Tampereen yliopisto

Konsta Jurvanen

# Web-teknologian vaiheittainen uudistus Sosiaali- ja terveysdatan integraatiojärjestelmälle

## Uusien ohjelmistokehysvaihtoehtojen arviointi ja vaiheittaisen käyttöönoton mahdollistava arkkitehtuurisuunnitelma

Diplomityö
Informaatioteknologian ja viestinnän tiedekunta (ITC)
Joulukuu 2025

# Tiivistelmä

Konsta Jurvanen

Web-teknologian vaiheittainen uudistus Sosiaali- ja terveysdatan integraatiojärjestelmälle: Uusien ohjelmistokehysvaihtoehtojen arviointi ja vaiheittaisen käyttöönoton mahdollistava arkkitehtuurisuunnitelma

Diplomityö
Tampereen yliopisto
Tietotekniikan DI-ohjelma
Joulukuu 2025

---

Atostek Oy on kehittänyt Kelan Kanta-palveluihin integroitumista helpottavan ERA-järjestelmän, jonka rinnalle on rakentunut monipuolinen web-käyttöliittymä sosiaali- ja terveydenhuollon ammattilaisille. Vuosia jatkuneen kehityksen aikana edullisinta on toistaiseksi ollut pitäytyä vanhassa ASP.NET MVC:tä hyödyntävässä käyttöliittymäteknologiassa, sillä välitön siirtymä vaatisi paljon työtä ja vanha käyttöliittymä on toiminit hyvin. Tämä on johtanut teknisen velan kertymiseen, mikä hidastaa kasvavissa määrin markkinoihin mukautumista ja rajoittaa vaativien käyttöliittymätoiminnallisuuksien kehittämistä. Tutkimuksen tavoitteena on luoda tieteellinen perusta suunnitelmille, jotka mahdollistavat teknisen velan takaisinmaksun asteittain uutta käyttöliittymäteknologiaa hyödyntäen.

Modernien käyttöliittymäteknologioiden vertailulla pyritään perustelemaan uuden ohjelmistokehyksen valinta, ja arkkitehtuurisuunnittelun tavoitteena on luoda kestävä pohja käyttöliittymän kehittämiselle ja vaiheittaiselle siirtymälle. Teknologioita arvioidaan ERAn tarpeiden pohjalta johdettujen kriteerien perusteella, joihin kuuluvat suorituskyky, ekosysteemi, luotettavuus, kehityskokemus, rakenteellisuus ja erityiset ominaisuudet. Kriteereittäin jaettu kirjallinen arviointi toteutetaan Angular-, React-, Vue-, Svelte- ja Blazor-teknologioille. Ominaisuuksissa huomioidaan myös teknologiakohtaiset korkean tason meta-tason ohjelmistokehykset Next, Nuxt ja SvelteKit, jotta toiminnallisuuksien taso vastaa kokonaisvaltaista Angular-kehystä.

Arkkitehtuurin osalta käyttöliittymälle havaittiin kolme eri tason integraatiotyyliä. Yksinkertaisena ratkaisuna malli A liittää ERAan itsenäisen yhden sivun sovelluksen, joka irtaantuu vanhasta järjestelmästä käyttöliittymän tasolla hyödyntäen navigointiin uudelleenohjauksia. Malli B erottaa myös palvelinpuolelle erillisen rajapintasovelluksen, mikä yksinkertaistaa useaan integraatiopalveluun yhdistymistä ja selkeyttää vastuunjakoa. Malli C rakentaa uudet komponentit nykyisten sivujen sisälle, jolloin uusi teknologia integroituu vanhoihin näkymiin selaimen JavaScript-sovelluksessa.

Tutkimuksen pohjalta tullaan kokeilemaan kehittäjäystävällistä Svelte-teknologiaa erillisen palvelun päälle rakentuvana yhden sivun sovelluksena. Vallittu arkkitehtuuri perustuu siis B-malliin, joka tukee modulaarisuutta ja vastuunjakoa. Lisäksi kiinnostavana liiketoimintamahdollisuutena tutkimuksessa havaittiin mahdollisuus micro-frontend-tekniikoiden hyödyntämiseen ERAn ulkopuolella, mutta myös nykyisen käyttöliittymän väliaikaiseen ehostukseen. Svelten valintaa perusteli sopeutuvuus tutkimuksen tarpeisiin ja arkkitehtuurisuunnitelmaan. Täten arkkitehtuurivaihtoehtoja yhdistelemällä löytyi ratkaisu, joka aloittaa teknisen velan takaisinmaksun puhtaalta pohjalta, mutta mahdollistaa osittaisen näkymien korvaamisen pienellä kynnyksellä. Turvallisena varavaihtoehtona kokeilun jälkeen toimii pelkkä B-mallin arkkitehtuuri ja Angular.

**Avainsanat:** Teknologiasiirtymä, web-kehitys, ohjelmistokehykset, vaiheittainen uudistus

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin Originality -ohjelmalla.

# Abstract

Konsta Jurvanen

Incremental web framework migration for healthcare data integration platform: Review of web frameworks and architecture plans for incremental technology migration

Master's thesis
Tampere University
Master of Science, Information Technology
December 2025

The goal of the thesis is to explore new web user interface (UI) technologies and architectural migration strategies for Atostek's data integration system, ERA, used by social and healthcare professionals. A key constraint is that the migration must proceed in small steps, keeping the old UI running alongside the modernised parts. The main technical challenge is to justify the choice of a new web UI framework and integrate it with the existing multi-page ASP.NET MVC application during the incremental transition. Atostek's main objectives are to increase UI development productivity and improve code maintainability, enabling faster responses to business requirements. In other words, the goal is to create plans for reducing the technical debt that has accumulated from decisions to continue development with old, insufficient technology.

The research consists of two execution phases and an analytical phase that combines their results. The first phase evaluates modern web frameworks by performance, ecosystem, stability, developer experience, structures and functionality, and unique features. The criteria are applied to Angular, React, Vue, Svelte, and Blazor. Meta-frameworks for React, Vue, and Svelte are included to align feature coverage with Angular. A limited review of less established technologies like HTMX and Modern.js ensures that other viable options are not overlooked. The second phase includes a comparison of plausible system architectures that allow for migrating the front-end functionality in granular steps while developing new features on top of the new technology. Three architectural approaches were identified, where model A introduces a standalone single-page application (SPA) integrated with redirections to the existing UI. Model B adds an extra application programming interface (API) layer for serving the business data of the UI. Model C embeds the new technology directly into the existing multi-page application (MPA).

The architectural plan for a future Proof-of-Concept phase is a compromise between ideas from all the analysed architectural models. Model A, with a new SPA, gives a fresh start for structured development. The additional API layer from model B provides important modularity and separation of concerns in an environment with two distinct backend services. Model C provides inspiration for combining micro-frontend technologies with the new SPA, enabling component-level migration.

