvio 23. Flutter-sovelluksen yleinen ulkoasu, joka muuttuu alustasta riippuen joko Android- tai iOS-näkymään. 59
Kuvio 24. React-Nativella kehitetyn sovelluksen ulkoasu Android- ja iOS-alustoilla. Flutterista poiketen React-Native ei sisäisesti käytä eri komponentteja eri alustoilla, vaan käyttää sen sijaan kohdealustalta löytyvää komponenttia esimerkiksi painikkeelle. 62
Kuvio 25. Swift-sovellus ja SwiftUI:n automaattinen teemoitus laitteen asetuksista riippuen. 65
Kuvio 26. Kotlin-sovellus ja Android UI:n automaattinen teemoitus. 67
Kuvio 27. Animaatiotestin keskimääräinen FPS eri sovelluksilla. 70
Kuvio 28. Animaatiotestin keskimääräinen CPU-käyttöaste eri sovelluksilla. 72
Kuvio 29. Animaatiotestin keskimääräinen RAM-käyttöaste eri sovelluksilla. 73
Kuvio 30. Animaatiotestin tulokset suhteessa natiiveihin sovelluksiin. 74

Kuvio 31. GPS-testin keskimääräinen suoritusaika eri sovelluksilla. 75
Kuvio 32. GPS-testin keskimääräinen CPU-käyttöaste eri sovelluksilla. 76
Kuvio 33. GPS-testin keskimääräinen RAM-käyttöaste eri sovelluksilla. 77
Kuvio 34. GPS-testin tulokset suhteessa natiiveihin sovelluksiin. 78
Kuvio 35. Kryptografisen testin keskimääräinen suoritusaika eri sovelluksilla. 79
Kuvio 36. Kryptografisen testin keskimääräinen CPU-käyttöaste eri sovelluksilla. 80
Kuvio 37. Kryptografisen testin keskimääräinen RAM-käyttöaste eri sovelluksilla. 81
Kuvio 38. Kryptografisen testin tulokset suhteessa natiiveihin sovelluksiin. 82
Kuvio 39. JSON-testin keskimääräinen suoritusaika eri sovelluksilla. 83
Kuvio 40. JSON-testin keskimääräinen CPU-käyttöaste eri sovelluksilla. 84
Kuvio 41. JSON-testin keskimääräinen RAM-käyttöaste eri sovelluksilla. 85
Kuvio 42. JSON-testin tulokset suhteessa natiiveihin sovelluksiin. 86

Taulukot

Taulukko 1. Sovelluskehittäjän kriteerit alustariippumattomille työkaluille. 40
Taulukko 2. Loppukäyttäjän kriteerit alustariippumattomille työkaluille. 41
Taulukko 3. Tutkielman Android- ja iOS-alustan testilaitteet. 56
Taulukko 4. Eri sovellustekniikat, niiden versiot sekä niiden tukemat Android- ja iOS-versiot. 58
Taulukko 5. Eri sovellusten sovelluspakettien koko ja laitteella käytetty tila. 87
Taulukko 6. Sovellusten käynnistymiseen kuluva aika eri alustoilla. 88
Taulukko 7. React-Nativen ja Flutterin pisteet sovelluskehittäjän kriteereillä. 89
Taulukko 8. React-Nativen ja Flutterin pisteet loppukäyttäjän kriteereillä. 92
Taulukko 9. React-Nativen ja Flutterin painotetut keskiarvot eri kriteeriryhmillä yhden desimaalin tarkkuudella. 94

Sisältö

1 JOHDANTO ... 1
1.1 Tutkimuskysymykset ... 2
1.2 Tutkielman rakenne ... 3

2 MOBIILIKEHITYKSEN NYKYTILANNE ... 4
2.1 Mobiilikehityksen lähestymistavat ... 7
2.1.1 Natiivit lähestymistavat ... 8
2.1.2 Web-pohjaiset lähestymistavat ... 8
2.1.3 Hybridilähestymistavat ... 10
2.1.4 Tulkilliset lähestymistavat ... 12
2.1.5 Ristiinkääntyvät ja widget-perusteiset lähestymistavat ... 14
2.1.6 Malliperusteiset lähestymistavat ... 16

3 MOBIILIKEHITYS ERI TEKNIIKOILLA ... 19
3.1 Swift ... 19
3.1.1 Xcode ... 20
3.1.2 Swift-sovellusten käyttöliittymät ... 20
3.2 Kotlin ... 22
3.2.1 Android Studio ... 22
3.2.2 Natiivien Android-sovellusten kehittäminen Kotlinilla ... 23
3.3 React-Native ... 24
3.3.1 Sovelluskehitys React-Nativella ... 25
3.3.2 Haasteita ... 27
3.4 Flutter ... 28
3.4.1 Sovelluskehitys Flutterilla ... 29
3.4.2 Haasteita ... 31

4 TUTKIMUSMETODOLOGIA ... 33
4.1 Kirjallisuuskatsaus ... 33
4.2 Kokeellinen tutkimus ... 34
4.2.1 Kokeellinen tutkimus tämän tutkielman kannalta ... 35
4.3 Vertailututkimus ... 36
4.3.1 Haasteet ... 37
4.3.2 Vertailututkimus tämän tutkielman kannalta ... 38
4.3.3 Kriteerit ... 39

5 TESTISOVELLUS ... 42
5.1 Testit ... 42
5.1.1 Animaatiotesti ... 43
5.1.2 GPS-testi ... 46
5.1.3 Kryptografinen testi ... 48
5.1.4 JSON-testi ... 50
5.1.5 Yleiset mittarit ... 51

5.2 Datan kerääminen 52
5.2.1 Android 53
5.2.2 iOS 55
5.2.3 Testilaitteet 56
5.3 Eri sovellukset eri alustoilla 57
5.3.1 Flutter 59
5.3.2 React-Native 62
5.3.3 Swift 65
5.3.4 Kotlin 67

6 TULOKSET 70
6.1 Animaatiotesti 70
6.2 GPS-testi 75
6.3 Kryptografinen testi 79
6.4 JSON-testi 83
6.5 Yleiset mittarit 87
6.6 Viitekehys 89
6.6.1 Alustariippumattomat tekniikat sovelluskehittäjän kriteereillä 89
6.6.2 Alustariippumattomat tekniikat loppukäyttäjän kriteereillä 92
6.6.3 Lopulliset tulokset 94

7 YHTEENVETO 96
7.1 Rajoitteet 97
7.2 Tulevaisuuden tutkimus 98

LÄHTEET 99

LIITTEET 106
A Flutter-sovelluksen alustariippumaton painikekomponentti 106
B Flutter-sovelluksen komponentti yhden kuvan animoimiseksi 107
C React-Native-sovelluksen kotinäkymä 108
D React-Native-sovelluksen komponentti yhden kuvan animoimiseksi 109
E Swift-sovelluksen animaatiotestin toteuttava komponentti 111
F Kotlin-sovelluksen yhden kuvan animaation asetukset 113

1 Johdanto

Älypuhelimien nopean yleistymisen ja jatkuvan suorituskyvyn kasvun seurauksena kilpailu erilaisten mobiilisovellusten välillä on vain kiihtynyt. Mobiilisovellusten perinteinen kehitys vaatii yleensä sovelluksen kehittämistä natiiveilla menetelmillä usealle eri alustalle, joka voi helposti johtaa ylimääräisiin kuluihin, eri alustoilla eroaviin sovelluksiin ja yleiseen tehottomuuteen, kun sama sovellus joudutaan kehittämään usealla eri tekniikalla. (Rahul Raj ja Seshu Babu Tolety 2012). Yksi nopeimmin kasvavista tekniikoista tämän välttämiseksi, on niin sanottujen alustariippumattomien (*engl.* cross-platform) mobiilikehitystekniikoiden käyttäminen. Alustariippumattomilla tekniikoilla sovelluksia voidaan tuottaa tehokkaammin, koska sovelluksen kehittämiseen tarvitaan teoriassa vain yksi koodikanta, johon lisätään mahdollisimman vähän alustakohtaisia muutoksia (Pinto ja Coutinho 2018).

Erilaisia alustariippumattomia tekniikoita on suuri määrä, jonka seurauksena ne mahdollistavat toimintansa useilla poikkeavilla lähestymistavoilla. Tästä johtuen eri alustariippumattomilla tekniikoilla on usein eroavia ominaisuuksia ja lähestymistavasta johtuvia rajoitteita, joiden vuoksi jokainen tekniikka ei välttämättä sovi kaikille sovelluksille, vaatien monimutkaisen vaatimusmäärittelyn ennen alustariippumattoman tekniikan valintaa (Biørn-Hansen, Grønli ja Ghinea 2018). Uudemmat alustariippumattomat lähestymistavat pyrkivät toimimaan yhä monipuolisemmissa käyttötapauksissa, mutta vähäisen olemassa olevan tutkimuksen vuoksi yrityksille tai sovelluskehittäjälle voi olla riskialtista käyttää näitä tekniikoita natiivien tekniikoiden sijaan.

Tämän johdosta tässä tutkielmassa keskitytään vertailemaan kahta uudempaa alustariippumatonta tekniikka eri lähestymistavoista sekä toisiaan että natiiveja vastakappaleitansa vastaan. Tekniikoiksi valittiin React-Native, joka edustaa tulkillista lähestymistapaa ja Flutter, joka edustaa widget-perusteista lähestymistapaa. Molemmat tekniikat lupaavat natiivia sovellusta vastaavan käyttöliittymän ulkoasun ja suorituskyvyn, jonka vuoksi erityisesti näitä tekniikoita on mielekästä vertailla. Tämän lisäksi varsinkin Flutterista on erittäin vähän olemassa olevaa suorituskykyyn painottuvaa tutkimusta, joka olisi toteutettu sekä Android-, että iOS-mobiilialustoille.

Tutkielman tarkoituksena on edellä mainittuja tekniikoita vertailemalla selvittää sovelluskehittäjän ja loppukäyttäjän näkökulmasta, mitä haasteita ja vahvuuksia React-Nativella ja Flutterilla on kehityksen, suorituskyvyn, ulkoasun ja muiden yleisten mittareiden puolesta. Näin voidaan luoda selkeä viitekehys siitä, missä tapauksissa kyseisiä tekniikoita voi tai ei voi käyttää, ja kuinka hyvin ne sopeutuvat yleiseen sovelluskehitykseen.

1.1 Tutkimuskysymykset

Koska tämän tutkielman tarkoituksena on luoda viitekehys, valittiin kaksi eri tutkimuskysymystä, joita ovat:

1. Mitä heikkouksia ja vahvuuksia alustariippumattomilla tekniikoilla kuten React-Native ja Flutter on verrattuna natiiveihin kehitysmenetelmiin mobiilikehityksessä?
2. Miten nämä tekniikat sopivat yleiseen mobiilikehitykseen?

Ensimmäinen tutkimuskysymys pyrkii luomaan tarkan kuvan siitä, mitkä ovat edellä mainittujen tekniikoiden tärkeimmät heikkoudet ja vahvuudet, joka on olennaista tekniikoiden toisiinsa ja natiiveihin tekniikoihin vertailun kannalta. Näiden heikkouksien ja vahvuuksien selvittäminen mahdollistaa myös selkeän kuvan luomisen siitä, minkä tyyppisissä sovelluksissa valittu tekniikka ei ole hyvä valinta.

Toinen tutkimuskysymys sen sijaan mittaa valittujen alustariippumattomien tekniikoiden sopimista yleiseen mobiilisovelluskehitykseen. Tällaisella kehityksellä tarkoitetaan yleisintä sovelluskategoriaa, jonka käyttötapaus on erilaisen sisällön selailu ja sen kanssa vuorovaikuttaminen.

Tutkimuskysymyksiin vastaamiseksi tässä tutkielmassa suunnitellaan ja toteutetaan kokeellisena tutkimuksena useita eri testejä sisältävä testisovellus, joka kehitetään erikseen sekä Flutterilla että React-Nativella. Testisovellus kehitetään myös Androidin Kotlinilla ja iOS:n Swiftillä, jotka ovat edellä mainittujen alustojen suositellut natiivit tekniikat. Näin saadaan neljä eri tekniikoilla toteutettua testisovellusta, jotka mahdollistavat tekniikoille toteutettavan vertailututkimuksen. Testisovelluksen tuloksia verrataan lopuksi kirjallisuuskatsauksessa kerättyyn informaatioon, jonka perusteella muodostetaan lopullinen viitekehys tutkimuskysy-

myksiin vastaamiseksi. Viitekehyksen luomiseen käytetty tutkimusmetodologia käydään läpi tarkemmin luvussa 4.

1.2 Tutkielman rakenne

Tutkielma koostuu johdannosta ja sen lisäksi kuudesta eri osasta, joista ensimmäisenä luvussa 2 käsitellään mobiilikehityksen nykytilaa ja erilaisia lähestymistapoja sekä natiivista että alustariippumattomasta näkökulmasta. Seuraavassa luvussa 3 käsitellään tarkemmin tutkielmaan valittuja natiiveja ja alustariippumattomia tekniikoita, tutustuen niiden kehitykseen, toimintaan ja yleisiin haasteisiin. Luvussa 4 käydään läpi tämän tutkielman tutkimusmetodologia ja sen haasteet sekä tekniikoiden vertailuun käytettävän vertailututkimuksen, että testisovelluksella toteutettavan kokeellisen tutkimuksen kannalta.

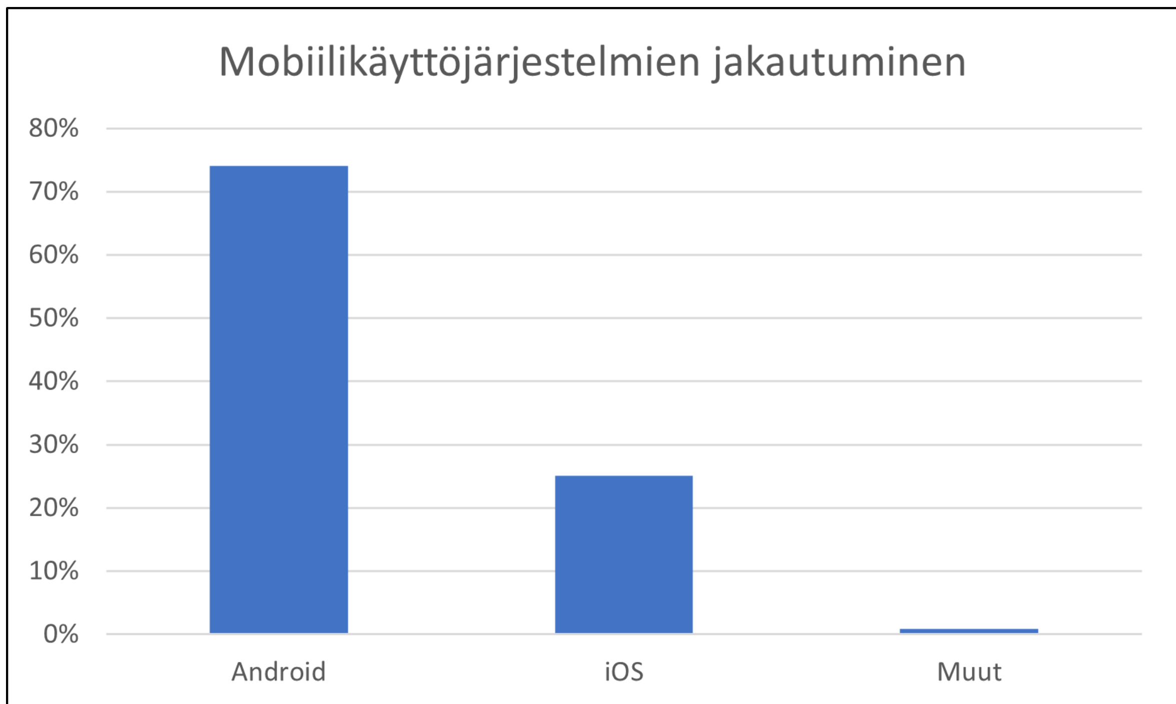
Tutkimusmetodologian jälkeen luvussa 5 tutustutaan testisovellukseen, sen eri testeihin ja niiden perusteluihin, jonka jälkeen luvussa 6 esitetään testisovelluksella tuotetut tulokset ja luodaan tutkimuskysymyksiin vastaava viitekehys. Viimeiseksi luvussa 7 käydään lävitse tutkielma tulosten yhteenveto, niiden rajoitteet ja mahdollisia tulevaisuuden tutkimuksen aiheita.

2 Mobiilikehityksen nykytilanne

Mobiilisovelluskehitys tuo mukanaan muutamia lisävaatimuksia verrattuna tavalliseen sovelluskehitykseen (Wasserman 2010). Näitä vaatimuksia ovat mm. kohdelaitteen muiden sovellusten väliset interaktiot, sensoreiden hyödyntäminen, erilaiset käyttöliittymät esimerkiksi työpöydän sovelluksiin verrattuna ja virrankulutuksen minimointi. Suurimpia haasteita ovat kuitenkin vaihtelevat laitteet ja niiden vaihteleva suorituskyky ja ohjelmisto. Tämä tekee varsinkin mobiilisovellusten testaamisesta haastavaa, sillä testaajan tulee huomioida eri laitteistokokoonpanot, käyttöjärjestelmäpohjat sekä mahdollisesti niiden eri versioiden luomat yhdistelmät (Wasserman 2010; Joorabchi, Mesbah ja Kruchten 2013) .

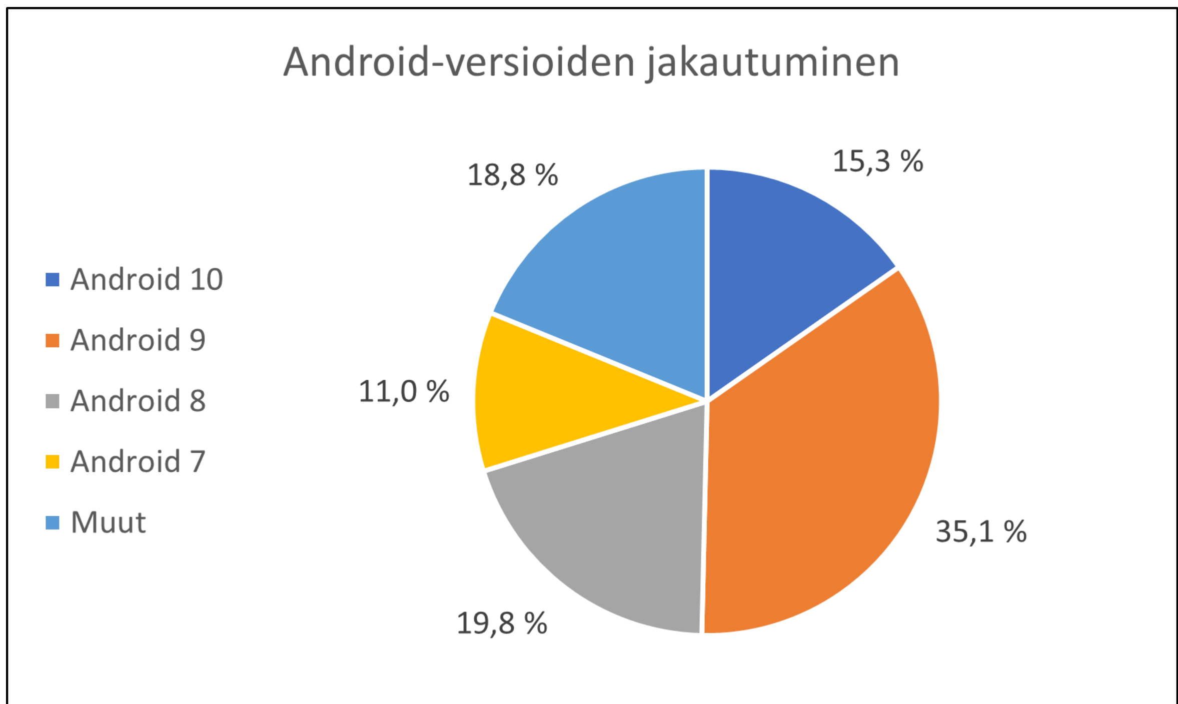
Tämä haaste korostuu varsinkin Googlen kehittämällä Android-käyttöjärjestelmällä, jonka laitekannan vaihtelu on huomattavasti laajempaa sen kilpailijoihin verrattuna. Tämän vuoksi kehitettäessä Androidille, voi kehittäjä resurssien puutteessa joutua valitsemaan, minkä tyyppisiä Android-laitteita tai versioita kehitettävä sovellus tukee (Fling 2009).

Vaikka laitekannan pirstaloituminen on yhä kasvussa, on erilaisten mobiilikäyttöjärjestelmien määrä siitä huolimatta vähentynyt huomattavasi viimeisen 10 vuoden aikana. Kuviosta 1 voidaan havaita StatCounter:in tuottaman analyysin tulokset, joiden mukaan lokakuussa 2020 Android omaa suurimman markkinaosuuden noin 74 prosentilla. Applen iOS seuraa selvänä kakkosena noin 25 prosentilla. Jäljelle jäävät mobiilikäyttöjärjestelmät omaavat vain vajaan yhden prosentin markkinoista.



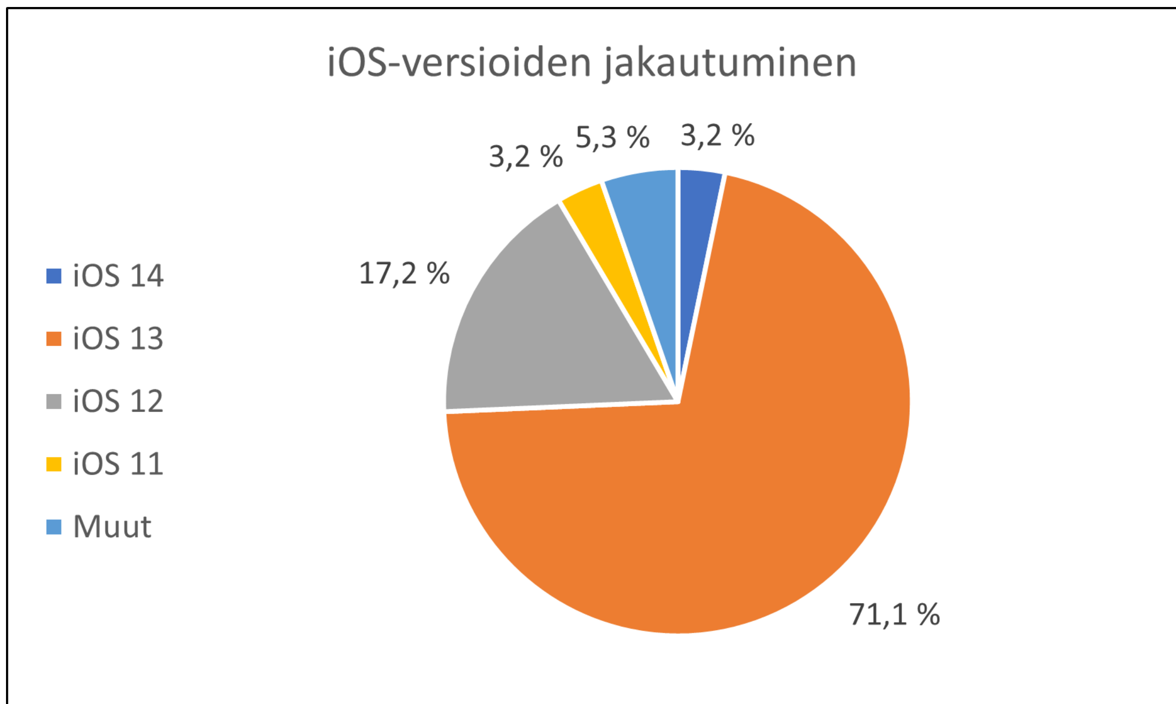
Kuvio 1. Mobiilikäyttöjärjestelmien maailmanlaajuinen jakautuminen vuonna 2020 lokakuussa (StatCounter 2020c).

Android-perusteisia laitteita on siis käytössä melkein kolme kertaa enemmän kuin iOS-perusteisia laitteita. Tämä tuo merkittäviä haasteita Android-kehitykseen, sillä kuten oheisesta kuviosta 2 voidaan havaita, edellä mainitusta 74 prosentista suurin osa käyttää yhä vanhempaa Android-versiota kuin vuonna 2018 elokuussa julkaistu Android 9. Tämän seurauksena sovelluskehittäjän pitää joko varautua tukemaan useita eri käyttöjärjestelmiä tai pahimmassa tapauksessa olla tukematta joitakin käyttöjärjestelmäversioita. (Wasserman 2010).



Kuvio 2. Android-versioiden jakautuminen vuonna 2020 lokakuussa (StatCounter 2020a).

Jos taas katsotaan iOS:n vastaavaa tilannetta oheisesta kuviosta 3, on tilanne täysin päinvastainen ja suurin osa laitteista käyttää vuoden 2019 syyskuussa julkaistua iOS-versio 13:ta tai uudempaa.



Kuvio 3. iOS-versioiden jakautuminen vuonna 2020 lokakuussa (StatCounter 2020b).

Laitekannan ja käyttöjärjestelmäversioiden pirstaloitumisen syyt eivät kuulu tämän tutkielman tutkimuskysymyksiin, mutta tutkielman kannalta on silti tärkeä ymmärtää käyttöjärjestelmäkohtainen tilanne. Esimerkiksi alustariippumattomat lähestymistavat kuten Web ja sen alle kuuluva PWA-malli eivät ole yhtä riippuvaisia käyttöjärjestelmän versiosta kuin natiivit kehitysmetodit (Biørn-Hansen ym. 2020). Tämän seurauksena varsinkin Androidilla voisi olla houkuttelevampaa kehittää sovellus jollakin alustariippumattomalla lähestymistavalla, sillä se voisi mahdollistaa suuremman yhteensopivuuden jo pelkästään yhdessä käyttöjärjestelmässä.

2.1 Mobiilikehityksen lähestymistavat

Nykytilanne alustariippumattomien mobiilikehityslähestymistapojen kanssa on melko monipuolinen. Kehittäjät pystyvät valitsemaan useista erilaisista työkaluista, jotka tarjoavat eroavia ominaisuuksia, vahvuuksia ja heikkouksia. Siksi päätös kehityksen lähestymistavasta perustuukin usein kehitettävän sovelluksen vaatimuksiin tai kohdealustaan (Xanthopoulos ja

Xinogalos 2013). Yleensä myös sovellusta kehittävä taho toimii lähestymistavan valinnan osapuolena ja jos heiltä löytyy jo valmista osaamista natiiveista tekniikoista, on todennäköistä, että sovellus kehitettäisiin niillä alustariippumattomien sijaan.

Seuraavissa kappaleissa käydään lävitse yleisimpiä mobiilikehityksen lähestymistapoja natiivista kehityksestä alustariippumattomaan.

2.1.1 Natiivit lähestymistavat

Natiivilla kehityksellä tarkoitetaan kehitystä, jossa hyödynnetään yhtä tai useampaa kohdekäyttöjärjestelmän tarjoamaa koodikantaa. Sovellusten kehitykseen käytetään yleensä käyttöjärjestelmän kehittäjän tarjoamaa ohjelmistokehityspakettia (*engl.* Software development kit), joka mahdollistaa kytkennän kohdealustaan matalammalla tasolla (Biørn-Hansen ym. 2020). Tämä tekee natiiveilla menetelmillä kehitetyistä sovelluksista yleensä nopeampia ja tehokkaampia kuin alustariippumattomilla menetelmillä kehitetyistä vastakappaleistaan sekä mahdollistaa pääsyn kaikkiin alustan laitteen toimintoihin (Pinto ja Coutinho 2018).

Natiivien kehitysmenetelmien suurin heikkous onkin se, että ne vaativat useammalle alustalle kehittäessä useamman eri ohjelmistokehityspaketin ymmärryksen sekä erillisten koodikantojen ylläpidon. Tämä voi johtaa eri alustojen eroavaan käyttökokemukseen, suurempiin kuluihin ja yleiseen epätehottomuuteen, kun sovelluksesta joudutaan kehittämään eri versioita eri alustoille (Pinto ja Coutinho 2018).

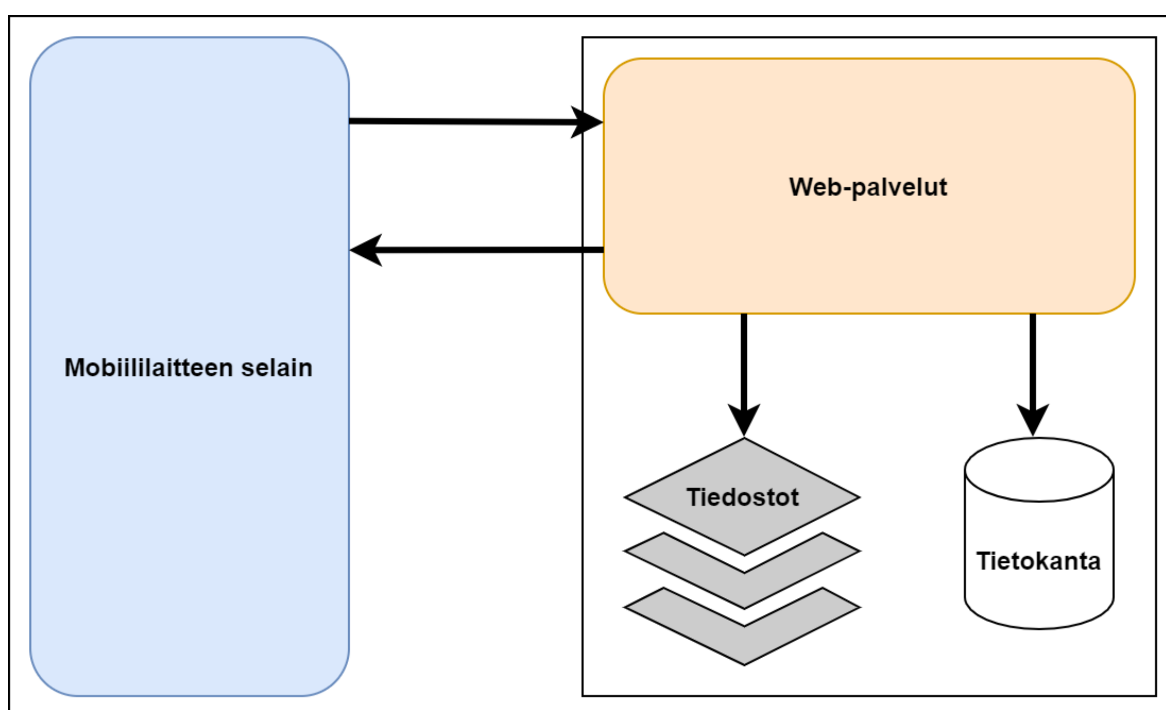
Kuten edellä mainittiin, tällä hetkellä suurinta markkinaosuutta mobiililaitteissa pitävät Applen tarjoama iOS-käyttöjärjestelmä ja Googlen Android-käyttöjärjestelmä. iOS tarjoaa ohjelmistokehityspaketteja Objective C:lle ja Swiftille, kun taas Android tukee koodipohjia kuten C++, Kotlin ja Java (Biørn-Hansen ym. 2020). Tämän tutkielman kannalta näistä merkittäviä ovat varsinkin Swift ja Kotlin, joihin tutustutaan tarkemmin luvuissa 3.1 ja 3.2.

2.1.2 Web-pohjaiset lähestymistavat

Web-pohjaiset sovellukset ovat yksi yksinkertaisimmista ja helpoiten toteutettavista alustariippumattomista lähestymistavoista. Web-pohjainen sovellus on yksinkertaisesti sanottuna

internetsivu, johon on lisätty sovelluslogiikkaa tai erityisiä alustasta riippuvaisia ominaisuuksia (Rahul Raj ja Seshu Babu Tolety 2012). Web-sovelluksen kehittämiseen käytetään yleensä samoja tekniikoita kuin web-kehitykseen, joka mahdollistaa luontevan siirtymisen pelkästä internetsivusta web-sovellukseen. Näitä tekniikoita ovat mm. HTML5, CSS ja Javascript, joista erityisesti Javascriptiä käytetään sovelluslogiikan lisäämiseen.

Tavallisesta sovelluksesta poiketen web-sovelluksen komponentteja ei yleensä asenneta laitteelle, vaan ne ladataan laitteen selaimella aina sovelluksen käynnistyksen yhteydessä (Rahul Raj ja Seshu Babu Tolety 2012).



Kuvio 4. Web-sovellusten rakenne (Rahul Raj ja Seshu Babu Tolety 2012).

Edellä mainitun toimintamallin vuoksi web-pohjaisilla sovelluksilla on haastavaa päästä käsiksi natiiveihin ominaisuuksiin, joka usein johtaa siihen, että kyseisten sovellusten käyttökokemus ei ole yhtä laadukas kuin muilla lähestymistavoilla. Tämän ratkaisemiseksi on pyritty kehittämään erilaisia vaihtoehtoja, ja esimerkiksi Google on tavoitellut niin sanotun PWA-mallin (Progressive Web Application) yleistämistä. PWA lisää web-sovellukseen taustalla ajettavan prosessin, metadataa, mahdollisuuden sovelluksen asentamiseen laitteelle ja kyvyn toimia ilman aktiivista verkkoyhteyttä (Biørn-Hansen ym. 2020). Tämä tekee web-

sovelluksesta huomattavasti monipuolisemman ja tarjoaa mahdollisuuden päästä käsiksi rajoitettuun joukkoon natiiveja ominaisuuksia, joita tavallinen web-sovellus ei pysty hyödyntämään.

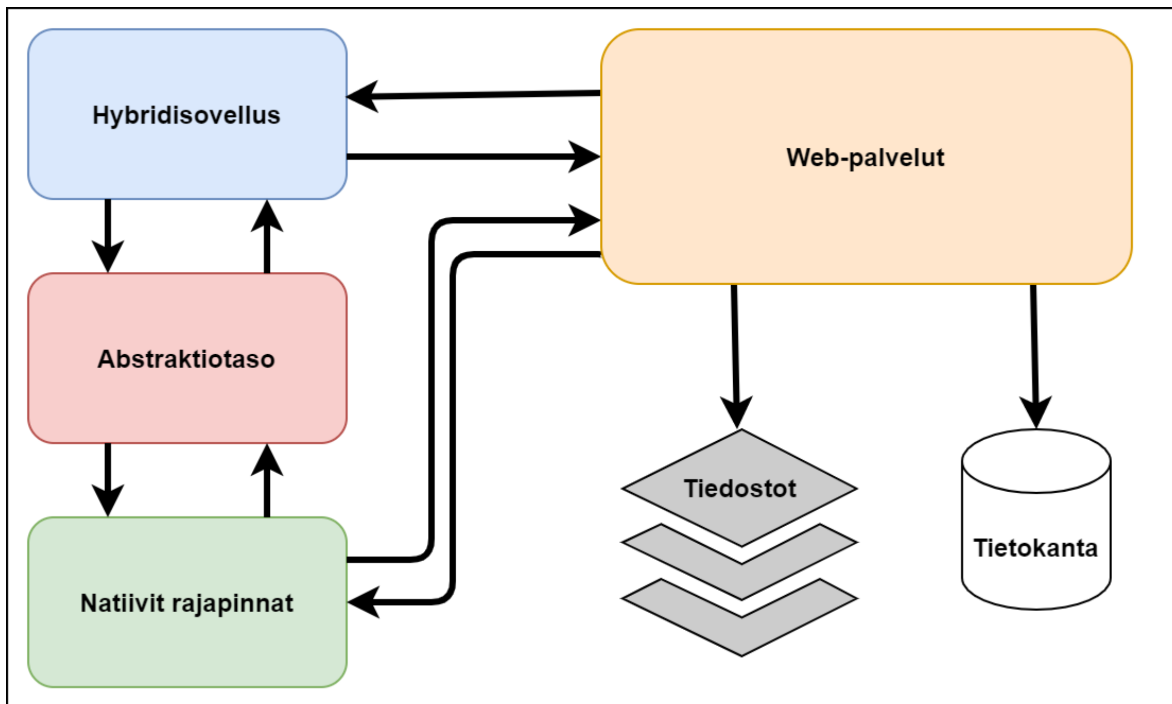
Vaikka PWA pyrkii tarjoamaan lisää natiiveja ominaisuuksia web-pohjaisille sovelluksille, on sen vastaanotto ollut rajoitettua. Koska sovellusta ajetaan selaimessa, joutuu se kohtamaan erilaisia rajoituksia, jotka ovat paikallaan verkkosivupohjaisten hyökkäyksien estämiseksi. Tämän vuoksi laitteen tarjoamat natiivit rajapinnat esimerkiksi kameralle tai GPS-sensorille ovat käyttöjärjestelmästä riippuen joko heikosti käytettävissä tai täysin käyttökelvottomia (Biørn-Hansen ym. 2020; Rahul Raj ja Seshu Babu Tolety 2012) .

Edellä mainittujen haasteiden lisäksi yksi merkittävimmistä web-pohjaisten sovellusten rajoitteista on, että käyttäjä ei voi asentaa niitä Applen tai Googlen tarjoamasta sovelluskaupasta (Rahul Raj ja Seshu Babu Tolety 2012). Tämä tekee niistä hankalammin saavutettavia, joka vaikeuttaa sovelluksen löytämistä, asentamista ja käyttämistä.

2.1.3 Hybridilähestymistavat

Hybridilähestymistapojen voidaan katsoa olevan evolutiivinen kehitysaskel web-lähestymistavoista. Sovellus ja varsinkin sen käyttöliittymä voidaan yhä kehittää web-teknologioilla kuten HTML5, CSS ja Javascript, mutta sovelluksen koodi niin sanotusti upotetaan natiivin sovelluksen sisälle (Biørn-Hansen, Grønli ja Ghinea 2018).

Käyttöliittymän web-pohjainen koodi asetetaan yleensä natiivista koodikannasta löytyvään web-selain komponenttiin, joka toimii samanlaisilla periaatteilla kuin web-pohjaisten sovellusten selain. Merkittävä ero web-pohjaisiin sovelluksiin kuitenkin on, että hybridisovelluksesta pystyy rakentamaan sovelluskaupasta asennettavan paketin, koska se käyttää pohjanaan natiivia koodia. Tätä tekniikkaa kutsutaan usein myös web-sovelluksen paketoimiseksi (oma suomennos, *engl.* wrapping), koska web-sovelluksen ympärille “kääritään” natiivia koodia käyttävä sovellus (Biørn-Hansen, Grønli ja Ghinea 2018).



Kuvio 5. Hybridisovelluksen rakenne (Rahul Raj ja Seshu Babu Tolety 2012).

Tämä natiiviin koodiin pakkaaminen tekee hybridisovelluksista erittäin yhteensopivia erilaisten laitteiden kanssa, sillä sovellus käyttää käyttöliittymän esittämiseen samaa komponenttia kuin laite käyttäisi oman nettiselaimensa esittämiseen. Laite ei kuitenkaan enää välttämättä tarvitse internet-yhteyttä, sillä toisin kuin tavallisilla web-sovelluksilla, suoritetaan koodia nyt laitteella verkon sijaan (Rahul Raj ja Seshu Babu Tolety 2012).

Hybridisovellukset ovat myös huomattavasti saavutettavampia, sillä ne voidaan lisätä eri tarjoajien sovelluskauppoihin. Lisäksi, koska koodi on pakattu natiivin sovelluksen sisälle, päästään käsiksi useisiin erilaisiin natiivien sovellusten käyttämiin rajapintoihin, kuten GPS, kamera ja laitteen sisäinen muisti (Biørn-Hansen, Grønli ja Ghinea 2018).

Koska hybridisovellus hyödyntää natiivia koodia, vaatii se kehittäjältä myös sen osaamista web-teknologioiden lisäksi. Ylimääräistä koodia web-sovelluksen käärimiseksi on melko paljon (Biørn-Hansen, Grønli ja Ghinea 2018), jonka vuoksi on kehitetty työkaluja, jotka pystyvät automaattisesti luomaan useamman alustan natiivin koodipohjan, johon web-sovellus kääritään. Yksi suosituimmista työkaluista tähän on Cordova (Biørn-Hansen ym. 2020),

joka helpottaa eri natiivien rajapintojen käyttöä tarjoamalla yksinkertaisia API-liitäntöjä. Muita hybridilähestymistapaa hyödyntäviä tekniikoita ovat mm: Capacitor, PhoneGap, Ionic Framework, Quasar Framework ja Intel App Framework (Biørn-Hansen, Grønli ja Ghinea 2018).