The technology analysis identified many solid framework options without a clear winner over others. The most promising technologies were Angular and Vue, closely followed by Svelte and React. However, the assessment criteria were used to create technology profiles that highlight the strengths and weaknesses of the technologies, enabling a reasoned assessment based on ERA's needs. Svelte was ultimately chosen for its intuitive developer experience, good performance, and suitability for the architectural migration plans that effectively achieve the research goals. Therefore, Svelte and modern micro-frontend strategies serve as an ambitious primary approach, while Angular and model B with an API layer provide a safe backup plan to stabilise ERA's UI development.

**Keywords:** Technology migration, Web frameworks, micro-frontends, Strangler Pattern

The originality of this thesis has been checked using the Turnitin Originality service.

# Alkusanat

*Kiitos ohjaajilleni, professori Kari Syställe yliopistolta, ja Atostekin ERA-järjestelmän tekniselle johtajalle Miika Järviselle Atostekin puolelta. Karia kiitän terävistä huomioista ja tutkimuksen kehitysehdotuksista sekä sujuvasta kommunikaatiosta. Kiitos Miikalle yleisistä vinkeistä diplomityön kirjoittamiseen liittyen ja arvokkaista näkemyksistä ERAa pitkään kehittäneenä asiantuntijana. Haluan kiittää myös Atostek Oy:tä työn sponsoroinnista sekä tuesta ja kannustuksesta diplomi-insinööriksi valmistumiseen. Aiheen muodostamiseen osallistuneet Devinder Singh, Mika Torhola, Miika Parvio ja ohjaaja Miika ovat myös ansainneet kiitoksensa vaivannäöstään työn alkuun saamisen vuoksi. Erityisesti Devinder on ollut ERAn osalta mukana tutkimusprojektin toteutusta seuraamassa ja tukemassa. Suuret kiitokset Tampereen yliopistolle ja erityisesti professori Kari Syställe työn tarkastuksista sekä ystävällisestä avustuksesta ja neuvonannosta diplomityön kirjoitukseen.*

Tampereella, 16. joulukuuta 2025

Konsta Jurvanen

# Sisällys

1 Johdanto . . . . . 1
1.1 Tausta . . . . . 1
1.2 Tavoitteet . . . . . 2
1.3 Keinot . . . . . 3
2 Tarpeet tutkimukselle . . . . . 5
2.1 Liiketoimintamotiivit . . . . . 5
2.2 Hallinnolliset tavoitteet . . . . . 7
2.3 Tavoitteet kehittäjän näkökulmasta . . . . . 8
2.4 Tekniset vaatimukset tavoitteiden pohjalta . . . . . 9
2.5 Koonti tarpeista . . . . . 9
3 Teoriaa web-teknologioiden kehittymisestä ja ohjelmistojen uudistamisesta . . . . . 11
3.1 Web-teknologioiden kehittyminen . . . . . 11
3.1.1 Verkkosivujen sisältö ja yhdistäminen . . . . . 11
3.1.2 Interaktiivisuus ja JavaScript . . . . . 12
3.1.3 Ohjelmistokehykset ja työkalut . . . . . 13
3.1.4 Meta-ohjelmistokehykset . . . . . 14
3.2 Ohjelmistot ja niiden uusiminen . . . . . 14
3.2.1 Huollettavuus, tekninen velka ja teknologia-siirtymä . 15
3.2.2 Arkkitehtuuri- ja suunnittelumallit . . . . . 16
4 Järjestelmän taustatiedot . . . . . 17
4.1 Palvelinohjelmat ja rajapinnat . . . . . 17
4.2 Käyttöliittymäteknologia . . . . . 18
4.3 Resurssit ja tietokanta . . . . . 19
4.4 Lähtökohdat teknologian uudistamiseen . . . . . 19
5 Tutkimusmenetelmät . . . . . 22

5.1 Kirjallisuuskatsaus teknologioiden soveltuvuudesta tarpeisiin . . . 22
5.1.1 Suorityskyky . . . 23
5.1.2 Ekosysteemi ja työkalut . . . 23
5.1.3 Stabiilisuus ja suosio . . . 23
5.1.4 Kehityksen mukavuus vs. oppimisen vaikeus . . . 24
5.1.5 Hyvien koodauskäytäntöjen tuki ja rakenteet . . . 24
5.1.6 Muuta huomioitavaa . . . 25
5.2 Arkkitehtuurin suunnittelu ja analysointi . . . 25
5.3 Muutoksen kannattavuuden arviointi . . . 27
6 Käyttöliittymäteknologioiden arviointi . . . 29
6.1 Angular . . . 29
6.1.1 Suorityskyky . . . 29
6.1.2 Stabiilisuus ja suosio . . . 30
6.1.3 Ekosysteemi ja työkalut . . . 30
6.1.4 Kehityksen mukavuus vs. oppimisen vaikeus . . . 31
6.1.5 Hyvien koodauskäytäntöjen tuki ja rakenteet . . . 31
6.1.6 Muuta huomioitavaa . . . 32
6.2 React . . . 32
6.2.1 Suorityskyky . . . 33
6.2.2 Stabiilisuus ja suosio . . . 34
6.2.3 Ekosysteemi ja työkalut . . . 35
6.2.4 Kehityksen mukavuus vs. oppimisen vaikeus . . . 35
6.2.5 Hyvien koodauskäytäntöjen tuki ja rakenteet . . . 36
6.2.6 Muuta huomioitavaa . . . 37
6.3 Vue.js . . . 38
6.3.1 Suorityskyky . . . 38
6.3.2 Stabiilisuus ja suosio . . . 38
6.3.3 Ekosysteemi ja työkalut . . . 39

6.3.4 Kehityksen mukavuus vs. oppimisen vaikeus . . . . . 39
6.3.5 Hyvien koodauskäytäntöjen tuki ja rakenteet . . . . . 40
6.3.6 Muuta huomioitavaa . . . . . 40
6.4 Svelte . . . . . 41
6.4.1 Suorityskyky . . . . . 41
6.4.2 Stabiilisuus ja suosio . . . . . 41
6.4.3 Ekosysteemi ja työkalut . . . . . 43
6.4.4 Kehityksen mukavuus vs. oppimisen vaikeus . . . . . 43
6.4.5 Hyvien koodauskäytäntöjen tuki ja rakenteet . . . . . 44
6.5 Blazor . . . . . 44
6.5.1 Suorityskyky . . . . . 45
6.5.2 Stabiilisuus ja suosio . . . . . 45
6.5.3 Ekosysteemi ja työkalut . . . . . 46
6.5.4 Kehityksen mukavuus vs. oppimisen vaikeus . . . . . 47
6.5.5 Hyvien koodauskäytäntöjen tuki ja rakenteet . . . . . 48
6.5.6 Muuta huomioitavaa . . . . . 48
6.6 Muut teknologiavaihtoehdot . . . . . 48
6.6.1 HTMX . . . . . 48
6.6.2 ReactJS.NET . . . . . 49
6.6.3 Modern.js . . . . . 49
6.6.4 Elm . . . . . 50
7 Arkkitehtuurianalyysi . . . . . 51
7.1 Malli A: Uusi käyttöliittymä yhden sivun sovelluksena . . . 52
7.1.1 Hyödyt . . . . . 52
7.1.2 Haitat . . . . . 53
7.1.3 Mahdollisuudet . . . . . 53
7.2 Malli B: Uusi palvelinrajapinta ja uusi yhden sivun sovellus 55
7.2.1 Hyödyt . . . . . 55
7.2.2 Haitat . . . . . 56

7.2.3 Mahdollisuudet . . . . . 57
7.3 Malli C: Käyttöliittymän rakentaminen nykyisen sisälle . . 57
7.3.1 Hyödyt . . . . . 58
7.3.2 Haitat . . . . . 58
7.3.3 Mahdollisuudet . . . . . 59
7.4 Sovellusten sisäiset arkkitehtuuriratkaisut . . . . . 59
7.4.1 Tietojen synkronointi ja tilan hallinta . . . . . 59
7.4.2 Reititys ja navigointi . . . . . 60
7.5 Arkkitehtuurin vaikutukset teknologiavalintaan . . . . . 60
7.6 Koonti . . . . . 61
8 Analyysi . . . . . 62
8.1 Teknologiavertailun koonti . . . . . 62
8.1.1 Suorituskyky . . . . . 62
8.1.2 Stabiilisuus ja suosio . . . . . 63
8.1.3 Ekosysteemi . . . . . 63
8.1.4 Rakenteellisuus ja skaalautuvuus . . . . . 64
8.1.5 Kehittämiskokemus ja tuottavuus . . . . . 64
8.1.6 Erityiset hyödyt tai haitat . . . . . 65
8.2 Teknologiavertailun analyysi . . . . . 65
8.3 Teknologioiden karsinta analyysin perusteella . . . . . 67
8.4 Arkkitehtuurimallien vertailu . . . . . 67
8.5 Päätös teknologiavertailun ja arkkitehtuurianalyysin pohjalta . . . . . 68
9 Jatkokehitys ja pohdinta . . . . . 70
9.1 Teknologioiden lisätutkimus . . . . . 70
9.2 Arvioinnin syventäminen ja jatkotutkimukset . . . . . 70
9.3 Proof-of-Concept, teknologian kokeilu . . . . . 71
9.4 Uuden teknologian käyttöönotto ja tyyliohjeet . . . . . 71
9.4.1 Meta-kehyksen käyttöönotto . . . . . 72

9.5 Käyttäjäkokemussuunnittelu . . . . . 72
9.6 Reflektio . . . . . 72
10 Yhteenveto . . . . . 73
Lähteet . . . . . 75
Liite A: Kaavioiden generointiin käytetty ohjelma . . . . . 79

# KUVALUETTELO

1.1 Kuvakaappaus ERA:n kirjautumisnäkymästä. . . . . 2

2.1 Hahmotelma projektin tavoitteista ja hyötyjen jaottelu eri osapuolten intressien kannalta . . . . . 6

2.2 Tutkimuksen motiivit ja tavoitteet tiivistettynä . . . . . 10

3.1 Visualisoitu esimerkki komponenttipuusta [59]. . . . . 13

4.1 Karkea hahmotelma ERA-järjestelmän nykyisestä arkkitehtuurista. . . . . 18

4.2 ERA-järjestelmän käyttöliitymän kannalta merkittävät resurssit havainnollistettuna . . . . . 20

4.3 ERAn multi-tenant luonteinen käyttäjäkohtainen tilanhallinta havainnollistettuna infrastruktuurin kannalta. . . . . 21

5.1 **Vaihtoehto A:** Uusi yhden sivun sovellus nykyisillä integraatiorajapinnalla . . . . . 26

5.2 **Vaihtoehto B:** Yhden sivun sovellus uudella palvelinpuolen rajapintakerroksella . . . . . 27

5.3 **Vaihtoehto C:** Vanhan käyttöliittymän sisään rakentuva arkkitehtuurimalli. . . . . 28

6.1 Kuvakaappaus Angularin omasta tutoriaaliympäristöstä, jossa loin AI:n avustuksella Angularin väreihin tyylitellyn komponentin. . . . . 32

6.2 Multi-Zones -tekniikan navigointia ja reititystä havainnollistava kaavio [51] . . . . . 38

6.3 Reactin ja Svelten konseptuaalisia eroja havainnollistava kuva hahmoteltuna teknologioiden dokumentaation pohjalta [31], [58] . . . . . 42

6.4 Esimerkkejä päivämäärävalitsimista Fluent UI Blazor -kirjastoa hyödyntäen [22] . . . . . 47

6.5 Htmx:n periaatteita havainnollistava meemi [33]. . . . . 49

7.1 **Vaihtoehto A:** Uusi yhden sivun sovellus nykyisillä integraatio-rajapinnalla. . . . . 52

7.2 **Vaihtoehto B:** Yhden sivun sovellus uudella palvelinpuolen rajapintakerroksella. . . . . 56

7.3 **Vaihtoehto C:** Vanhan käyttöliittymän sisään rakentuva arkkitehtuurimalli. . . . . 58

8.1 Teknologia-arvioinnin tulokset ominaisuusprofiileina . . . . . 66

# TAULUKKOLUETTELO

8.1 Teknologiavertailun koontitaulukko . . . . . 62
8.2 Teknologioiden stabiilisuuden jäsentelevä koontitaulukko . . . 64

# Ohjelma- ja algoritmiluettelo

6.1 Esimerkki HTMX:n tarjoamasta mahdollisuudesta sisällyttää JavaScript-koodia. [29] . . . . . 49
A.1 Osa 1: Teknologiayhteenvedon kuvan 8.1 tuottanut Python-ohjelma . . . . . 80
A.2 Osa 2: Teknologiayhteenvedon kuvan 8.1 tuottanut Python-ohjelma . . . . . 81
A.3 Osa 3: Teknologiayhteenvedon kuvan 8.1 tuottanut Python-ohjelma . . . . . 82