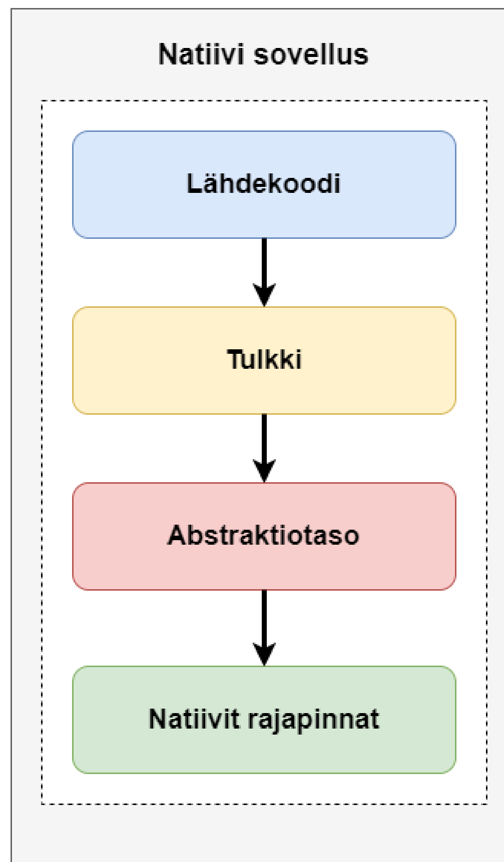
Hybridilähestymistapojen heikkouksiin kuuluu yleisesti heikompi suorituskyky natiiveihin sovelluksiin verrattuna, koska koodia ajetaan yhä verkkoselaimen kaltaisessa komponentissa, vaikka dataa ei verkon ylitse haettaisikaan. Hybridisovellukset eivät myöskään pysty web-sovellusten tavoin käyttämään natiiveja käyttöliittymäkomponentteja, tehden niiden käyttökokemuksesta heikomman (Rahul Raj ja Seshu Babu Tolety 2012). Tämän lisäksi hybridisovelluksilla on uniikkeja sovellushaavoittuvuuksia, koska useat käärimiseen käytetyt työkalut muuttavat tapaa, joilla yleisiä sovelluslogiikan virheitä käsitellään (Zuo, Wu ja Guo 2015). Pahimmassa tapauksessa kehittäjän voi olla mahdotonta korjata haavoittuvuutta itse, jos työkalu ei mahdollista virheenkäsittelyn muuttamista.

2.1.4 Tulkilliset lähestymistavat

Siinä missä hybridisovellukset olivat evolutiivinen askel web-sovelluksille, ovat tulkkia hyödyntävät sovellukset kehitysaskel hybridisovelluksille. Näissä tulkillisen lähestymistavan (oma suomennos, *engl.* interpreted approach) sovelluksissa koodia voidaan jälleen tuottaa web-kehitykseen tarkoitetuilla työkaluilla, mutta tällä kertaa koodin suorittamista ei tarvitse rajoittaa vain natiivin alustan selainkomponenttiin (Biørn-Hansen ym. 2020). Sovellusten kehittämiseen voidaan myös käyttää muita vaihtoehtoisia ohjelmointikieliä Javascriptin lisäksi. Esimerkiksi C# tai jokin merkkikieli kuten Qt tukevat molemmat tulkillista lähestymistapaa (Biørn-Hansen ym. 2020).

Mobiilikehityksessä tulkin sisältäviä lähestymistapoja kutsutaan usein myös ajonaikaisiksi ja web-natiivi-lähestymistymistavoiksi (oma suomennos, *engl.* runtime approach & web-native approach) (Biørn-Hansen ym. 2020). Tämä johtuu siitä, että tulkilliset lähestymistavat hyödyntävät ajonaikaista ohjelmointikielen tulkkia, joka kääntää esimerkiksi Javascriptillä tuotetun koodin aina tarvittaessa natiiville koodille ymmärrettäväksi. Vaikka tulkki kääntääkin esimerkiksi käyttöliittymän koodin natiiviksi koodiksi, ei se kuitenkaan tuota natiivia tavu-

koodia. Tämän vuoksi tulkilliset lähestymistavat vaativat vielä erillisen abstraktiotason tulkin ja natiivin koodin väliin (Rahul Raj ja Seshu Babu Tolety 2012).



Kuvio 6. Tulkillisen sovelluksen rakenne (Rahul Raj ja Seshu Babu Tolety 2012).

Käytettävät tulkit vaihtelevat kohdealustan ja tulkilliseen kehitykseen hyödynnettävien työkalujen mukaan. Pääsääntöisesti iOS-pohjaiset laitteet käyttävät kuitenkin JavascriptCore-tulkkia, kun taas Android-pohjaiset laitteet käyttävät V8:a (Biørn-Hansen ym. 2020; Biørn-Hansen, Grønli ja Ghinea 2018) . Jotta tulkki pystyy kommunikoimaan natiivin ja ei-natiivin koodin välillä, hyödyntää se siltaamiseksi (oma suomennos, *engl.* bridging) kutsuttua tekniikkaa. Nämä sillat välittävät kutsuja natiivin ja esimerkiksi Javascriptillä kehitetyn koodipohjan välillä ja toimivat varsin samankaltaisesti joidenkin hybridilähestymistapojen käyttämien API-siltojen kanssa (Biørn-Hansen, Grønli ja Ghinea 2018).

Kuten aikaisemmin mainittiin, eivät tulkilliset sovellukset ole enää rajoitettuja vain selainkomponenttiin. Tämä tekee niiden käyttöliittymistä ja käyttökokemuksesta laadukkaampia,

sillä web-tekniikoilla tuotettu sovelluslogiikka pystytään nyt kääntämään natiiveiksi komponenteiksi ja käyttöliittymä on ainakin ulkoisesti täysin verrattavissa natiiviin sovellukseen (Biørn-Hansen, Grønli ja Ghinea 2018). Kaikkia laitteen natiiveja ominaisuuksia ei kuitenkaan pystytä käyttämään ja hybridisovellusten tavoin rajoitteeksi tulee valittuun tulkilliseen lähestymistapaan luodut sillat.

Tämä siltojen spesifikaatioiden vaihtelu erilaisten sovelluskehysten, kuten React-Nativen ja Titanium Appceleratorin välillä aiheuttaa myös merkittävästi suurempia ongelmia verrattuna hybridisovellusten työkaluihin (Biørn-Hansen, Grønli ja Ghinea 2018). Toisin kuin hybridisovelluksilla, ei tulkillisten sovellusten kehitystyökaluille ole kehittynyt yhtä yhteistä pohjaa. Näin ollen erilaiset sillat natiiviin koodiin on toteutettu eri tavalla työkalusta riippuen, tehden jo kehitetyn sovelluksen siirtämisestä toiseen työkaluun haastavaa (Biørn-Hansen, Grønli ja Ghinea 2018). Tätä ongelmaa vain lisää se, että tulkillisten sovellusten kehittämiseen on todella useita eri työkaluja, joilla saattaa olla hyvinkin eroavia ominaisuuksia (Pinto ja Coutinho 2018). React-Nativen ja Titanium Appceleratorin lisäksi näitä työkaluja ja tekniikoita ovat mm: NativeScript, MoSync, Adobe AIR, LuaView, Jasonette ja Fusetools (Biørn-Hansen, Grønli ja Ghinea 2018).

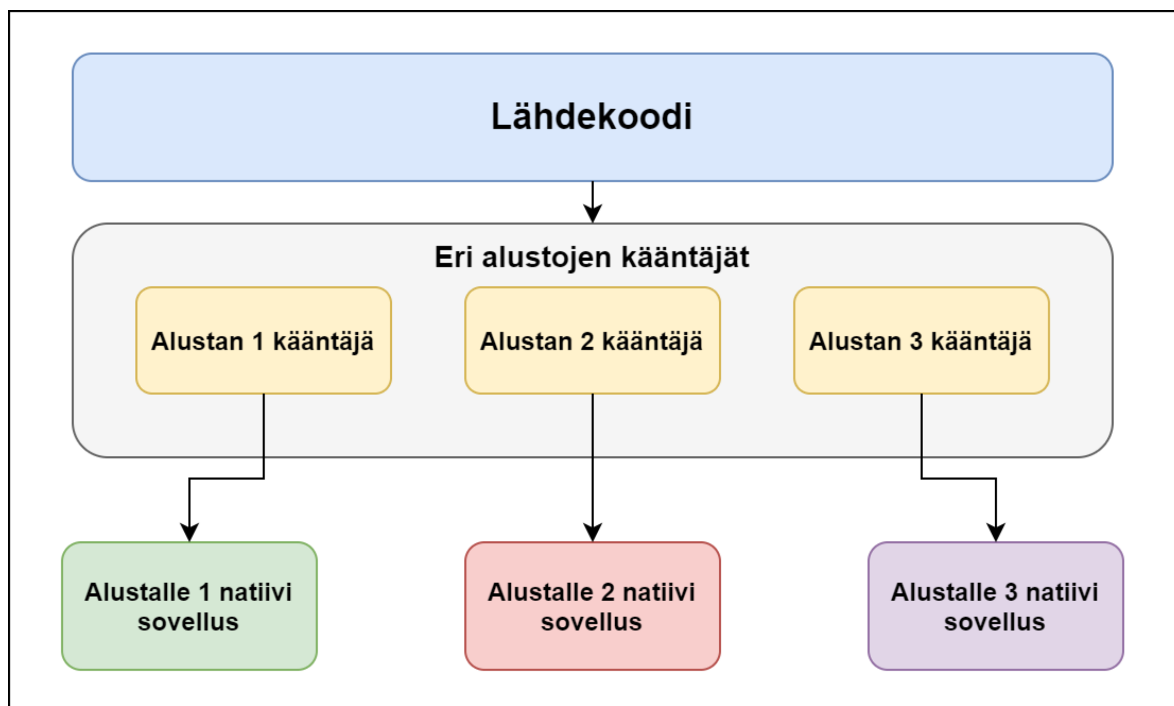
Muita tulkillisten sovellusten haasteita ovat mm. suorituskyvyn ongelmat, jotka johtuvat pääasiassa eri siltojen viestien aiheuttamista viiveistä ja suorituksen ajonaikaisuudesta (Rahul Raj ja Seshu Babu Tolety 2012). On myös huomattavaa, että suurin osa tulkillisten sovellusten kehitykseen käytettävistä ohjelmistokehyksistä vaatii jonkin verran alustakohtaista koodia, koska sovellukset käyttävät nyt eri alustoilla eroavia natiiveja UI-komponentteja. Tämä lisää työmäärää esimerkiksi web-sovelluksiin verrattuna, mutta tarjoaa samalla ylivertaisen käyttäjäkokemuksen.

2.1.5 Ristiinkääntyvät ja widget-perusteiset lähestymistavat

Ristiinkääntyvät lähestymistavat (oma suomennos, *engl.* crosscompiled approach) eroavat aikaisemmin mainituista lähestymistavoista varsin merkittävästi. Kun edelliset lähestymistavat ovat pääasiassa keskittyneet web-teknologioilla kehittämiseen ja yhä syvempään natiiviin integraatioon siirtymiseen, luopuvat ristiinkääntyvät lähestymistavat web-teknologioista

ja olemassa olevista natiiveista UI-komponenteista kokonaan. Ristiinkääntyvät sovellukset luovat sen sijaan uusia natiiveja UI-komponentteja kopioimalla iOS- ja Android-alustojen olemassa olevia komponentteja, jotka sitten piirretään valitun ohjelmistokehyksen omalla renderöintimoottorilla (Shah, Sinha ja Mishra 2019).

Jokaiselle ristiinkääntyvän sovelluksen alustalle on oma kääntäjänsä, joka kykenee kääntämään saman lähdekoodin kohdealustoille natiiviksi tavukoodiksi (Rahul Raj ja Seshu Babu Tolety 2012). Koska ristiinkääntyvät lähestymistavat sisältävät omat kääntäjänsä, eivät ne myöskään ole samalla tavalla riippuvaisia edellisten lähestymistapojen tulkeista tai silloista (Biørn-Hansen, Grønli ja Ghinea 2018).



Kuvio 7. Ristiinkääntyvän sovelluksen rakenne (Rahul Raj ja Seshu Babu Tolety 2012).

Teoriassa tämä tarjoaa ristiinkääntyville lähestymistavoilla parhaan suorituskyvyn kaikista edellä mainituista alustariippumattomista lähestymistavoista, sillä ne kykenevät kommunikoimaan laitteen natiivien rajapintojen kanssa ilman erillisten siltojen tai tulkkien luomia viiveitä (Biørn-Hansen, Grønli ja Ghinea 2018). Ero esimerkiksi tulkillisten lähestymistapojen tarjoamiin käyttöliittymiin on kuitenkin merkittävä, sillä jokainen käyttöliittymän komponentti tulee nyt erikseen tuottaa valittua ohjelmistokehystä varten. Tämän vuoksi ristiinkään-

tyviä lähestymistapoja kutsutaan usein myös widget-perusteisiksi, sillä jokainen ohjelmistokehyksen tarjoama komponentti voidaan nähdä yhtenä widgettinä (Shah, Sinha ja Mishra 2019).

Ristiinkääntyvillä menetelmillä tuotetun sovelluksen voidaan katsoa olevan kaksipuolinen miekka, sillä se tarjoaa hyvin lähelle natiivia sovellusta vastaavaa suorituskykyä, mutta samalla yksikään sen natiiveista käyttöliittymäkomponenteista ei ole alustan oma. Lisäksi, koska ristiinkääntyvät sovellukset vaativat omat kääntäjänsä ja komponenttinsa, voi valmiista sovelluspaketista tulla varsin suurikokoinen verrattuna muihin lähestymistapoihin (Biørn-Hansen ym. 2020).

Yleisiä ristiinkääntyviä lähestymistapoja Flutterin lisäksi ovat mm: Xamarin, Xojo, RAD Studio (Delphi) ja Codename One (Biørn-Hansen ym. 2020; Biørn-Hansen, Grønli ja Ghinea 2018) . Näistä varsinkin Flutter on uudempi ja tuntemattomampi ja se on siksi tämän tutkielman kannalta varsin relevantti.

2.1.6 Malliperusteiset lähestymistavat

Malliperusteinen lähestymistapa (oma suomennos, *engl.* Model-Driven Approach) perustuu malliperusteiseen ohjelmistokehitykseen, joka on yksi yleisistä ohjelmistokehityksen paradigmoista (Biørn-Hansen, Grønli ja Ghinea 2018). Malliperusteisessa kehityksessä sovellusta ja sen toimintaa kuvataan useilla erilaisilla malleilla, joita sitten hyödynnetään lähdekoodin generoimiseen (Xanthopoulos ja Xinogalos 2013).

Alustariippumattomien sovellusten kehityksessä malliperusteisten lähestymistapojen hyöty on niiden kyvyssä tarjota sovelluskehitystä pelkkää lähdekoodia korkeammalta tasolta. Tämä sallii joiltain osin helpomman sovelluskehityksen, sillä kehitys voidaan toteuttaa koodittomasti malleja luoden, yleensä jonkinlaisella graafisella käyttöliittymällä (Biørn-Hansen ym. 2020; Rieger 2018) . Tämä taas mahdollistaa sovelluksen kehityksen myös henkilöille, joilla ei ole yhtä vahvaa teknistä sovelluskehityksen osaamista.

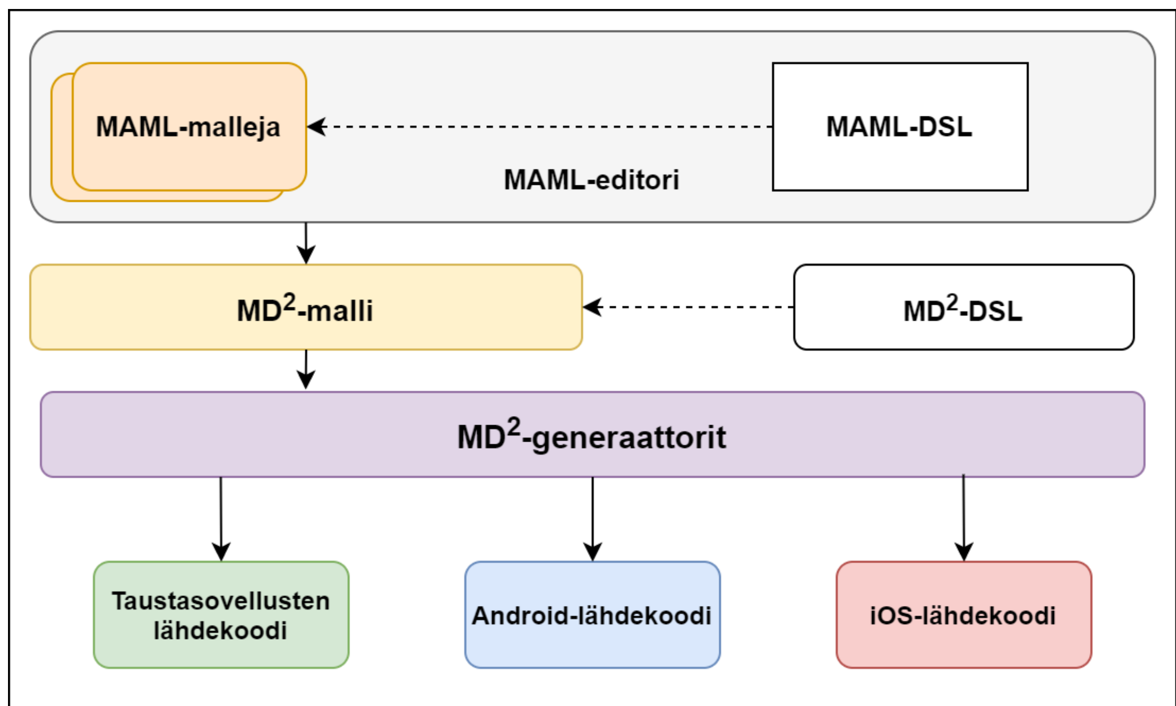
Malliperusteiset lähestymistavat vaativat kuitenkin yleensä jonkin työkalukohtaisen DSL:n (*engl.* Domain-Specific Language) ymmärrystä graafisen käyttöliittymän ohella (Biørn-Hansen ym. 2020; Biørn-Hansen, Grønli ja Ghinea 2018) . DSL toimii perinteisen koodikielen, ku-

ten esimerkiksi Javan korvaajana ja sitä käytetään varsinkin monimutkaisemman sovelluslogiikan määrittämiseen. Kehittäjän näkökulmasta työkalukohtaiset DSL-mallit ovat yksi malliperusteisen lähestymistavan heikkouksista, sillä eri työkalujen erilaiset DSL:lät tekevät osaamisen siirtämisen toisesta DSL:stä toiseen hankalaksi (Biørn-Hansen, Grønli ja Ghinea 2018).

Työkalusta riippuen DSL myös muodostetaan eri tavoilla. DSL voi esimerkiksi olla työkalun valmistajan itse kehittämä, kuten MD2-työkalussa (Heitkötter, Majchrzak ja Kuchen 2013). Tämän lisäksi DSL voi olla asiakkaan itse kehittämä, joka antaa asiakkaalle vapaat kädet kielen kanssa ja mahdollistaa asiakkaan omiin tarpeisiin perustuvalla kielellä tapahtuvan kehityksen. Tällaista palvelua tarjoaa esimerkiksi MetaCase (tunnetaan myös nimellä MetaEdit+), joka kutsuu kehitystapaa DSM:ksi (*engl.* Domain-Specific Modeling) (Kelly ja Tolvanen 2008). DSM-mallia suositellaan varsinkin tilanteisiin, jossa kehitystä toteutetaan pitkällä aikavälillä saman tietojärjestelmän alueella. Tällöin DSM mahdollistaa luotettavamman kehitysajan arvioinnin, koodipohjan korkeamman abstrahoinnin ja sen tehokkaamman uudelleenkäytön.

Ideaalissa tilanteessa malliperusteisilla lähestymistavoilla kehitetty sovellus kääntyy suoraan natiiviksi koodiksi, jolloin suorituskyvyn ja käyttöliittymän pitäisi olla identtisiä natiiveilla kehitysmenetelmillä tuotetun sovelluksen kanssa (Biørn-Hansen ym. 2020). Todellisuudessa erilaiset työkalut tarjoavat varsin erityyppisiä lopputuloksia ja valmiit sovellukset ovat usein lähempänä hybridi- tai tulkillisilla menetelmillä tuotettuja sovelluksia (Biørn-Hansen ym. 2020). Lisäksi, koska kehitys tehdään pääasiassa pelkillä malleilla, on sovelluksen kehitys rajoittunut työkalun tarjoamiin lisäominaisuuksiin. Tämän vuoksi malliperusteisen mobiilikehityksen hyödyt riippuvat vahvasti tilanteesta, sillä jos sovellus vaatisi jotain tiettyä toimintoa, johon valittu työkalu ei valmiiksi pysty, voi olla nopeampaa vain kehittää sovellus jollakin muulla lähestymistavalla (Biørn-Hansen ym. 2020).

Yleisimpiä malliperusteisen kehityksen työkaluja ovat mm. MD2, MAML, DSL Bubble, MetaCase, Applause ja Mendix (Biørn-Hansen ym. 2020; Biørn-Hansen, Grønli ja Ghinea 2018; Xanthopoulos ja Xinogalos 2013) . On myös tärkeää huomata, että suurin osa malliperusteisista työkaluista, jotka vaativat vähiten lähdekoodiin perustuvaa osaamista ovat yleensä maksullisia (Xanthopoulos ja Xinogalos 2013).



Kuvio 8. Esimerkki malliperusteisen MD2-sovelluksen rakenteesta (Rieger 2018).

3 Mobiilikehitys eri tekniikoilla

Tässä luvussa käydään tarkemmin läpi tutkielman kannalta merkittävät natiivit ja alustariippumattomat tekniikat. Jokaisessa tekniikassa tutustutaan sen historiaan, toimintaan, kehityskokemukseen, tulevaisuuden näkymiin ja yleisiin haasteisiin.

Seuraavassa kappaleessa tutustutaan ensimmäisenä Applen ekosysteemille natiiviin Swiftiin.

3.1 Swift

Swift on Applen kehittämä ja vuonna 2014 julkaisema yleiskäyttöinen avoimen lähdekoodin koodikieli, joka soveltuu ajettavaksi erilaisilla laitteilla puhelimista palvelimiin. Swift toimii yhteensopivasti Applen aiemmin voimakkaasti tukeman Objective-C koodikielen kanssa, johon verrattuna Swift tarjoaa useita suorituskykyyn ja koodin turvallisuuden liittyviä parannuksia (Apple Developer 2020a).

Ominaisuuksiltaan Swift pyrkii toteuttamaan useita moderneja ohjelmointiparadigmoja (Swift 2020). Näitä ovat mm:

- Muuttujien alustus ennen käyttöä.
- Kokonaislukujen tarkistus ylivuotojen varalta.
- Null-arvojen määritetty käyttö.
- Muistin automaattisen ylläpito.
- Virhekäsittely, joka mahdollistaa virhetilanteista palautumisen hallitusti.

Swift-koodi käännetään tavukoodiksi LLVM-kääntäjällä, joka tarjoaa optimoitua suorituskykyä, joskin Swiftin erinäiset turvallisuusominaisuudet voivat aiheuttaa valintoja turvallisuuden ja suorituskyvyn välillä (Swift 2020). Swift myös vaatii vähintään version 7.0 iOS:stä (Swift 2021), mutta kuten kuviosta 3 voidaan havaita, toteutuu tämä vaatimus suurimmalla osalla käyttäjistä. Eri käyttöjärjestelmäversioiden tuen kannalta suurempi rajoittaja onkin käyttöliittymän kehittämiseen valittu ohjelmistokehys, joihin tutustutaan kappaleessa 3.1.2.

3.1.1 Xcode

Apple suosittelee Swift-sovelluksien kehittämiseen Xcode IDE:a, jota voi kuitenkin käyttää vain macOS-käyttöjärjestelmällä. Vaikka Swiftillä voi kehittää ilman macOS-käyttöjärjestelmää (Swift 2020), on varsinkin iOS-mobiilisovellusten kehittäminen ilman sitä mahdotonta. Useat mobiilisovellusten käyttämät kirjastot on kehitetty vain macOS:llä käytettäviksi, jonka vuoksi sovelluksen julkaisuun vaadittu sovelluspaketti on mahdotonta luoda millään muulla käyttöjärjestelmällä. Lisäksi pelkän sovelluksen simuloitu ajokin on mahdotonta ilman macOS:ää.

Xcode tarjoaa työkalut kaikkien eri Apple-alustojen sovellusten kehittämiseen, jonka lisäksi se tarjoaa myös useita eri pohjia erityyppisille sovelluksille, nopeuttaen prototyypin kehittämistä. Sovellusten käyttöliittymiä voi kehittää joko raakana koodina tai graafisilla työkaluilla, jotka mahdollistavat eri käyttöliittymäkomponenttien lisäämisen drag'n'drop-periaatteella. Xcode toimii myös integroidusti Applen tarjoaman iOS-simulaattorin kanssa, joka mahdollistaa ohjelmakoodin ajamisen simuloidusti. Kyseessä on kuitenkin vain simulaattori, jonka vuoksi se tarjoaa vähemmän toiminnallisuutta verrattuna Android-kehityksessä käytettävään emulaattoriin.

3.1.2 Swift-sovellusten käyttöliittymät

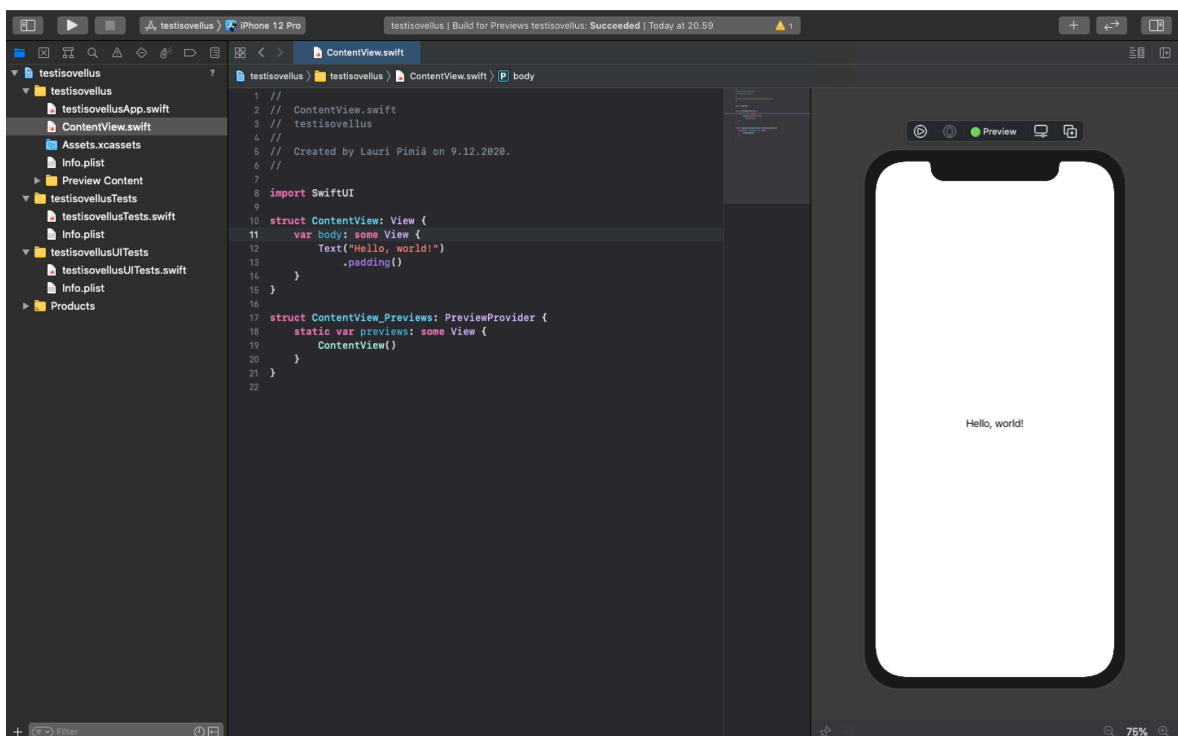
Käyttöliittymien kehittämiseen Apple suosittelee SwiftUI:n käyttämistä, joka mm. tarjoaa tuen jokaiselle Apple-alustalle, deklaratiivisen syntaksin käytölle, dynaamiselle tyypitykselle, lokalisaatiolle ja paremmalle saavutettavuudelle (Apple Developer 2020b). SwiftUI:n on tarkoitus yksinkertaistaa aikaisemman UIKit-käyttöliittymäkirjaston käyttöä, joka on ollut käyttöliittymien kehittämiseen tarkoitettu kirjasto Objective-C:stä lähtien (Blewitt 2016). SwiftUI hyödyntää yhä UIKit:n tarjoamia rajapintoja, joten sen tarkoitus ei ole kokonaan korvata UIKit-kirjastoa.

Ohjelmistokehitysparadigmojen puolesta SwiftUI eroaa UIKit:stä varsinkin siksi, että UIKit:llä kehitys toimi yleensä MVC-paradigman (*engl.* Model-View-Controller) mukaisesti (Apple Developer 2020c), kun taas SwiftUI on enemmän abstrahoitu ja käyttöliittymä luodaan kokonaan erilaisilla malleilla ja näkymillä (Apple Developer 2020b). SwiftUI:ta on kuitenkin mahdollista kehittää myös MVC-periaatteilla.

Kenties suurin rajoite SwiftUI:n käyttöön on, että se vaatii vähintään iOS:n version 13 (Apple Developer 2021). Vaikka Applen laitteista yli 70 prosenttia näyttäisi toteuttavan tämän vaatimuksen, on mahdollista, että SwiftUI:n käyttöönotto hidastuu, kunnes suurempi osa laitteista on päivitetty. Lisäksi, koska SwiftUI julkaistiin vasta vuonna 2019, on siihen saatavilla huomattavasti vähemmän dokumentaatiota UIKit:n verrattuna.

Sovellusten testaamiseen Apple suosittelee käyttöliittymäkirjastosta riippumatta XCTest-kirjastoa, joka tarjoaa työkalut yksikkö-, suorituskyky- ja käyttöliittymätestien luomiseen (Apple Developer 2020d). XCTest tukee myös jatkuvaa integraatiota, tehden ohjelmistotuotannosta nopeampaa.

Kuviossa 9 nähdään SwiftUI:lla ja Xcodella kehitetty Hello World -sovellus. Samassa kuviossa nähdään myös SwiftUI:n syntaksia, projektin rakenne ja muutosten mukaan päivittyvä esikatselu käyttöliittymälle.



Kuvio 9. Hello World -projekti SwiftUI:lla ja Xcodella.

3.2 Kotlin

Kotlin on staattisesti tyypitetty yleiskäyttöinen avoimen lähdekoodin ohjelmointikieli, joka on ohjelmistoyritys JetBrains:n kehittämä ja vuonna 2011 julkaisema. Kotlin on myös Googlen suosittelema tekniikka natiivien Android-sovellusten kehittämiseen, ohittaen Javan vuonna 2019 (TechCrunch 2019). Google siirtyi tukemaan Kotlinia ensisijaisesti sen luettavuuden ja tiivistetymmän rakenteen, Null-arvojen turvallisemman käsittelyn, Javan kanssa 100 prosenttisen yhteensopivuuden ja helppojen asynkronisten kutsujen käsittelyn vuoksi (Android Developer 2020c).

Kotlin on myös alustariippumaton kieli, sillä sen Java-yhteensopivuuden vuoksi Kotlinia on mahdollista kääntää jokaiselle Java-virtuaalikoneita tukevalle alustalle. JVM-kääntämisen lisäksi Kotlin tukee myös Javascriptiksi ja natiiviksi tavukoodiksi kääntämistä (Kotlin 2020). Varsinkin natiiviksi tavukoodiksi kääntyvä Kotlin Native on tämän tutkielman kannalta alustariippumaton kieli, joka vastaa tulkillista lähestymistapaa. Kotlin Native vaatii kuitenkin merkittäviä alustakohtaisia muutoksia, sillä vaikka ydinominaisuudet kääntyvät suoraan natiiviksi tavukoodiksi, tulee käyttöliittymät kehittää erikseen alustakohtaisilla ohjelmointikielillä (Kotlin 2020). Esimerkiksi Applen iOS:n tapauksessa tämä tarkoittaisi käyttöliittymän kehittämistä SwiftUI:lla, ja muun sovelluslogiikan kehittämistä Kotlin Nativella.

Kotlinille on lisäksi juuri julkaistu Kotlin Multiplatform Mobile, joka on arkkitehtuurimalli alustariippumattomien sovellusten rakentamiseen Kotlin Nativea hyödyntäen (JetBrains Blog 2020). Kotlin Multiplatform helpottaa mm. eri alustojen yhteisen koodin hallitsemista ja testaamista. Esimerkiksi suoratoistopalvelu Netflix on ottanut Kotlin Multiplatformin käyttöönsä iOS- ja Android-sovelluksissaan ja yli 50 prosenttia heidän sovelluskoodistaan on alustariippumatonta (Netflix Technology Blog 2020).

Sen alustariippumattomista ominaisuuksista huolimatta Kotlinia käytetään tässä tutkielmassa vain natiivin Android-testisovelluksen kehittämiseen.

3.2.1 Android Studio

Android-sovellusten kehittämiseen Kotlinilla Google suosittelee Android Studio IDE:a. Android Studio perustuu paljolti JetBrains:n kehittämään IntelliJ IDEA:n, joka on suosittu IDE var-

sinkin Java-kehityksen parissa. Kotlin-sovelluksiin Android Studio tarjoaa mm. nopean ja ominaisuuspainotteisen Android-emulaattorin, yleisen kehitysympäristön eri Android-versioille, suuren määrän erilaisia työkaluja suorituskyvyn arvioimiseen ja testaamiseen, sekä integraation tunnettujen versiohallintatarjoajien ja Google Cloud -alustan kanssa (Android Developer 2020b).

Applen Xcoden tavoin myös Android Studio tarjoaa erilaisia aloituspohjia erityyppisille projekteille. Nämä aloituspohjat vaihtelevat sovelluksen tyylin, mutta myös kohdealustan mukaan (esimerkiksi puhelin, tabletti tai äly-tv). Vastaavasti myös sovellusten käyttöliittymiä voi käytetystä tekniikasta riippuen kehittää joko pelkällä koodilla tai graafisella käyttöliittymällä.

3.2.2 Natiivien Android-sovellusten kehittäminen Kotlinilla

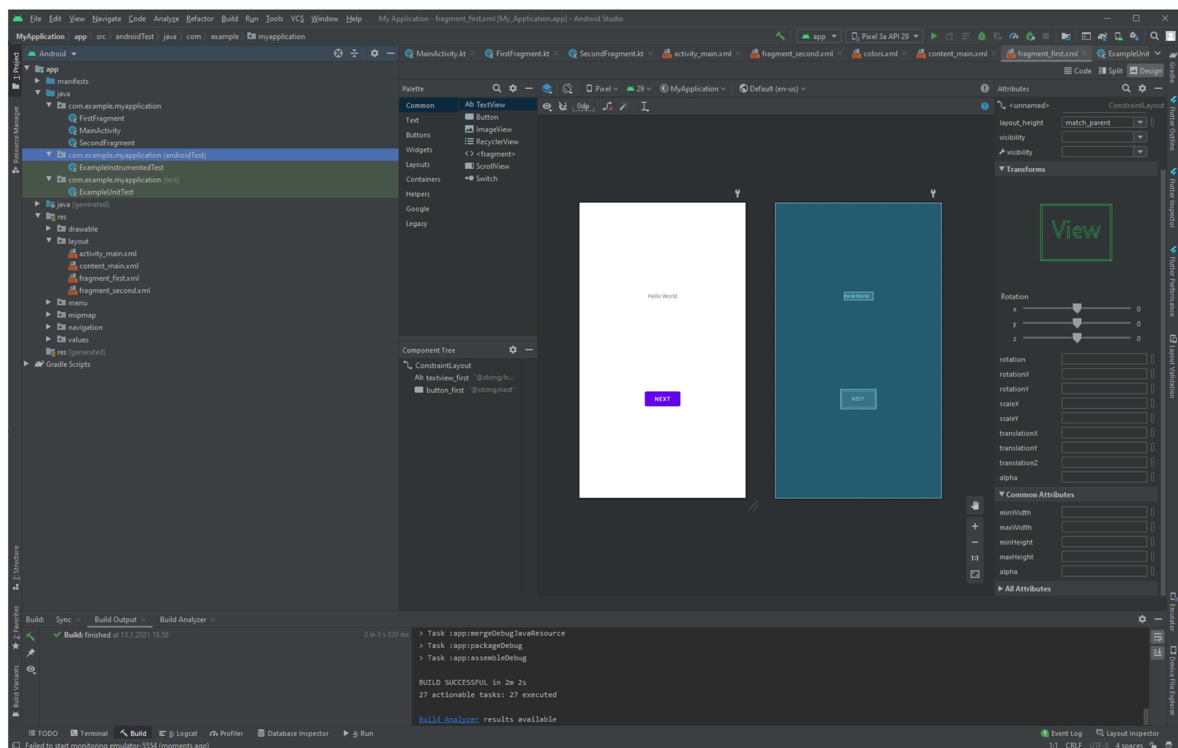
Kotlin-sovelluksen rakenne jakautuu kolmelle ylätasolle, joita ovat: sovelluslogiikka, käyttöliittymä ja testit. Mikäli sovellus kehitetään Android Studiolla, saadaan näille tasoille valmiita pohjia heti projektin alustuksen yhteydessä. Sovellusta kehittäessä Kotlin tarjoaa Swiftin tavoin hot-reload-ominaisuuden, joka mahdollistaa muutoksen näkymisen emulaattorissa heti, kun se tallennetaan. Jotkin muutokset voivat kuitenkin vaatia koko sovelluksen uudelleenkäynnistämisen ja esimerkiksi sovelluksen testejä ei voida ajaa sovelluksen kanssa rinnakkain (Kotlin 2020).

Kotlin ei vielä tarjoa valmista ohjelmistokirjastoa käyttöliittymien kehittämiseen, jonka vuoksi Kotlinilla kehitetyt Android-sovellukset hyödyntävät pääasiassa Androidin JavaFX-pohjaista käyttöliittymäkirjastoa. Käyttöliittymä kehitetään XML-tiedostoilla, joita voi muokata joko suoraan tai graafisilla työkaluilla (Android Developer 2020a).

JetBrains on kuitenkin kehittämässä Jetpack Compose-käyttöliittymäkirjastoa, jonka on jatkossa tarkoitus korvata vanha Android UI-rakenne. Jetpack Compose on tällä hetkellä betatestauksessa ja se tarjoaa deklaratiivisen tavan kehittää käyttöliittymiä, joka on hyvin samankaltainen esimerkiksi Flutterin kanssa (Android Developer 2020d).

Kuviossa 10 nähdään Kotlinilla ja Android Studiolla kehitetty Hello World -sovellus. Swiftin vastaavan tavoin kuviossa nähdään myös projektin rakennetta ja UI:n kehittämiseen tarkoi-

tettu graafinen työkalu.



Kuvio 10. Hello World -projekti Kotlinilla ja Android Studiolla.

3.3 React-Native

React-Native on avoimen lähdekoodin Javascript-ohjelmistokirjasto, jonka Facebook Inc. kehitti alun perin alustariippumattomia mobiilisovelluksia varten. Ajan kanssa React-Native on kuitenkin laajentunut tukemaan myös erilaisia työpöytäsovellusympäristöjä, joskin näiden kehitys ei ole samalla tasolla mobiiliympäristöihin verrattuna (React-Native 2020d). React-Native on myös varsin samankaltainen Facebookin aikaisemmin kehittämän React.js-ohjelmistokirjaston kanssa ja ne erottaakin toisistaan pääasiassa se, että koska React.js on tarkoitettu vain web-kehitykseen, ei sillä ole tulkkia, joka mahdollistaisi eri natiivien laiterajapintojen käytön (Eisenman 2015).

React-Native kehitys kuuluu siis aikaisemmin kappaleessa 2.1.4 mainittuun tulkilliseen lähestymistapaan, joka mahdollistaa React-Native-sovelluksille hyvän suorituskyvyn, natiivia sovellusta vastaavan ulkoasun ja pääsyyn natiiveihin sovellusrajapintoihin (Biørn-Hansen

ym. 2020; Biørn-Hansen, Grønli ja Ghinea 2018) . Koska React-Native perustuu Javascriptiin, jakaa se osan Javascriptin ominaisuuksista kuten dynaamisen tyypityksen. Kehittäjä voi kuitenkin halutessaan käyttää kehityksessä Typescriptiä, joka mahdollistaa tarkemman tyypityksen (React-Native 2021b).