# Lyhenteet ja merkinnät

| | |
|---|---|
| AJAX | Asynchronous JavaScript And XML, tapa kommunikoida web-käyttöliittymän ja palvelimen välillä HTTP:n välityksellä. |
| API | Ohjelmistorajapinta, (engl. Application Programming Interface) |
| B2B | Yritysten välinen liiketoiminta, (engl. Business-to-Business) |
| C# | Microsoftin kehittämä olio-orientoitunut ohjelmointikieli. Lausutaan ”*c-sharp*.” |
| CDA R2 | Hl7 Clinical Document Architecture, Release 2 |
| CSR | Selainpuolen kuvantaminen, (engl. Client Side Rendering) |
| CSS | Cascading Style Sheets first |
| DOM | Document Object Model, verkkosivuilla näytettävän sisällön mallintava tietorakenne, joka koostuu HTML-elementeistä |
| ERA | Atostekin kaupallinen tuote, terveys- ja sosiaalihuollon datan arkistointijärjestelmä, joka on yhteydessä Kelan Kanta-palveluihin |
| FHIR R4 | Hl7 Fast Health Interoperability Resources, Release 4 |
| HMR | Hot Module Replacement, tekniikka web-sovelluksen kehityksenaikaiseen koodimuutosten automatisoituun päivittämiseen ilman koko sovelluksen uudelleen-lataamista ja renderöintiä |
| HTML | HyperText Markup Language, avoimesti standardoitu merkintäkieli, jolla määritellään verkkosivulla näkyvät elementit ja niiden metatiedot. |
| HTTP | HyperText Transfer Protocol, avoimesti standardoitu merkintäkieli, jolla määritellään verkkosivulla näkyvät elementit ja niiden metatiedot. |
| IoC | Inversion of Control, käänteinen ojelmistofilosofia verrattuna imperatiiviseen koodiin, jolla on selkeä entry point (yleensä main-funktio) |
| ISO | Kansainvälinen standardointiorganisaatio |
| JSON | JavaScript olioiden rakenteen mukainen merkintäkieli tiedon tallentamiseen ja välittämiseen. (Engl. JavaScript Object Notation) |
| JSX | React-teknologiassa käytetty HTML-kieltä mukaileva JavaScript syntaksin laajennos (engl. JavaScript XML) |
| LTS | Long-Term Support, pitkän aikavälin tuki, yleisesti käytetty termi ohjelmistoversioiden ylläpitolupauksissa. |

| | |
|---|---|
| ORM | Objekti-relaatio kartoitus, (engl. Object Relational Mapping, ORM) |
| RFC | Teknologioiden käytäntö pyytää yhteisöltä palautetta ja ehdotuksia tulevien ominaisuuksien kehittämiseen (engl. Request For Comments) |
| SaaS | Ohjelmiston jakelumalli palveluna, (engl. Software as a Service |
| SemVer | Semanttinen ohjelmistojen versionumeroiden järjestelmä (engl. Semantic Versioning) |
| SEO | Hakukoneoptimointi, (engl. Search Engine Optimization) |
| SPA | Yhden sivun sovellus, (engl. Single Page Application) |
| SSR | Palvelinpuolen kuvantaminen, (engl. Server Side Rendering) |
| XML | Merkintäkieli tiedon rakenteiseen tallentamiseen ja välittämiseen. (Engl, Extensible Markup Language) |

# Tekoälyn käyttö tässä työssä

Opinnäytteessäni on käytetty tekoälysovelluksia:

☒ Kyllä
☐ Ei

Ilmoitukseni mukaan olen käyttänyt opinnäytteessäni tutkielmaprosessin aikana seuraavia tekoälysovelluksia:

| Sovellus | Versio |
|---|---|
| Azure AI (GPT) | 3.5, 4o |
| ChatGPT | 3.5 |
| Writefull (GPT) | - |
| LibreChat (Claude Opus) | 4 |
| LibreChat (GPT) | 5, 5.1 |

## Tekoälyn käyttötarkoitus

Tekoälyä on käytetty inspiraation lähteenä teknologioiden tutkinnassa ja arkkitehtuurin suunnittelussa. Tekoälyn tuottaman sisällön avulla on etsitty virallisia lähteitä, joista kaikki faktat on tarkistettu ennen niiden kirjaamista opinnäytteeseen. Tekoälyn tuottamaa tekstiä ei ole suoraan kopioitu tähän dokumenttiin. Käytin tekoälyä tehostaakseni lähteenhakuprosessia, koska pystyin kuvailemaan tekoälylle tarkasti väitteen, jolle halusin löytää tieteellisen lähteen. Erityisesti Module Federation -tekniikkaan liittyen kyselin tekoälyltä myös ymmärrystäni kehittäviä kysymyksiä.

Overleaf-työkalun sisältämää Writefull-kielentarkastusominaisuutta on hyödynnetty kirjoitusvirheiden etsimiseen ja korjaamiseen kirjoittamisen lomassa. Writefull-toiminnosta on käytetty vain ilmaisversiota, joka huomauttaa yleisistä kielioppivirheistä ja ehdottaa korjauksia tekoälyn avulla.

Tekoälyä käytettiin apuna datan visualisoinnissa pyytäen tekoälyltä python-ohjelmaa kaavioiden generointiin tekoälylle annetun datan pohjalta. Tekoälyn pohjaa korjailtiin itsenäisesti ja tekoälylle korjauspyyntöjä antaen, jotta ohjelma pystyi tuottamaan halutunlaiset kaaviot.

Työn loppuvaiheessa tekoälyä on käytetty apuna työn tehokkaaseen viimeistelyyn. Tekoälyä pyydettiin ensin generoimaan optimaalinen kehotepohja työn valmiusastetta ja kriittisimpiä puutteita analysoimaan. Sen jälkeen tekoälylle liitettiin työ pdf-muodossa analysoitavaksi ja tekoälyn luomalla kehotteella saatiin arvio luku-kohtaisesta valmiusasteesta ja tehokkaimmista tavoista paikata aukkoja tekstissä viimeistelyä varten.

## Osiot, joissa tekoälyä on käytetty

Tekoälyä on käytetty inspiraation lähteenä ja tiedonhakuapuna teknologia-arvioinnissa luvussa 6. Tekoälyä käytettiin teknologioiden yhteenvedossa alaluvun 8.2 kuvan 8.1 generointiin käytetyn liitteessä A esitetyn Python-ohjelman kehittämisessä. Tekoälyä on käytetty virtuaalisena ohjaajana lukua 7 kirjottaessa. Tekoälyltä pyydettiin ideoita kappaleen uudelleenjäsentelyyn, kun keskittyminen herpaantui yksityiskohtiin. Lopulta jäsentely tehtiin oman idean pohjalta, mutta tekoäly auttoi prosessissa. Englanninkielisessä tiivistelmässä tekoälyltä pyydettiin ehdotuksia alkuperäisen itsekirjoitetun tiivistelmän lauserakenteiden tiivistämiseen, jotta laaja tutkimus saatiin kuvailtua yhdellä tekstisivulla.

## Riskien tiedostaminen

Olen tietoinen siitä, että olen täysin vastuussa koko opinnäytteeni sisällöstä, mukaan lukien osat, joiden tuottamisessa on hyödynnetty tekoälyä, ja hyväksyn vastuun mahdollisista tästä seuranneista eettisten ohjeiden rikkomuksista.

# 1 Johdanto

Tämä tutkimus käsittelee web-käyttöliittymän uudistamista Business-to-Business (B2B) -tyyliseen dataintegraatiojärjestelmään. Uudistuksella ei tässä tapauksessa tarkoiteta järjestelmän refaktorointia tai koko käyttöliittymän siirtämistä uuteen teknologiaan. Sen sijaan tavoitteena on uuden teknologian asteittainen käyttöönotto vanhan käyttöliittymän laajennoksena. Vanhassa käyttöliittymässä ei ole akuutteja ongelmia, mutta tulevaisuuden kehitystä halutaan tehostaa ja kehitysmahdollisuuksia parantaa. Tutkimus toimii vanhassa käyttöliittymäteknologiassa pysymisen aiheuttaman teknisen velan takaisinmaksun suunnitteluvaiheena.

Tutkimus kohdistuu Atostekin ERA-nimiseen ohjelmistoon, jota kutsutaan jatkossa ERA-järjestelmäksi tai lyhyesti ERAksi. Käsitellyt konseptit pätevät myös yleisesti erityisesti ASP.NET MVC -teknologialla kehitettyjen suurikokoisten web-käyttöliittymien uusimiseen. Erityisesti teknologioiden arviointi koskee yleisesti modernien web-kehitystyökalujen ominaisuuksia, vaikka arvioinnissa keskitytään tämän tutkimuksen kannalta keskeisiin kriteereihin. Yrityskäyttöön kriittistä teknologia-arvioinnissa on erityisesti käyttöönoton turvallisuus teknologian jatkuvuutta ja tulevaisuutta ajatellen. Arkkitehtuuriratkaisuja käsitellessä päästään myös syventymään edistyneisiin käyttöliittymien integrointitekniikoihin, jotka liittyvät erityisesti niin kutsuttuihin Micro-Frontends -arkkitehtuureihin.

## 1.1 Tausta

Atostek Oy on pääosin ohjelmistokonsultaatioon keskittyvä yritys. Atostekin liiketoiminnassa merkittävässä asemassa on konsultaation ohella myös yrityksen oma tuote ERA ja sen mahdollistama konsultaatiotyö. ERA on terveys- ja sosiaalihuollon datan arkistointijärjestelmä, joka säilöö potilastietoja CDA R2 -muodossa Kelan Kanta-palveluihin. ERA:a käyttää lukuisat suomalaiset terveydenhuollon yksityiset ja julkiset toimijat päivittäin. Suurin osa asiakkaista hyödyntää ERA:a tietojen arkistointiin järjestelmäintegraation kautta, jolloin käyttöliittymänä toimii asiakkaan oma potilastietojärjestelmä. Kuukausittaisia istuntoja ERA:n omasta web-käyttöliittymästä luodaan silti useita tuhansia. Osa asiakkaista hyödyntää ERA:n käyttöliittymästä vain tiettyä osuutta. Esimerkiksi reseptienkirjauskäyttöliittymä on paljon käytetty näkymä ERA:n omasta käyttöliittymästä. ERA toimii Software-as-a-Service (SaaS) -periaatteella, joten yksittäisiä käyttöliittymänäkymiä voidaan toimittaa yksinkertaisesti konfiguroimalla asiakkaalle ja käyttäjälle oikeudet tiettyyn toimintoon. Tämän jälkeen käyttäjä pystyy kirjautumaan toimikortin avulla ERAn käyttöliittymään kuvassa 1.1 esitetyltä sivulta.

Tämä tutkimus keskittyy esiselvitystyöhön ERA-järjestelmän graafisen web-käyttöliittymän uudistamiseen. Käyttöliittymä on toteutettu vanhalla teknologialla, jonka kyvykkyydet koetaan riittämättömiksi. ERA sisältää pääosin yksinkertaisia käyttöliittymäelementtejä, ja monimutkaiset komponentit, kuten

**Kuva 1.1:** Kuvakaappaus ERA:n kirjautumisnäkymästä.

visuaaliset aikajanakaaviot, on koettu työläiksi toteuttaa nykyisillä työkaluilla. Täten kiinnostuksena on uuden teknologian käyttöönotto. Uudella teknologialla tarkoitetaan tässä kontekstissa uutta web-käyttöliittymien kehittämiseen tarkoitettua ohjelmistokehystä.

Nykyinen käyttöliittymä sisältää paljon erilaisia terveyteen ja sosiaalihuoltoon liittyviä toimintoja, kuten reseptien kirjausta, koronatodistusten muodostamista ja hoitoon liittyvien valtuutusten myöntämistä. Niinpä koko käyttöliittymä olisi hyvin työlästä ja täten myös kallista uusia kerralla. Nykyinen käyttöliittymä on kuitenkin turhan vanhanaikainen ja kehitys kömpelöä, joten vaiheittaista teknologiavaihdosta halutaan tutkia. Teknologian vaihtamisen lisäksi mielenkiintona on uusien teknologioiden tuomat näkymien integrointimahdollisuudet esimerkiksi muiden toimittajien potilastietojärjestelmiin, mikä mahdollistaisi Atostekin markkinaroolin kasvattamista korkeamman asteen järjestelmätoimittajana käyttöliittymien avulla. ERA ei pysty korvaamaan kokonaisia potilastietojärjestelmiä, vaan se keskittyy Kanta-järjestelmiin liittyvien tietojen käsittelyyn. Täten osittainen integrointi nähdään tehokkaimpana myyntisuuntana, joka huomioisi sekä käyttäjäkokemuksen optimoinnin että Atostekin roolin ohjelmistojen tuottajana.

## 1.2 Tavoitteet

Tavoitteena projektilla onkin selvittää mahdollisuuksia modernisoida käyttöliittymää mahdollisimman pienellä kynnyksellä. Uuden teknologian ja arkkitehtuurin tuomilla mahdollisuuksilla pyritään lisäämään mukautuvuutta asiakkaiden tarpeisiin ja purkamaan vanhan teknologian aiheuttamia rajoitteita uusien toiminnallisuuksien kehittämiseen. Mikäli ERAlle saadaan esimerkiksi suuri asiakas, jolla on tietyt vaatimukset käyttöliittymälle, voidaan yksittäinen käyttöliittymä uudel-

leensuunnitella ja refaktoroida mahdollisimman pienellä työllä uuden sovelluksen puolelle. Täten sovellus modernisoituu asteittain pitkällä aikavälillä.

Tutkimuksella halutaan arvioida, millaisella teknologialla ja arkkitehtuuriratkaisulla asteittainen uusiminen on helpointa. Konkreettisena tavoitteena on parantaa käyttäjäkokemusta, prosessia ja kehittäjäkokemusta. Luku 2 käsittelee tavoiteltuja konkreettisia tavoitteita korkealla tasolla ja perustelee niiden odotettuja vaikutuksia kokonaisvaltaisesti.