React-Nativella kehitetyn sovelluksen käyttöliittymä ja sovelluslogiikka ajetaan automaattisesti eri säikeissä, joka mahdollistaa tasaisen suorituskyvyn taustalla pyörivistä logiikasta huolimatta, parantaen loppukäyttäjän kokemusta (React-Native 2020c). Käyttöliittymän päivitykset tapahtuvat reaktiivisesti, eli kun jokin käyttöliittymään vaikuttava arvo muuttuu, päivittyy käyttöliittymä automaattisesti. Tämä käyttöliittymän elämänkaari on jälleen hyvin samankaltainen kuin React.js:ssä ja suurimpana erona on, että React-Native päivittää käyttöliittymän kohdeympäristön UI-komponenteilla, kun taas React.js luo sen HTML- ja CSS-tiedostojen perusteella (Eisenman 2015).

Seuraavassa kappaleessa tutustutaan tarkemmin React-Nativella kehitettävien sovellusten toimintaan.

3.3.1 Sovelluskehitys React-Nativella

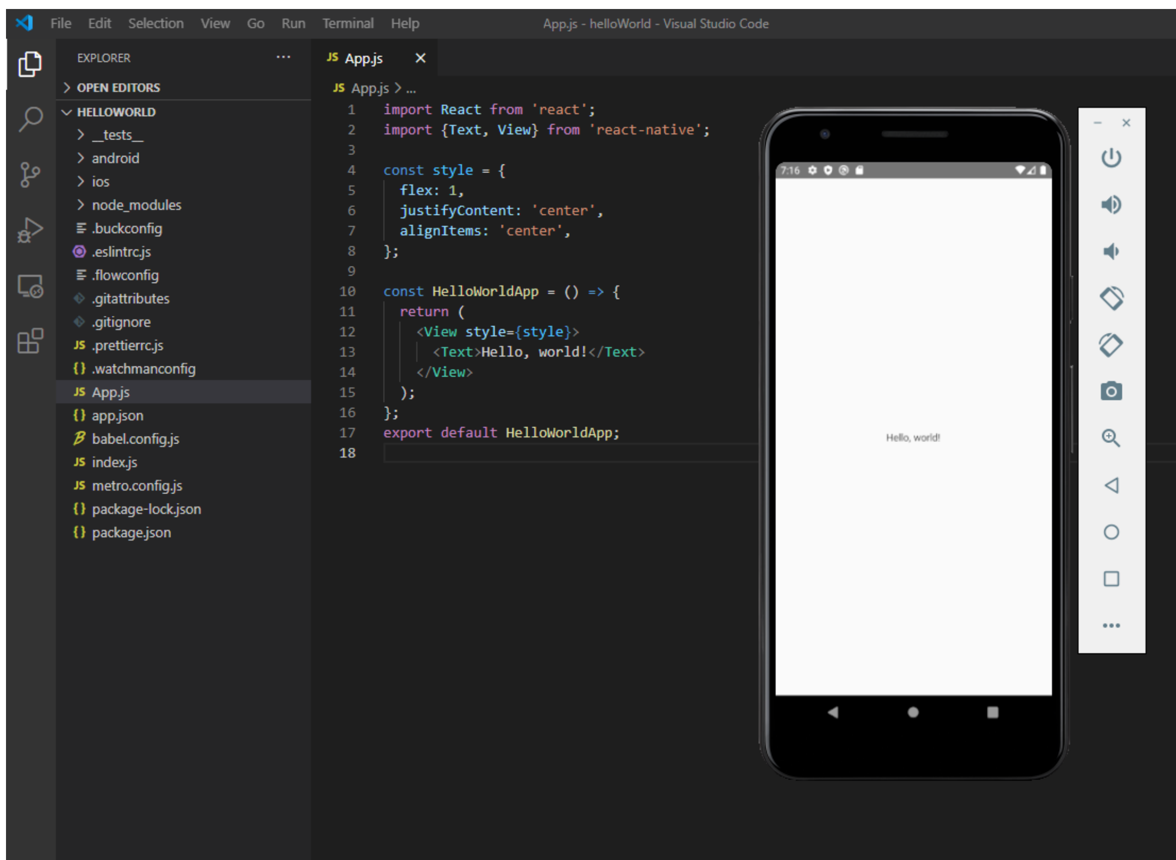
Koska React-Native on niin samankaltainen React.js:n kanssa, on se selkeä valinta kehittäjälle, joka haluaisi kehittää mobiilisovelluksia ja omaa aikaisempaa kokemusta React.js:stä. Tämä tekee React-Nativesta hyvän valinnan myös yrityksille, sillä on suuri todennäköisyys, että web-kehittäjät pystyvät oppimaan ja omaksumaan React-Nativella kehittämisen nopeasti. Varsinkin jos verrataan tilanteeseen, jossa pitäisi alkaa kehittämään usealle alustalle samaa sovellusta, eroavilla natiiveilla kielillä.

React-Nativen kehitys on myös melko joustavaa, sillä vaikka kehittäjä tarvitseekin Android Studion tai Xcoden sovelluksen ajon simuloimista varten, voi React-Nativea kehittää melkein millä tahansa suositulla IDE:lla (React-Native 2020c). Tämä voi kuitenkin heikentää tarjolla olevien debug-työkalujen määrää. Swiftin ja Kotlinin tavoin myös React-Native tarjoaa niin sanotun hot-reload-toiminnallisuuden, jonka pitäisi päivittää emulaattorissa ajettava sovellus sekunneissa siitä, kun muutos tehdään.

Kehitettävän sovelluksen ulkoasu ja toimintalogiikka ovat React-Nativessa Kotlinista ja Swif-

tistä poiketen samassa komponentissa. Kaikki React-Nativen toiminnallisuus onkin komponenttien sisällä, jotka voivat myös sisältää muita komponentteja (Eisenman 2015). Komponentit rakentuvat yleensä kahdesta osasta, joita ovat näkymä ja sitä ohjaava logiikka. Koska eri kohdealustojen erot määritetään yleensä näkymässä, on koodin uudelleenkäytön kannalta olennaista, että koodin logiikasta pyritään tekemään mahdollisimman alustariippumatonta (React-Native 2020b). Komponenteissa määritetty ulkoasu ja toimintalogiikka siirtyvät sitten tulkille, joka ohjaa käännetyn tavukoodin, kappaleessa 2.1.4 mainittuja siltoja pitkin natiiveille rajapinnoille, mahdollistaen natiivien komponenttien käytön (Eisenman 2015).

Komponenttien lisäksi React-Native-projektiin kuuluu laitekohtaiset asetukset (iOS ja Android), joita voidaan hyödyntää, mikäli halutaan päästä suoraan käsiksi natiiveihin laiterajapintoihin. Tämä projektirakenne, Hello World -sovellus ja sen luova komponentti näkymineen kuvataan kuviossa 11.



Kuvio 11. Hello World -projekti React-Nativella.

Merkittävä ero React-Nativen ja aiemmissa kappaleissa mainittujen natiivien kielien välillä, on sen kolmannen osapuolten kirjastojen käyttö. React-Nativelle löytyy tuhansia kirjastoja ja muunneltuja komponentteja, mutta osa näistä kirjastoista toimii vain tietyllä kohdealustalla. Tämä tarkoittaa sitä, että vaikka React-Native itsessään on alustariippumaton, vaatii kolmannen osapuolen kirjastojen käyttö tarkkuutta (Eisenman 2015). Tämä kolmannen osapuolien kirjastojen käyttö korostuu myös siksi, että React-Nativen mukana tulevat komponentit ja niiden toiminnallisuus ovat yksinkertaisempia verrattuna sen natiiveihin vastakappaleisiin.

3.3.2 Haasteita

Vaikka React-Native tarjoaakin monia merkittäviä hyötyjä alustariippumattomuudellaan ja tulkillisen lähestymistavan mahdollistamalla käyttökokemuksella, on tärkeää tunnistaa siihen liittyviä riskejä ennen mahdollisen projektin aloittamista. Yksi suurimmista riskeistä onkin React-Nativen uutuus, sillä se saavutti tuen iOS:lle ja Androidille vasta vuonna 2015 (Eisenman 2015). Tämä voi johtaa siihen, että mahdollinen projekti vaatisi useita riippuvuuksia kolmannen osapuolen kirjastoihin, koska React-Native ei oletuksena vielä tarjoaisi vaadittuja ominaisuuksia. Kun otetaan huomioon, että kolmannen osapuolen Javascript-kirjastoilla on suurempi riski olla vanhentuneita, huonosti toteutettuja, lisenssikelvottomia tai turvattomia (Decan, Mens ja Constantinou 2018), on tämä merkittävä riski sovelluksen toiminnallisuuden toteutuksen näkökulmasta.

Lisäksi, koska React-Native käyttää natiiveja komponentteja, on se samalla riippuvainen niiden olemassaolosta (React-Native 2020b). Tilanteessa, jossa natiivin komponentin rajapinta muuttuisi, olisi mahdollista, että kehitetty sovellus muuttuisi tai lakkaisi toimimasta vain koska käyttäjä päivitti käyttöjärjestelmäänsä. Tämä tarkoittaa, että sovellusta tulee testata tehokkaammin, sillä mahdollisia kohdealustoja ja niiden eri versioita on nyt huomattavasti enemmän. Kun otetaan vielä huomioon, että suurin määrä eriävästä koodista on komponenttien näkymissä, voi koodin luettavuus kärsiä, mikäli eri käyttöjärjestelmäversiot luovat lisää poikkeuksia kohdealustojen ohelle.

React-Native myy itseään suorituskyvyllä, joka näyttäisi olevan lähempänä natiivisti kehitettyä sovellusta kuin monet muut alustariippumattomat vaihtoehdot (Biørn-Hansen ym. 2020).

Tämä suorituskyky edellyttää kuitenkin ajonaikaisen tulkin käyttöä, joka yleensä vaatii enemmän resursseja. Tutkimuksessaan Biørn-Hansen ym. (2020) havaitsivat, että React-Native käytti keskimäärin enemmän prosessorin resursseja kuin yksikään muista tutkimuksen alustariippumattomista tekniikoista. Toisaalta tutkimuksesta ei selviä miten paljon tämä ero vaikuttaisi esimerkiksi laitteen akkukestoon ja suorituskyky vs. akkukesto voi olla tilannekohtainen kysymys.

3.4 Flutter

Flutter on avoimen lähdekoodin kehitysalusta alustariippumattomien sovellusten kehittämiseen varsinkin iOS- ja Android-käyttöjärjestelmille, joskin React-Nativen tavoin myös Flutter on viime aikoina laajentunut tukemaan erilaisia työpöytäympäristöjä (Flutter 2020b). Flutter on pääasiassa Googlen kehittämä ja se on yksi uusimmista alustariippumattomien sovellusten kehitysalustoista, sillä ensimmäinen vakaa Flutter versio Android- ja iOS-tuella julkaistiin vasta vuoden 2018 joulukuussa.

Flutter eroaa monista muista alustariippumattomista lähestymistavoista siinä, että sen koodipohja ei perustu johonkin web-tekniikkaan. Sen sijaan Flutter käyttää C:tä, C++:a ja sitä varten luotua Dart-kieltä, joka on oliosuuntautunut, luokkapohjainen, C-syntaksia mukaileva kieli ja on tarkoitettu varsinkin käyttöliittymien luomista varten (Dart 2020). Dart voidaan kääntää joko natiiviksi tavukoodiksi tai Javascriptiksi ja kääntäminen tapahtuu yleensä AOT:na (*engl.* Ahead of Time compiling), joka mahdollistaa koodin optimoinnin ennen suorittamista, parantaen suorituskykyä. Dart tukee myös JIT-kääntämistä (*engl.* Just in Time compiling), jossa koodin kääntäminen natiiviksi tehdään kokonaan vasta suoritusvaiheessa. AOT:hen verrattuna JIT tarjoaa huonompaa suorituskykyä, mutta kääntäminen on nopeampaa, joka on hyödyllistä Flutter-sovelluksen kehitysvaiheessa kun koodi muuttuu jatkuvasti.

Kuten kappaleessa 2.1.5 mainittiin, kuuluu Flutter niin sanottuun widget-perusteiseen lähestymistapaan. Tämä tarkoittaa, että Flutterilla luotavat käyttöliittymät koostuvat useista widget-komponenteista, jotka voivat myös sisältää toisia komponentteja. Flutterin lähestymistavassa kaikki kohdealustan natiivit käyttöliittymäkomponentit on korvattu omilla widget-komponenteilla, jotka Flutter piirtää käyttäen omaa Skia-grafiikkamoottoriaan (Flutter 2020b).

Tämä oman grafiikkamoottorin käyttö tekee Flutterista varsin poikkeuksellisen, sillä yksikään käyttöliittymän komponenteista ei ole “oikea” natiivi komponentti, mutta ne on luotu mahdollisimman tarkasti alkuperäisten komponenttien avulla, tehden erosta hankalasti havaittavan.

3.4.1 Sovelluskehitys Flutterilla

Flutterin syntaksi muistuttaa paljolti SwiftUI:n tai JetPack Compose:n syntaksia. Koodi on kuvailevaa ja pyrkii mahdollistamaan käyttöliittymien nopean luomisen ilman suurta määrää ylimääräistä koodia. Tämä web-teknologioista poikkeava lähestymistapa voi tosin rajoittaa mm. yrityksiltä löytyvää valmista osaamista, esimerkiksi React-Nativeen verrattuna. Toisaalta Flutter on varsin helppo oppia ja esimerkiksi käyttöliittymän ulkoasun muokkaamiseen tarvittava osaaminen on hyvin samankaltaista web-kehityksen kanssa (Flutter 2020b). Flutter tarjoaa myös dokumentaatiossaan käteviä siirtymäohjeita eri kokemusta omaaville kehittäjille, helpottaen aikaisemman kokemuksen hyödyntämistä (Flutter 2020c).

Itse Flutter-sovelluksen kehittämisen voi tehdä melkein millä tahansa IDE:lla, joka antaa kehittäjille paljon valinnanvaraa. Tosin, kuten myös React-Nativen tapauksessa, voivat debug-työkalut olla vaihtelevia käytetystä IDE:sta riippuen. React-Nativen tavoin myös Flutter vaatii joko Android Studion tai Xcoden asennetuksi, jotta kehitettävää sovellusta voidaan ajaa. Vastaavasti, myös Flutter tarjoaa hot-reload-toiminnallisuuden sovelluskehityksen nopeuttamiseksi.

Uuden Flutter-projektin rakenne sisältää valmiit kansiot ja mallit testeille sekä Dart-koodille. Mikäli kehittäjä niin haluaa, voidaan myös lisätä Kotlin- ja Swift-tuki, joka mahdollistaa natiivin koodin kirjoittamisen Dart-koodin lisäksi (Flutter 2020b). Tätä tukea ei kuitenkaan ole pakko lisätä ja sen poisjättäminen voi vähentää valmiin sovelluspaketin kokoa.

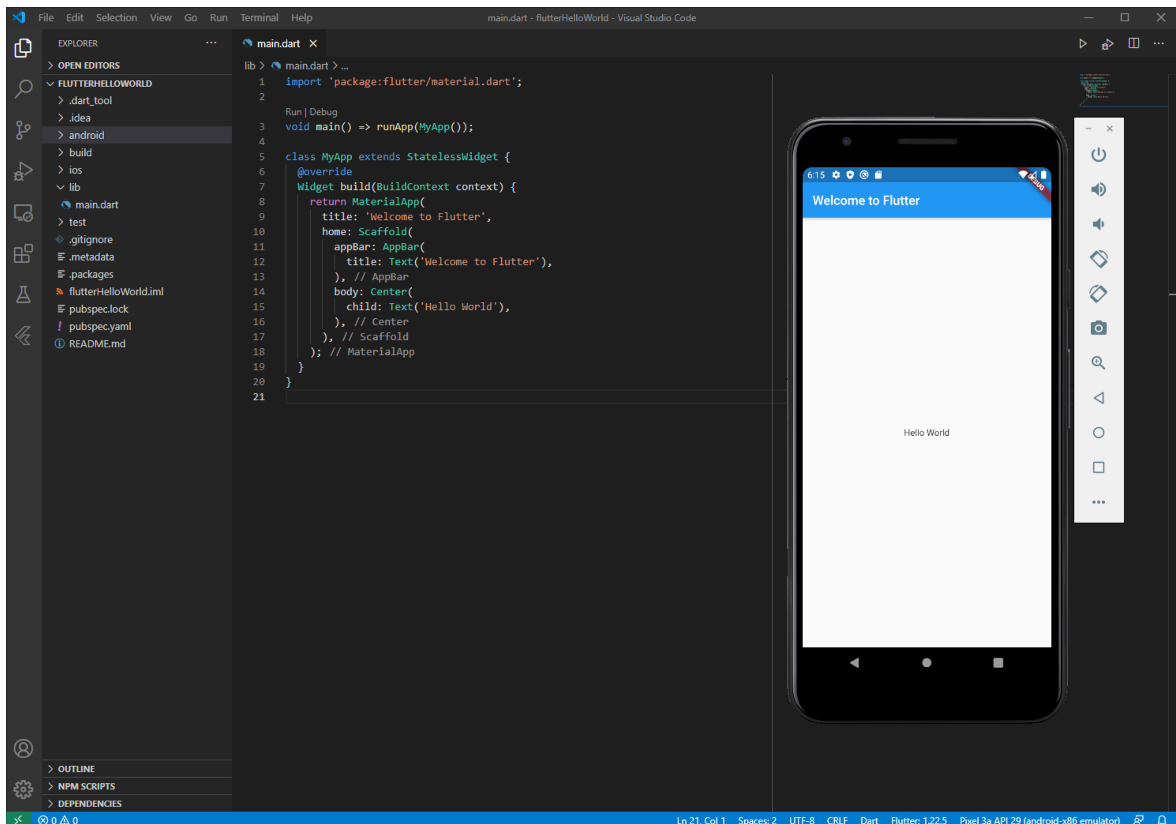
Flutterissa melkein kaikki käyttöliittymän osat ovat widgettejä ja siksi suurin osa kirjoitettavasta koodista onkin erilaisten widgettien luomista. Sovelluksen käyttöliittymän voi siis ajatella puurakenteena, joka koostuu useista sisäkkäisistä widgeteistä (Flutter 2020b). Jokaisella widgetillä on build-metodi, jossa määritetty käyttöliittymä lopulta piirretään grafiikkamoottorilla. Widget-tyyppejä on sekä tilattomia että tilallisia, joka tekee koodin uudelleen-

käytöstä ja erittäin muunneltujen widgettien luomisesta tehokasta. Itse widgetillä on myös oma reaktiivinen elämänkaarensa, joka toimii jälleen hyvin samankaltaisesti React-Nativen kanssa (Flutter 2020b).

Flutter tarjoaa joukon valmiita käyttöliittymäkomponentteja, mutta mikäli kehittäjä ei löydä niistä haluamaansa, voi hän käyttää myös suurta joukkoa kolmansien osapuolien valmistamia komponentteja ja kirjastoja (Payne 2019). Google ylläpitää yhtä suosituimmista tietokannoista ("Pub.dev" 2021), joka mahdollistaa edellä mainittujen jakamisen. Flutterin kehittäjät poimivat myös palveluun ladattujen komponenttien joukoista suosikkejaan, joka helpottaa laadukkaiden ja turvallisten kolmannen osapuolen komponenttien käyttöä.

Koodin uudelleenkäyttö Flutterissa on riippuvaista käyttöliittymistä ja niiden kohdealustojen UI-komponenttien määrästä. Eri alustojen käyttöliittymät voivat käyttää omia komponenttejaan (Payne 2019), mutta käyttöliittymien logiikan voi jakaa molempien alustojen kanssa. Tämä suosii jälleen yleistetyn sovelluslogiikan kehittämistä, joskin React-Nativesta eroten voidaan molemmille alustoille nyt tehdä täysin identtiset käyttöliittymät käyttämättä yhtäkään natiivia UI-komponenttia (Payne 2019). Tämä on mahdollista siksi, että Flutter koostuu omista UI-komponenteistaan, jotka voivat näyttää samalta eri alustojen natiivien komponenttien kanssa, mutta toimivat silti alustasta riippumatta (Biørn-Hansen ym. 2020).

Alla olevasta kuviosta 12 voidaan havaita Flutter-projektin rakenne ja esimerkki Hello World -sovelluksesta.



Kuvio 12. Hello World -projekti Flutterilla.

3.4.2 Haasteita

Kuten edellisestä kappaleesta voidaan havaita, on Flutter joiltakin osin joustavampi alustariippumattoman kehityksen suhteen verrattuna React-Nativeen. Tästä huolimatta myös Flutterilla on haasteita, jotka tulee ottaa huomioon ennen sovelluskehityksen aloittamista.

Kuten React-Native, myös Flutter kärsii uutuudestaan ja vaatii siksi useissa tapauksissa kolmannen osapuolen kirjastojen käyttöä. Varsinkin alkuun Flutter vaati enemmän työtä iOS-puolen kanssa, sillä Google priorisoi oman Android-alustansa ominaisuuksia. Tämä tilanne on tosin parantunut huomattavasti ja esimerkiksi Flutterin 1.22 päivitys toi tuen uusimmille Android- ja iOS-versioille samanaikaisesti (Flutter 2020a).

Flutterin uutuuden vuoksi myös yrityksille voi olla hankalaa löytää korkeamman tai edes yleisen tason osaamista. Kuten edellisessä kappaleessa mainittiin, vaikuttaa osaamisen löytymiseen myös se, että React-Nativella kehitys muistuttaa paljolti web-kehitystä, ja vaikka

Dart käyttääkin monia web-kehityksen konsepteja, on se silti varsin erilainen. Lisäksi Dart itsessään on varsin uusi ja kokee yhä suuria muutoksia, jotka voivat hankaloittaa koodin ylläpitämistä. On myös otettava huomioon, että Flutter on yhä vahvasti riippuvainen Googlen tuesta. Google on viime aikoina aggressiivisesti lopettanut erilaisten sivuprojektien tukemisen (“Killed by Google” 2021) ja yrityksenä voi olla haastava päätös valita tekniikka, jonka tulevaisuus ei ole varma.

Koska Flutterissa Dart-koodilla toteutetaan käyttöliittymä ja sovelluslogiikka, pitäisi varsinkin sovelluskehityksen alkuvaiheen olla nopeampaa ja helpompaa (Freitas 2019; Payne 2019) . Tämä voi kuitenkin vaikeuttaa monimutkaisten käyttöliittymien rakentamista ja myöhempää ylläpitämistä, sillä nyt logiikka ja käyttöliittymä eivät ole yhtä helposti erotettavissa. Flutter ei myöskään tarjoa omaa graafista työkalua käyttöliittymän rakentamiseen, joka rajoittaa käyttöliittymäkehityksen vain henkilöille, joilla on Dart osaamista.

Edellisessä kappaleessa mainittiin, että Flutter käyttää käyttöliittymissään kopioita aidoista natiiveista UI-komponenteista. Tämä antaa sille uniikin kyvyn pysyä samanlaisena, vaikka “oikea” natiivi UI-komponentti muuttuisi järjestelmäpäivityksen vuoksi. Tämän seurauksena Flutter ei kuitenkaan pysty heti hyödyntämään mahdollisia uusia ominaisuuksia, joita järjestelmäpäivitys voisi lisätä. Mikäli kehittäjällä on osaamista, voi hän toteuttaa puuttuvia rajapintoja natiivilla koodilla (Flutter 2021), mutta muussa tapauksessa ainut vaihtoehto on odottaa, kunnes Flutter saa päivityksen tai joku kolmas osapuoli kehittää vastaavan komponentin. Tämä taas voi altistaa Flutterin samoille ongelmille kuin React-Nativen tapauksessa, joskin Flutterin kolmannen osapuolen kirjastojen laadusta ei vielä ole tutkimusta. Lisäksi, koska Flutter ei kolmannen osapuolen kirjastoista huolimatta käytä “oikeita” natiiveja UI-komponentteja, ei sovelluksen käyttökokemus välttämättä kykene vastaamaan täysin natiivin sovelluksen käyttökokemusta.

Viimeiseksi, koska Flutter tarvitsee jokaista käyttöliittymäkomponenttia varten oman UI-komponentin, voi sovelluspaketeista tulla suurempia, kuin muilla verrattavilla teknologiolla. Esimerkiksi Biørn-Hansen ym. (2020) havaitsivat tutkimuksessaan Flutterilla kehitetyn sovelluksen kokonaiskooksi 32,8 megatavua, kun taas React-Nativella kehitetyn sovelluksen koko oli 9,7 megatavua. Flutter on kuitenkin panostanut tähän ja versiossa 1.22 on luvattu parannuksia sovelluspakettien kokoihin (Flutter 2020a).

4 Tutkimusmetodologia

Tässä luvussa käydään läpi tutkielmassa käytettäviä tutkimusmenetelmiä, niiden valintaan käytettyjä perusteluja, heikkouksia ja miten niitä sovelletaan tämän tutkielman kannalta. Tutkimusmenetelmällä tarkoitetaan tutkielman kontekstissa lähestymistapoja ja käytänteitä, joita tutkimuksen tuottamiseen vaaditaan.

Tutkimusmenetelmien valinta on usein riippuvainen teetettävän tutkimuksen tyypistä ja tavoitteista. Tämän tutkielman tutkimus voidaan luokitella konstruktiiviseksi tutkimukseksi, jolla pyritään vastaamaan kappaleessa 1.1 määriteltyihin tutkimuskysymyksiin (Hassani 2017). Toteutettava tutkimus voidaan sen sijaan määritellä tavoitteiden osalta kuvailevaksi ja tutkivaksi.

Kuvailevalla tutkimuksella pyritään kuvaamaan olemassa olevan tutkimuksen perusteella löytyneitä ja odotettavia tutkimustuloksia. Tämän jälkeen siirrytään tutkivaan osuuteen, jossa aikaisemmin löydettyjä oletuksia voidaan todeta ja vertailla sekä havaita uusia haasteita, joita olemassa oleva tutkimus ei ole vielä ottanut huomioon (Hassani 2017).

Seuraavissa luvuissa käydään lävitse tutkielman kolme eri tutkimusmenetelmää, joita ovat: kirjallisuuskatsaus, kokeellinen tutkimus ja vertailututkimus.

4.1 Kirjallisuuskatsaus

Kirjallisuuskatsauksen tarkoituksena tässä tutkielmassa on tuottaa taustatietoa ja dataa vertailututkimusta varten. Tutkielmaa varten harkittiin myös systemaattista kirjallisuuskatsausta, mutta se nähtiin resurssien ja tutkielman tavoitteiden kannalta tehottomana. Systemaattinen kirjallisuuskatsaus ja sen tulokset olisivat perustuneet kokonaan löydettyyn kirjallisuuteen (Petersen ym. 2008), jota varsinkin uudemmista alustariippumattomista työkaluista voi löytyä heikosti. Kevyempi kirjallisuuskatsaus ja kokeellinen vertailututkimus ratkaisevat tämän ongelman, sillä tutkimustulokset eivät ole kokonaan riippuvaisia olemassa olevasta kirjallisuudesta.

Kirjallisuuskatsaus tehtiin käyttäen IEEE:n, ACM:n ja Springer:n tarjoamia tietokantoja, joi-

den lisäksi hyödynnettiin Google Scholar-hakukonetta. Kirjallisuuskatsaukseen valitut lähteet pyrittiin valitsemaan vertaisarvioitujen tutkimusten ja luotettavien julkaisijoiden joukosta. Löytyneet tieteelliset lähteet pyrittiin lisäksi pitämään mahdollisimman tuoreina, sillä tutkimusaiheen nopean kehityksen vuoksi on mahdollista, että lähteet antaisivat vanhentunutta tai väärää tietoa. Kirjallisuuskatsauksen hakusanoina toimivat:

- Mobile development
- Cross-platform development issues
- Flutter
- React-Native
- Android/iOS development
- Native development
- Kotlin
- Swift

Näitä hakusanoja käytettiin sekä erikseen että yhdistelminä. Tieteellisten artikkeleiden lisäksi kirjallisuuskatsauksessa hyödynnettiin myös eri työkaluista toteutettua kirjallisuutta ja tarvittaessa eri työkalujen teknistä dokumentaatiota. Tämä oli olennaista, sillä kirjallisuuskatsauksen suurimmaksi haasteeksi osoittautui varsinkin Flutteriin kohdistuvan tieteellisen tutkimuksen löytäminen. Tämä johtui erityisesti Flutterin uutuudesta, jonka vuoksi Flutter onkin yksi tämän tutkielman tärkeimmistä tutkimuskohteista.

Tutkittavasta tekniikasta huolimatta, suurimman arvon saivat yleensä tieteelliset lähteet ja tekninen dokumentaatio toimi toissijaisena lähteenä. Tätä sääntöä kuitenkin sovellettiin niin, että valitun tekniikan ominaisuuksien puolesta tekninen dokumentaatio oli tärkeämpää, koska se oli usein merkittävästi ajantasaisempaa verrattuna olemassa olevaan tutkimukseen.

4.2 Kokeellinen tutkimus

Kokeellisen tutkimuksen määritelmä tietotekniikassa on kiistelty, mutta yhdeksi määritelmäksi sille voidaan antaa “ei itsestään selvän sovelluksen tai laitteiston rakentaminen, tutkiminen ja/tai käytön havainnollistaminen” (oma suomennos, Hassani 2017). Tämä on varsin monitulkintainen ja laaja määritelmä, mutta se antaa kuvan siitä mihin kokeellisella tutki-

muksella pyritään vastaamaan. Onnistuneen kokeellisen tutkimuksen tuloksien pitäisi myös olla laajoja ja niiden validiteetin pitäisi olla helposti määritettävissä (Moher ja Schneider 1982).

Moher ja Schneider (1982) jaottelevat kokeellisen tutkimuksen viiteen eri osa-alueeseen, joita ovat kokeellinen suunnittelu, kohteen ominaisuuksien havaitseminen, havaintoympäristön laatiminen, suorituskyvyn vaatimusten laatiminen ja suorituskyvyn mittaaminen. Tässä tutkielmassa pyritään soveltamaan useampaa näistä osa-alueista.

4.2.1 Kokeellinen tutkimus tämän tutkielman kannalta

Tässä tutkielmassa kokeellista tutkimusta hyödynnetään testisovelluksen suunnitteluun, kehittämiseen ja arviointiin. Testisovellus kehitetään jokaisella neljästä tutkittavasta tekniikasta, jonka jälkeen kehityskokemusta ja eri työkaluilla tuotettujen sovellusten arvioinnista kerättyä dataa hyödynnetään tutkielman päätutkimusmenetelmän, eli vertailututkimuksen avulla.

Testisovelluksen rakenteeseen ja sen suunnitteluun perehdytään tarkemmin kappaleessa 5.1, mutta sovelluksen ominaisuuksien suunnittelussa hyödynnetään pääasiassa keskimääräistä joukkoa ominaisuuksia, joita yleisillä sovelluksilla nykypäivänä on. Näillä ominaisuuksilla voidaan tuottaa sovellus, josta kerättävä data vastaa tarkemmin oikean maailman käyttötapausta. Sovelluksen monimutkaisuus rajataan kuitenkin niin, että se on mahdollista tuottaa neljällä eri tekniikalla ilman kohtuutonta riskiä siitä, että kehittäjän taidot voivat merkittävästi vaikuttaa tutkimustuloksiin. Sovelluksen tulee siis olla tarpeeksi yksinkertainen, mutta samalla monipuolinen.

Kokeellisen tutkimuksen hyödyntäminen ei ole tutkielman tutkimuskysymysten kannalta välttämätöntä, mutta se tarjoaa arvokasta dataa varsinkin uusimpien alustariippumattomien tekniikoiden kannalta, joihin kirjallisuuskatsaus ei välttämättä vielä pysty antamaan tarpeeksi tietoa. Lisäksi, koska verrattavat alustariippumattomat työkalut ovat varsin uusia, on erityisesti niiden suorituskyky saattanut merkittävästi muuttua edelliseen tutkimukseen verrattuna. Tämänkaltaisen datan keräämiseen testisovellus on hyvä vaihtoehto, kunhan erilaisia suorituskykyyn perustuvia arvoja painotetaan yleisen sovelluksen näkökulmasta. Testisovel-

lus tuottaa myös lisäarvoa, koska se mahdollistaa tutkimustulosten uusimisen, mikäli tutkimuskohteet kokisivat merkittäviä muutoksia.

Seuraavassa kappaleessa käydään lävitse tutkielman päätutkimusmenetelmä eli vertailututkimus.

4.3 Vertailututkimus

Vertailututkimus mahdollistaa eri tutkimuskohteiden vertailun erilaisten ennalta määriteltyjen kriteerien avulla. Tämä on tutkielman tutkimuskysymysten kannalta mielekästä, mutta myös jatkotutkimuksen kannalta arvokasta, sillä mikäli arviointikriteerit on laadukkaasti määriteltyjä, pystytään sama tutkimus toteuttamaan samoilla kriteereillä tarvittaessa uudelleen (Esser ja Vliegenthart 2017).

Esser ja Vliegenthart (2017) määrittelevät laadukkaalle vertailututkimukselle muutamia vaatimuksia. Ensinnäkin vertailututkimuksen tavoitteiden pitää olla tarkasti määriteltyjä ja niiden tulisi samalla toimia määräävinä tekijöinä koko tutkimuksen rakenteen kannalta. Toiseksi, eri vertailuun käytettyjen kriteerien tulee olla tarkasti määriteltyjä, perusteltuja ja niiden erilaiset kontekstit selvitettyjä. Kolmanneksi, kun eri osa-alueita verrataan, tulee vertailu suorittaa tasapuolisesti jokaista kohtaan. Tällä tarkoitetaan, että verrattavilla kohteilla tulee olla ainakin yksi sama vertailukohde keskenään. Neljänneksi, kaikkia vertailukohteita tulee verrata samoilla periaatteilla, eli vertailun logiikka ei saa muuttua vertailukohteen mukaan.

Esser ja Vliegenthart (2017) mukaan vertailututkimuksen tuloksena saadaan tietämystä vertailukohteiden samankaltaisuuksista, eroista ja konteksteista, jotka aiheuttavat poikkeamia vertailukohteiden välillä. Tämä mahdollistaa kohteiden luokittelun ja arvostelun, joka on tämän tutkielman tutkimuskysymysten kannalta oleellista.

Koska vertailututkimus on tämän tutkielman päätutkimusmenetelmä, on tärkeää, että mahdollisiin vertailututkimukselle uniikkeihin tutkimusvirheisiin on varauduttu ja että niitä osataan ehkäistä. Seuraavassa luvussa käydään läpi näitä virhetilanteita ja miten niitä ehkäistään tässä tutkielmassa.

4.3.1 Haasteet

Vertailututkimuksella on sen luonteen vuoksi uniikkeja virhetilanteita, joiden välttäminen on tärkeää luotettavan tutkimustuloksen kannalta. Vaikka nämä virhetilanteet kohdistuvat erityisesti vertailututkimukseen, ovat monet riippuvaisia myös kokeellisen osuuden oikeaoppisesta toteutuksesta. Artikkelissaan Esser ja Vliegenthart (2017) määrittelevät vertailututkimuksen haasteiksi käsitteelliset, mittaukselliset, instrumentaaliset ja otoskohtaiset virheet.

Käsitteellisellä virheellä tarkoitetaan tilannetta, jossa käsite ei välttämättä tarkoita samaa asiaa eri kulttuurien tai alojen välillä (Esser ja Vliegenthart 2017). Tämä virhetilanne on mahdollinen tutkielman kannalta, sillä esimerkiksi testisovelluksen arviointiin käytettävät kriteerit voivat helposti olla monitulkinnallisia (esimerkiksi sujuvuus tai ulkonäkö). Tähän voidaan kuitenkin varautua valitsemalla mahdollisimman selkeästi määriteltyjä kriteerejä. Riskiä pienentää myös se, että kokeen eri sovelluksia verrataan toisiaan vastaan, joka varmistaa, että kaikkia sovelluksia verrataan samalla käsitteellä.

Mittauksellisilla virheillä tarkoitetaan virheitä, jotka johtuvat mitatun arvon vääristymisestä (Esser ja Vliegenthart 2017). Vääristyminen voi johtua useista eri syistä, mutta tässä tutkielmassa varsinkin testisovelluksesta otettavat mittaukset ovat riskialttiita. Keinoja mittauksellisen virheen riskin lieventämiseen on esimerkiksi mittausten tekeminen standardoidusti ja objektiivisesti. Tällä tarkoitetaan, että jokaista sovellusta mitataan samalla tavalla ja samoissa tilanteissa. Esimerkiksi suorituskykyä mittaavien testien tulee kestää yhtä pitkään ja niillä tulee olla samat olosuhteet kuin muilla mitattavilla sovelluksilla.

Instrumentaaliset virheet ovat hyvin samankaltaisia mittauksellisten virheiden kanssa, mutta ne johtuvat pääasiassa virheellisesti käytetystä mittausprosessista tai työkalusta (Esser ja Vliegenthart 2017). Instrumentaaliset virheet tässä tutkielmassa voisivat johtua esimerkiksi suorituskyvyn mittaamiseen käytettävien työkalujen väärinkäytöstä tai niiden tulosten väärinymmärtämisestä. Toisaalta, mikäli väärinkäytös kohdistuu jokaiseen mittauskohteeseen, pitäisi virheen esiintyä jokaisessa vertailukohteessa, tehden vertailusta silti mahdollisen. Instrumentaalisilta virheiltä pyritään silti välttymään työkalujen oikeaoppisella käytöllä ja tulosten monikertaisella varmistamisella.

Otoskohtaisella virheellä tarkoitetaan tilannetta, jossa jokin mitattava kohde osuu suurem-

malla todennäköisyydellä mitattavaan satunnaisotokseen (Esser ja Vliegenthart 2017). Tämä tilanne ei ole tutkielman kannalta suuri huoli, sillä mitattavat kohteet eivät ole satunnaisia. Yksi otosvirheeksi laskettava tilanne voi kuitenkin esiintyä, mikäli testisovelluksen suorituskykyä mitattaessa eri sovellukset mitattaisiin niin peräkkäin, että edellisen sovelluksen ajaminen vaikuttaisi testattavan sovelluksen tuloksiin. Siksi on tärkeää, että mittausotokset poimitaan standardoiduilla aikaväleillä ja olosuhteilla, jotta edellisen sovelluksen vaikutus voidaan eliminoida.

4.3.2 Vertailututkimus tämän tutkielman kannalta

Tässä tutkielmassa vertailututkimus toimii primäärisenä tutkimusmenetelmänä kirjallisuuskatsauksen ja kokeellisen tutkimuksen lisäksi. Kuten aikaisemmassa luvussa 1.1 perusteltiin, tutkielman vertailukohteina toimivat neljä eri mobiilikehitystekniikkaa, joita ovat natiivien tekniikoiden osalta Kotlin ja Swift sekä alustariippumattomien tekniikoiden osalta React-Native ja Flutter. Näiden kohteiden vertailuun käytettävä data koostuu kokeessa havaituista tuloksista sekä kirjallisuuskatsauksen perusteella löytyneestä datasta.

Vaikka tässä vertailututkimuksessa pyritäänkin käyttämään hyväksi nähtyjä menetelmiä laadukkuuden ja luotettavuuden säilyttämiseksi, on tärkeää, että myös kokeellisen tutkimuksen osuus on mahdollisimman tarkasti rajattu ja toteutettu. Kokeellisen tutkimuksen testisovellus tuottaa vain osan vertailtavasta datasta, mutta väärin tuotettuna se voisi silti merkittävästi vääristää tutkimustuloksia hyvistä vertailututkimuksen periaatteista huolimatta.

Vertailuun käytettävä data tulee olemaan sekä laadullista että määrällistä, jonka lisäksi eri vertailukriteereille asetetaan eroavia painoarvoja. Nämä painoarvot ovat olennaisia siksi, että alustariippumaton kehitys saattaa vaatia valintoja eri ominaisuuksien välillä. Näillä ominaisuuksilla on erilaisia merkittävyyksiä sekä yleisen sovelluksen loppukäyttäjän että sovellusta kehittävän osapuolen kannalta, jonka vuoksi eri kriteerit pyrkivät arvioimaan yleistä sovellusta molempien sovelluskehittäjän ja loppukäyttäjän näkökulmista.

Nämä vertailuun käytettävät sovelluskehittäjän ja loppukäyttäjän kriteerit käydään tarkemmin läpi seuraavassa luvussa.

4.3.3 Kriteerit

Laadukkaiden kriteerien määrittäminen on oleellista tutkimustulosten luotettavuuden ja myös tulevaisuuden tutkimuksen helpon tuottamisen vuoksi. Eri alustariippumattomia työkaluja on mielekästä arvioida usealla eri kriteerillä, jotka voidaan jakaa karkeasti sovelluskehittäjälle ja loppukäyttäjälle merkittäviin kriteereihin.

Sovelluskehittäjälle keskeiset kriteerit arvioivat alustariippumatonta tekniikkaa kehittäjälle merkittävistä näkökulmista, joista suurin osa selviää jo ennen kehittämisen aloittamista. Nämä kriteerit arvioivat pääasiassa sovelluksen kehittämisen haasteita ja onnistumisia, mutta myös sen jatkokehityksen mahdollisuuksia. Loppukäyttäjän kriteerit sen sijaan perustuvat enemmän valmiin sovelluksen ulkonäköön ja suorituskykyyn. Alustariippumattomilla tekniikoilla tuotettua sovellusta on myös tärkeä arvioida sen käyttöliittymän samankaltaisuuden kannalta natiiviin sovellukseen verrattuna, sillä alustariippumattomuuden on tarkoitus olla loppukäyttäjälle näkymätön muutos (Biørn-Hansen ym. 2020).

Seuraavissa taulukoissa 1 ja 2 esitetään sovelluskehittäjän ja loppukäyttäjän kriteerit kuvauksineen ja painoarvoineen. Kriteereiden luomiseksi on käytetty useissa eri lähteissä mainittuja vertailukohteita ja luotuja kriteerejä vertaillaan samankaltaisella periaatteella kuin Heitkötter, Hanschke ja Majchrzak (2013) tuottamassa tutkimuksessa. Eri alustariippumattomilla tekniikoilla kehitettyjä sovelluksia arvioidaan taulukoiden kriteereillä arvosanalla yhdestä viiteen, jossa viisi tarkoittaa, että tekniikka vastaa tai ylittää natiivin tekniikan omaisuudet ja yksi, että tekniikka on huomattavasti heikompi verrattuna natiiviin sovellukseen.

Tämän lisäksi kriteereihin on lisätty painoarvot, jotka kohdistavat arvosanan yleiselle nykyaikaiselle sovellukselle, joka määritellään kappaleessa 5.1. Näiden kriteerien perusteella eri tekniikoista voidaan luoda viitekehys, joka osoittaa niiden vahvuudet ja heikkoudet sekä miten ne soveltuisivat näillä painoarvoilla kohdistetun sovelluksen kehitykseen.

Taulukko 1: Sovelluskehittäjän kriteerit alustariippumattomille työkaluille.

Kriteeri	Kuvaus	Painoarvo
Suorituskyky	Sovelluskehittäjän näkökulmasta valitun työkalun tarjoama suorituskyky määrittää kuinka monimutkaisen tai vaativan sovelluksesta pystyy tekemään.	3
Koodipohjan helppokäyttöisyys	Yksi tärkeimmistä kriteereistä sovelluskehittäjälle. Tähän kriteeriin lasketaan koodipohjan oppimisen vaativuus uudelle kehittäjälle, mahdollisen olemassa olevan koodin omaksumiseen vaadittu aika ja koodin yleinen selkeys ja loogisuus. Tämä kaikki vaikuttaa merkittävästi sovelluksen kehitykseen liittyviin kuluihin.	5
Ylläpidettävyys	Kuvastaa miten helppo tuotettua koodia on ylläpitää, miten hyvin valitulle työkalulle on saatavilla tukea ja minkälaiset tulevaisuuden näkymät työkalulla on.	4
Jatkokehitys	Miten helppo sovellusta on laajentaa tietystä tilasta eteenpäin.	2
Alustakohtaiset erot	Yksi alustariippumattomien tekniikoiden perusperiaatteista, eli kuinka paljon koodi vaatii eroja, jotka ovat alustakohtaisia.	4
Natiivit rajapinnat	Kuinka hyvin natiiveja rajapintoja voidaan hyödyntää (esimerkiksi kamera tai GPS).	3
Käyttöjärjestelmien tuki	Kuinka useaa versiota kohdekäyttöjärjestelmästä sovellus tukee (esimerkiksi Android 9 ja Android 10).	2

Taulukko 2: Loppukäyttäjän kriteerit alustariippumattomille työkaluille.

Kriteeri	Kuvaus	Painoarvo
UI:n yhdenmukaisuus	Pystyykö sovelluksen ulkonäöstä erottamaan, että kyseessä ei ole natiivisti kehitetty sovellus.	3
UI:n suorituskyky	Pystyykö sovelluksen käytön responsiivisuudesta havaitsemaan hitautta verrattuna natiivisti kehitettyyn sovellukseen.	4
Sovelluksen suorituskyky	Pystyykö sovelluksen käytössä tapahtuvien raskaiden työkuormien prosessoinin hitauden havaitsemaan natiiviin sovellukseen verrattuna.	4
Resurssien käyttö	Kuinka paljon resursseja sovellus vaatii verrattuna natiiviin sovellukseen ja kuinka suuri vaikutus tällä on akkukestoon.	3
Sovelluksen koko	Kuinka suuri sovellus on verrattuna natiivisti kehitettyyn sovellukseen (sovelluksen laitteelta viemä tila ja latauksen koko).	1

5 Testisovellus

Tässä luvussa käydään läpi tutkielmaa varten kehitettyä testisovellusta, sen eroja eri kohdealustoilla ja sen eri testien perusteluja sekä toimintaperiaatteita. Lisäksi käydään läpi, kuinka testisovelluksesta ja sen eri testeistä kerätään dataa ja millä työvälineillä.

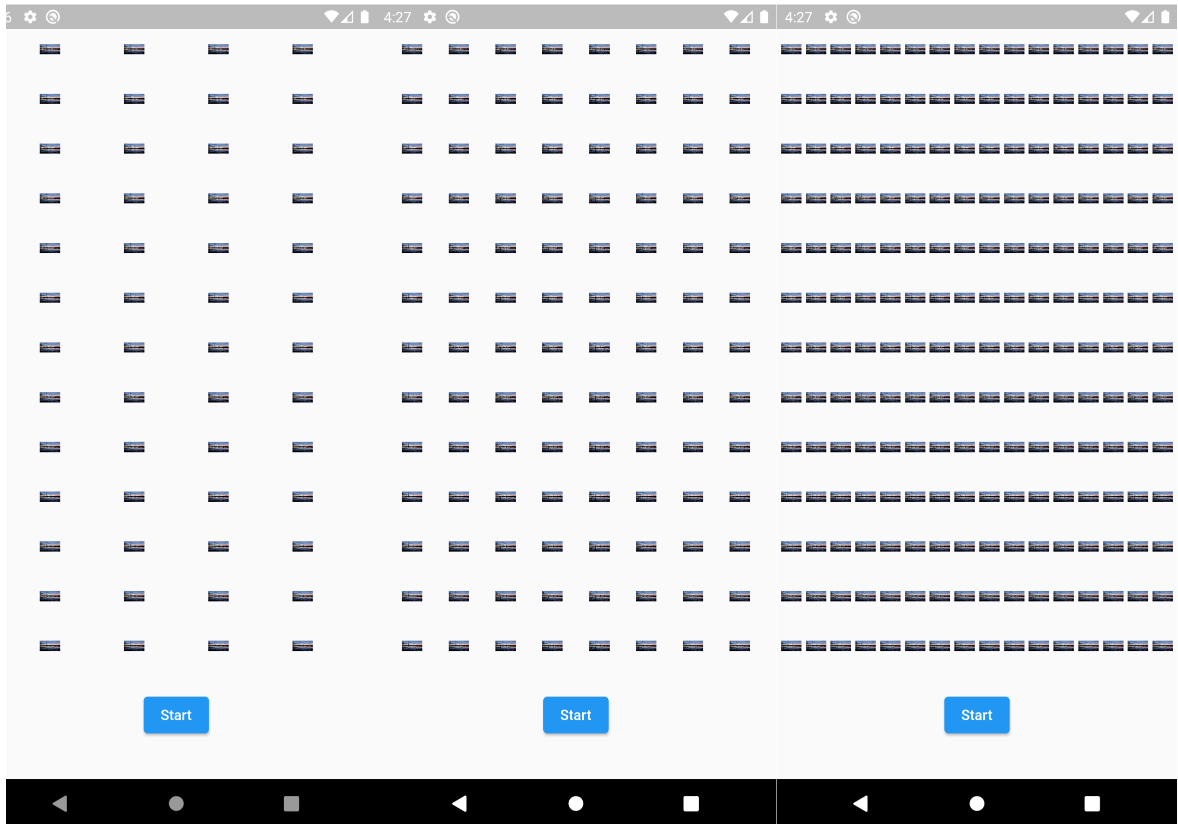
Jokainen tässä tutkielmassa toteutetuista testisovelluksista löytyy oheisesta git-versionhallinnasta (Pimiä 2021). Samassa versionhallinnassa on myös testeissä käytetyt resurssit, kuten kuvat ja JSON-tiedosto.

5.1 Testit

Testisovelluksen testeillä pyritään kuvastamaan erilaisia mittareita, jotka ovat merkittäviä nykyaikaisten sovellusten näkökulmasta. Nykyaikaisilla sovelluksilla tarkoitetaan tässä kontekstissa tämän hetken suosittuja sovelluksia, joista voidaan havaita selkeitä kategorioita. Esimerkiksi, jos tarkastellaan Googlen Play -sovelluskaupan suosituimpia sovelluksia (Google Play 2021), voidaan havaita että suurin osa listan sovelluksista toteuttaa jonkin kaltaista median hakemista ja esittämistä, eikä esimerkiksi datan prosessoimista. Tilanne on vastaavanlainen myös Applen App Store -sovelluskaupassa (Appfigures 2021). Tästä voidaan arvioida, että sovelluksen ulkoasun suorituskyky saattaa nykytilanteessa olla tärkeämpi, kuin esimerkiksi prosessoinnin suorituskyky.

Testien suunnittelussa pyrittiin myös hyödyntämään mahdollisia aikaisempien tutkimuksien tuloksia ja havaintoja, joita voidaan sitten verrata tällä testisovelluksella saavutettaviin tuloksiin. Seuraavissa luvuissa käydään läpi testisovelluksen neljä eri testikategoriaa ja erilaiset yleiset mittarit, joita sovelluksen arvioimiseen käytetään.

5.1.1 Animaatiotesti



Kuvio 13. Eri intensiteettien animaatiotestit Flutterilla kehitetyssä sovelluksessa ennen testien käynnistämistä.

Testisovellus sisältää kolme erilaista animaatiotestiä, joista jokainen kohdistaa eri tasoisen rasituksen testilaitteelle. Näiden testien ulkoasu ennen suoritusta voidaan havaita kuviosta 13. Testissä esitetään useita sarakkeita kuvia, joista jokaiseen kohdistetaan erikseen koon skaalaaminen niin, että kuvan korkeus on 10dp ja leveys 20dp. Dp:llä tarkoitetaan yksikköä, joka muuttuu dynaamisesti laitteen näytön koon mukaan. Kuvien varsinainen tarkoitus on kuitenkin toteuttaa animaatiota, joka kiertää kuvaa loputtomasti joko vasemmalle tai oikealle sarakkeesta riippuen niin, että yksi kierros kestää 500 millisekuntia. Testi pyrittiin pitämään samanaikaisesti yksinkertaisena, mutta tehokkaana, mahdollistaen sen kopioimisen eri kehitystekniikoilla siten, että kehityksen aikaiset virheet voitiin minimoida. Eri intensiteettitasojen tarkoituksena taas on mahdollistaa suorituskyvyn skaalautumisen tutkiminen.

Testin tarkoituksena on mitata eri kehitystekniikoiden kykyä aloittaa nopeasti useitakin ani-

maatioita, sekä niiden kykyä ylläpitää kyseisiä animaatioita. Suorituskyvyn mittareina toimivat kohdelaitteen prosessorin ja muistin käyttö sekä testin aikana saavutettavat näytön virkistykset eli kuvat per sekunti, joka yleisemmin tunnetaan lyhenteellä FPS (*engl.* Frames per second). Google suosittelee, että sovellusten pitäisi saavuttaa ainakin 60 FPS, jotta käyttäjäkokemus olisi mahdollisimman sulava (Android Developer 2021). Mikäli sovellus ei saavuta tätä nopeutta, joutuu se pudottamaan kuvia, joka taas näkyy käyttäjälle tökkimisenä ja hitautena.

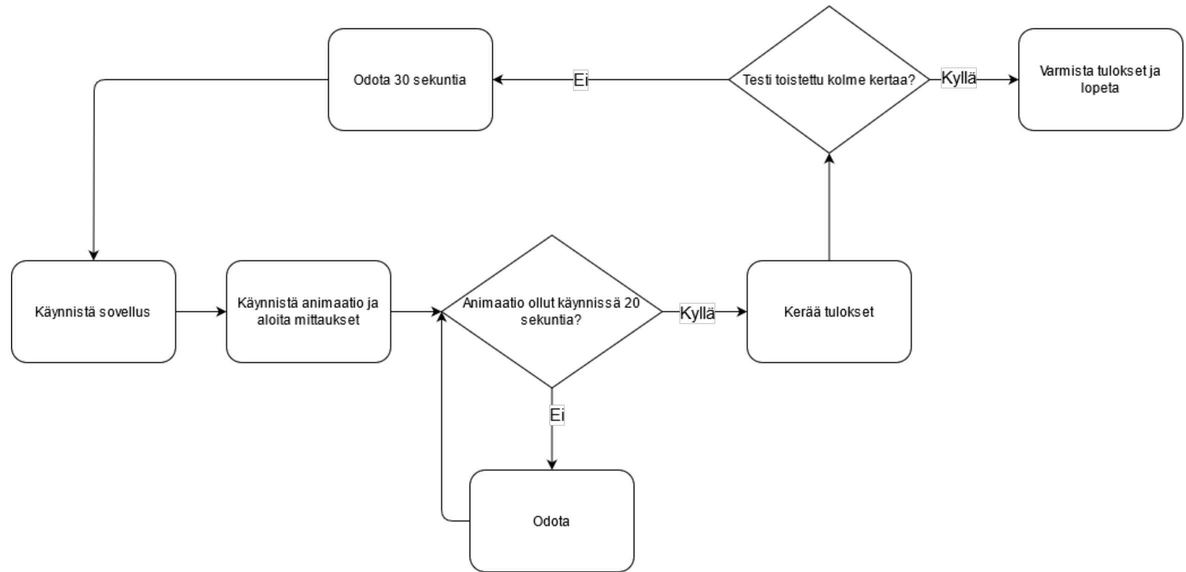
Tämän vuoksi tärkeitä mittareita ovat saavutettava kuvien määrä ja siitä saatava keskiarvo-FPS, sillä ne kuvastavat suoraan kuinka monimutkaisia animaatioita sovellukseen voidaan lisätä ilman, että käyttäjäkokemus kärsii. Prosessorin ja muistin käyttö sen sijaan kuvastaa kuinka paljon virtaa laite joutuu käyttämään kyseisen käyttäjäkokemuksen tuottamiseen. Tässä testissä haluttiin alun perin kerätä dataa myös näytönohjaimen kuormasta, mutta Android ei valitettavasti tarjoa työkalua kyseisen datan keruuta varten.

Tämänkaltaisia animaatiotestejä on toteutettu tutkimuksissa myös kolmannen osapuolen animaatiokirjastoa käyttäen (esimerkiksi Lottie) (Hidayat ja Sungkowo 2020; Biørn-Hansen, Grønli ja Ghinea 2019) . Niiden käyttäminen suljettiin kuitenkin tässä tutkielmassa pois siksi, että Lottie ei täysin tukenut Flutteria ja sen eri versioita oli toteutettu eri tavoilla, tehden saavutettavista tuloksista epävarmoja, koska erot saattaisivat johtua Lottien eri toteutuksista. Tähän testiin valittu kiertoanimaatio sen sijaan löytyy jokaiselta kohdealustalta ja kehitystekniikalta valmiiksi, ja sen pitäisi siksi olla laadukkaampi suorituskyvyn vertailuun.

Animaatiotesti toteutetaan alla olevan kuvion 14 mukaisessa järjestyksessä. Testi toistetaan kolme kertaa ja jokaisen kerran välissä on rauhoitusjakso ja sovelluksen uudelleenkäynnistys, joilla varmistetaan, että edellinen testi ei vaikuta seuraavaan suoritukseen, esimerkiksi laitteen lämpötilan kautta. Tulokset muodostetaan sitten kolmen suorituskerran keskiarvoista.

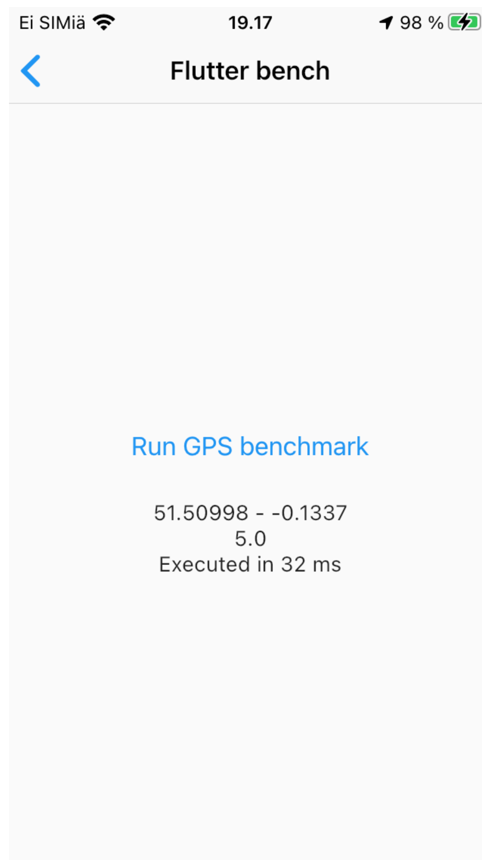
Työkaluihin, joilla tulokset kerätään, palataan myöhemmässä luvussa 5.2, mutta animaatiotestin kannalta on oleellista, että tuloksia aletaan keräämään heti sillä hetkellä, kun käyttäjä käynnistää animaation. Tämä on tärkeää, koska varsinkin alkuun saavutettava FPS on käyttäjälle helpoiten erotettavissa. On kuitenkin todennäköistä, että animaation suorituskyky pa-

ranee ajan kanssa, kun laite, jolla sovellusta ajetaan nostaa prosessorin tai näytönohjaimen nopeutta.



Kuvio 14. Animaatiotestin kulkukaavio.

5.1.2 GPS-testi



Kuvio 15. GPS-testi ja sen tulokset Flutterilla toteutetussa testisovelluksessa iOS:llä.

GPS-rajapinta on yksi yleisimmin käytetyistä natiiveista rajapinnoista, jonka vuoksi on mielekästä testata sen nopeutta ja tehokkuutta eri tekniikoilla. Tässä tutkielmassa toteutettu testi pyrkii hakemaan laitteen sen hetkisen sijainnin ja mittaamaan sen hakemiseen kuluvan ajan. Hakuun kuluva aika on mittari, joka voi vaikuttaa merkittävästi käyttäjäkokemukseen tapauksissa, joissa sovellus tarvitsee käyttäjän sijainnin ennen jatkamista. Nopeuden lisäksi mitataan myös prosessorin ja muistin käyttöä, joka jälleen kertoo varsinkin virrankulutuksen tehokkuudesta.

Testiin millisekunteina kuluvaa aikaa mitataan monotonisilla kelloilla, jotka mahdollistavat mahdollisimman tarkan ja stabiilin ajan ilman vaihtelua reaaliaikaisiin kelloihin verrattuna (Love 2013). Jokaisen sovelluksen haluttu sijainnin tarkkuus asetettiin myös samaksi. Androidin tapauksessa käytettiin keskitason tarkkuutta, kun taas iOS:n tapauksessa käytet-

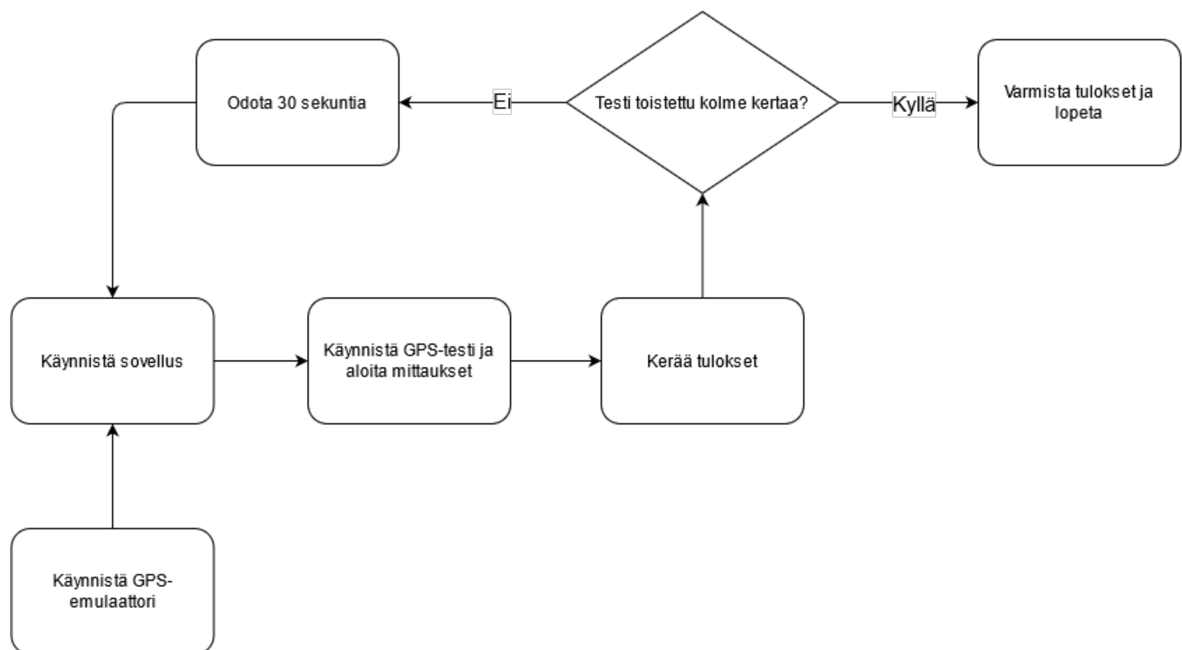
tiin parasta mahdollista tarkkuutta. Myös Androidin oli alun perin tarkoitus käyttää parasta mahdollista tarkkuutta, mutta kyseinen tarkkuusasetus tuotti eroavia tarkkuuksia eri tekniikoilla kehitetyille sovelluksille, kun taas keskitason tarkkuudella tätä ongelmaa ei esiintynyt. Edellä mainittu ero ei kuitenkaan testin kannalta ole ongelma, koska tutkielmassa ei yritetä mitata Androidin nopeutta iOS:n verrattuna.

Koska testissä halutaan mitata erityisesti sijainnin rajapinnalta hakemiseen kuluvaa aikaa, käytetään testissä hyväksi GPS-emulaatiota, joka mahdollistaa täysin identtisten koordinaattien antamisen eri sovelluksille. Näin vältytään sijainnista ja esimerkiksi GPS-satelliittien liikkeestä aiheutuvista eroista, sillä jokainen sovellus saa täysin identtisen sijainnin. GPS-emulointi toteutettiin alustasta riippuen joko Xcoden tai Android Studion tarjoamilla työkaluilla.

GPS-testiin harkittiin myös toteutusta, joka hakisi sijaintia jatkuvasti taustalla, mutta tämänkaltaisen testin tulokset todettiin liian epäluotettavaksi, varsinkin erilaisten Android-laitteiden virrankulutuksen optimointiin liittyvien tekijöiden vuoksi, jotka olisivat tehneet testitulokset riippuvaiseksi laitteesta, ei tekniikasta. Jokaisen sovelluksen GPS-kirjasto olisi silti mahdollistanut sijainnin hakemisen taustalla, kunhan taustahakuun vaadittu oikeus lisättiin sovellukseen.

Tämän GPS-testin kaltaisia testejä on myös toteutettu aikaisemmissa tutkimuksissa. Esimerkiksi Biørn-Hansen ym. (2020) havaitsivat tutkimuksessaan eroja eri alustariippumattomilla tekniikoilla, joskin tutkimuksesta ei täysin selviä käytettiinkö testissä sijainnin emulointia sen tulosten tasoittamiseksi. Joka tapauksessa Biørn-Hansen ym. (2020) tutkimuksen tulokset ovat hyvä vertailukohde tämän testin tuloksille.

GPS-testin kulku on eroistaan huolimatta hyvin samankaltainen animaatiotestin kanssa. Poikkeuksena on vain GPS-emulaation käynnistäminen ennen sovellusta ja sovelluksen suorituksen kestäminen niin kauan kuin laitteella kestää sijainnin hakemisessa. Tämän jälkeen talteen otetaan suoritusaika ja muut edellä mainitut mittarit, joita aletaan mittaamaan siitä hetkestä, kun käyttäjä käynnistää testin.



Kuvio 16. GPS-testin kulkukaavio.

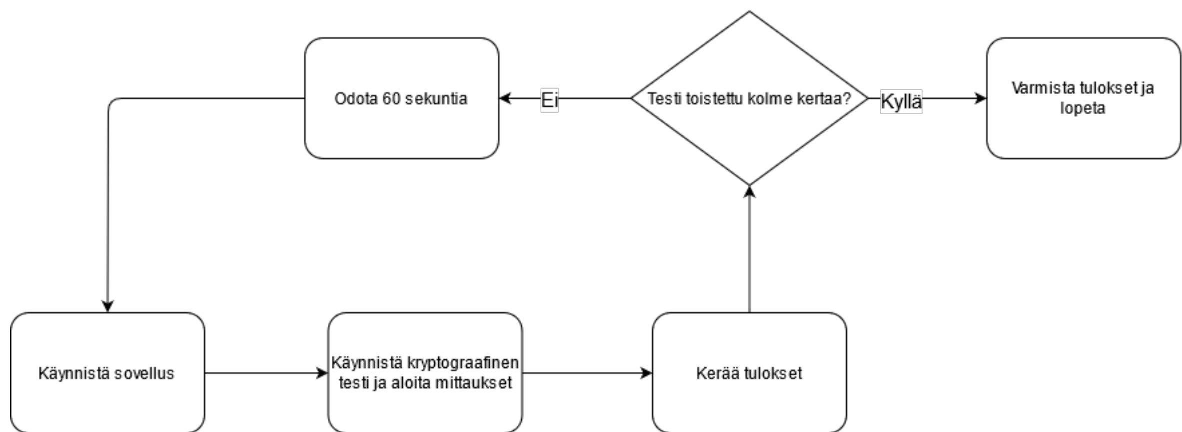
5.1.3 Kryptografinen testi

Kryptografinen testi on testisovelluksen toinen raakaa suorituskykyä arvioiva testi, joka mittaa suorituskykyä tilanteessa, jossa sovellus joutuu käyttämään natiiveja rajapintoja mahdollisimman nopeassa tahdissa. Testissä lasketaan 800000 uniikkia HMAC-SHA256 tiivistettä, joka rasittaa voimakkaasti varsinkin prosessoria. GPS-testin tavoin tässäkin testissä mitataan aikaa monotonisia kelloja hyödyntäen, jonka lisäksi mitataan myös prosessorin ja muistin resurssien käyttöä.

Tämän testin tarkoituksena ei ole kuvata realistista tilannetta oikeassa sovelluksessa, vaan sen sijaan valitun tekniikan skaalautuvuutta tilanteeseen, jossa sovelluksen pitäisi tehdä prosessori-intensiivistä laskentaa. Testi on tärkeä myös siksi, että Flutter ja React-Native käyttävät natiiveja rajapintoja eri tavalla, joka voi vaikuttaa merkittävästi tämänkaltaiseen suorituskykyyn. Kryptografiselle testille ei löytynyt suoraan vastaavaa tutkimusta, johon tuloksia voisi verrata, mutta sen eri variaatioita voidaan löytää suosituista benchmark-sovelluksista, kuten esimerkiksi Geekbench:stä (Geekbench 2019).

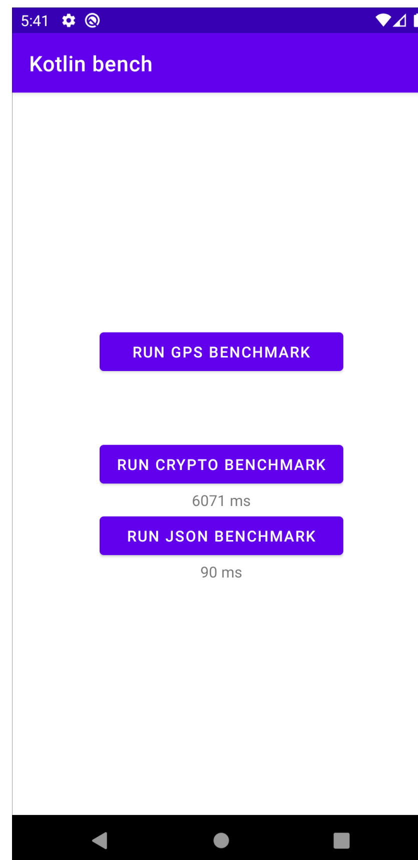
Kryptografisen testin kulkua voidaan havainnoida kuviosta 17. Kulku on jälleen sovelluksen

käytön kannalta hyvin samankaltainen kuin aikaisemmissa testeissä, mutta poikkeuksena on, että eri testikierrosten välillä odotetaan nyt 60 sekuntia. Tämä johtuu siitä, että testin todettiin olevan hyvin raskas ja aiemmalla 30 sekunnin rauhoitusajalla havaittiin, että seuraavat testit olivat toistettavasti hieman hitaampia kuin edelliset. Tämä osoitti, että laite ei ehtinyt palautua edellisestä testistä tarpeeksi, joka olisi siten vääristänyt testien tulokset. Uudella 60 sekunnin rauhoitusajalla vääristymää ei enää havaittu.



Kuvio 17. Kryptografisen testin kulkukaavio.

5.1.4 JSON-testi



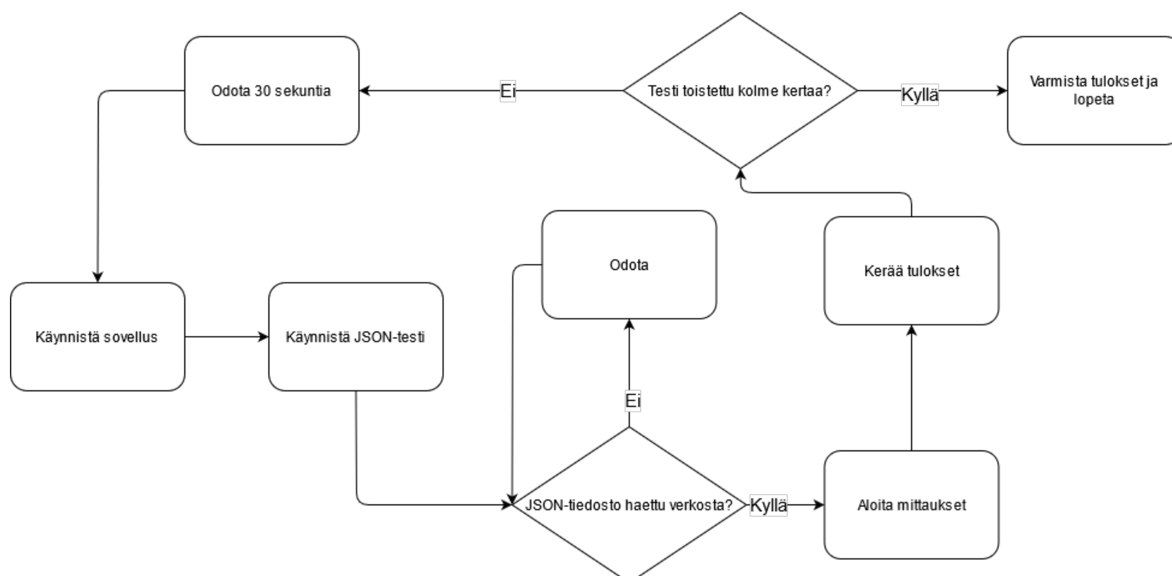
Kuvio 18. Kryptografisen ja JSON-testin ajalliset tulokset Kotlinilla tuotetussa sovelluksessa.

Kryptografisen testin ohella yleistä suorituskykyä mitataan myös testillä, jossa laitteen muistiin luetaan iso JSON-tyyppinen tiedosto. JSON-tiedostot ovat hyvin yleisiä esimerkiksi verkkopohjaisten REST-rajapintojen käytössä ja niiden käsittely vaatii alustariippumattomalta sovellukselta natiivin rajapinnan kanssa työskentelyä.

Testissä mitataan samoja parametreja kuin kryptografisessa testissä niin, että sovellus lataa ensin verkosta noin kymmenen megatavun kokoisen JSON-tiedoston, jonka se sitten pyrkii lukemaan mahdollisimman nopeasti muistiin. Suoritukseen kuluvaa aikaa aletaan kuitenkin mittaamaan vasta verkosta lataamisen jälkeen, sillä testin tarkoituksena on mitata varsinkin natiivien rajapintojen käyttämisestä aiheutuvaa viivettä. Lisäksi verkosta lataamisen nopeuteen voi vaikuttaa suuri joukko tekijöitä, joita olisi hankala minimoida testin tasavertaisuuden

saavuttamiseksi.

Tämän testin merkittävä ero kryptografiseen testiin on, että se vaatii yhden ison tehtävän käsittelyä monen lyhyen tehtävän sijaan. Tämänkaltainen tehtävä on myös realistisempi, sillä jos kehitettäisiin sovellus, jolla halutaan pystyä esittämään suuria määriä dataa, vaaditaan tällaista suorituskykyä. Koska tämän testin suoritus ei ole yhtä raskas kuin kryptografisen, on myös käytetty rauhoitusaika taas 30 sekuntia. Testissä on kuitenkin tärkeää kerätä dataa vasta verkosta lataamisen jälkeen, jonka valmistuminen on onneksi helppo havaita käytetyillä suorituskyvyn profilointityökaluilla. Näihin työkaluihin palataan tarkemmin myöhemmässä luvussa.



Kuvio 19. JSON-testin kulkukaavio.

5.1.5 Yleiset mittarit

Edellä mainittujen neljän testin ohella sovelluksesta mitataan myös muutamia yleisiä muuttujia. Näitä ovat valmiin sovelluspaketin koko (.APK Androidilla ja .IPA iOS:llä), sovelluksen käynnistymiseen kuluva aika sekä tila jonka valmis sovellus vie laitteelta. Näistä varsinkin sovelluksen käynnistykseen kuluva aika on merkittävä muuttuja käyttäjäkokemuksen kannalta, sillä sovelluksen viemä tila on nykyään vähemmän relevantti puhelimien tallennustilan kasvaessa. Käynnistykseen kuluvaa aikaa mitattiin Androidilla adb-komentorivityökalun (*engl.* Android Debug Bridge) komennolla "`adb shell am start`

`-S -W <paketin_nimi>`”, kun taas iOS:n käynnistysajan pystyi keräämään Xcoden profilointityökaluilla.

Myös sovelluspaketin koko on nykyään vähemmän relevantti muuttuja, mutta siitä löytyy aikaisempaa tutkimusta (Biørn-Hansen ym. 2020), jonka vuoksi on mielekästä tutkia onko siinä tekniikoiden päivittymisen mukana tapahtunut muutoksia. Sovelluspaketit, joiden kokoa verrattiin, rakennettiin kaikki Release-tilassa eli muodossa, jossa ne olisivat valmiita sovelluskauppoihin.

5.2 Datan kerääminen

Jotta testisovelluksen testien tuloksista saadaan mahdollisimman luotettavia, on tärkeää, että työkalut, joilla testien dataa kerätään ovat tarkkoja ja eivät vaikuta merkittävästi testien tuloksiin. On myös olennaista, että testit ja datan keruu on helposti toistettavissa. Pääasiassa käytetyt työkalut koostuivat Applen Xcoden ja Googlen Android Studion tarjoamista suorituskyvyn profilointityökaluista, mutta varsinkin Androidin kanssa jouduttiin käyttämään myös komentorivillä toimivaa dumpsys-työkalua. Tämän lisäksi Android-alustan testaaminen vaati useamman testikierroksen eri sovelluspaketeilla, jonka syitä perustellaan kappaleessa 5.2.1.

Ennen testien suorittamista kummallakin alustalla sammutettiin kaikki muut sovellukset, laitteiden akut ladattiin täyteen ja mobiilidata kytkettiin pois päältä, jättäen päälle vain WLAN-yhteyden ja GPS:n. Sovellusten suorituskykyä mittaavat testit, eli suoritusaikaan ja saavutettuun FPS:n liittyvät testit ajettiin aina niille parhaimman suorituskyvyn takaavassa konfiguraatiossa. Yleisemmät muuttujat, kuten prosessorin ja muistin käyttöaste mitattiin sen sijaan alustasta riippuen hieman eroavilla konfiguraatioilla, joihin perehdytään tarkemmin tulevissa kappaleissa.

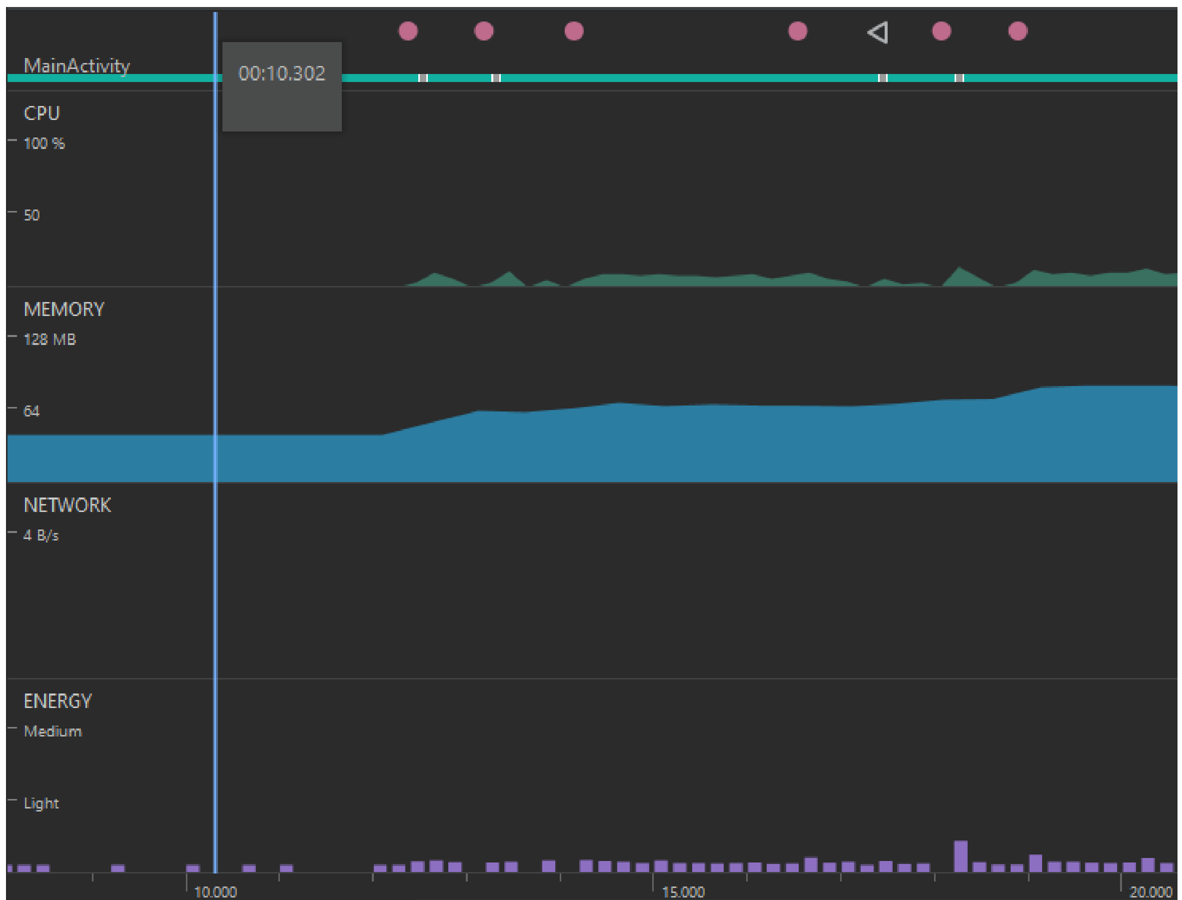
Seuraavissa kappaleissa käydään lävitse, kuinka dataa kerättiin kohdealustakohtaisesti ja millä laitteilla testisovelluksia ajettiin.