Tässä työssä ei siis keskitytä sovelluksen sisäiseen käyttäjäkokemukseen liittyviin parannuksiin, vaan halutaan mahdollistaa tehokkaampi käyttäjäkokemuksen kehitys, jossa asiakkaiden tarpeisiin pystytään mukautumaan nopeasti ja laadukkaasti. Hyötyjä haetaan siis koko käyttöliittymäkehitysprosessiin. Kun saatavilla on kehittyneempiä kirjastokomponentteja ja käyttöliittymän perusta on rakennettu huollettavammalle koodipohjalle, käyttäjäkokemuksen asiantuntijoiden suunnitelmia voidaan toteuttaa monipuolisemmin ja vähemmällä työllä.

## 1.3 Keinot

Jotta muutoksen tarpeet ymmärretään syvällisesti, on tehtävä pohjatyötä nykyiseen ERA-järjestelmään tutustuen sekä teknisesti että ulkopuolisesti. Teknisesti on ymmärrettävä käytetyn ohjelmistokehyksen periaatteet ja rakenteet sekä niiden toteutunut käyttö nykyisessä järjestelmässä. Ulkopuolisesti ominaisuuksiin on myös tutustuttava, jotta ymmärretään tarpeita uusille näkymille ja komponenteille, ja voidaan kohdentaa teknologioiden arviointia relevantteihin ominaisuuksiin. Teknisen tutustumisen päätuotoksena kirjoitetaan luku 4 lähtötilanteesta ja ulkopuolisiin ominaisuuksiin perehtyessä kartoitetaan tutkimuksen tavoitteita jäsentelevän luvun 2 -sisältöä. Tavoitteita liiketoiminnan näkökulmasta kartoitetaan erityisesti palavereissa ERAn kehitystä johtavien työntekijöiden kanssa. Nämä tavoitteet pyritään jäsentelemään vaatimuksiksi käyttöliittymäuudistukselle ja tuotoksena kirjoitetaan luvun 5 listaamat teknologioiden arviointikriteerit. Yleisesti web-kehitykseen liittyvien konseptien ja työkalujen mieleenpalautteluun ja oppimiseen keskitytään työn alkuvaiheessa, jotta muutoksen teknologiavaihtoehtoja osataan arvioida oikeista näkökulmista. Tutkimuksen kannalta olennaista teoriaa pohjustetaan luvussa 3.

Tämän jälkeen luvussa 6 tehdään selvitystyötä mahdollisista teknologiavaihtoehdoista, joilla vaiheittainen käyttöliittymien refaktorointi voitaisiin toteuttaa. Teknologioihin tutustutaan kattavasti ja valitaan käytännössä mahdolliset vaihtoehdot vertailtaviksi. Teknologioiden arvionti sisältää kirjallisuuskatsauksen teknologioiden omiin dokumentaatiosivustoihin sekä teknologioita käsitteleviin tieteellisiin teksteihin ja oppaisiin. Vertailun tuloksena tehdään parhaan teknologiavaihtoehdon valinta. Teknologian objektiivinen valinta vaatii myös selvitystyötä mahdollisista arkkitehtuuriratkaisuista, joilla nykyiseen järjestelmään voidaan liittää uusi käyttöliittymämoduuli. Teknologiavertailun tuotettua kokonaiskuvan nykyisestä teknologiatarjonnasta aloitetaan arkkitehtuuriratkaisun pohdinta ja vaihtoehtojen vertailu. Arkkitehtuurivaihtoehtojen suunnittelu ja niiden arviointi toteutetaan luvussa 7. Arkkitehtuureista valitaan paras luvussa 2 määriteltyjen tavoitteiden pohjalta, minkä jälkeen varmistetaan vielä arkkiteh-

tuuriratkaisun yhteensopivuus parhaimmaksi valitun teknologian kanssa. Mikäli arkkitehtuuri- ja teknologiavalinnat eivät kohtaa, joudutaan vielä molempien hyötyjä ja haittoja priorisoimaan keskenään, jotta voidaan valita paras yhdistelmä uuden käyttöliittymän toteutukselle. Teknologiavalinnan ja arkkitehtuuriratkaisun yhteistoimintaa arvioidaan luvussa 10.

Tutkimuksen päätuotoksena on päätös PoC (engl. Proof-of-Concept) -tyyliseen käyttöliittymäratkaisuun käytettävästä teknologiasta ja teknologian integroivasta arkkitehtuurimallista. Valinta ei siis ole lopullinen, vaan jatkokehitykseen kuuluvan PoC-toteutuksen pohjalta pyritään validoimaan alkuperäisiä tutkimustuloksia tai toteamaan tarvetta tarkemmalle jatkotutkimukselle. Mikäli päätetään pysyä vanhassa teknologiaratkaisussa, pyritään tutkimuksen pohjalta löytämään käytäntöjä, joilla tutkimuksen tavoitteita voitaisiin edistää parantamalla vanhoja prosesseja ja kehitystapoja etenkin uusia käyttöliittymänäkymiä luodessa. Jos kuitenkin liiketoiminnan kannalta hyödyllinen uusi teknologiaratkaisu löydetään, uuden teknologian käytölle ja kehittämiselle muodostetaan myös ohjelmoinnin tyyliohjeet, joilla varmistetaan teknologian vaihdoksen hyötyjen pysyvyys ja täten muutoksen tuomien etujen maksimointi. Tyyliohjeiden kirjoittaminen ei kuulu tämän tutkimuksen sisältöön, mutta niitä pohjustetaan kevyesti jatkokehitystä pohtivassa luvussa 9.

# 2 Tarpeet tutkimukselle

Tässä luvussa käsitellään vielä johdantoa tarkemmin tutkimusta motivoineita liiketoimintaan vaikuttavia tarpeita. Tutkimuksen kohteena on suuri muutos yksityisen ohjelmistoyrityksen merkittävässä omassa tuotteessa ja sen kehitysprosessissa. Täten muutosta koskevien päätösten on oltava tarkkaan harkittuja, ja myös tarpeet vaativat erillistä jäsentelyä. Asiakkaalle halutaan tarjota mahdollisimman laadukas tuote, mutta myös minimoida tarvittavan työn määrä [30]. Jotta liiketoiminnallista menestystä voidaan kasvattaa mahdollisimman optimaalisesti, on huomioitava tavoitteissa sekä tavoitellut hyödyt, että muutoksesta aiheutuvat säästöt ja kustannukset [30]. J. E. Hannayn hyöty-kustannuspohjaista kehitysfilosofiaa halutaankin toteuttaa tässä tutkimuksessa kahdella tasolla; uuden käyttöliittymäosuuden käyttöönottokustannukset halutaan minimoida, mutta monimutkaisempien arkkitehtuurien edutkin halutaan huomioida, eikä turvautua vain hetkellisesti halvimmalta vaikuttavaan ratkaisuun. Lisäksi kehitystyössä ja henkilöstöresurssien käytössä tavoitellaan pitkän aikavälin säästöjä, mutta myös tuotteen kiinnostavuutta loppukäyttäjälle halutaan lisätä. Tätä jaottelua on havainnollistettu kuvassa 2.1.