5.2.1 Android

Android-alustalla prosessorin ja muistin käyttötason keräämiseen käytettiin Android Studion tarjoamia profilointityökaluja, jotka näkyvät kuviossa 20. Android Studion työkalut mahdollistivat tämän statistiikan keräämisen ja suodattamisen kaikkien testien osalta tehokkaasti, sillä työkalusta pystyi helposti havaitsemaan täsmällisen hetken, kun käyttäjä käynnisti minkä tahansa testeistä. Esimerkiksi animaatiotestin tapauksessa käynnistyshetkestä poimittiin tasan 20 sekunnin otos, kun taas muiden testien osalta kerättiin niin pitkä otos kuin testin suorittaminen vaati.

Valitettavasti Android Studion profilointityökaluja oli mahdollista käyttää vain, jos rakennetulle Android-sovelluspaketille oli asetettu debuggable-lippu. Vaikka tämän lipun asettaminen oli mahdollista samanaikaisesti sovelluksen parhaimman suorituskyvyn takaavan Release-koontiversion kanssa, aiheuttaa se mahdollista ylimääräistä rasitusta, joka voisi vaikuttaa suorituskykytestien tuloksiin. Tämän vuoksi testaamiseen käytettiin Biørn-Hansen ym. (2020) käyttämää menetelmää, jossa suorituskykyyn painottuvat testit (FPS ja suoritusaika) suoritettiin ensin ilman debuggable-lippua, jonka jälkeen toteutettiin uusi suoritus debuggable-lipun kanssa ja kerättiin sovelluksen ajamiseen liittyvä data (prosessorin ja muistin käyttöaste).

Tulokset saattaisivat olla erilaisia ilman debuggable-lippua, mutta se ei tässä tapauksessa haittaa, sillä tutkielmassa verrataan tekniikoita toisiinsa alustakohtaisesti ja kaikki tekniikat kärsivät samasta suorituskyvyn menetyksestä.



Kuvio 20. Android Studion suorituskyvyn profilointityökalu. Yläreunan punaiset pisteet kuvaavat kosketustapahtumia, joiden perusteella testien aloitus voidaan havaita.

Keskimääräisen FPS:n osalta Android vaati dumpsys- ja gfxinfo-työkalujen käyttämistä. Työkaluja käytettiin komentoriviltä adb:n ylitse komennolla "`adb shell dumpsys gfxinfo <paketin_nimi>`", joka mahdollisti keskimääräisen FPS:n laskemiseksi vaadittujen kuvien määrän keruun myös ilman profilointityökalujen vaatimaa debuggable-lippua. Koska tätä dataa kerättiin vain animaatiotestiä varten, voitiin gfxinfo alustaa juuri ennen testin ajamista, jonka jälkeen se alkaa keräämään dataa heti kun näyttö päivittyy seuraavan kerran. Tämä päivittyminen tapahtuu täsmälleen sillä hetkellä, kun käyttäjä aloittaa testin, joka mahdollistaa työkalun käyttämisen samanaikaisesti muiden profilointityökalujen kanssa. Gfxinfotyökalusta saatava data piti kuitenkin tulostaa ulos täsmälleen sillä hetkellä kun 20 sekuntia animaatiotestiä oli kulunut, jotta kerättävä data saataisiin oikealta väliltä. Tähän hyödynnettiin Autohotkey-skriptiä, joka käynnisti testin adb:n yli lähetetyllä kosketuskomennolla ja

tulosti gfxinfon tulokset täsmälleen 20 sekunnin kuluttua.

```
Applications Graphics Acceleration Info:
Uptime: 103675139 Realtime: 180552186

** Graphics info for pid 29213 [com.example.gradu_kotlin_bench] **

Stats since: 103653812410249ns
Total frames rendered: 919
Janky frames: 60 (6.53%)
50th percentile: 9ms
90th percentile: 10ms
95th percentile: 11ms
99th percentile: 12ms
Number Missed Vsync: 0
Number High input latency: 913
Number Slow UI thread: 3
Number Slow bitmap uploads: 0
Number Slow issue draw commands: 3
Number Frame deadline missed: 6
```

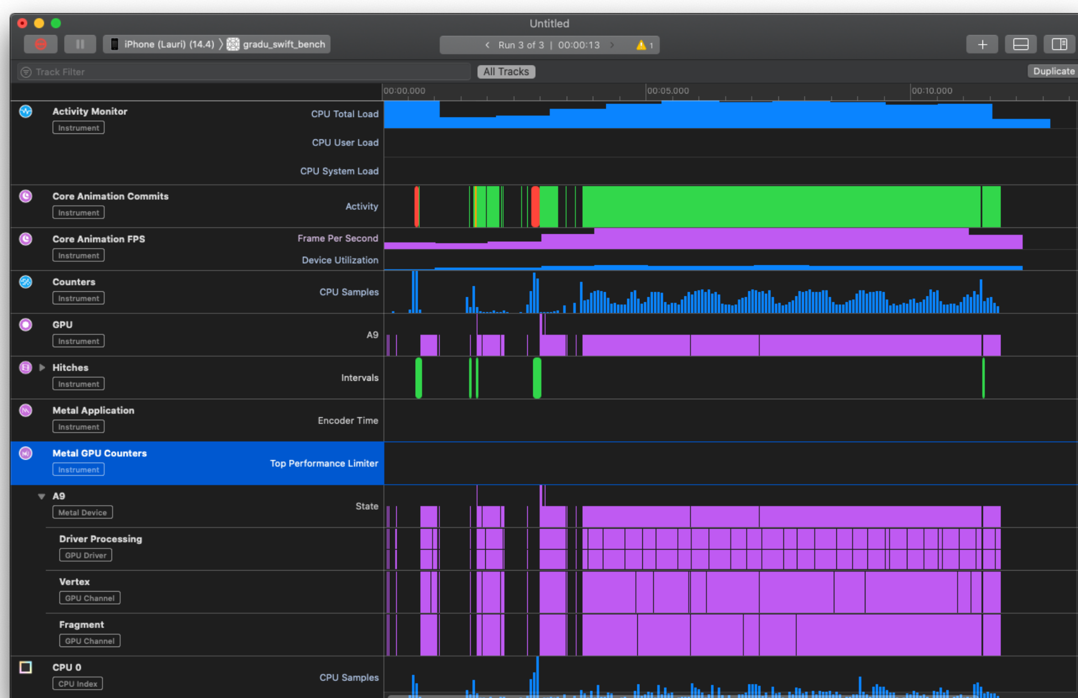
Kuvio 21. Osa gfxinfo-työkalun tulostamasta datasta. Huomioitavia kohtia varsinkin kokonaiskuvien (*engl.* total frames) ja pudotettujen kuvien (*engl.* janky frames) määrä.

Edellä mainitut työkalut toimivat kaikkien muiden, paitsi Flutterilla tuotetun testisovelluksen animaatiotestin kanssa. Koska Flutter käyttää Skia-renderöintimoottoria, ei gfxinfo pysty havaitsemaan sovelluksen animaatioiden kuvien päivittymistä, jonka seurauksena gfxinfo tulostaa tyhjiä tietoja. Tämän vuoksi Flutterin tuloksiin käytettiin kirjaston omia profilointityökaluja, jotka tarjosivat kuitenkin samat tiedot kuin gfxinfo. Flutter tarjosi myös ainoana tekniikkana oman Profile-koontiversion, jonka suorituskyky vastasi Release-versiota, mutta mahdollisti yhä profiloinnin Flutterin omilla työkaluilla.

5.2.2 iOS

iOS:n osalta datan keruu oli yksinkertaisempaa, sillä kaikkien iOS:llä ajettavien testisovellusten statistiikka pystyttiin keräämään Applen Xcoden omilla profilointityökaluilla ilman erilaisia koontiversioita. Xcode tarjosi huomattavasti laajempaa statistiikkaa verrattuna Android Studioon, joka samalla mahdollisti kaiken datan saamisen ilman erillisten tai esimerkiksi Flutterin tapauksessa kirjaston omien työkalujen käyttöä. Flutterin tulokset var-

mennettiin kuitenkin myös kirjaston omien työkalujen avulla, jotta saavutettiin varmuus työkalujen tulosten yhdenvertaisuudesta. Itse testien data kerättiin samalla menetelmällä kuin Androidilla, eli datan keruu ajoitettiin testien aloituksesta syntyviin tapahtumiin.



Kuvio 22. Esimerkki Xcoden profilointityökalujen tulosteesta.

5.2.3 Testilaitteet

Testilaitteiksi valittiin Androidin puolelta Oneplus 7 Pro ja iOS:n puolelta Apple iPhone SE. Näiden laitteiden tarkemmat ominaisuudet ovat näkyvissä taulukossa 3.

Taulukko 3: Tutkielman Android- ja iOS-alustan testilaitteet.

Nimi	Julkaisuvuosi	Piirisarja	Muisti	OS	Virkistystaajuus
Apple iPhone SE	2016	Apple A9 (14nm)	2 GB	iOS 14.4	60hz
Oneplus 7 Pro	2019	Snapdragon 855 (7nm)	8 GB	Android 10	90hz

Laitevalinnassa tärkeintä oli, että laitteita oli molemmilta alustoilta. Tässä tapauksessa laitteilla on myös noin kolmen vuoden ikäero, joka tekee odotettavaksi sen, että uudempi laite voi tuottaa parempia tuloksia. Toisaalta ero ei välttämättä ole niin suuri kuin voisi etukäteen kuvitella, sillä iOS- ja Android-laitteet toimivat varsin eri tavoin. Laitteiden iän lisäksi merkittävä ero on Oneplus 7 Pro:n näytön korkeampi virkistystaajuus. Tämä tarkoittaa, että laitteen sovellukset tähtäävät sekunnissa 90 virkistykseen 60 sijaan, joka tekee Android-sovellusten animaatiotestin suorituksesta teoriassa raskaamman. Laite vaihtelee oletuksena 60 ja 90 virkistyksen välillä, mutta tämä asetus asetettiin testin ajaksi pysyvästi 90 hertsiin, jotta vaihteluilta varsinkin animaatiotestin aikana pystyttiin välttymään.

Vaikka animaatiotesti on teoriassa Androidilla raskaampi, ei se tutkielman kannalta haittaa, koska testeillä ei ole tarkoitus osoittaa miten laitteet toimivat toisiaan vastaan. Sen sijaan on tarkoitus vain mitata miten eri tekniikoilla toteutetut testisovellukset vertautuvat keskenään samalla alustalla. Voidaan kuitenkin päätellä, että esimerkiksi iPhonella ajettavat testit todennäköisesti antaisivat parempia tuloksia uudemmilla laitteilla samaan tapaan kuin Androidilla ajetut testit voisivat antaa huonompia tuloksia vanhemmilla laitteilla.

5.3 Eri sovellukset eri alustoilla

Seuraavissa kappaleissa käydään lävitse, kuinka eri testit toteutettiin tutkielman eri kehitystekniikoilla, mitä havaintoja sovelluksia kehittäessä tehtiin sekä miten eri tekniikat vaikuttivat sovelluksen ominaisuuksiin. Flutterilla ja React-Nativella tuotetuista sovelluksista pyrittiin tekemään ulkoasultaan natiivin näköiset, mutta erinäisten asetusten, kuten värien annettiin pysyä oletuksina. Koska tässä tutkielmassa keskitytään mobiilikehitykseen, kehitettiin sovelluksia primäärisesti vain älypuhelinalustoille. Tästä huolimatta kaikki sovellukset toimivat myös kohdealustojen kanssa samanlaista arkkitehtuuria käyttävillä laitteilla, kuten tableteilla ja ARM-perusteisilla tietokoneilla kuten Googlen Chromebook- ja Applen M1 -sarjan kannettavilla.

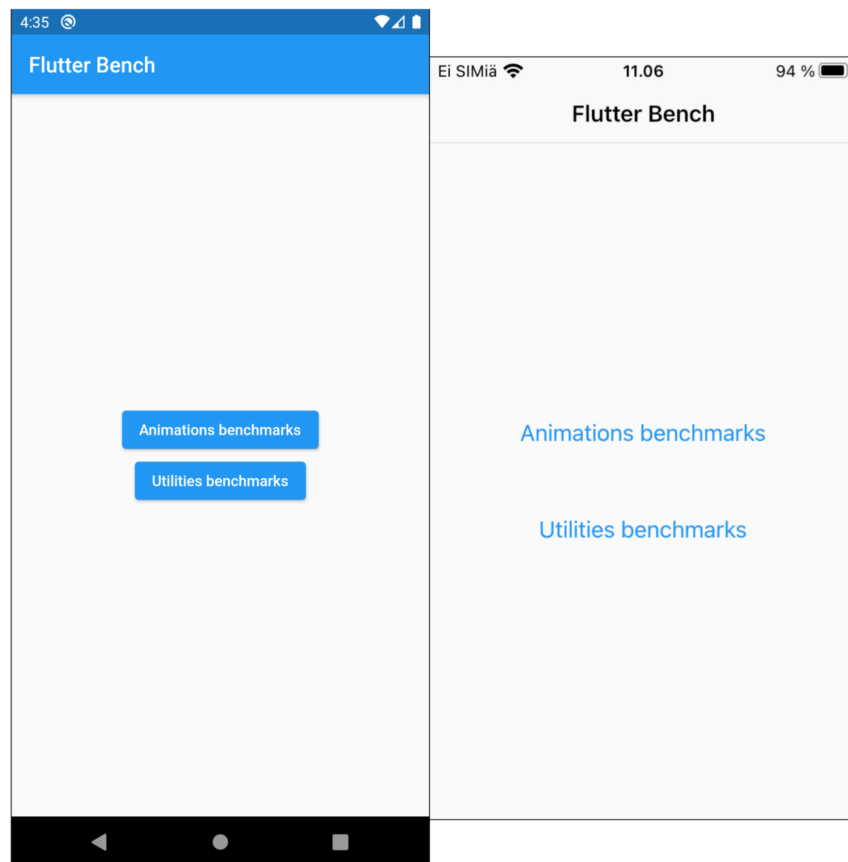
Sovellukset pyrittiin kaikki toteuttamaan eri tekniikoiden parhaita menetelmiä hyödyntäen ja toteutukset tehtiin iteratiivisella syklillä, jossa sovelluksia vuorotellen kehitettiin ja testattiin, jonka jälkeen testaamisessa ilmentyneitä parannuskohteita siirryttiin taas kehittämään.

Sovellusten kehitykseen käytettyjen ohjelmistokirjastojen versiot sekä niiden tukemat kohdealustojen versiot näkyvät taulukossa 4.

Taulukko 4: Eri sovellustekniikat, niiden versiot sekä niiden tukemat Android- ja iOS-versiot.

Nimi	Versio	Tuetut Android-versiot	Tuetut iOS-versiot
Flutter	1.22.6	Android 4.1 ja uudempi	iOS 10 ja uudempi
React-Native	0.63.4	Android 4.1 ja uudempi	iOS 10 ja uudempi
Swift	5.1	N/A	iOS 13 ja uudempi
Kotlin	1.4.30	Android 4.1 ja uudempi	N/A

5.3.1 Flutter



Kuvio 23. Flutter-sovelluksen yleinen ulkoasu, joka muuttuu alustasta riippuen joko Android- tai iOS-näkymään.

Ensimmäinen tutkielman sovelluksista kehitettiin Flutterilla. Flutter on tutkielman tutkittavista tekniikoista uusin, jonka pystyi havaitsemaan varsinkin sen monipuolisista ja helppokäyttöisistä kehitystyökaluista sekä modernista deklaratiivisesta koodirakenteesta. Flutter tarjosi myös ylivoimaisesti nopeimman kehityskokemuksen sovelluksen ulkoasulle, sillä sen tarjoama hot-reload-toiminnallisuus oli erittäin nopea varsinkin, kun verrattiin natiiveihin sovelluksiin. Sovellus kehitettiin Android Studiolla, joskin sen testaaminen iOS:llä vaati Xcoden käyttöä.

Käyttöliittymien kehitykseen Flutter tarjoaa suuren joukon valmiita UI-komponentteja, joiden tarkoitus on muistuttaa aitoja natiiveja komponentteja. Suurinta osaa näistä komponenteista pitää kuitenkin käyttää alustakohtaisesti ja kehittäjän tulee itse asettaa, käytetäänkö esi-

merkiksi iOS:n vai Androidin painikekomponenttia. Tämänkaltainen logiikka oli kuitenkin helppo siirtää omaan komponenttiinsa, joka mahdollisti alustan mukaan ulkoasua muuttavan UI-komponentin. Koska Flutter luo omat UI-komponenttinsa, on molempien kohdealustojen ulkoasua myös mahdollista testata pelkällä iOS- tai Android-laitteella.

Alustakohtaisten komponenttien lisäksi Flutter asetti oletuksena enemmän tyyliasetuksia React-Nativeen verrattuna. Esimerkiksi painikkeet saivat automaattisesti marginaaleja ja rivit sekä sarakkeet sisennyksiä. Tämä antaa sovelluksille yleisesti paremman ulkoasun, mutta saattaa myös toisaalta vaivata kehittäjiä, jotka joutuvat nyt alustamaan ylimääräisiä asetuksia. Lisäksi nämä oletusasetukset vaihtelevat eri alustojen komponenttien mukaan (esimerkiksi Android vs. iOS painike), joka tarkoittaa että eri alustoilta pitää nollata eri asetuksia. Esimerkki Flutter-testisovelluksen alustariippumattoman painikkeen koodista löytyy liitteestä A, jossa näkyy kuinka käytetty alusta havaitaan ja kuinka eri alustojen painikkeille annettiin eri tyylejä.

Testisovelluksen ulkoasusta riippuvainen animaatiotesti toteutettiin Flutterin omaa animaatiokirjastoa ja siihen liittyviä hyviä periaatteita käyttäen. Koska Flutter joutuu oletuksena uudelleenrakentamaan koko ulkoasun yhden komponentin päivittyessä, oli suorituskyvyn kannalta tärkeää, että animaatio toteutettiin tavalla, joka mahdollisti vain animaation alaisten komponenttien, eli tässä tapauksessa kuvien päivityksen. Jokainen kuva ja sen animaatio asetettiin sen lisäksi omaan komponenttiinsa, joka mahdollisti niiden määrän dynaamisen muuttamisen. Yhden kuvan ja sen animaation sisältävän komponentin koodi näkyy liitteessä B.

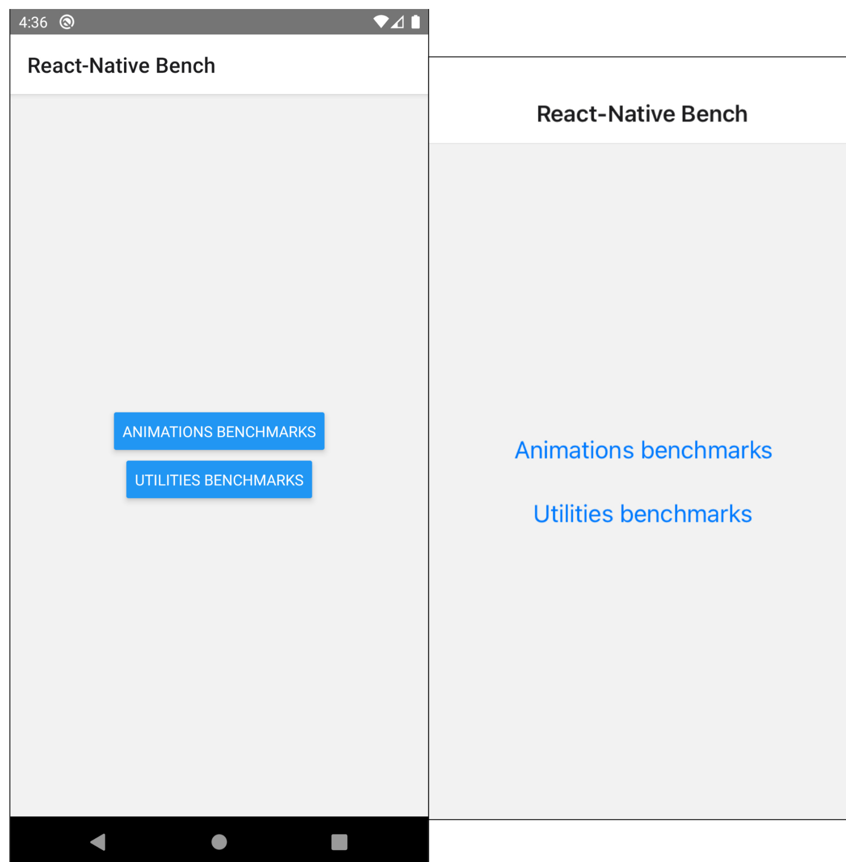
GPS-testiin käytettiin kolmannen osapuolen kehittämää geolocator-komponenttia, joka noudettiin kappaleessa 3.3.1 mainitusta pub.dev tietokannasta. Flutter ei tarjonnut omaa GPS-komponenttiaan, mutta suositteli edellä mainittua, jonka vuoksi sitä päädyttiin käyttämään tässä testissä. Geolocator tarjosi suuren määrän konfiguroitavia asetuksia, mutta se erottui muista tekniikoista edukseen varsinkin siksi, että se osasi automaattisesti hoitaa sijainnin käyttöoikeuksien hankkimisen. Geolocator osasi myös asettaa alustakohtaisia asetuksia ilman, että niitä piti erikseen mainita, joskin tämän tutkielman tapauksessa Android- ja iOS-alustat käyttivät eroavia tarkkuusasetuksia kuten kappaleessa 5.1.2 mainittiin.

Myös kryptografinen testi oli yksinkertaista toteuttaa Flutterin omalla crypto-komponentilla. Flutter olisi vaihtoehtoisesti tukenut myös natiivien rajapintojen käyttämistä suoraan Kotlinilla ja Swiftillä, mutta testien tekeminen näitä käyttäen olisi rikkonut alustariippumattomuuden periaatetta. Kryptografisen testin tavoin myös JSON-testi toteutettiin Flutterin omilla kirjastoilla ja Flutter tarjosi JSON-tiedoston dekoodaamisen, joko mallilla ennalta määriteltyyn luokkaan tai tuntemattomaan objekti-tietoluokkaan. Tämän tutkielman kaikissa testeissä JSON dekoodattiin tuntemattomana objektina muistiin.

Aikaisemmin mainitun erittäin nopean hot-reload-toiminnallisuuden lisäksi Flutter tarjosi myös hot-restart-ominaisuuden, joka näytti muutokset, joihin hot-reload ei toiminut ilman koko sovelluksen uudelleen kokoamista. Ylipäänsä Flutterilla sovelluksen kokoaminen oli hyvin automatisoitua ja ainut vaihe joka vaati alustakohtaista manuaalista työtä, oli erilaisten oikeuksien lisääminen (esimerkiksi sijainnin käyttöoikeus).

Flutter tarjoaa kolme erilaista koontiversiota, joita ovat: Debug, Profile ja Release. Eri testien mittaukset tehtiin pääasiassa Release-versiolla, paitsi kappaleessa 5.2.1 mainittu Android-alustan FPS-testi, joka tehtiin Profile-versiolla. Profile-version pitäisi silti olla yhdenvertainen Release-version suorituskyvyn kanssa.

5.3.2 React-Native



Kuvio 24. React-Nativella kehitetyn sovelluksen ulkoasu Android- ja iOS-alustoilla. Flutterista poiketen React-Native ei sisäisesti käytä eri komponentteja eri alustoilla, vaan käyttää sen sijaan kohdealustalta löytyvää komponenttia esimerkiksi painikkeelle.

Seuraavaksi kehitettiin React-Nativella toteutettu testisovellus, joka ohjelmistokirjaston ikäerosta huolimatta oli Flutteriin verrattuna varsin samankaltainen kehityksen kannalta. Kaikki UI-komponentit koostuivat muista komponenteista ja myös React-Native tarjosi erittäin nopean hot-reload-toiminnallisuuden käyttöliittymän kehittämiseen (joskaan ei yhtä nopean verrattuna Flutteriin). React-Native sovellus kehitettiin Visual Studio Codella, jonka lisäksi iOS-alustan testaamiseen käytettiin jälleen Xcodea.

Merkittävin ero React-Nativen ja Flutterin välillä on, että React-Native perustuu tulkilliseen lähestymistapaan, jonka vuoksi se käyttää täysin natiiveja UI-komponentteja verrattuna Flutterin omiin UI-komponentteihin. Tämän seurauksena React-Nativen UI-komponentit

toimivat teoriassa täysin identtisesti natiivisti kehitetyn sovelluksen kanssa. Tämä ei kuitenkaan toteudu täysin käytännössä, sillä näitä natiiveja komponentteja on tarkoitus käyttää React-Nativen omien komponenttien kautta. Esimerkiksi Button-komponentti rakentaa automaattisesti alustakohtaisen natiivin painikkeen, jonka luominen Flutterilla vaati enemmän manuaalista työtä. Esimerkki tästä löytyy liitteestä C, jossa toteutetaan kuvion 24 näkymä.

Monet React-Nativen tarjoamat UI-komponentit ovat hyvin rajoittuneita muokattavuudeltaan ja jopa React-Nativen oma dokumentaatio suosittelee, että Button-komponentin sijaan käytettäisiin jotain muuta komponenttia kuten Pressable tai Touchable (React-Native 2020a). Alkuperäinen Button-komponentti ei salli esimerkiksi sen sisältämän tekstin muuttamista pieniin kirjaimiin Androidilla tai sen taustavärin muuttamista iOS:llä. Ylipäänsä, kun verrataan Flutteriin, on monilla UI-elementeillä vähemmän automaattisia ulkoasuasetuksia ja tiettyjen komponenttien rajoittuneisuus korostaa tätä ongelmaa, kun kehittäjän täytyy nyt luoda haluamansa painike täysin tyhjästä vain, koska hän halusi muuttaa painikkeen taustaväriä. On kuitenkin selvää, että mikäli kehittäjän ei tarvitse tehdä paljon ulkoasumuutoksia, on React-Nativella nopeampaa tuottaa ulkoasultaan natiivia muistuttava sovellus, sillä kehittäjän ei tarvitse huolehtia siitä, että eri alustoilla näytetään erilainen komponentti.

Animaatiotestin osalta React-Native-sovellus vastasi Flutterin toteutusta ja animaatio sekä sen kuva erotettiin omaan komponenttiinsa. Myös React-Native päivittää kaikki elementit, kun näkymä päivittyy, jonka vuoksi oli jälleen oleellista erottaa animoitavat elementit sekä varmistaa että animaatio oli toteutettu niin, että vain animaation arvot muuttuvat koko elementin sijaan. Lisäksi, jotta React-Native käyttäisi natiiveja animaatioita, oli parametri `useNativeDriver` asetettava todeksi. React-Native-sovelluksen kuvat ja niiden animaatiot luovan komponentin koodi löytyy liitteestä D.

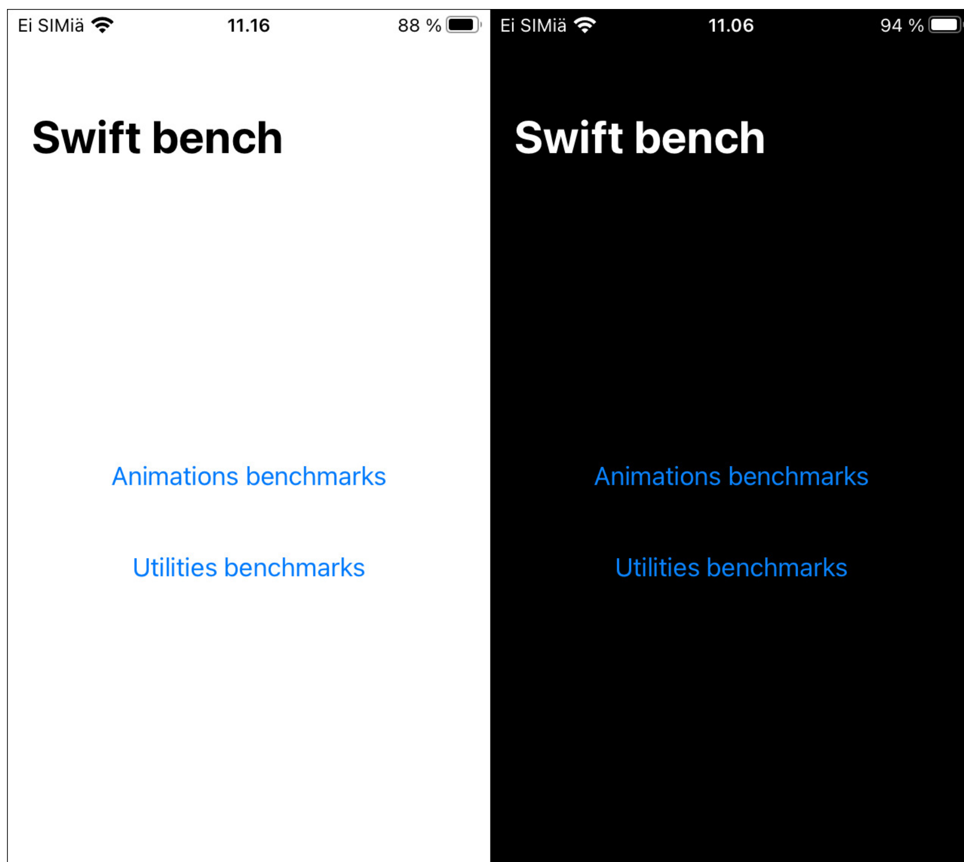
Kryptografinen testi ja GPS-testi vaativat molemmat kolmannen osapuolen kirjastojen käyttöä, sillä React-Native ei oletuksena tarjonnut kumpaakaan. Laadukkaiden kirjastojen löytäminen oli huomattavasti vaikeampaa verrattuna Flutteriin, koska React-Native ei erityisemmin suositellut eri kirjastoja ja osa dokumentaatioon linkitetyistä kirjastoista ei edes tukenut uusimpia kohdealustaversioita. GPS-testi tehtiin lopulta react-native-location-kirjastolla ja kryptografinen testi CryptoES-kirjastolla. Molemmat vaativat huomattavasti enemmän manuaalista asentamista Flutteriin verrattuna varsinkin iOS:n tapauksessa, sillä kirjastoja piti

manuaalisesti linkittää iOS:llä kääntyvään projektiin. Kirjastojen asentamisen jälkeen sijaintikirjasto vaati myös erillistä logiikkaa sijainnin käyttöoikeuden kysymiseen käyttäjältä, mutta muilta osin molempia kirjastoja oli yhtä helppo käyttää kuin Flutterin vastaavia. Myöskään Flutterin kanssa identtisen JSON-testin toteutus ei tuottanut vaikeuksia.

Yleiseltä kehityskokemukseltaan React-Native oli hieman heikompi Flutteriin verrattuna, sillä sen tarjoama hot-reload oli epäluotettava ja sovellusta ei Flutterista poiketen voinut käynnistää uudelleen ilman uudelleenkoontia. Lisäksi, koska React-Native käyttää natiiveja UI-komponentteja, on sen iOS-version ulkoasun testaaminen haastavaa, jos sovellusta kehitetään jollain muulla kuin macOS-käyttöjärjestelmällä. Tämä on tosin ratkaistavissa sillä, että koko sovellus kehitettäisiin macOS-käyttöjärjestelmän alla, jolloin jokaista versiota pystyttäisiin testaamaan jo emulaattorissa. Natiivien rajapintojen osalta myös React-Native tarjosi mahdollisuuden upottaa sovellukseen natiivia koodia joko Kotlinilla tai Swiftillä.

React-Native tarjoaa kaksi eri koontiversiota, joita ovat: Debug ja Release. Kaikki mittaukset tehtiin Release-versiolla, sillä se mahdollisti profiloinnin ja oli samalla optimoiduin sovellusversio. Flutterista eroten React-Native mahdollisti iOS-sovelluksen koonnin natiivia bittikoodia hyödyntäen, joka antaa Applen optimoida sovelluspakettia tehokkaammin ja tekee sovelluksen väärinkäytöstä ja purkamisesta väärissä käsissä hankalampaa (Apple 2019, 2020) . Tämä on ominaisuus, jota Apple tulee tulevaisuudessa vaatimaan kaikilta sovelluskaupan sovelluksiltaan ja Flutter ei sitä vielä tue. Bittikoodiksi kääntäminen ei kuitenkaan vaikuta sovelluksen suorituskykyyn.

5.3.3 Swift



Kuvio 25. Swift-sovellus ja SwiftUI:n automaattinen teemoitus laitteen asetuksista riippuen.

Seuraavana kehitettiin tutkielman iOS-alustan natiivi sovellus, joka toteutettiin Swiftillä. Natiivien sovelluksien tarkoituksena on tässä tutkielmassa toimia suorituskyvyn, mutta samalla myös kehityskokemuksen vertailukappaleena. Swift-sovellus kehitettiin ja profiloitiin täysin Xcodea ja sen työkaluja hyödyntäen.

Koska tässä sovelluksessa käyttöliittymän luomiseen käytettiin uutta UIKit:n korvaavaa SwiftUI-kirjastoa, oli sen kirjoittaminen deklaratiivista ja hyvin samankaltaista varsinkin Flutterin kanssa. Myös SwiftUI tarjosi hot-reload-toiminnallisuuden, mutta sitä pystyi hyödyntämään vain sovelluksen käyttöliittymän esikatselussa, ei emulaattorissa ajettavassa sovelluksessa. Tämä priorisoi ensimmäisenä kehittämään koko käyttöliittymän ja vasta sen jälkeen sovelluslogiikan. Itse sovelluksen kokoaminen emulaattoriin oli kuitenkin nopeampaa verrattuna Flutteriin tai React-Nativeen. Toisaalta Flutterin hot-restart ajoi useissa tilanteissa saman

asian, ollen samalla nopeampi.

Koska Swift-sovellus on täysin natiivi, oli käyttöliittymäkomponenttien käyttäminen ja muokkaaminen SwiftUI:n avulla helpompaa verrattuna React-Nativeen. Tämän lisäksi SwiftUI:lla toteutettu sovellus tuki automaattisesti tummaa tilaa, joka osasi älykkäästi muuttaa sovelluksen värejä kohdelaitteen asetusten mukaan. Ero tumman ja vaalean tilan välillä voidaan havaita kuviossa 25.

Sovelluksen animaatiotesti toteutettiin samalla kaavalla kuin aiemmin kehitetyissä sovelluksissa, joskin tällä kertaa kuvaa ja animaatiota ei eroteltu erilliseen komponenttiinsa, sillä SwiftUI osasi älykkäästi päivittää vain komponentteja, jotka muuttuivat. Flutteriin ja React-Nativeen verrattuna SwiftUI vaati myös vähemmän koodia käyttöliittymän tilan muuttamiseen, joka paransi koodin luettavuutta. Tämä animaatiotestin toteuttava komponentti näkyy liitteessä E.

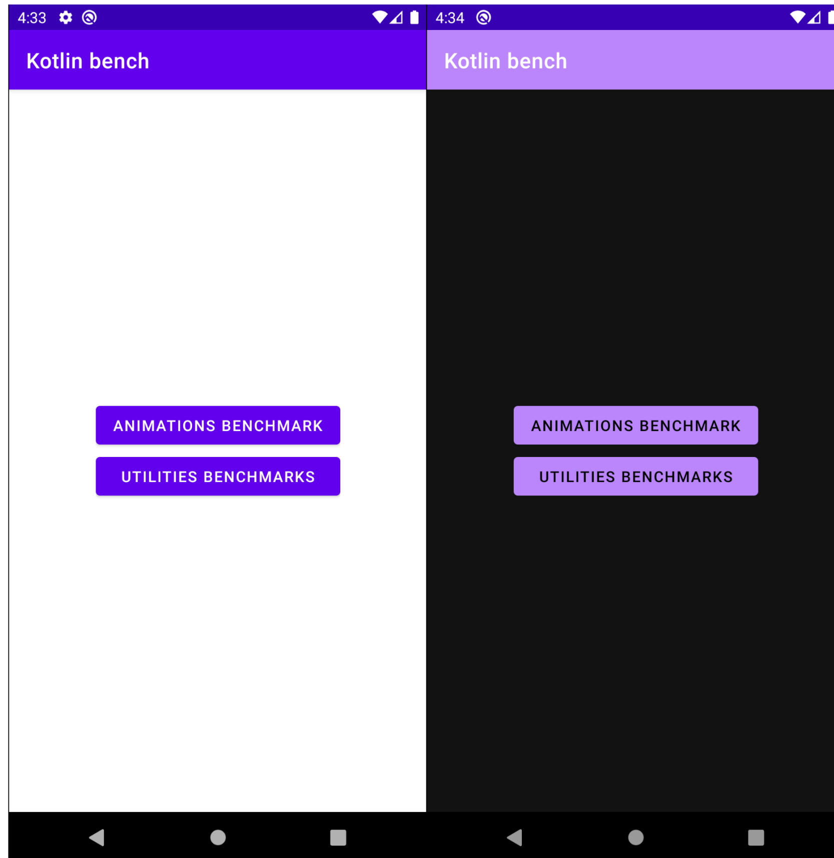
Kryptografinen testi toteutettiin Swiftin omalla CryptoKit-kirjastolla, kun taas GPS-testi käytti Swiftin CoreLocation-kirjastoa. Sijainnin hankkiminen Swiftillä oli kuitenkin huomattavasti monimutkaisempi prosessi ja vaati enemmän koodia verrattuna Flutteriin tai React-Nativeen. Kryptografinen testi ja JSON-testi sen sijaan olivat molemmat yksinkertaisia toteuttaa eivätkä vaatineet merkittävästi enemmän tai vähemmän koodia verrattuna aikaisempiin sovelluksiin.

Swift-sovellusta kehitettäessä merkittävin haaste oli, että suuri osa olemassa olevasta dokumentaatiosta oli tarkoitettu vanhalle UIKit-käyttöliittymäkirjastolle. Tämän seurauksena SwiftUI esimerkkejä oli haastavaa löytää varsinkin, koska myös UIKit-sovellukset käyttävät pohjanaan Swiftiä. Lisähaastetta aiheutti myös vanhan ja uuden käyttöliittymäkehityksen erilaisuus, sillä SwiftUI-käyttöliittymiä kehitetään deklaratiivisesti, kun taas UIKit-käyttöliittymiä kehitettiin pääasiassa Storyboard-työkalulla, jossa eri elementtejä asetettiin drag'n'drop-periaatteella. Käyttöliittymien kehitys oli silti nopeaa, joskin Flutterin ja React-Nativen mukainen myös emulaattorissa toimiva hot-reload-toiminnallisuus olisi tehnyt siitä nopeampaa. SwiftUI:n käyttäminen asettaa myös merkittävän rajoitteen tuetuille iOS-versioille, sillä SwiftUI:ta tuetaan virallisesti vasta iOS-versiosta 13 eteenpäin.

Swift tarjoaa Flutterin tavoin kolme koontiversiota, joita ovat: Debug, Profile ja Release.

Swift-sovelluksen kaikki testi suoritettiin Release-versiolla, jonka lisäksi valmis Swift-sovellus kääntyi automaattisesti React-Nativen tavoin bittikoodiksi.

5.3.4 Kotlin



Kuvio 26. Kotlin-sovellus ja Android UI:n automaattinen teemoitus.

Viimeisenä sovelluksena kehitettiin Android-alustalle natiivi Kotlin-sovellus. Sovellus toteutettiin kokonaan Android Studiolla ja sen käyttöliittymäkirjastona toimi Android UI. Vaihtoehtoinen käyttöliittymäkirjasto olisi ollut Jetpack Compose, mutta sitä päätettiin olla käyttämättä, koska sen kehitys on yhä beta-vaiheessa.

Kotlin-sovelluksen käyttöliittymän kehittäminen erosi merkittävästi muista tässä tutkielmassa kehitetyistä sovelluksista. Kotlinin käyttämä Android UI ei ole deklaratiivinen, vaan käyttöliittymät on tarkoitus luoda XML-tiedostoilla ja graafisella työkalulla, joka näkyy kuviossa 10. Käyttöliittymän ulkoasun lisäksi myös sovelluksen sisäinen reititys eri näkymien välil-

lä luodaan Android UI:n työkaluilla, joka aiheuttaa hyppimistä Kotlinilla tuotetun koodin ja Android UI:n työkalujen välillä. Android UI oli kuitenkin helppokäyttöinen ja se mahdollistaa monimutkaisenkin käyttöliittymän luomisen drag'n'drop-toiminnallisuudellaan.

Swiftin tavoin Android UI:n hot-reload toimi vain ulkoasun esikatseluun emulaattorissa ajettavan sovelluksen sijaan. Kotlin tarjosi kuitenkin Flutterin kanssa identtisen hot-restart-toiminnallisuuden, joka auttoi välttämään sovelluksen uudelleenkokoamista sovelluslogiikan muuttuessa. Myös Android UI osasi automaattisesti luoda sekä tumman että vaalean teeman, joka vaihtui kohdelaitteen asetusten mukaan. Tämä voidaan havaita kuviosta 26.

Koska sovelluksessa käytettiin Android UI:ta, oli animaatiotesti mahdollista toteuttaa tekemällä kaikki kolme intensiteettiä eri näkymiin dynaamisen generoimisen sijaan. Näin ollen eri näkymien luominen oli aikaisempiin sovelluksiin verrattuna nopeampaa. Itse animaatiot asetettiin kuitenkin jokaiseen kuvaan dynaamisesti ja ne aktivoitiin kaikki samanaikaisesti, kun käyttäjä käynnistää testin. Kotlinin animaatiotestiä kehittäessä kohdattiin yllättäen matalaa suorituskykyä, joka johtui siitä, että Kotlin ei jostain syystä automaattisesti käyttänyt animaatioiden kanssa laitteistokiihdytystä. Tämän aktivoiminen vaati vain yhden rivin koodia, mutta on hämmästyttävää, että Kotlin ei automaattisesti kytke suorituskyvyn kannalta näin tärkeää ominaisuutta päälle. Kyseinen laitteistokiihdytyksen aktivoiminen animaatiolle näkyy liitteessä F.

Tämän lisäksi Kotlin-sovelluksen kanssa haasteelliseksi osoittautui Kotlin-natiivien kirjastojen löytäminen. Koska Kotlin on yhä melko uusi, oli suurin osa olemassa olevista kirjastoista kehitetty Javalla. Tämä ei ollut suuri ongelma, koska Kotlin on täysin yhteensopiva Javan kanssa, mutta sovelluksen tulevaisuuden näkymien ja suorituskyvyn kannalta olisi parempi, jos kaikkeen löytyisi Kotlin-natiivi vaihtoehto. Esimerkiksi kryptografinen testi jouduttiin kehittämään Java-kirjastolla, kun taas sijaintiin löytyi Kotlin-kirjasto. Kotlinilla esiintyi siis samankaltaista ongelmaa, kuin SwiftUI:lla, jossa suurin osa dokumentaatiosta oli vain edeltävälle tekniikalle (Swiftillä UIKit:lle ja Kotlinilla Javalle). Kotlin sisälsi kuitenkin erittäin tehokkaan työkalun, joka osasi tunnistaa ja automaattisesti muuttaa Java-koodia Kotlin-koodiksi.

React-Nativen tavoin Kotlin tarjoaa kaksi eri koontiversiota, joita ovat: Debug ja Release.

Release-versio on mahdollista koota debuggable-lipulla, joka luo Kotlinin Profile-version. Testit tehtiin jälleen pääasiassa Release-versiolla, jonka lisäksi osa testeistä toteutettiin Profile-versiolla kappaleen 5.2.1 mukaisesti.

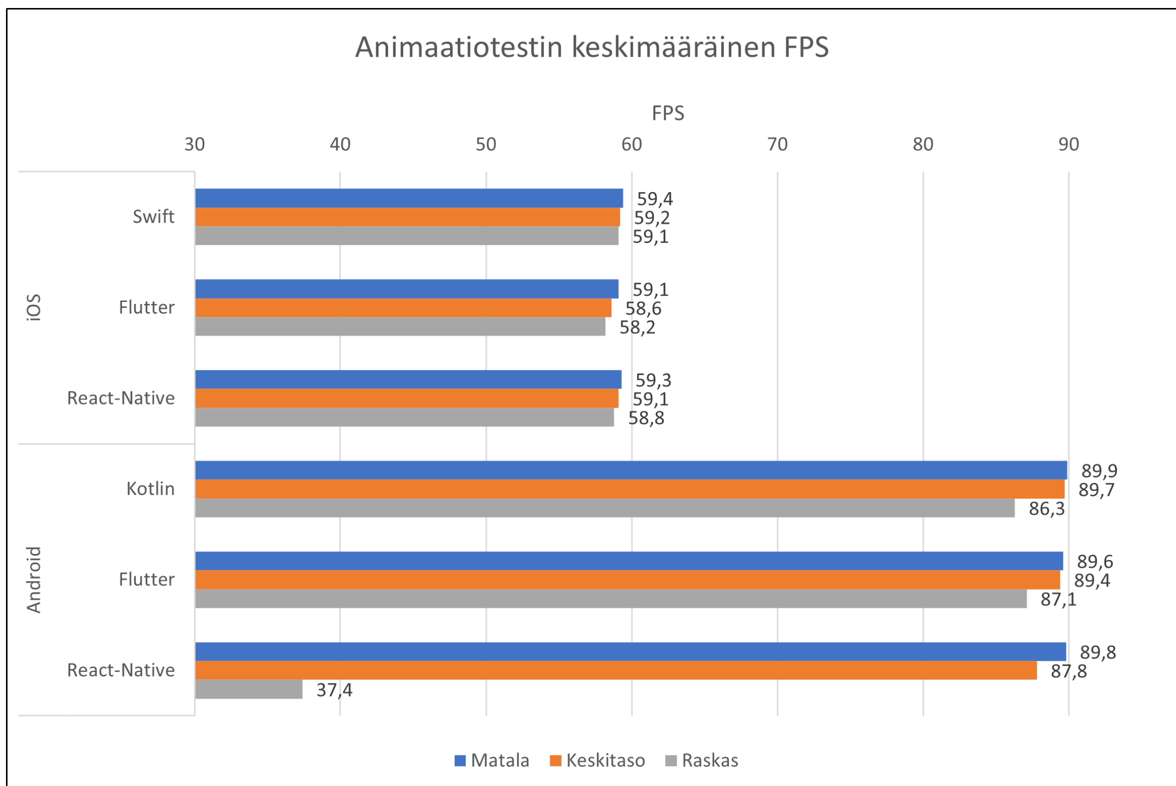
Ylipäänsä Kotlin-sovelluksen kehitys erosi eniten aikaisemmasta kolmesta sovelluksesta varsinkin käyttöliittymän osalta. Deklaratiivinen kehitys mahdollistaa nopeamman ulkoasun prototyypityksen, kun taas Android UI:n työkalut mahdollistavat tarkemman hienosäädön. Merkittävä ero aikaisempiin sovelluksiin on myös, että Kotlin vaati huomattavasti enemmän ylimääräistä koodia Android UI:n ja Kotlinin välisen interaktion toteuttamisen vuoksi. Kotlin vaati yleisesti myös enemmän koodia sen staattisen tyypityksen ja Javaa muistuttavan rakenteen seurauksena.

6 Tulokset

Seuraavissa kappaleissa käydään läpi testisovelluksen testien tulokset sekä pohditaan näiden tuloksien syitä ja niiden rajoitteita. Eri tuloksista kootaan lopuksi viitekehys sovelluskehittäjän, loppukäyttäjän ja kokonaisuuden kannalta, jota käytetään tekniikoiden lopulliseen arviointiin.

Sovelluspaketin kokoa ja laitteelta käytettyä tilaa lukuun ottamatta kaikki testit perustuvat eri suorituskertojen keskiarvoihin. Näiden tulokset pyöristettiin sitten testistä riippuen joko yhden desimaalin tai seuraavan kokonaisluvun tarkkuuteen.

6.1 Animaatiotesti

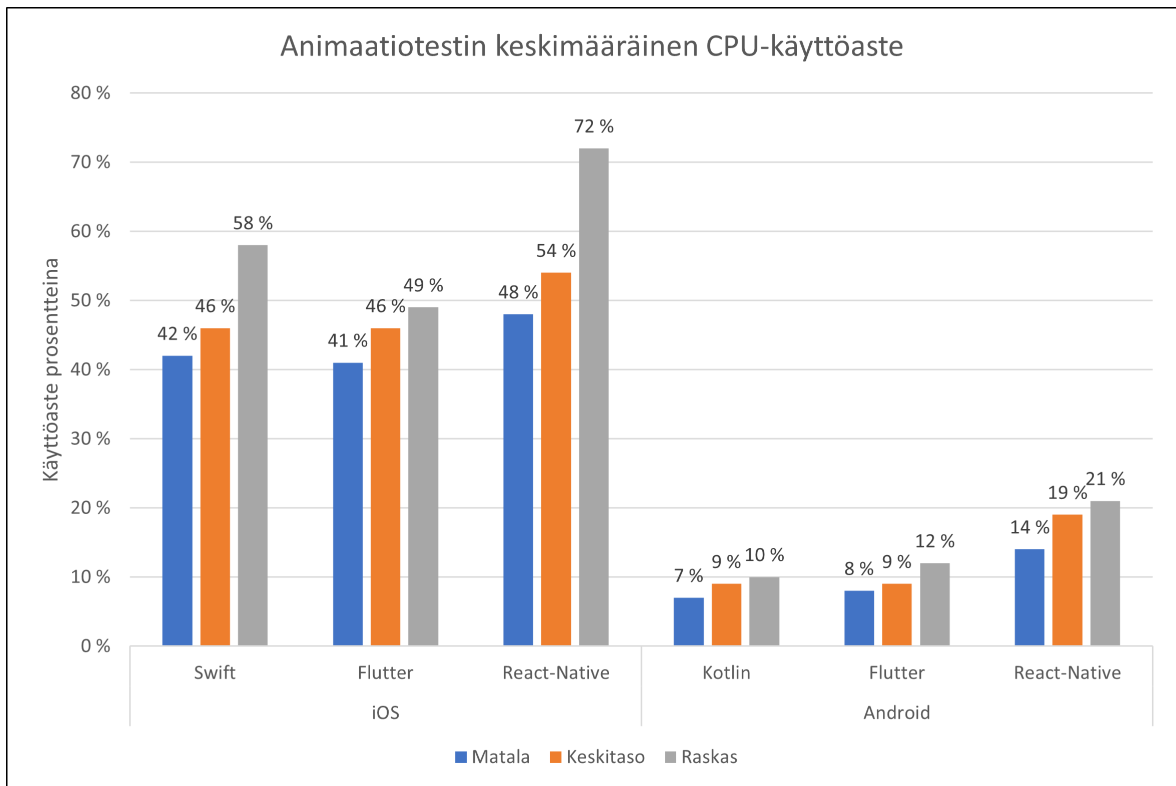


Kuvio 27. Animaatiotestin keskimääräinen FPS eri sovelluksilla.

Animaatiotesti oli testisovelluksen ainut graafinen testi, mutta sen tulokset olivat tästä huolimatta yhdet merkittävimmistä, sillä animaatioissa saavutettavat huonot tulokset näkyisivät käyttäjällä helpoiten. Kuten kuviosta 27 voidaan havaita, olivat iOS:n testisovellukset keskenään varsin samanveroisia saavutetun FPS:n osalta. Eri intensiteettien välillä natiivi Swift-sovellus oli keskimäärin aina paras suorituskyvyltään React-Nativen ollessa toisena ja Flutterin kolmantena. Flutter- ja React-Native-sovellus olivat kuitenkin erittäin lähellä natiivia suorituskykyä, ja loppukäyttäjän olisi hankala havaita eroa oikean maailman käyttötapauksissa.

Samaa ei kuitenkaan voinut sanoa Android-sovelluksista, joissa varsinkin React-Nativen keskimääräinen FPS putosi merkittävästi raskaimman intensiteetin animaatiossa. Kaikki alustan sovellukset menettivät enemmän suorituskykyä keskitason ja raskaan testin välillä kuin iOS-vastakappaleensa, joka saattaa osoittaa, että iOS on tehokkaampi animaatioiden kanssa. Toisaalta Android-sovellukset joutuivat työskentelemään huomattavasti kovemmin, sillä niiden tuli testilaitteen maksimoimiseksi saavuttaa 90 FPS 60 sijaan.

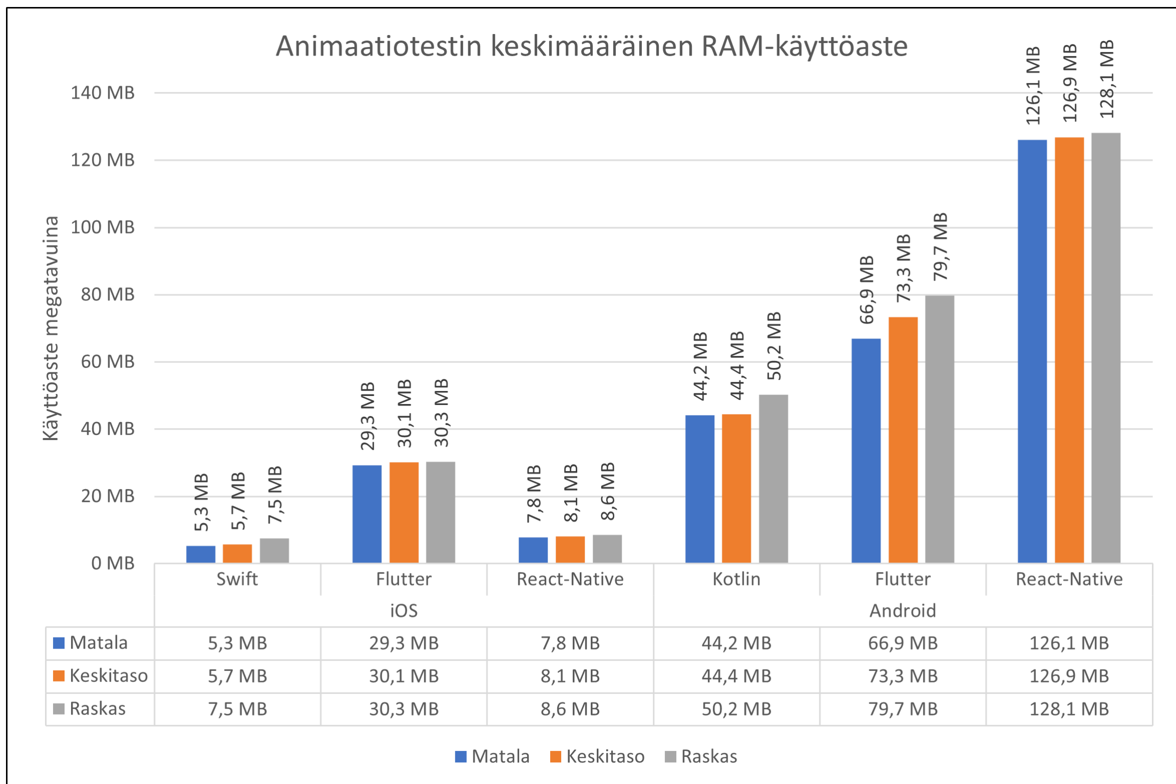
Ylipäänsä natiivi Kotlin-sovellus ja alustariippumaton Flutter-sovellus olivat erittäin lähellä toisiaan suorituskyvyssä ja käyttäjän olisi vaikea huomata animaation suorituskyvystä kumpaa tekniikka käytettiin. Sen sijaan React-Nativen merkittävä pudotus raskaassa animaatiotestissä osoittaa, että varsinkin Androidilla React-Native ei välttämättä skaalaudu yhtä hyvin verrattuna Flutteriin ja Kotliniin, jonka seurauksena sen käyttäminen olisi todennäköisesti käyttäjän havaittavissa tarpeeksi raskaassa animaatioympäristössä.



Kuvio 28. Animaatiotestin keskimääräinen CPU-käyttöaste eri sovelluksilla.

Prosessorin käytöltä Android-alustan natiivi Kotlin-sovellus oli hieman tehokkaampi alustariippumattomiin sovelluksiin verrattuna. Flutter oli hyvin lähellä samaa tehokkuutta, mutta React-Native sen sijaan vaati keskimäärin kaksinkertaisen määrän prosessorin resursseja, antaen samalla huonomman FPS-arvon.

iOS:n puolella tilanne oli myös mielenkiintoinen, sillä Flutter oli keskimäärin tehokkaampi kuin natiivi Swift-sovellus, varsinkin raskaassa animaatiotestissä. Kuten Android-sovelluksessa, React-Native vaati kuitenkin jälleen huomattavasti enemmän resursseja verrattuna Swift- tai Flutter-sovellukseen.



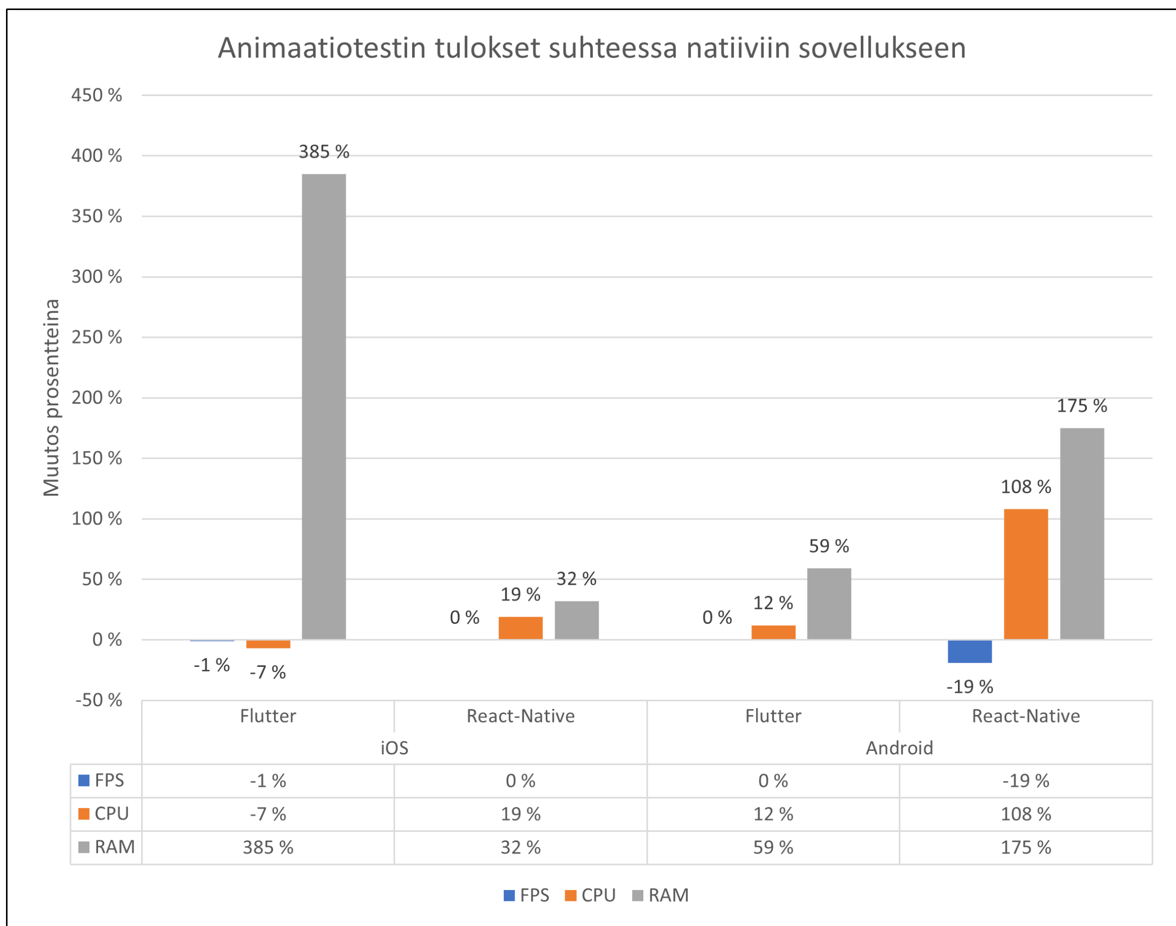
	Swift	Flutter	React-Native	Kotlin	Flutter	React-Native
	iOS			Android		
Matala	5,3 MB	29,3 MB	7,8 MB	44,2 MB	66,9 MB	126,1 MB
Keskitaso	5,7 MB	30,1 MB	8,1 MB	44,4 MB	73,3 MB	126,9 MB
Raskas	7,5 MB	30,3 MB	8,6 MB	50,2 MB	79,7 MB	128,1 MB

Kuvio 29. Animaatiotestin keskimääräinen RAM-käyttöaste eri sovelluksilla.

Muistin käytön osalta Swift oli selkeästi tehokkain iOS-sovelluksista React-Nativen seuratessa lähellä seuraavana. Mielenkiintoisesti Flutter vaati keskimäärin jopa yli kolminkertaisen määrän muistia muihin iOS-sovelluksiin verrattuna. Itse animaatiotestin raskaus ei näyttänyt vaikuttavan käytetyn muistin määrään kovin merkittävästi yhdelläkään iOS:n sovelluksista.

Kuten iOS:llä, myös Android-sovelluksista natiivi Kotlin-sovellus oli tehokkain, mutta tällä kertaa Flutter oli toisena ja React-Native viimeisenä käyttäen huomattavasti enemmän muistia edellä mainittuihin verrattuna. On mielenkiintoista, miksi React-Native on niin paljon tehokkaampi iOS-alustalla Android-vastakappaleeseensa verrattuna. Toisaalta ero saattaa johtua myös siitä, että Android-alustan testilaitteella mahdollisen muistin määrä on huomattavasti suurempi ja sovellus saattaa vain allokoida tietyn suhteutetun määrän muistia sen saatavuuden mukaan. Ero voi myös johtua Android-alustan testeissä vaaditun debuggable-lipun asetuksesta, joskin muut Android-alustan sovellukset antavat tästä huolimatta kilpailukykyi-

sempiä tuloksia.



	Flutter (iOS)	React-Native (iOS)	Flutter (Android)	React-Native (Android)
FPS	-1 %	0 %	0 %	-19 %
CPU	-7 %	19 %	12 %	108 %
RAM	385 %	32 %	59 %	175 %

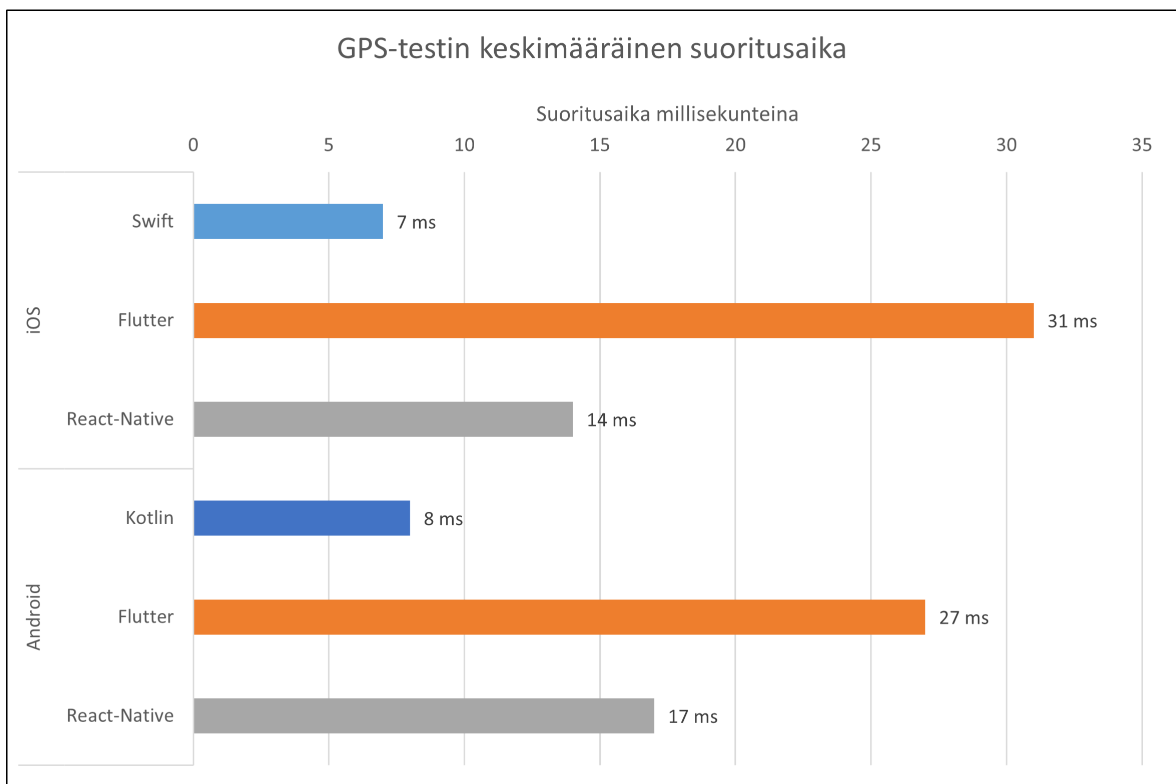
Kuvio 30. Animaatiotestin tulokset suhteessa natiiveihin sovelluksiin.

Kuviossa 30 Flutter- ja React-Native-sovellusten tulokset on suhteutettu molempien alustojen natiiveja sovelluksia vastaan. Kuten kuviosta voidaan havaita, oli iOS:llä suorituskyvyltään paras alustariippumaton sovellus React-Native, saavuttaen Swift-sovellusta vastaavan FPS:n, ja käyttäen 19 prosenttia enemmän prosessoria ja 32 prosenttia enemmän muistia. Flutter saavutti melkein saman suorituskyvyn käyttäen keskimäärin jopa 7 prosenttia vähemmän prosessoria kuin Swift-sovellus, mutta kuitenkin massiivisen 385 prosenttia enemmän muistia, joka varsinkin iOS:llä on merkittävää pienemmän muistikapasiteetin vuoksi.

Androidilla Flutter oli päinvastaisesti suorituskykyisempi React-Nativeen verrattuna, käyttäen 12 prosenttia enemmän prosessorin resursseja ja 59 prosenttia enemmän muistia kuin natiivi Kotlin-sovellus. React-Native saavutti keskimäärin 19 prosenttia huonomman suori-

tuskyvyn Kotliniin verrattuna, ja samalla vaati 108 prosenttia enemmän prosessoria ja 175 prosenttia enemmän muistia. Tämä näyttäisi vahvistavan teoriaa, että React-Native ei ole yhtä tehokas Android-alustan visuaalisissa työtaakoissa verrattuna iOS:n vastaaviin. Flutter sen sijaan näyttäisi suorituskyvyltään olevan hyvin lähellä natiivia sovellusta alustasta riippumatta, joskin se vaatii silti enemmän resursseja vastaavan suorituskyvyn saavuttamiseksi.

6.2 GPS-testi

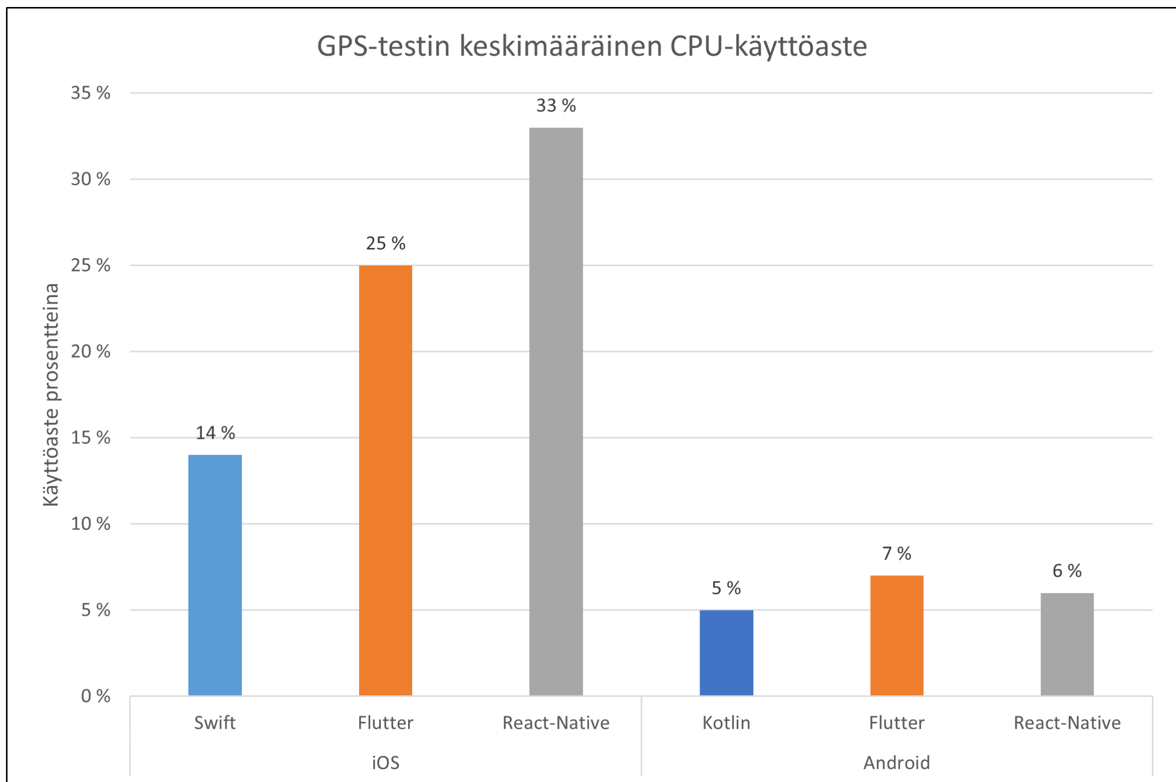


Kuvio 31. GPS-testin keskimääräinen suoritusaika eri sovelluksilla.

Kuten kuviosta 31 nähdään, olivat natiivit sovellukset GPS-testin osalta selkeästi nopeimpia. Swift- ja Kotlin-sovellukset olivat keskimäärin jopa kaksi kertaa nopeampia alustariippumattomiin vastakappaleisiinsa verrattuna, joka oli toisaalta odotettu tulos, sillä niiden suora kommunikointi laitteen rajapintojen kanssa pitäisi olla nopeampaa.

Kuitenkin yllättäen Flutter oli molemmilla alustoilla hitain, jonka lisäksi se käytti keskimäärin yli kolme kertaa enemmän aikaa saman sijainnin saamiseen kuin natiivi sovellus. Tämä

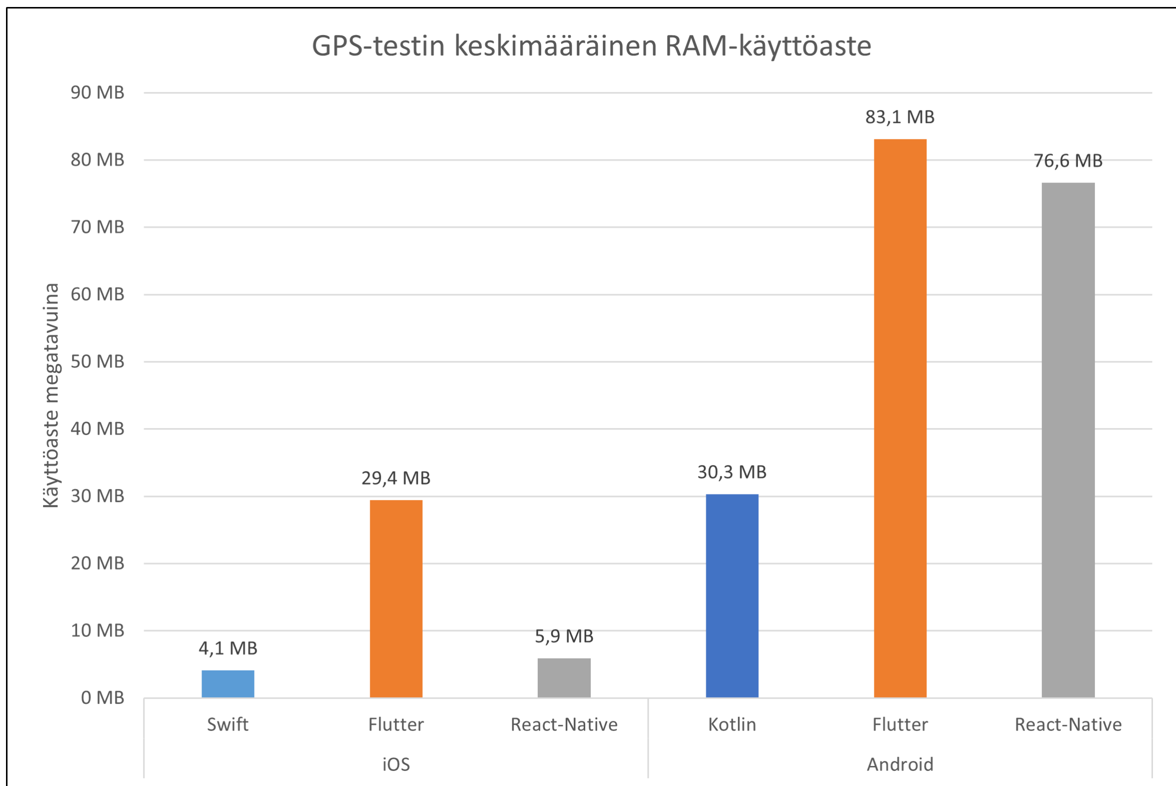
oli yllättävää, koska React-Nativen odotettiin kärsivän tässä testissä sen API-siltojen sekä tulkin käytön vuoksi. Vaikka React-Native oli hitaampi kuin kumpikaan natiiveista sovelluksista, oli se silti huomattavasti nopeampi kuin Flutter.



Kuvio 32. GPS-testin keskimääräinen CPU-käyttöaste eri sovelluksilla.

Prosessorin käytön osalta natiivit sovellukset olivat jälleen tehokkaampia verrattuna alustariippumattomiin sovelluksiin. Alustakohtaisesti voitiin kuitenkin havaita eroja, sillä React-Native vaati huomattavasti enemmän resursseja iOS:llä muihin sovelluksiin verrattuna. Myös Flutter vaati enemmän resursseja Swiftiin verrattuna, mutta kuitenkin vähemmän kuin React-Native.

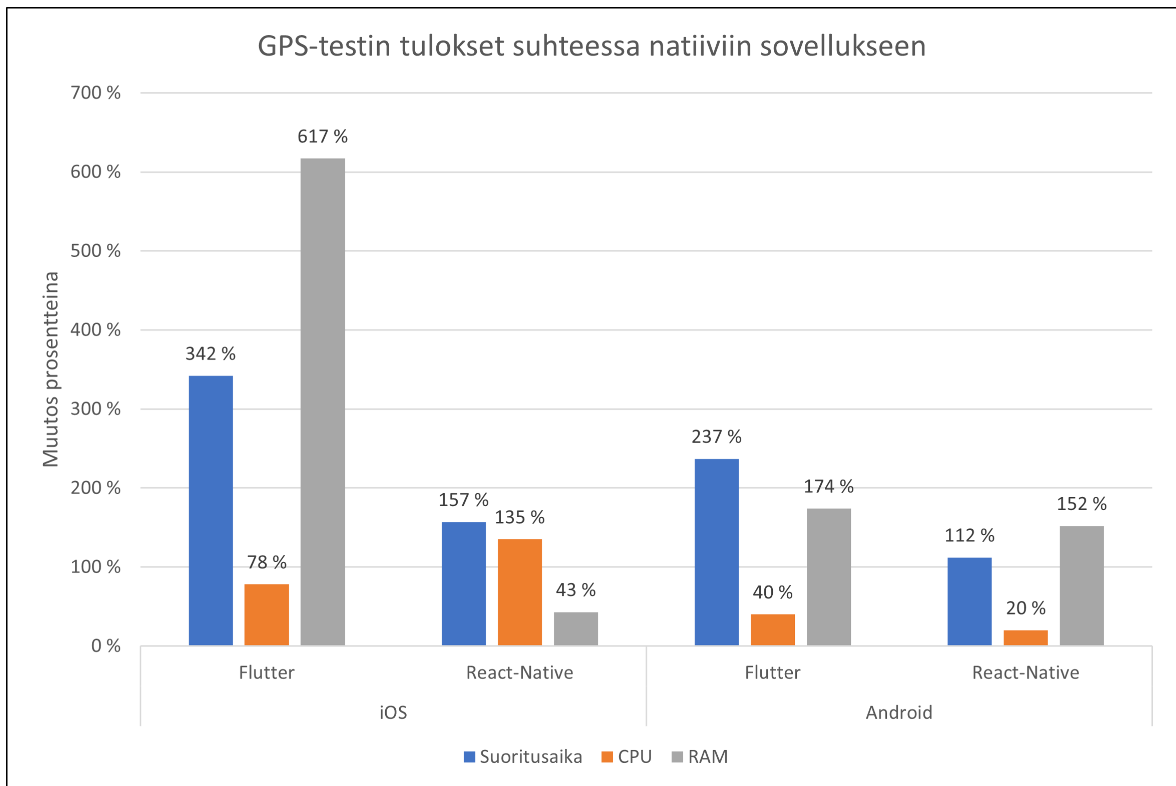
Tämä ei toistunut Android-alustalla, jonka sovellusten prosessorin käyttö oli jokaisella hyvin samalla tasolla. Flutter vei keskimäärin eniten resursseja, mutta ero natiiviin Kotliniin ja React-Nativeen oli hyvin pieni. Android- ja iOS-alustoilla havaittu prosessorin käyttöasteen ero saattaa myös johtua alustojen eri tarkkuusasetuksista, jotka perusteltiin kappaleessa 5.1.2.



Kuvio 33. GPS-testin keskimääräinen RAM-käyttöaste eri sovelluksilla.

Muistin käytön osalta tehottomin oli jälleen Flutter-sovellus, joka vei molemmilla alustoilla enemmän muistia muihin sovelluksiin verrattuna. Ero oli animaatiotestin tavoin suurempi iOS:llä, jolla Flutter vei huomattavasti enemmän muistia kuin Swift tai React-Native. Flutter oli myös Androidilla merkittävästi tehottomampi kuin Kotlin, mutta ero React-Nativeen ei ollut yhtä merkittävä iOS:ään verrattuna.

React-Nativen tehokkuus iOS:llä oli hyvin lähellä natiivia sovellusta, mutta Androidilla React-Native oli animaatiotestin tavoin huomattavasti tehottomampi.



Kuvio 34. GPS-testin tulokset suhteessa natiiveihin sovelluksiin.