Tuotteen markkina-arvon maksimointiin pohjautuvia tavoitteita selvennetään alaluvussa 2.1. Jotta liiketoiminnan vaatimuksia voidaan saavuttaa, täytyy henkilöstöresursseja pystyä valjastamaan tehokkaasti ja kestävästi liiketoimintatarpeiden pariin. Näitä tarpeita tarkentaa alaluku 2.2. Jotta henkilöstöresurssit saadaan toimimaan tehokkaasti liiketoimintatarpeiden tavoittelua ajatellen, täytyy itse kehitystyöhön olla hyvät lähtökohdat kehitettävän tuotteen ja työkalujen osalta. Tällaisten tarpeiden kartoittamiseen keskittyy alaluku 2.3. Täten tarpeet puretaan korkealta motiivitasolta lähtien kehitystyön tavoitteisiin, jotka toimivat teknologian ja arkkitehtuurin tutkimuksen lähtökohtina.

Suunniteltu muutos koskee koko tuotteeseen käytettäviä työkaluja ja rakennuspalikoita, eli tuotteen ohjelmistokehystä ja kirjastoja. Täten muutoksella on vaikutuksia useisiin projektiin liittyviin sidosryhmiin. Muutokset näkyvät sekä yrityksen ulkopuolella, että sen sisällä. Kaupalliset asiakkaat tulevat ottamaan käyttöönsä uudenlaisia käyttöliittymänäkymiä ja toimintoja. Tuotteen kehittämiseen osallistuvat työntekijät taas joutuvat muuttamaan toimintatapojaan ja opettelemaan uusia työkaluja, mutta saavat myös kehittämistä nopeuttavia ja helpottavia resursseja käyttöönsä.

## 2.1 Liiketoimintamotiivit

Kaupallisia motiiveja pohtiessa keskitytään suoraan loppukäyttäjille ja asiakkaille näkyviin hyötyihin, joiden perimmäinen tavoite on ERAn myynnin helpottaminen ja asiakastyytyväisyyden ylläpitäminen. Kaupallisiin motiiveihin liittyen keskeisiä kiinnostuksen kohteita ovat täten loppukäyttäjien käyttäjäprofiilit, eli sosiaali- ja terveydenhuollon ammattilaisten mieltymykset ja työskentelytavat sekä heidän

**Kuva 2.1:** Hahmotelma projektin tavoitteista ja hyötyjen jaottelu eri osapuolten intressien kannalta

käytössä olevat sote-alan tiedon arkistointiin liittyvät ulkopuoliset järjestelmät. Myös yleisesti suorityskyvyn kehittymistä ja käyttöliittymän tyyllikkyyttä tavoitellaan tuotteen houkuttelevuuden lisäämiseksi ja myynnin helpottamiseksi.

Käyttäjinä esimerkiksi lääkärit ovat Atostekille kertyneen kokemuksen perusteella hyvinkin erilaisia kuin esimerkiksi tavallisen verkkosivun selailijat. Lääkärit vaikuttavat Atostekin kokemuksen mukaan pitävän näkymistä, joissa on esillä

paljon tietoa ja vähemmän tarvetta uusien sivujen availulle. Vaikka käyttäjäkokemus ei ole tämän työn tutkimuskohde, täytyy siihen vaikuttavat seikat huomioida teknologioiden ja arkkitehtuuriratkaisujen mahdollisuuksia arvioidessa, jotta käyttäjäkokemuksen kehitystyötä ei ainakaan rajoiteta tulevaisuudessa. Lääkäreille on myös tunnistettu konkreettisia tarpeita uusille näkymäkokonaisuuksille, joiden tulevaa kehitystä halutaan edesauttaa. Kokonaisvaltaisemmilla käyttöliittymäominaisuuksilla tavoitteena on kasvattaa roolia korkeamman tason järjestelmätoimittajana, eikä vain myydä yksittäisiä käyttöliittymätoimintoja suurien potilastietojärjestelmien rinnalle. Toistaiseksi ERAn tärkeä kilpailuvaltti on Kanta-integraatiomahdollisuudet rajapinnan kautta ilman erillisiä teknisiä ja byrokraattisia vaatimuksia, kuten käyttöoikeuksien hallintaa, yhteistestausta Kelan kanssa tai tietoturvasertifiointia [67]. Kokonaisia loppukäyttäjän työtehtäviä palvelevat käyttöliittymät voisivat kasvattaa Atostekin roolia korkean tason järjestelmätoimittajana lisäten täten ERAn tuottavuutta.

Muihin järjestelmiin liittyen tutkimuksessa on tärkeää analysoida etenkin ERA-tuotteen suhdetta suuriin potilastietojärjestelmiin, jotka ovat ominaisuuksiltaan ERAa merkittävästi laaja-alaisempia. ERAn toiminnallisuudet keskittyvät tietojen Kanta-arkistointiin, ja se voikin tuoda täydentävästi lisäominaisuuksia potilastietojärjestelmiin, mutta ei pysty niitä täysin korvaamaan. Täten potilastietojärjestelmät ovat ERAn liiketoiminnallisesta näkökulmasta lähempänä myyntimahdollisuuksia kuin kilpailijoita. Osaa nykyisenkin järjestelmän käyttöliittymänäkymistä hyödynnetään jo sellaisinaan aktiivisesti ilman koko ERA-järjestelmän asiakkuutta. Tästä esimerkkinä ERAn reseptinkirjausnäkymä, jota käyttääkseen lääkärin on erikseen kirjauduttava ERAan, kun taas muut potilaskäynnin aikaiset kirjaukset tehdään erilliseen potilastietojärjestelmään. Tämä voi olla lääkärin näkökulmasta epäkäytännöllinen toimintamalli, joten sujuvampaa ERAn osittaiskäyttöä halutaan tälläkin tutkimuksella edesauttaa. Lääkärien tyytyväisyyden tavoittelun avulla halutaan tässä tilanteessa helpottaa ERAn myyntiä sote-alan toimijoille.