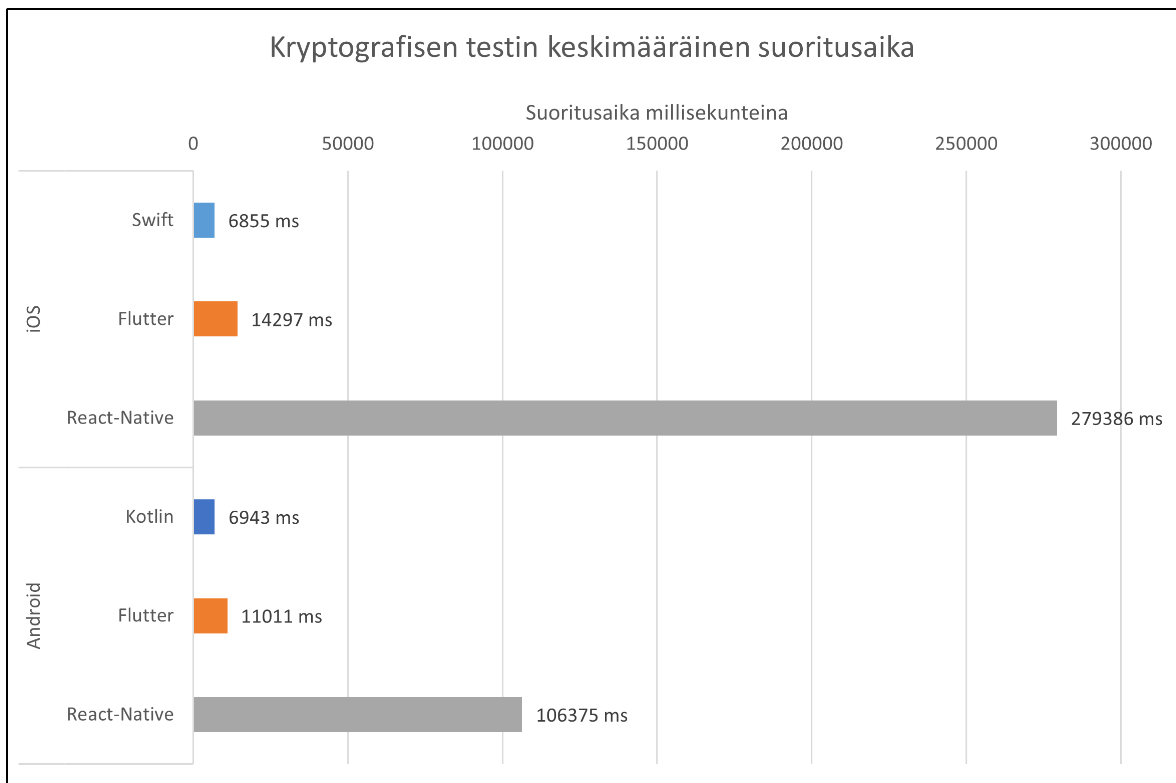
Flutterin suhteellinen tehottomuus tässä testissä voidaan havaita kuviosta 34, jossa Flutter on iOS:llä React-Nativeen ja natiiviin sovellukseen verrattuna selkeästi tehottomampi, käyttäen yli kaksi kertaa enemmän aikaa sijainnin hakemiseen ja yli kuusinkertaisen määrän muistia. Flutter käyttää kuitenkin jälleen vähemmän prosessoria React-Nativeen verrattuna, joskin tämä näkyy nyt merkittävästi hitaampana suorituksena. Molemmat sovellukset ovat silti tehottomampia ja merkittävästi hitaampia natiiviin vastakappaleeseensa verrattuna.

Android-alustalla Flutterin ja React-Nativen ero keskenään on pienempi, mutta Flutter on silti hitain ja käyttää keskimääräisesti eniten prosessoria ja muistia. Näistä tuloksista voidaan siis päätellä, että ainakin GPS:n osalta React-Native on Flutteriin verrattuna parempi vaihtoehto molemmilla alustoilla. React-Native häviää Flutterille ainoastaan prosessorin käytössä iOS:llä, mutta antaa tätä vastaan yli kaksi kertaa nopeamman suorituksen Flutteriin verrattuna.

Suoritusajan osalta tulos on erilainen verrattuna Biørn-Hansen ym. (2020) tutkimuksessa

käytetyn GPS-testin tuloksiin, jossa React-Native-sovellus oli keskimäärin hieman hitaampi Flutteriin verrattuna. Toisaalta muistin ja prosessorin käytön osalta tutkimuksen tulokset ovat tämän tutkielman mukaiset, joka mahdollistaa eron syyksi myös erilaisen testimetodologian. Biørn-Hansen ym. (2020) eivät tutkimuksessaan käy kovin tarkasti läpi, kuinka heidän GPS-testi on toteutettu.

6.3 Kryptografinen testi



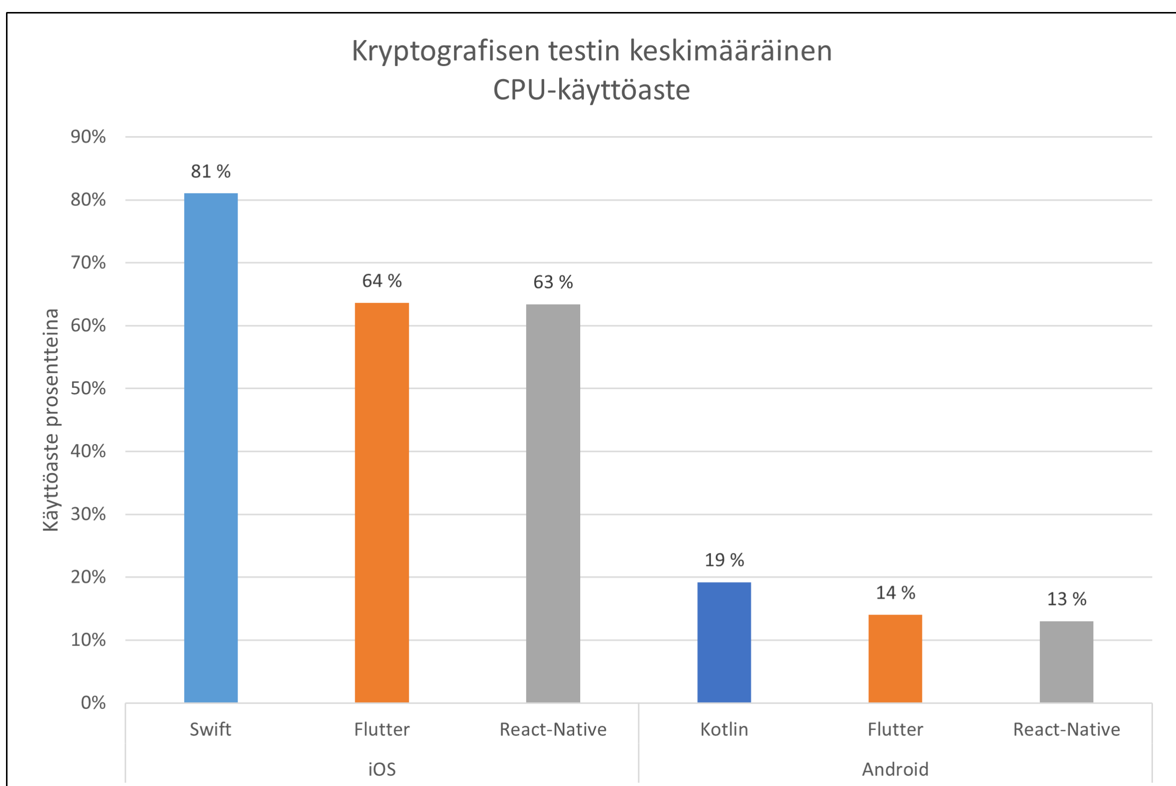
Kuvio 35. Kryptografisen testin keskimääräinen suoritusaika eri sovelluksilla.

Kryptografinen testi oli testisovelluksen testeistä ylivoimaisesti raskain prosessorin kannalta ja siksi sen tuloksilla voidaan arvioida suorituskyvyn skaalautumista käyttötapauksissa, joissa laitteella halutaan tehdä keskimääräistä raskaampaa prosessointia.

Kuten kuviosta 35 voidaan havaita, olivat natiivit sovellukset jälleen ylivoimaisesti nopeimpia. Seuraavana oli Flutter, joka molemmilla alustoilla vaati keskimäärin yli kaksi kertaa enemmän suoritusaikaa. Merkittävästi hitaampi molemmilla alustoilla oli kuitenkin React-

Native, joka sekä iOS:llä että Androidilla vaati yli kymmenen kertaa enemmän suoritusaikaa natiiviin sovellukseen verrattuna. Tämä testi havainnollistaa selkeästi tulkillisen lähestymistavan tehottomuutta tilanteessa, jossa sovellus joutuu nopeaan tahtiin lähettämään kutsuja natiiveille rajapinnoille, ja jonka seurauksena React-Nativen tulokset kärsivät tässä testissä merkittävästi.

React-Native oli niin hidas, että iOS-alustalla voitiin havaita lämpötilasta johtuvaa prosessorin nopeuden hidastamista, joka pidensi suoritusta entisestään. Tämä on osittain syyllinen React-Nativen selvästi hitaampaan tulokseen iOS:llä Androidiin verrattuna, sillä Android-sovellus ei joutunut laskemaan prosessorin nopeutta liian korkean lämpötilan vuoksi.

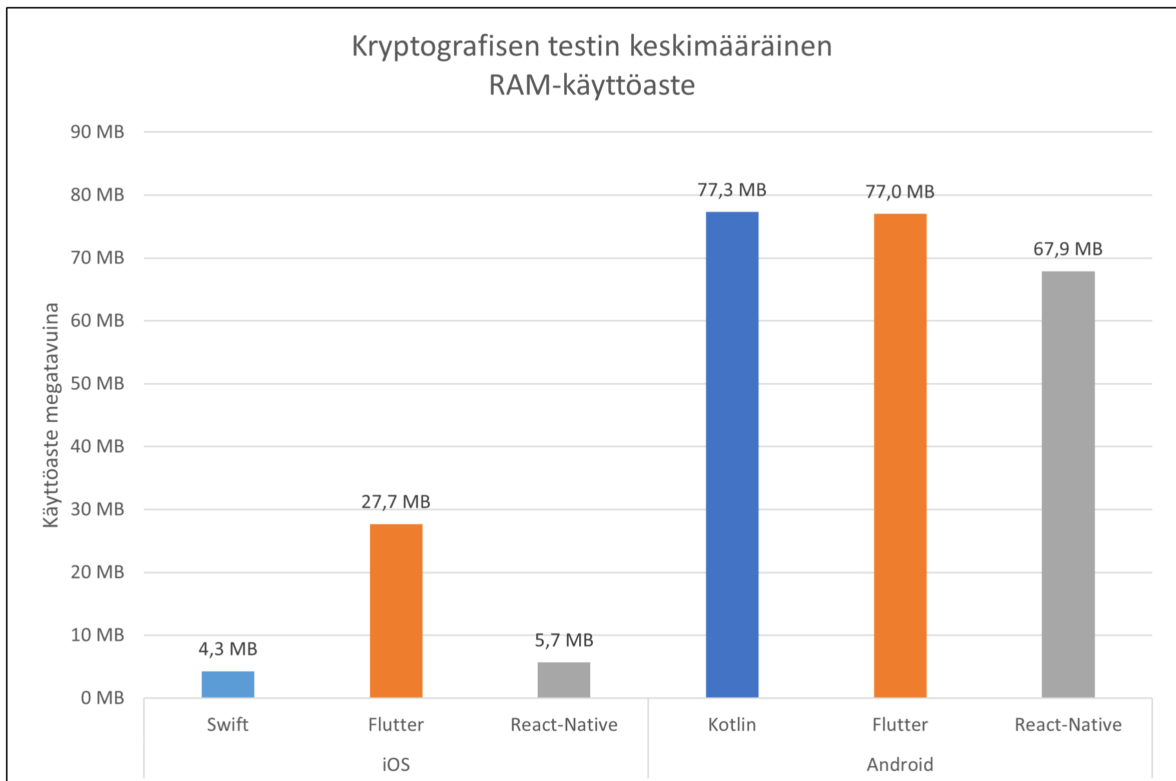


Kuvio 36. Kryptografisen testin keskimääräinen CPU-käyttöaste eri sovelluksilla.

Prosessorin käytön osalta iOS:n natiivi Swift-sovellus vei selvästi eniten tehoa, jonka voidaan katsoa johtuvan sen paremmasta optimoinnista, mahdollistaen raskaamman prosessorin käytön lyhyemmän suoritusajan saavuttamiseksi. Flutter- ja React-Native-sovellukset veivät molemmat vähemmän resursseja kuin Swift ja keskenään lähes saman verran resursseja,

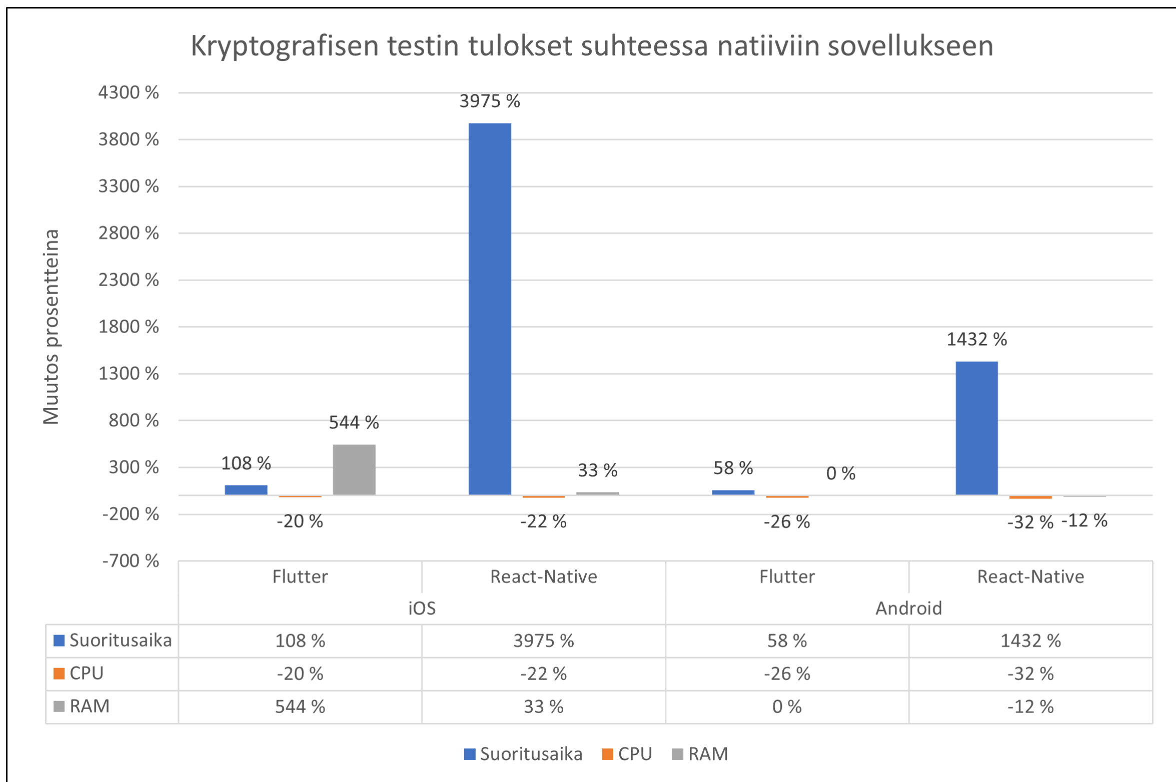
mutta Flutter oli siitä huolimatta kymmeniä kertoja nopeampi kuin React-Native.

Android-sovelluksilla tilanne oli hyvin samankaltainen, natiivin Kotlin-sovelluksen vieden eniten prosessorin resursseja, jonka jälkeen Flutter- ja React-Native-sovellukset veivät keskenään melkein saman verran resursseja. Jälleen kerran Flutter oli silti huomattavasti nopeampi samalla määrällä resursseja kuin React-Native.



Kuvio 37. Kryptografisen testin keskimääräinen RAM-käyttöaste eri sovelluksilla.

Kuten kuviosta 37 voidaan havaita, on tilanne iOS:llä muistin osalta identtinen GPS-testin kanssa. Flutter vie huomattavasti enemmän muistia muihin tekniikoihin verrattuna, kun taas React-Native vie vain hieman enemmän muistia kuin natiivi Swift. Androidilla tilanne on kuitenkin täysin päinvastainen ja molemmat Flutter ja React-Native vievät keskimäärin vähemmän muistia kuin natiivi Kotlin-sovellus, joskin ero varsinkin Flutter- ja Kotlin-sovelluksen välillä on erittäin pieni.



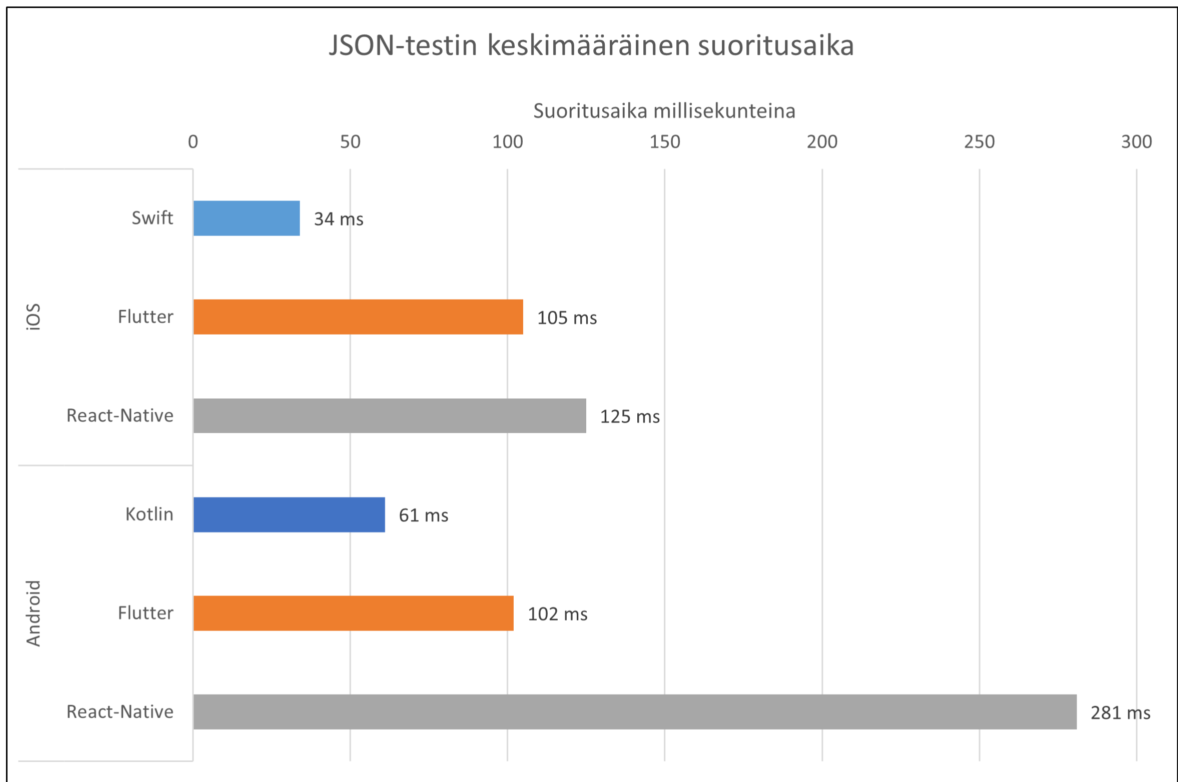
	iOS Flutter	iOS React-Native	Android Flutter	Android React-Native
Suoritusaika	108 %	3975 %	58 %	1432 %
CPU	-20 %	-22 %	-26 %	-32 %
RAM	544 %	33 %	0 %	-12 %

Kuvio 38. Kryptografisen testin tulokset suhteessa natiiveihin sovelluksiin.

Kun tarkastellaan Flutter- ja React-Native-sovelluksien suhdetta kumpaankin natiiviin sovellukseen, on selvää, että Flutter on näistä kahdesta parempi alustariippumaton vaihtoehto molemmilla alustoilla. Flutter on hitaampi kuin natiivit sovellukset, mutta käyttää keskimäärin vähemmän prosessorikapasiteettia ja sen ainut ongelma onkin iOS:n tapauksessa merkittävä muistin kulutus.

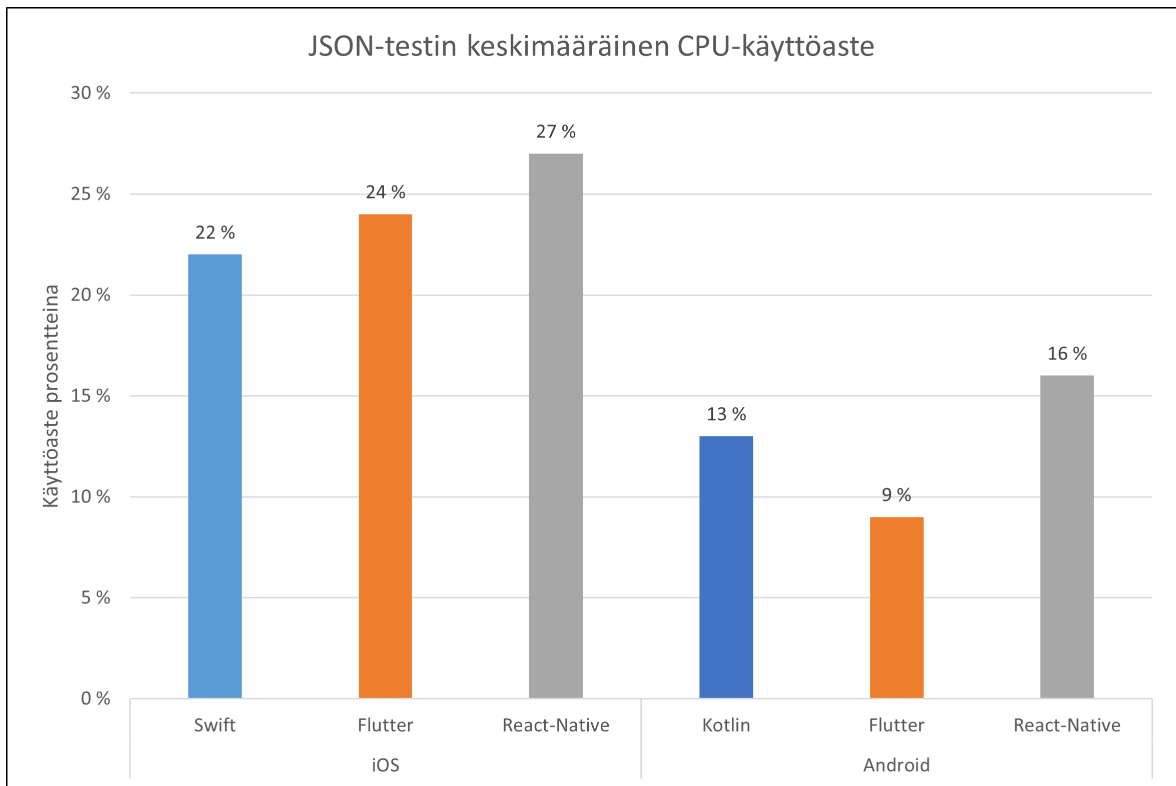
React-Native sen sijaan olisi tämänkaltaisessa työkuormassa lähes käyttökelvoton sen erittäin hitaan suorituksen vuoksi. Toisaalta kryptografinen testi esittää pahinta mahdollista tapausta, jonka vuoksi seuraavan kappaleen JSON-testin tulokset ovat parempia kuvastamaan suorituskykyä realistisemmalla työkuormalla.

6.4 JSON-testi



Kuvio 39. JSON-testin keskimääräinen suoritusaika eri sovelluksilla.

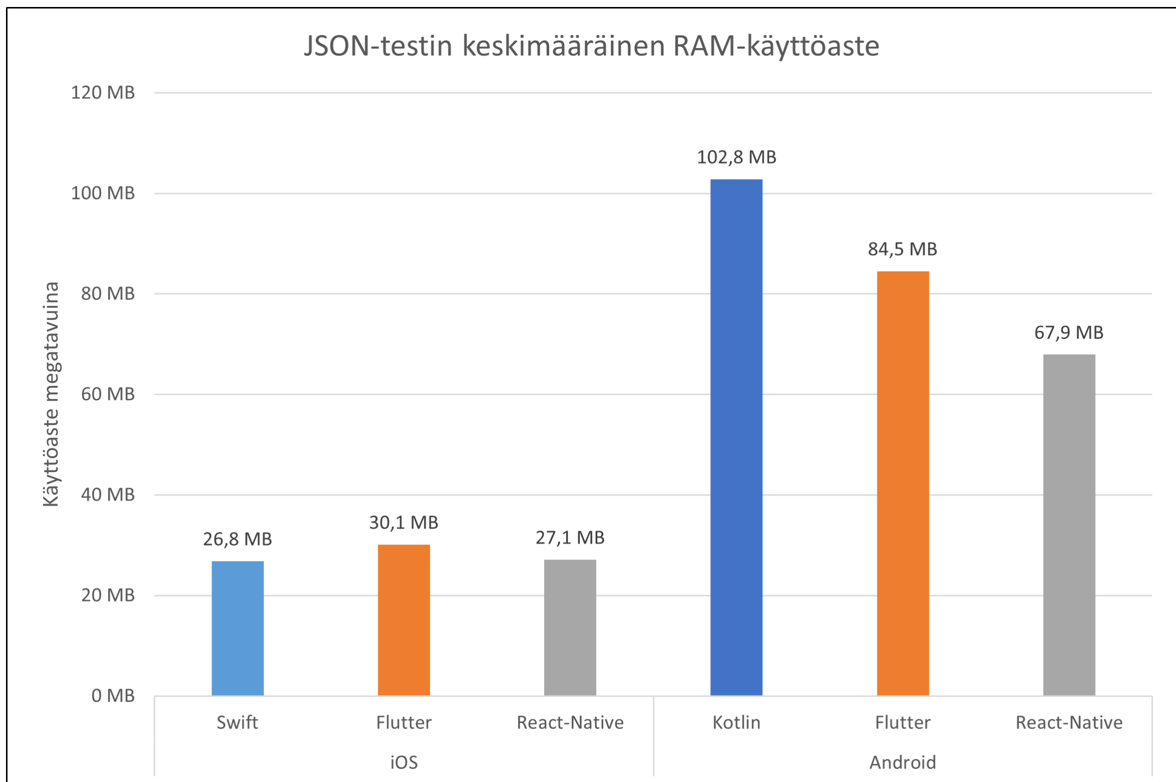
Kuten kryptografisessa testissä, natiivit sovellukset ovat selkeästi nopeimpia myös JSON-testissä. Flutter on jälleen molemmilla alustoilla toisiksi nopein, mutta tällä kertaa React-Native on iOS:llä huomattavasti kilpailukykyisempi. Androidilla React-Native on silti Flutteriin verrattuna melkein kolme kertaa hitaampi, mutta suoritus ei kestä poikkeuksellisen kauan kuten kryptografisessa testissä.



Kuvio 40. JSON-testin keskimääräinen CPU-käyttöaste eri sovelluksilla.

Prosessorin osalta Swift on iOS:llä tehokkain, Flutter toisiksi tehokkain ja React-Native tehottomin. Vaikka eri iOS-sovellusten resurssien käyttö on varsin samankaltaista, käyttää React-Native silti niitä havaittavasti enemmän verrattuna Flutteriin ja varsinkin Swiftiin.

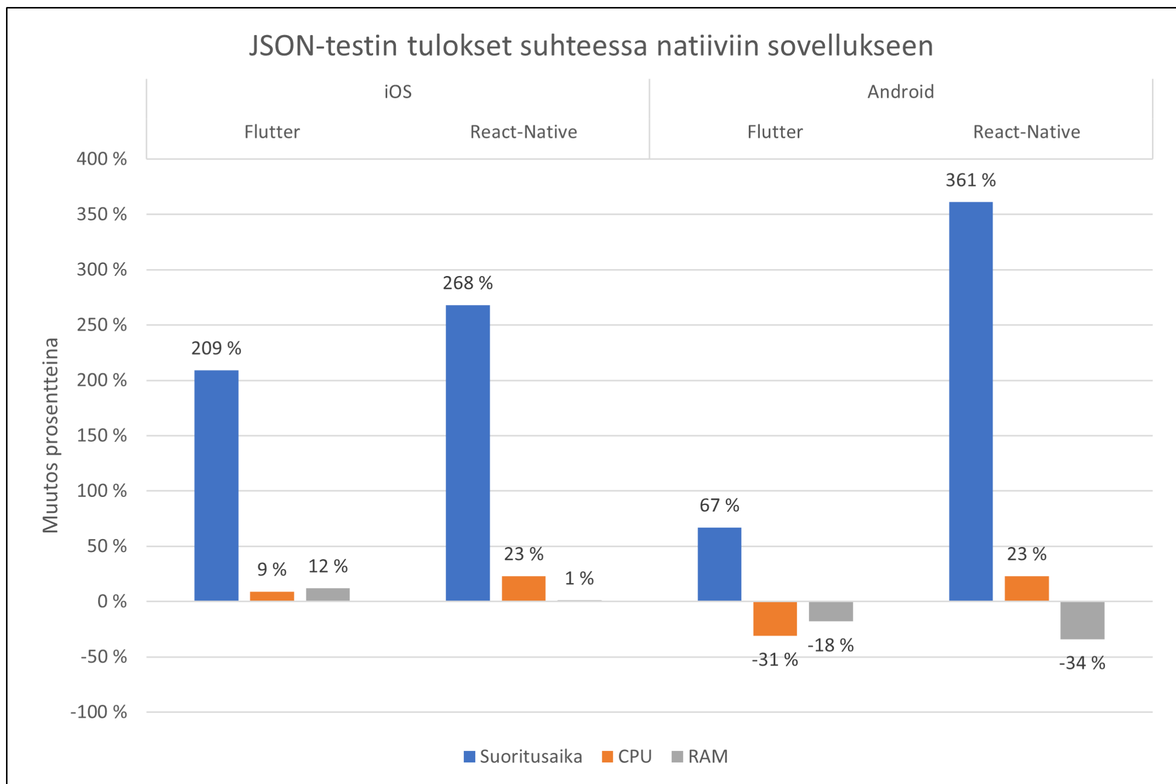
React-Native käyttää eniten resursseja myös Androidilla, joskin tällä kertaa se on lähempänä natiivin Kotlin-sovelluksen kulutusta. Yllättäen Flutter-sovellus käyttää Android-sovelluksista selvästi vähiten resursseja, joskin tämä heijastuu natiiviin Kotlin-sovellukseen verrattuna melkein kaksinkertaisena suoritusaikana.



Kuvio 41. JSON-testin keskimääräinen RAM-käyttöaste eri sovelluksilla.

Kryptografiseen ja GPS-testiin verrattuna JSON-testin iOS:n tulokset eroavat siten, että tällä kertaa kaikki tekniikat käyttävät suhteellisen saman määrän muistia. Swift on niukasti tehokkain ja Flutter tehottomin, mutta vaaditut resurssit ovat silti erittäin lähellä toisiaan varsinkin, kun verrataan aikaisempien testien tuloksissa havaittuihin eroihin.

Android-sovelluksilla tilanne taas muistuttaa kryptografisen testin tuloksia siten, että natiivi Kotlin-sovellus vie jälleen eniten muistia, mutta tällä kertaa Flutter ja React-Native vievät molemmat selvästi vähemmän muistia natiiviin sovellukseen verrattuna. React-Native vie taas vähemmän muistia kuin Flutter, mutta toisaalta sen suoritusaika on näillä resursseilla huomattavasti pidempi.



Kuvio 42. JSON-testin tulokset suhteessa natiiveihin sovelluksiin.

Kun tarkastellaan Flutter- ja React-Native-sovelluksia suhteessa natiiveihin vastakappaleisiinsa, voidaan havaita, että Flutter näyttäisi taas olevan parempi vaihtoehto molemmilla alustoilla, joskin ero on huomattavasti pienempi kuin kryptografisessa testissä. iOS:llä Flutter vie ainoastaan enemmän muistia kuin React-Native ja on muuten kaikin tavoin parempi, kun taas Androidilla Flutter vie selvästi vähemmän muistia ja prosessorin resursseja, ollen samalla vain 67 prosenttia hitaampi natiiviin Kotlin-sovellukseen verrattuna.

JSON-testi kuitenkin osoittaa, että vaikka React-Nativen tulokset olivat todella heikot raskaassa kryptografisessa testissä, ovat sen tulokset silti kilpailukykyiset oikean maailman käyttötapausta paremmin kuvaavassa testissä. Toisaalta tulokset ovat varsinkin Androidilla silti merkittävästi heikommat kuin Flutterilla. JSON-testin tuloksia voidaan myös verrata Biørn-Hansen ym. (2020) tutkimuksen tietojärjestelmän lukemiseen perustuvaan testiin, jossa tästä testistä eroten React-Native oli keskimäärin Flutteria nopeampi ja käytti vähemmän muistia. React-Native käytti kuitenkin myös Biørn-Hansen ym. (2020) tutkimuksessa enemmän prosessorin resursseja.

Syy Flutterin nopeampaan suoritukseen tämän tutkielman testissä on todennäköisesti eroava testimetodologia sekä Flutterin eri päivitykset, joita Biørn-Hansen ym. (2020) teettämän tutkimuksen ja tämän tutkielman välillä on valmistunut.

6.5 Yleiset mittarit

Suorituskykyyn kohdistuvien testien lisäksi jokaisesta testisovelluksesta kerättiin kolme yleistä muuttujaa, joita ovat: asennuspaketin koko, sovelluksen laitteelta viemä tila ja sovelluksen käynnistymiseen kuluva aika. Taulukossa 5 on kuvattu molempien alustojen kaikkien sovellusten koottujen sovelluspakettien koko (Androidilla .APK ja iOS:llä .IPA). Sovelluspaketin koko kuvastaa sitä määrää, jonka käyttäjä joutuisi lataamaan, kun sovellus asennetaan sovelluskaupasta. Taulukossa 5 on myös sovellusten asennuksen jälkeen laitteelta käyttämä tila.

Taulukko 5: Eri sovellusten sovelluspakettien koko ja laitteella käytetty tila.

Tekniikka	APK	IPA	Käytetty tila Androidilla	Käytetty tila iOS:llä
Kotlin	3,9 MB	N/A	12,51 MB	N/A
Swift	N/A	178 KB	N/A	651 KB
Flutter	7,8 MB	87,9 MB	64,4 MB	212,9 MB
React-Native	9,2 MB	15,7 MB	39,6 MB	53,6 MB

Androidin puolella Kotlin on selvästi tehokkain sekä asennuspaketin koon että laitteelta käytetyn tilan osalta. Seuraavaksi tehokkain on Flutter, joka ylitti odotukset päihittäen myös React-Nativen Biørn-Hansen ym. (2020) tutkimuksesta poiketen. Tämä ero johtui todennäköisesti Flutterin version 1.22 mukana tulleista optimoinneista (Flutter 2020a), jotka lisättiin Biørn-Hansen ym. (2020) tutkimuksen jälkeen.

Tilanne on iOS:llä erilainen, sillä vaikka molemmat alustariippumattomat sovellukset häviävät jälleen erittäin merkittävästi natiiville Swift-sovellukselle, on Flutterin .IPA-paketti jo huomattavasti suurempi edes React-Nativen sovelluspakettiin verrattuna. Tämä suuri ero johtuu todennäköisesti Flutterin widget-perusteisesta luonteesta, jonka vuoksi sen pitää tuo-

da sovelluspaketissa jokainen sovelluksessa käytettävä komponentti. On kuitenkin epäselvää, miksi ero on niin paljon suurempi iOS:llä Androidiin verrattuna, jonka vuoksi tarkempi syyllinen saattaakin olla .APK- ja .IPA-tiedostojen poikkeava kompressointi.

Myös React-Nativen sovelluspaketti on iOS:llä suurempi suhteessa Androidin vastaavaan, jonka yksi syy saattaa olla iOS-sovellusten automaattinen vahva salaus. Tämä vahva salaus tekee sovelluspaketista turvallisemman, mutta sen kompressoitumisesta epätehokkaan. Ero React-Nativen Android-sovelluspakettiin on silti suhteellisen pieni ja paketin koko on hyvin lähellä Biørn-Hansen ym. (2020) saavuttamaa 9,7 megatavua.

Laitteelta käytetyn tilan kannalta tilanne on hyvin samankaltainen. Androidilla Kotlin-sovellus vie huomattavasti vähemmän tilaa kuin seuraavaksi vähiten vievä React-Native. React-Nativen käyttämä tila on kuitenkin vähäistä, kun verrataan sitä Flutterin viemään tilaan, joka on melkein kaksinkertainen pelkästään React-Nativeen verrattuna. Flutterin tapauksessa merkittävästi suurempi tilan käyttö johtuu todennäköisesti jälleen sen widget-perusteisesta lähestymistavasta.

Vastaavasti iOS:llä tilanne on hyvin samankaltainen, joskin tällä kertaa Flutter vie vielä enemmän tilaa muihin sovelluksiin verrattuna. Myös React-Native vie merkittävästi enemmän tilaa kuin natiivi Swift, joka jälleen havainnollistaa vain natiiville koodille mahdollisen vahvan optimoinnin merkityksen.

Taulukko 6: Sovellusten käynnistymiseen kuluva aika eri alustoilla.

Tekniikka	Käynnistymiseen käytetty aika Androidilla	Käynnistymiseen käytetty aika iOS:llä
Kotlin	146 ms	N/A
Swift	N/A	61 ms
Flutter	295 ms	171 ms
React-Native	168 ms	99 ms

Taulukosta 6 voidaan havaita sovellusten käynnistymiseen kuluva aika millisekunnin tarkkuudella eri alustoilla. Käynnistysajat menivät samassa järjestyksessä molemmilla kohdealustoilla, joilla natiivit sovellukset olivat nopeimpia, React-Native-sovellukset toisiksi nopeimpia ja Flutter-sovellukset hitaimpia. Flutter oli molemmilla alustoilla myös selkeästi hitaampi muihin tekniikoihin verrattuna ja sen käynnistyksen hitauden pystyi havaitsemaan jo silmämääräisestikin.

Kuten sovelluksen koossa, myös tämä viive johtuu todennäköisesti Flutterin lähestymistavasta, jossa sen tulee käynnistää oma grafiikkamoottorinsa ennen sovelluksen käynnistämistä. Natiivit sovellukset ja React-Native käyttävät grafiikan esittämiseen laitteen omaa grafiikkamoottoria, joka antaa niille selvän etulyöntiaseman. Vaikka React-Native-sovellukset olivat natiiveihin sovelluksiin verrattuna hitaampia käynnistymään, ei niiden eroa kuitenkaan voinut havaita silmämääräisesti.

6.6 Viitekehys

Tässä kappaleessa käydään läpi molempien tämän tutkielman alustariippumattomien tekniikoiden pisteet ja niiden perustelut aikaisemmassa kappaleessa 4.3.3 esitetyillä sovelluskehittäjän ja loppukäyttäjän kriteereillä. Näiden kriteerien perusteella luodaan viitekehys, joka mahdollistaa tekniikoiden vahvuuksien ja heikkouksien havainnoinnin. Lopuksi lasketaan React-Nativen ja Flutterin kokonaispisteet molemmilla näkökulmilla, jotta saadaan selville kumpi tekniikoista on keskimääräisesti parempi yleiseen sovelluskehitykseen.

6.6.1 Alustariippumattomat tekniikat sovelluskehittäjän kriteereillä

Taulukko 7: React-Nativen ja Flutterin pisteet sovelluskehittäjän kriteereillä.

Kriteeri	React-Native	Flutter	Painoarvo
Suorituskyky	3	4	3
Koodipohjan helppokäyttöisyys	4	3	5
Ylläpidettävyys	4	3	4

Kriteeri	React-Native	Flutter	Painoarvo
Jatkokehitys	4	4	2
Alustakohtaiset erot	4	3	4
Natiivit rajapinnat	4	4	3
Käyttöjärjestelmien tuki	4	4	2

Sovelluskehittäjän näkökulmasta ensimmäinen kriteeri oli tekniikan yleinen suorituskyky, jossa React-Native sai kolme pistettä ja Flutter neljä pistettä. Perustelut tälle pisteytykselle olivat, että Flutter oli yleisesti melkein jokaisessa suorituskykytestissä ainakin hieman parempi kuin React-Native, mutta ei kuitenkaan yhtä tehokas kuin natiivi sovellus. React-Native pysyi yleisesti tarpeeksi hyvin mukana ansaitakseen kolme pistettä, ja sen ainut merkittävä heikkous oli kryptografisessa testissä, joka kuvaa erittäin raskasta työkuormaa sovelluksella. Molemmat tekniikat mahdollistavat myös suorituskyvyllisesti raskaiden tehtävien erottelun eri säikeisiin, jotta UI:n suorituskyky pystyttiin säilyttämään. Tämä ei vaikuta suoritusaikaan, mutta antaa loppukäyttäjälle sulavamman käyttökokemuksen.

Seuraavana kriteerinä oli koodipohjan helppokäyttöisyys, jossa React-Native sai neljä pistettä ja Flutter kolme pistettä. React-Nativen pisteitä perustellaan varsinkin sen samankaltaisuudella web-teknologioihin, joka tekee osaamisen löytämisestä ja kehittämisestä helppoa. React-Nativelle harkittiin myös viittä pistettä, mutta se päätettiin olla antamatta, koska React-Nativella on Flutteriin verrattuna haasteita eri alustoilla testaamisessa, mikäli kehitysalusta ei ole macOS. Tämän lisäksi osa natiiveihin rajapintoihin liittyvistä toiminnoista vaati React-Nativella enemmän manuaalista työtä Flutteriin verrattuna.

React-Nativen käyttämästä Javascriptistä poiketen Flutterin Dart on varsin tuntematon ja vaikka sen oppiminen ei ole erityisen vaikeaa koodipohjan modernien ominaisuuksien vuoksi, on sille hankala löytää valmiita osaajia Flutterin uutuuden seurauksena. Tämän vuoksi Flutterille päädyttiin antamaan vain kolme pistettä.

Seuraavat kaksi kriteeriä “Ylläpidettävyys” ja “Jatkokehitys” ovat myös vahvasti liitoksissa koodipohjan helppokäyttöisyyteen. React-Nativelle annettiin ylläpidettävyydestä neljä pistettä ja Flutterille kolme pistettä. Päätös perustui jälleen React-Nativen koodipohjan tarjoa-

maan osaamiseen, mutta sen lisäksi sen vakiintuneeseen käyttäjäkuntaan, joka varmistaa, että tekniikalla on jatkuva tuki vielä useita vuosia. Flutterin tulevaisuus sen sijaan ei ole yhtä varmalla pohjalla sen uutuuden vuoksi ja vaikka sen koodista on helppo tehdä ylläpidettävää ja siihen löytyy työkaluja, on tässä vaiheessa vielä hankala luvata samankaltaista tukea, joka React-Nativella jo on.

Jatkokehityksen osalta molemmille tekniikoille annettiin neljä pistettä. Molemmat tekniikat perustuvat komponentteihin, jotka koostuvat pienemmistä komponenteista. Tämä tarkoittaa, että sovelluksista syntyy luonnollisesti modulaarisia kokonaisuuksia, joita oikeanlaisen kehityksen seurauksena pitäisi olla helppo laajentaa tai refaktoroida. Yksi rajoittava tekijä jatkokehitykselle voisi kuitenkin olla, että jompikumpi tekniikoista ei tukisi uutta tai vanhempaa käyttöjärjestelmäversiota, jonka vuoksi uusia ominaisuuksia ei voitaisi toteuttaa tai ne voitaisiin toteuttaa vain osalle käyttöjärjestelmäversioista.

Seuraava kriteeri “Alustakohtaiset erot” kuvastaa sitä, kuinka paljon alustakohtaisia koodin muutoksia alustariippumaton sovellus vaatii eri alustojen välillä sekä sovelluslogiikassa, että käyttöliittymässä. React-Nativelle annettiin tässä kategoriassa neljä pistettä ja Flutterille kolme pistettä. Molemmat tekniikat mahdollistivat sovelluslogiikan yhdistämisen alustojen välillä, mutta vain React-Native mahdollisti täysin natiivien komponenttien hyödyntämisen ilman, että komponentille tarvitsi tehdä alustakohtaisia muutoksia. Flutter sen sijaan tarjoaa kyllä komponentin, joka näyttää samalta molemmilla alustoilla, mutta mikäli halutaan tehdä molemmille alustoille natiivilta näyttävä komponentti, tuli Flutterille luoda logiikkaa, joka valitsi oikean komponentin alustan mukaan.

Natiivien rajapintojen hyödyntämisen mahdollisuuksia kuvaavalla kriteerillä molemmat tekniikat saivat neljä pistettä. Molemmille tekniikoille löytyy valmiita kirjastoja yleisten rajapintojen, kuten kameran, GPS:n, NFC:n, Bluetoothin ja salausvaraston käyttämiseen. Tämän lisäksi molemmat tekniikat mahdollistavat teoriassa minkä tahansa natiivin rajapinnan käyttämisen, mikäli sille kirjoitetaan tuki natiivilla kielellä. Tuetut natiivit kielet ovat molemmilla tekniikoilla Kotlin Androidille ja Swift iOS:lle.

Viimeinen sovelluskehittäjän näkökulman kriteeri “Käyttöjärjestelmien tuki” kuvastaa tekniikan tukea eri käyttöjärjestelmäversioille. Tämä tuki on olennainen tuettavien laitteiden

kannalta, koska kuten luvussa 2 mainittiin, varsinkin Androidilla laitekanta sisältää usein vanhempia käyttöjärjestelmäversioita iOS:ään verrattuna. Molemmat tekniikat saivat jälleen neljä pistettä, sillä sekä React-Native että Flutter tukivat täysin identtisiä käyttöjärjestelmäversioiden joukkoja.

6.6.2 Alustariippumattomat tekniikat loppukäyttäjän kriteereillä

Taulukko 8: React-Nativen ja Flutterin pisteet loppukäyttäjän kriteereillä.

Kriteeri	React-Native	Flutter	Painoarvo
UI:n yhdenmukaisuus	4	3	3
UI:n suorituskyky	3	3	4
Sovelluksen suorituskyky	3	4	4
Resurssien käyttö	3	4	3
Sovelluksen koko	3	2	1

Ensimmäinen loppukäyttäjän kriteeri “UI:n yhdenmukaisuus” kuvastaa alustariippumattomien tekniikoiden teoreettista kykyä toistaa natiivin sovelluksen ulkoasua. React-Native sai tässä kategoriassa neljä pistettä, kun taas Flutter kolme pistettä. React-Native sai korkeammat pisteet, koska sen käyttämät UI-elementit ovat identtisiä natiivien sovellusten kanssa molemmilla alustoilla. Tämän vuoksi React-Native mahdollistaa teoriassa täysin identtisen sovelluksen tekemisen natiiviin sovellukseen verrattuna. React-Native ei kuitenkaan mahdollista jokaisen natiivin komponentin käyttöä suoraan ja mikäli halutaan päästä käsiksi monimutkaisempiin ja erikoistuneempiin komponentteihin, joudutaan niiden tuki hankkimaan joko kolmannen osapuolen kautta tai kehittämällä se itse alustan natiivilla kielellä.

Flutter sai vähemmän pisteitä kuin React-Native, koska yksikään sen UI-komponenteista ei ole täysin aito natiivi komponentti, vaan tarkka kopio. Vaikka suurin osa Flutterin komponenteista on laadukkaasti kopioitu, pystyy niissä havaitsemaan pieniä eroja esimerkiksi animaatioissa. Tämän lisäksi myös Flutter kärsii samasta monimutkaisempien ja erikoistuneempien

komponenttien ongelmasta kuin React-Native, jonka ohella Flutterin uutuuden vuoksi sille ei vielä ole saatavilla yhtä suurta joukkoa muokattuja komponentteja.

Seuraava kriteeri "UI:n suorituskyky" toimii jatkona edelliselle kriteerille ja kuvastaa sovelluksen käyttöliittymän sulavuutta, kun käyttäjä esimerkiksi liikuttaa eri UI-elementtejä tai sovelluksessa on käynnissä animaatio. Molemmat React-Native ja Flutter saivat tässä kategoriassa kolme pistettä. Vaikka React-Native hävisi Flutterille ja natiiville sovellukselle animaatiotesteissä varsinkin Androidilla, suoriutui se matalan ja keskitason testeistä niin hyvin, että ongelmia tuskin havaittaisiin yleisissä sovelluskäyttötapauksissa. Flutter sen sijaan oli hyvin lähellä natiivin sovelluksen animaatiosuorituskykyä, mutta pisteitä vähennettiin, koska Flutter-sovellus käynnistyi selvästi hitainten, joka on varsinkin loppukäyttäjälle selkeä heikkous sovelluksessa. Flutteriin verrattuna React-Native ei kärsinyt hitaasta UI:n käynnistymisestä ja sitä oli hankala erottaa natiivista sovelluksesta.

Kriteeri "Sovelluksen suorituskyky" on samankaltainen sovelluskehittäjän kriteerin "Suorituskyky" kanssa. Loppukäyttäjän suorituskyvyssä keskitytään kuitenkin tarkemmin varsinkin erilaisten yleisten työkuormien suoritusaikaan, kun taas sovelluskehittäjän suorituskyvyssä arvioitiin koko tekniikan yleistä suorituskykyä UI:n ja sovelluslogiikan osalta. React-Native sai tässä kategoriassa kolme pistettä, koska se oli keskimäärin hitain suoritusajan osalta ja varsinkin yleistä työkuormaa kuvaavassa JSON-testissä se oli selkeästi hitaampi Android-alustan Flutter-sovellukseen verrattuna. Flutter sen sijaan sai neljä pistettä, koska se oli nopeampi kuin React-Native kaikessa paitsi GPS-testissä. Flutter ei kuitenkaan pystynyt vastaamaan natiivin sovelluksen suorituskykyä.

Seuraava kriteeri "Resurssien käyttö", kuvastaa sovellusten hyötysuhdetta ja tehokkuutta. Mitä tehokkaampi sovellus, sitä vähemmän aikaa ja resursseja laitteelta sen tulisi vaatia, joka parantaa merkittävästi akkukestoa varsinkin sovelluksilla, jotka ovat jatkuvasti auki. React-Native sai tässä kategoriassa kolme pistettä, sillä vaikka sen muistin käyttöaste olikin hyvin alhaista, oli sen akunkeston kannalta tärkeämpi prosessorin käyttöaste lähes jokaisessa testissä korkein. Tätä suurta prosessorin käyttöastetta pahensi se, että React-Native oli usein myös sovelluksista hitain. React-Nativeen verrattuna Flutter sai neljä pistettä, koska sen prosessorin käyttö oli joko lähellä natiivia sovellusta tai ainakin huomattavasti alempana React-Nativeen verrattuna. Flutterin heikkous oli sen suuri muistin käyttö, mutta tämä ei ole

yhtä iso ongelma akkukeston kannalta ja varsinkin uudemmat laitteet ovat harvoin muistirajoittuneita.

Viimeinen loppukäyttäjän kriteeri “Sovelluksen koko” kuvastaa sovelluksen ladattavaa kokoa ja sen laitteelta viemää tilaa. Tämä kriteeri on kaikista kriteereistä alhaisimmalla painolla, koska nykyaikaiset laitteet ovat harvoin tilarajoittuneita ja nopeat verkkoyhteydet ovat yleistyneet. React-Native sai tässä kategoriassa kolme pistettä, koska sen sovelluspaketit eivät olleet poikkeuksellisen isoja eikä se myöskään vienyt kohtuuttomasti tilaa laitteelta Androidilla tai iOS:llä. React-Native oli silti huomattavasti tehottomampi verrattuna natiiviin sovellukseen varsinkin iOS:llä, mutta tästä huolimatta tehokkaampi Flutteriin verrattuna. Flutter sen sijaan sai tässä kategoriassa vain kaksi pistettä, koska se vaati molemmilla alustoilla laitteelta eniten tilaa, jonka lisäksi sen sovelluspaketti iOS:llä oli massiivinen verrattuna natiiviin sovellukseen.

6.6.3 Lopulliset tulokset

Taulukko 9: React-Nativen ja Flutterin painotetut keskiarvot eri kriteeriryhmillä yhden desimaalin tarkkuudella.

Kriteerien tyyppi	React-Native	Flutter
Sovelluskehittäjän kriteerit	3,9	3,4
Loppukäyttäjän kriteerit	3,2	3,4

React-Nativen ja Flutterin painotetut keskiarvot voidaan nähdä taulukosta 9. React-Native voitti sovelluskehittäjän kriteereissä Flutterin 3,4 pistettä 3,9 pisteellä, kun taas Flutter voitti niukasti 3,4 pisteellä React-Nativen 3,2 pistettä loppukäyttäjän kriteereissä. Kun molemmat kriteeriryhmät lasketaan yhteen, voittaa React-Native 7,1 pisteellä Flutterin 6,8 pistettä. React-Native näyttäisi siis näiden kriteerien mukaan olevan keskimäärin parempi valinta yleiseen sovelluskehitykseen, joskin ero on varsin pieni.

React-Nativen vahvuuksia olivat varsinkin sen koodipohjan helppokäyttöisyys ja ylläpidettävyys, jotka ovat oleellisia, kun kehittäjä valitsee tekniikkaa sovellusta varten. Flutter tarjoaa

pääasiassa samat ominaisuudet, mutta ei sen uutuuden ja web-tekniikoista eroavan lähestymistavan vuoksi pysty tarjoamaan samankaltaista helppoutta ja olemassa olevaa osaamista kuin React-Native. React-Native kuitenkin kärsii varsinkin suorituskyvyssä, mikäli sille asetetaan raskaampia työkuormia ja sen tulkillinen lähestymistapa tarkoittaa, että se vaatii enemmän resursseja prosessorilta, joka taas kuluttaa enemmän virtaa. Tästä huolimatta React-Nativen muihin vahvuuksiin kuului sen natiivien komponenttien mahdollistama UI:n ulkoasu ja myös alustakohtaisten erojen vähyys verrattuna Flutteriin.

Flutter sen sijaan oli vahvempi varsinkin suorituskyvyn ja resurssien tehokkuuden kannalta sekä sovelluskehittäjän että loppukäyttäjän näkökulmasta. Flutter myös skaalautuu paremmin raskaisiin tehtäviin React-Nativeen verrattuna ja kykenee siksi mahdollistamaan joitakin sovelluksia, joihin React-Native ei pysty. Flutter ei kuitenkaan pysty ulkoasultaan täysin vastaamaan natiivia sovellusta sen widget-perusteisen lähestymistavan vuoksi, joka tarkoittaa, että loppukäyttäjä voi havaita eron sen ja natiiviin sovelluksen välillä.

Kuten tässä tutkielmassa on useaan kertaan mainittu, yksi alustariippumattoman sovelluksen tärkeimmistä piirteistä on olla käyttäjälle näkymätön muutos (Biørn-Hansen ym. 2020). React-Native pystyy tähän ulkoasun osalta, mutta kärsii suorituskyvyssä, kun taas Flutter toimii päinvastoin tarjoten hyvää suorituskykyä, mutta kärsien ulkoasun osalta. React-Native näyttäisi niukasti paremmalta valinnalta yleiseen sovelluskehitykseen sen vakiintuneen käytön vuoksi, mutta Flutter voi tulevaisuudessa olla samassa asemassa tarjoten samalla paremmin skaalautuvaa suorituskykyä.

Tämän vuoksi oikea tekniikka sovellukselle riippuu yhä kehittäjästä ja kehitettävän sovelluksen vaatimuksista. Mikäli sovellus halutaan kehittää alustariippumattomana, ovat molemmat Flutter ja React-Native yleisesti hyviä valintoja, mutta molemmat tekniikat sekä alustariippumattomuus kokonaisuutena vaativat kehityksestä huolimatta vielä kompromisseja natiiveihin sovelluksiin verrattuna.

7 Yhteenveto

Tässä tutkielmassa selvitettiin mitä eri haasteita alustariippumattomilla tekniikoilla ja lähestymistavoilla on mobiilikehityksessä sekä sovelluskehittäjän että loppukäyttäjän näkökulmista. Eri lähestymistavoista valittiin kaksi eri tekniikkaa: Flutter ja React-Native, joita käyttäen kehitettiin testisovellus, joka mahdollisti tekniikoiden vertailun toisiaan ja Android- sekä iOS-alustojen natiiveja tekniikoita vastaan. Tämän vertailun avulla muodostettiin lopulta viitekehys, joka antoi selkeän kuvan molempien tekniikoiden vahvuuksista ja heikkouksista.

Testisovelluksella teetetyssä kokeessa havaittiin, että molemmat React-Native ja Flutter kärsivät erilaisista haasteista verrattuna natiiveihin vastakappaleisiinsa. React-Nativen tapauksessa suurimmat haasteet olivat heikentynyt suorituskyky ja resurssien käyttö, kun taas Flutterin tapauksessa suurimpia haasteita olivat sen uutuudesta johtuvat kehityshaasteet sekä ulkoasun kyvyttömyys vastata täysin natiivia sovellusta. Kokeessa molemmat Flutter ja React-Native todettiin kuitenkin haasteistaan huolimatta toimiviksi tekniikoiksi yleisille sovelluksille.

React-Native oli sovelluskehittäjän näkökulmasta keskimäärin parempi vaihtoehto varsinkin sen vakiintuneen käyttäjäkunnan vuoksi. Sen sijaan loppukäyttäjän näkökulmasta Flutter oli keskimäärin parempi vaihtoehto sen paremman suorituskyvyn ja tehokkaan resurssien hallinnan vuoksi. Tästä huolimatta React-Native todettiin niukasti paremmaksi vaihtoehdoksi yleiseen sovelluskehitykseen, kun molempien näkökulmien pisteet laskettiin yhteen.

Vaikka React-Native todettiin keskimäärin paremmaksi vaihtoehdoksi, oli sen voitto pääasiassa sen turvallisen tulevaisuuden näkymän ansiota. Tämän vuoksi tulos voi muutamassa vuodessa olla toinen, kun Flutter vakiintuu, hyötyen varsinkin paremmin skaalautuvasta suorituskyvystään.

On selvää, että alustariippumattomien tekniikoiden soveliaisuus mobiilikehitykseen on yhä riippuvainen kehitettävän sovelluksen vaatimuksista, sillä ne vaativat edelleen kompromisseja natiiveihin tekniikoihin verrattuna. Kompromissit, joita Flutter ja React-Native vaativat ovat kuitenkin melko tapauskohtaisia, jonka vuoksi on odotettavissa, että yhä useampi so-

vellus kehitetään tulevaisuudessa alustariippumatomasti. Loppujen lopuksi sekä sovellusta kehittävä taho että loppukäyttäjä hyötyvät pienemmistä kehityskustannuksista ja tasapuolisemmista lopputuloksista alustojen välillä.

7.1 Rajoitteet

Tutkielman rajoitteiden kannalta merkittävimpiä muuttujia ovat tämän tutkielman testisovelluksen rakenne verrattuna muihin tutkimuksiin sekä eri tekniikoiden eri versiot. Vaikka testisovelluksen testeistä pyrittiin tekemään samankaltaisia aikaisemmassa tutkimuksessa esiintyneiden testisovellusten kanssa, eivät niiden tulokset ole täysin keskenään vertailukelpoisia sovelluskohtaisten erojen vuoksi. Varsinkin Flutterin uutuuden vuoksi oli myös hankala löytää aikaisempaa tutkimusta, jonka seurauksena osa tämän tutkielman testien tuloksista oli hankala suhteuttaa aikaisempaan tutkimukseen. Tämän lisäksi eri tekniikoiden eri versiot voivat merkittävästikin vaikuttaa testituloksiin, jonka vuoksi tuloksia ei tulisi verrata eri versioiden välillä, ellei tarkoituksena ole nimenomaan tutkia valitun tekniikan suorituskykyä eri versioiden välillä.

Testisovelluksen kehityksen osalta on myös mahdollista, että kehittäjän virheestä johtuen testitulokset olisivat huonompia tai parempia kuin niiden pitäisi olla. Tämän virheen mahdollisuutta pyrittiin välttämään käyttämällä kunkin tekniikan hyviä käytäntöjä ja pitäen testit tarpeeksi yksinkertaisina.

Kuten kappaleessa 4.3.1 mainittiin, on vertailututkimuksella useita sille uniikkeja virhemahdollisuuksia kuten käsitteelliset, mittaukselliset, instrumentaaliset ja otoskohtaiset virheet. Jokaista näistä pyrittiin lieventämään parhaaksi havaituilla tavoilla, joihin kuului mm: selkeät kokeiden kriteerit, helposti toistettavat kokeet, rauhoitusajat kokeiden välillä ja mittaustyökalujen omien ohjeiden ja rajoitteiden tarkka noudattaminen.

Testisovelluksen testien ajamiseen käytettiin kahta eri laitetta, jonka vuoksi on myös mahdollista, että valitut alustariippumattomat tekniikat skaalautuisivat erilaisilla laitteilla tämän tutkielman tuloksista poikkeavalla tavalla. Tämän tutkielman tutkimuskysymykset koskivat kuitenkin eri tekniikoiden käyttäytymistä samalla alustalla eri alustojen vertailun sijaan, jonka vuoksi eri alustojen erot eivät ole tämän tutkielman kannalta haitallisia. Testeistä voidaan

silti tehdä päätelmiä jonkin tekniikan suorituskyvyn skaalautumisesta eri alustoilla, mutta näitä päätelmiä ei voi varmentaa ilman suurempaa joukkoa laitteita.

7.2 Tulevaisuuden tutkimus

Tässä tutkielmassa rajoituttiin vain kahteen eri alustariippumattomaan tekniikkaan kahdesta eri lähestymistavasta, jonka vuoksi olisi mielekästä lisätä tulevaisuuden tutkimukseen myös muita lähestymistapoja kuten malliperusteiset- ja hybridilähestymistavat. Tämän lisäksi uudet tekniikat kuten Kotlin Multiplatform Mobile tarjoavat tuoreita vertailukohteita vakiintuneille tekniikoille.

Alustariippumattomat tekniikat kokevat usein merkittäviä suorituskykyparannuksia eri versioiden välillä, jonka vuoksi myös tämän tutkielman tulokset olisi järkevää kerätä uudelleen muutaman vuoden kuluttua. Esimerkiksi React-Nativelle on juuri lisätty tuki Hermes Javascript-moottorille (React-Native 2021a), joka voi parantaa React-Nativen tässä tutkielmassa saavuttamaa suorituskykyä. Myös Flutterista on juuri julkaistu versio 2 (Google Developers 2021), joka parantaa sen tukea erityisesti työpöytäsovelluksille. Olisi siksi mielekästä nähdä tutkimusta alustariippumattomista sovelluksista erilaisissa työpöytäympäristöissä kuten Windows ja Ubuntu.

Tässä tutkielmassa havaittiin myös eroja React-Nativen ja Flutterin suorituskyvyssä eri alustoilla, joita ei kuitenkaan voitu todeta kovin varmaksi vähäisen testilaitteiden määrän vuoksi. Olisi siis mielekästä nähdä tarkempaa tutkimusta siitä, miten eri kohdealustat vaikuttavat eri alustariippumattomien tekniikoiden suorituskykyyn.

Tutkielmassa eri tekniikoiden havaitut haasteet perustuivat pääosin suorituskykyyn, resurssien käyttöön ja kehittämisen haasteisiin, mutta olisi lisäksi mielekästä nähdä laajempaa tutkimusta siitä, pystyykö loppukäyttäjä todella havaitsemaan eron alustariippumattoman ja natiivin sovelluksen välillä. Tämänkaltainen käyttäjätestaus auttaisi myös validoimaan tämän tutkielman tekniikoiden haasteita sekä suorituskyvyssä että käyttöliittymien toiminnassa.

Lähteet

Android Developer. 2020a. “Android Layout”. Viitattu 13. tammikuuta 2021. https://developer.android.com/guide/topics/ui/declaring-layout.

———. 2020b. “Android Studio”. Viitattu 17. joulukuuta 2020. https://developer.android.com/studio/intro.

———. 2020c. “Androids Kotlin-first approach”. Viitattu 15. joulukuuta 2020. https://developer.android.com/kotlin/first.

———. 2020d. “Jetpack Compose”. Viitattu 13. tammikuuta 2021. https://developer.android.com/jetpack/compose.

———. 2021. “UI Performance”. Viitattu 21. helmikuuta 2021. https://developer.android.com/training/testing/performance.

Appfigures. 2021. “Top Ranked iOS App Store Apps”. Viitattu 4. huhtikuuta 2021. https://appfigures.com/top-apps/ios-app-store/united-states/iphone/top-overall.

Apple. 2019. “Distribution options”. Viitattu 13. huhtikuuta 2021. https://help.apple.com/xcode/mac/11.0/index.html?localePath=en.lproj#/devde46df08a.

———. 2020. “What is app thinning?” Viitattu 13. huhtikuuta 2021. https://help.apple.com/xcode/mac/current/#/devbbdc5ce4f.

Apple Developer. 2020a. “Swift”. Viitattu 7. joulukuuta 2020. https://developer.apple.com/swift/.

———. 2020b. “SwiftUI”. Viitattu 13. joulukuuta 2020. https://developer.apple.com/xcode/swiftui/.

———. 2020c. “UIKit”. Viitattu 13. joulukuuta 2020. https://developer.apple.com/documentation/uikit/about_app_development_with_uikit.

Apple Developer. 2020d. “XCTest”. Viitattu 13. joulukuuta 2020. https://developer.apple.com/documentation/xctest.

———. 2021. “SwiftUI”. Viitattu 3. huhtikuuta 2021. https://developer.apple.com/documentation/swiftui/.

Biørn-Hansen, Andreas, Tor-Morten Grønli ja Gheorghita Ghinea. 2018. “A Survey and Taxonomy of Core Concepts and Research Challenges in Cross-Platform Mobile Development”. *ACM Comput. Surv.* (New York, NY, USA) 51, numero 5 (marraskuu). ISSN: 0360-0300. doi:10.1145/3241739. https://doi.org/10.1145/3241739.

———. 2019. “Animations in cross-platform mobile applications: An evaluation of tools, metrics and performance”. *Sensors* 19 (9): 2081.

Biørn-Hansen, Andreas, Christoph Rieger, Tor-Morten Grønli, Tim A. Majchrzak ja Gheorghita Ghinea. 2020. “An empirical investigation of performance overhead in cross-platform mobile development frameworks”. *Empirical Software Engineering* (kesäkuu). doi:10.1007/s10664-020-09827-6.

Blewitt, Alex. 2016. *Swift Essentials.* Packt Publishing Ltd.

Dart. 2020. “Documentation”. Viitattu 18. tammikuuta 2021. https://dart.dev/guides.

Decan, Alexandre, Tom Mens ja Eleni Constantinou. 2018. “On the Impact of Security Vulnerabilities in the Npm Package Dependency Network”. Teoksessa *Proceedings of the 15th International Conference on Mining Software Repositories,* 181–191. MSR ’18. Gothenburg, Sweden: Association for Computing Machinery. ISBN: 9781450357166. doi:10.1145/3196398.3196401. https://doi.org/10.1145/3196398.3196401.

Eisenman, Bonnie. 2015. *Learning react native: Building native mobile apps with JavaScript.* "O’Reilly Media, Inc.".

Esser, Frank, ja Rens Vliegenthart. 2017. “Comparative Research Methods”. Teoksessa *The International Encyclopedia of Communication Research Methods,* 1–22. American Cancer Society. ISBN: 9781118901731. doi:10.1002/9781118901731.iecrm0035. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781118901731.iecrm0035. https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118901731.iecrm0035.

Fling, Brian. 2009. *Mobile Design and Development: Practical Concepts and Techniques for Creating Mobile Sites and Web Apps - Animal Guide.* 1st. O’Reilly Media, Inc. ISBN: 0596155441.

Flutter. 2020a. “Announcing Flutter 1.22”. Viitattu 24. tammikuuta 2021. https://medium.com/flutter/announcing-flutter-1-22-44f146009e5f.

———. 2020b. “Documentation”. Viitattu 18. tammikuuta 2021. https://flutter.dev/docs.

———. 2020c. “From another platform?” Viitattu 21. tammikuuta 2021. https://flutter.dev/docs/get-started/flutter-for/android-devs.

———. 2021. “FAQ”. Viitattu 24. tammikuuta 2021. https://flutter.dev/docs/resources/faq.

Freitas, Ed. 2019. *Flutter Succinctly.* Morrisville: Syncfusion Inc.

Geekbench. 2019. “CPU workloads”. Viitattu 21. helmikuuta 2021. https://www.geekbench.com/doc/geekbench5-cpu-workloads.pdf.

Google Developers. 2021. “Announcing Flutter 2”. Viitattu 29. maaliskuuta 2021. https://developers.googleblog.com/2021/03/announcing-flutter-2.html.

Google Play. 2021. “Top Apps”. Viitattu 21. helmikuuta 2021. https://play.google.com/store/apps/top?hl=fi&gl=US.

Hassani, Hossein. 2017. “Research Methods in Computer Science: The Challenges and Issues”. *CoRR* abs/1703.04080. arXiv: 1703.04080. http://arxiv.org/abs/1703.04080.

Heitkötter, Henning, Sebastian Hanschke ja Tim A. Majchrzak. 2013. "Evaluating Cross-Platform Development Approaches for Mobile Applications". Teoksessa *Web Information Systems and Technologies,* toimittanut José Cordeiro ja Karl-Heinz Krempels, 120–138. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN: 978-3-642-36608-6.

Heitkötter, Henning, Tim A. Majchrzak ja Herbert Kuchen. 2013. "Cross-Platform Model-Driven Development of Mobile Applications with Md2". Teoksessa *Proceedings of the 28th Annual ACM Symposium on Applied Computing.* New York, NY, USA: Association for Computing Machinery. ISBN: 9781450316569. doi:10.1145/2480362.2480464. https://doi-org.ezproxy.jyu.fi/10.1145/2480362.2480464.

Hidayat, T., ja B. D. Sungkowo. 2020. "Comparison of Memory Consumptive Against the Use of Various Image Formats for App Onboarding Animation Assets on Android with Lottie JSON". Teoksessa *2020 3rd International Conference on Computer and Informatics Engineering (IC2IE),* 376–381. doi:10.1109/IC2IE50715.2020.9274612.

JetBrains Blog. 2020. "Kotlin Multiplatform Mobile Goes Alpha". Viitattu 17. joulukuuta 2020. https://blog.jetbrains.com/kotlin/2020/08/kotlin-multiplatform-mobile-goes-alpha/.

Joorabchi, M. E., A. Mesbah ja P. Kruchten. 2013. "Real Challenges in Mobile App Development". Teoksessa *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement,* 15–24. doi:10.1109/ESEM.2013.9.

Kelly, Steven, ja Juha-Pekka Tolvanen. 2008. *Domain-specific modeling: enabling full code generation.* John Wiley & Sons.

"Killed by Google". 2021. Viitattu 25. helmikuuta 2021. https://killedbygoogle.com/.

Kotlin. 2020. "Documentation". Viitattu 17. joulukuuta 2020. https://kotlinlang.org/docs/.

Love, Robert. 2013. *Linux System Programming, 2nd Edition.* O'Reilly Media, Inc. ISBN: 9781449339531.

Moher, Thomas, ja G. Michael Schneider. 1982. “Methodology and experimental research in software engineering”. *International Journal of Man-Machine Studies* 16 (1): 65–87. ISSN: 0020-7373. doi:https://doi.org/10.1016/S0020-7373(82)80072-2. http://www.sciencedirect.com/science/article/pii/S0020737382800722.

Netflix Technology Blog. 2020. “Netflix Android and iOS Studio Apps now powered by Kotlin Multiplatform”. Viitattu 17. joulukuuta 2020. https://netflixtechblog.com/netflix-android-and-ios-studio-apps-kotlin-multiplatform-d6d4d8d25d23.

Payne, Rap. 2019. *Beginning App Development with Flutter: Create Cross-Platform Mobile Apps.* Apress. doi:10.1007/978-1-4842-5181-2.

Petersen, Kai, Robert Feldt, Shahid Mujtaba ja Michael Mattsson. 2008. “Systematic Mapping Studies in Software Engineering”. Teoksessa *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering,* 68–77. EASE’08. Italy: BCS Learning / Development Ltd.

Pimiä, Lauri. 2021. “Git-repository for the benchmark apps”. Viitattu 6. huhtikuuta 2021. https://github.com/naabvb/TIES502-thesis-benchmarks.

Pinto, C. M., ja C. Coutinho. 2018. “From Native to Cross-platform Hybrid Development”. Teoksessa *2018 International Conference on Intelligent Systems (IS),* 669–676.

“Pub.dev”. 2021. Viitattu 21. tammikuuta 2021. https://pub.dev/.

Rahul Raj, C. P., ja Seshu Babu Tolety. 2012. “A study on approaches to build cross-platform mobile applications and criteria to select appropriate approach”. Teoksessa *2012 Annual IEEE India Conference (INDICON),* 625–629.

React-Native. 2020a. “Button”. Viitattu 28. helmikuuta 2021. https://reactnative.dev/docs/button.

———. 2020b. “Components”. Viitattu 17. tammikuuta 2021. https://reactnative.dev/docs/intro-react-native-components.

React-Native. 2020c. "Documentation". Viitattu 13. tammikuuta 2021. https://reactnative.dev/docs/getting-started.

———. 2020d. "Out-of-Tree Platforms". Viitattu 14. tammikuuta 2021. https://reactnative.dev/docs/out-of-tree-platforms.

———. 2021a. "Using Hermes". Viitattu 28. maaliskuuta 2021. https://reactnative.dev/docs/hermes.

———. 2021b. "Using Typescript". Viitattu 3. huhtikuuta 2021. https://reactnative.dev/docs/typescript.

Rieger, Christoph. 2018. "Evaluating a Graphical Model-Driven Approach to Codeless Business App Development". Tammikuu. doi:10.24251/HICSS.2018.717.

Shah, K., H. Sinha ja P. Mishra. 2019. "Analysis of Cross-Platform Mobile App Development Tools". Teoksessa *2019 IEEE 5th International Conference for Convergence in Technology (I2CT),* 1–7.

StatCounter. 2020a. "Android Market Share Worldwide". Viitattu 30. lokakuuta 2020. https://gs.statcounter.com/os-version-market-share/android/mobile-tablet/worldwide#monthly-201910-202010-bar.

———. 2020b. "iOS Market Share Worldwide". Viitattu 30. lokakuuta 2020. https://gs.statcounter.com/os-version-market-share/ios/mobile-tablet/worldwide#monthly-201910-202010-bar.

———. 2020c. "Mobile Operating System Market Share Worldwide". Viitattu 28. lokakuuta 2020. https://gs.statcounter.com/os-market-share/mobile/worldwide#monthly-201910-202010-bar.

Swift. 2020. "Swift Documentation". Viitattu 8. joulukuuta 2020. https://swift.org/documentation/.

———. 2021. "Platform Support". Viitattu 3. huhtikuuta 2021. https://swift.org/platform-support/.

TechCrunch. 2019. “Kotlin is now Googles preferred language for Android app development”. Viitattu 15. joulukuuta 2020. https://techcrunch.com/2019/05/07/kotlin-is-now-googles-preferred-language-for-android-app-development.

Wasserman, Anthony I. 2010. “Software Engineering Issues for Mobile Application Development”. Teoksessa *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research,* 397–400. FoSER ’10. Santa Fe, New Mexico, USA: Association for Computing Machinery. ISBN: 9781450304276. doi:10.1145/1882362.1882443. https://doi-org.ezproxy.jyu.fi/10.1145/1882362.1882443.

Xanthopoulos, Spyros, ja Stelios Xinogalos. 2013. “A Comparative Analysis of Cross-Platform Development Approaches for Mobile Applications”. Teoksessa *Proceedings of the 6th Balkan Conference in Informatics,* 213–220. BCI ’13. Thessaloniki, Greece: Association for Computing Machinery. ISBN: 9781450318518. doi:10.1145/2490257.2490292. https://doi.org/10.1145/2490257.2490292.

Zuo, Chaoshun, Jianliang Wu ja Shanqing Guo. 2015. “Automatically Detecting SSL Error-Handling Vulnerabilities in Hybrid Mobile Web Apps”. Teoksessa *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security,* 591–596. ASIA CCS ’15. Singapore, Republic of Singapore: Association for Computing Machinery. ISBN: 9781450332453. doi:10.1145/2714576.2714583. https://doi-org.ezproxy.jyu.fi/10.1145/2714576.2714583.

Liitteet

A Flutter-sovelluksen alustariippumaton painikekomponentti

```
import 'dart:io';
import 'package:flutter/cupertino.dart';
import 'package:flutter/material.dart';

class OSSpecificButton extends StatelessWidget {
  OSSpecificButton(this.callback, this.text);

  final String text;
  final VoidCallback callback;
  final bool isIOS = Platform.isIOS;

  @override
  Widget build(BuildContext context) {
    return (isIOS
        ? Container(
            margin: EdgeInsets.only(top: 10.0, bottom: 10.0),
            child: CupertinoButton(
              child: Text(text),
              onPressed: callback,
            ),
          )
        : ElevatedButton(
            child: Text(text),
            onPressed: callback,
          ));
  }
}
```

B Flutter-sovelluksen komponentti yhden kuvan animoimiseksi

```
import 'package:flutter/material.dart';
import 'package:flutter/widgets.dart';

class SpinnerWidget extends StatelessWidget {
  SpinnerWidget(this.controller, this.columns);

  final AnimationController controller;
  final int columns;
  final Image kuva =
      Image(width: 20, height: 10, image: AssetImage('assets/testikuva.jpg'));

  @override
  Widget build(BuildContext context) {
    return (Row(
      mainAxisSize: MainAxisSize.max,
      mainAxisAlignment: MainAxisAlignment.spaceEvenly,
      children: [
        for (int i = 0; i < this.columns; i++)
          AnimatedBuilder(
              builder: (BuildContext context, Widget _widget) {
                return new Transform.rotate(
                    angle:
                        i % 2 == 0 ? controller.value : controller.value * -1,
                    child: _widget);
              },
              animation: controller,
              child: kuva),
      ],
    ));
  }
}
```

C React-Native-sovelluksen kotinäkymä

```
function HomeScreen({navigation}) {
  return (
    <View style={{flex: 1, alignItems: 'center', justifyContent: 'center'}}>
      <View style={{marginBottom: 10}}>
        <Button
          title="Animations benchmarks"
          onPress={() => navigation.navigate('animationsselection')}
          style={{padding: 30}}
        />
      </View>
      <View>
        <Button
          title="Utilities benchmarks"
          onPress={() => navigation.navigate('utilities')}
        />
      </View>
    </View>
  );
}
```

D React-Native-sovelluksen komponentti yhden kuvan animoimiseksi

```
import React, {Component} from 'react';
import {Animated, Easing, View} from 'react-native';
import testikuva from '../assets/testikuva.jpg';

export default class SpinnerImage extends Component {
  spinValue = new Animated.Value(0);

  anim = Animated.loop(
    Animated.timing(this.spinValue, {
      toValue: 1,
      duration: 500,
      easing: Easing.linear,
      useNativeDriver: true,
    }),
  );

  componentDidUpdate() {
    if (this.props.spin === true) this.anim.start();
    if (this.props.spin === false) this.spinValue.setValue(0);
  }

  render() {
    const spin = this.spinValue.interpolate({
      inputRange: [0, 1],
      outputRange: ['0deg', '360deg'],
    });
    const spin2 = this.spinValue.interpolate({
      inputRange: [0, 1],
      outputRange: ['360deg', '0deg'],
    });
    var images = [];
    for (let i = 0; i < this.props.columns; i++) {
      images.push(
        <Animated.Image
          key={i}
          style={{
```

```
            transform: [{rotate: i % 2 == 0 ? spin : spin2}],
            width: 20,
            height: 10,
          }}
          source={testikuva}
        />,
      );
    }

    return (
      <View
        style={{
          alignItems: ’center’,
          flexDirection: ’row’,
          justifyContent: ’space-evenly’,
          marginVertical: 9,
        }}>
        {images}
      </View>
    );
  }
}
```

E Swift-sovelluksen animaatiotestin toteuttava komponentti

```
import SwiftUI

struct AnimationsView: View {
    var columns: Int
    var amount: Int
    @State private var isActive = false
    var animation: Animation {
        Animation.linear(duration: 0.5).repeatForever(autoreverses: false)
    }

    func getAngle(index: Int) -> Double {
        if (index % 2 == 0) {
            return isActive ? 0 : 360
        }
        else {
            return isActive ? 360 : 0
        }
    }

    var body: some View {
        let flexibleColumn = Array(repeating: GridItem(.flexible(), spacing: 2),
                                   count: columns)
        VStack{
            LazyVGrid(columns: flexibleColumn, content: {
                ForEach((1..<amount)) { i in
                    Image("testikuva")
                        .resizable()
                        .frame(width: 20, height: 10)
                        .rotationEffect(Angle.degrees(getAngle(index: i)))
                        .animation(isActive ? animation : .default)


                }

            })
            Button(isActive ? "Stop" : "Start") {
```

```
                self.isActive = !isActive
            }
        }
    }
}
```

F Kotlin-sovelluksen yhden kuvan animaation asetukset

```
private fun createAnim(index: Int, view: View) {
        view.setLayerType(View.LAYER_TYPE_HARDWARE, null)
        var imageViewObjectAnimator = ObjectAnimator.ofFloat(
            view,
            "rotation", 360f, 0f
        )
        if (index % 2 == 0) {
            imageViewObjectAnimator = ObjectAnimator.ofFloat(
                view,
                "rotation", 0f, 360f
            )
        }
        imageViewObjectAnimator.duration = 500
        imageViewObjectAnimator.repeatCount = ObjectAnimator.INFINITE
        imageViewObjectAnimator.repeatMode = ObjectAnimator.RESTART
        imageViewObjectAnimator.interpolator = LinearInterpolator()
        animations.add(imageViewObjectAnimator)
}
```