## 2.2 Hallinnolliset tavoitteet

Käyttöliittymäuudistuksella ei tavoitella ainoastaan teknisiä ja itse tuotteessa havaittavia hyötyjä. Näiden taustalla huomioidaan myös ohjelmistotuotantoprosessin johtamiseen ja resursointiin liittyvät hyödyt, jotka mahdollistavat tuotteen nopeamman, laadukkaamman ja kustannustehokkaamman kehittämisen. Kehitysprosessista halutaan tehdä ketterämpää, eli uusia ominaisuuksia halutaan pystyä julkaisemaan nopeammin, kun uusia kehitystarpeita huomataan. Tähän liittyen tutkimuksessa huomioidaan teknologian helppokäyttöisyys, teknologialla syntyvän koodin huollettavuus sekä valitun arkkitehtuurin sitoutuneisuus muuhun järjestelmään. Näillä kehitysaskeleilla nykyisillä kehittäjäresursseilla on tarkoitus tuottaa enemmän arvoa samalla työpanoksella ja parantaa markkinamukautuvuutta (engl. time-to-market) [56]. Markkinamukautuvuus on tärkeää Atostekin kilpailukyvyn ylläpidon kannalta, sillä Kelan Kanta-järjestelmän vaatimuksiin mukautuminen ensimmäisenä voi tuoda kilpailuetua.

Helpolla ja miellyttävällä kehityskokemuksella tavoitellaan myös etuja kehittäjien hankinnassa ja pysyvyydessä. Projektiin halutaan löytää helpommin uusia ke-

hittäjiä sekä yrityksen sisällä että uusien työntekijöiden rekrytoinnissa. Atostek pyrkii kunnioittamaan myös kehittäjien toiveita projektinvaihdoksille, jos kehittäjä ei koe projektiaan mielekkääksi. Täten kehittäjäystävällinen teknologia on tärkeä myös kehitystiimin muutostarpeiden minimoimiseksi. Kun muutostarpeita esiintyy, helppokäyttöinen teknologia voi tehdä tiedonsiirrosta ja perehdyttämisprosessista nopeampaa, joten henkilöstöresurssien hallinnan kannalta tavoitellut hyödyt ovat monipuolisia.

## 2.3 Tavoitteet kehittäjän näkökulmasta

Ohjelmiston kehitykseen liittyvät tavoitteet hyödyntävät monesti myös kaupallisten ja hallinnollisten tavoitteiden saavuttamisessa. Esimerkiksi kehittämisen tavoitteena oleva koodin huollettavuuden kehittyminen voi parantaa hallinnollisella puolella resursointikykyä, sillä samat kehittäjät viihtyvät projektissa todennäköisesti kauemmin ja pystyvät työskentelemään tehokkaammin. Lisäksi se voi myös tuottaa vähemmän virhealtista koodia, mikä näkyy loppukäyttäjälle vakaasti toimivana tuotteena. Tarkemmin ohjelmiston huollettavuuden konseptia käsitteenä käsitellään työssä myöhemmin luvussa 3, mutta lyhyesti sillä tarkoitetaan tässä luvussa ohjelmiston laatua palveluntarjoajan, eli Atostekin ja sen työntekijöiden näkökulmasta. Kehitykseen liittyviä tavoitteita käsitellään kuitenkin omana alalukunaan konkreettisesti kehittäjien päivittäiseen työskentelyyn liittyvällä tasolla.

Työskentely nykyisellä teknologiayhdistelmällä, eli ASP.NET MVC -ohjelmistokehyksellä ja jQuery-kirjastolla, on johtanut haasteisiin käyttöliittymäkehityksessä. ASP.NET MVC toimii palvelinpuolella palvellen lomaketyylisiä näkymiä [4]. Selaimessa jQuery-kirjastoa hyödyntävä JavaScript ottaa vastuun käyttöliittymän päivittämisestä. Yksi nykyisen teknologiapinon heikkouksista on jQuery-kirjaston jättämät vapaudet kirjoittaa monenlaista ja epäjärjestelmällistä koodia. Interaktiivisuuden kehittämiseen kaivattaisiin rakennetta esimerkiksi ohjelmistokehyksen tai vahvojen tyyliohjeiden muodossa, jotta järjestelmään perehtyminen olisi helpompaa ja kehitys systemaattisempaa. Kun eri näkymien toiminnot on rakennettu yhtenäisillä periaatteilla, yksittäisten toimintojen oppiminen vaatii vähemmän toimintalogiikan sisäistämistä. Toinen nykyisen järjestelmän haasteista on ASP.NET MVC:n verkkorajapintafunktioiden sekoittuminen käyttöliittymänäkymiä palvelevien funktioiden kanssa. ERA-järjestelmään kaivataan selkeämpää erottelua rajapintatoiminnallisuuksien ja käyttöliittymään liittyvän koodin välille, jotta eri kehittäjät voivat työskennellä rinnakkain eri osa-alueisiin keskittyen.

Kehittäjän näkökulmasta uudella teknologialla pyritään lisäämään koodaamisen tehokkuutta, johon voi vaikuttaa koodin laadun ja uudelleenkäytettävyyden lisäksi teknologian ominaisuudet, kuten kompakti syntaksi ja intuitiiviset toiminnot. Tavoitteena on löytää produktiivisuutta edistävä työkalu, jolla työntekijä voi keskittyä liiketoimintalogiikan keskeisten haasteiden ratkaisemiseen eikä teknologian kanssa painimiseen. ERA:ssa on käytössä myös katselmointiprosessi, jossa jokainen ohjelmistoon tehty muutos tarkastetaan toisen työntekijän toimesta. Teknologian intuitiivisuus helpottaa täten myös katselmointiprosessia, mikä johtaa työskentelyn tehostumiseen.

Kun teknologian uudistamisen johdosta vanhan ja uuden välille saadaan selkeä

erottelu, voidaan pitää huoli uuden koodin laadusta tyyliohjeiden, koodikatselmoinnin ja mahdollisesti ohjelmistokehyksen ominaisuuksien avulla. Uutta koodia halutaan tuottaa modulaarisesti lisäten koodin uusiokäyttötasoa ja vähentäen refaktoroinnin tarvetta tulevia muokkauksia tehdessä. Oman koodin uusiokäytön lisäksi tavoitteena olisi myös kolmansien osapuolten kirjastojen, valmiskomponenttien ja komponenttipohjien paremmat käyttömahdollisuudet. Nykyisille teknologiavalinnoille on ollut vaikea löytää moderneja komponenttipohjia esimerkiksi datan visualisointiin, minkä johdosta on päädytty tekemään kokonaan erillinen sovellus ERA:n rinnalle erääseen visuaalisesti vaativaan käyttöliittymätoimintoon. Tätä halutaan välttää jatkossa, joten ERA:aan itseensä halutaan uuden teknologian yhteydessä nykyistä paremmat kirjastotarjonnat, jotka tarjoavat kattavat rakennuspalikat uusien käyttöliittymäominaisuuksien toteuttamiseen.

## 2.4 Tekniset vaatimukset tavoitteiden pohjalta

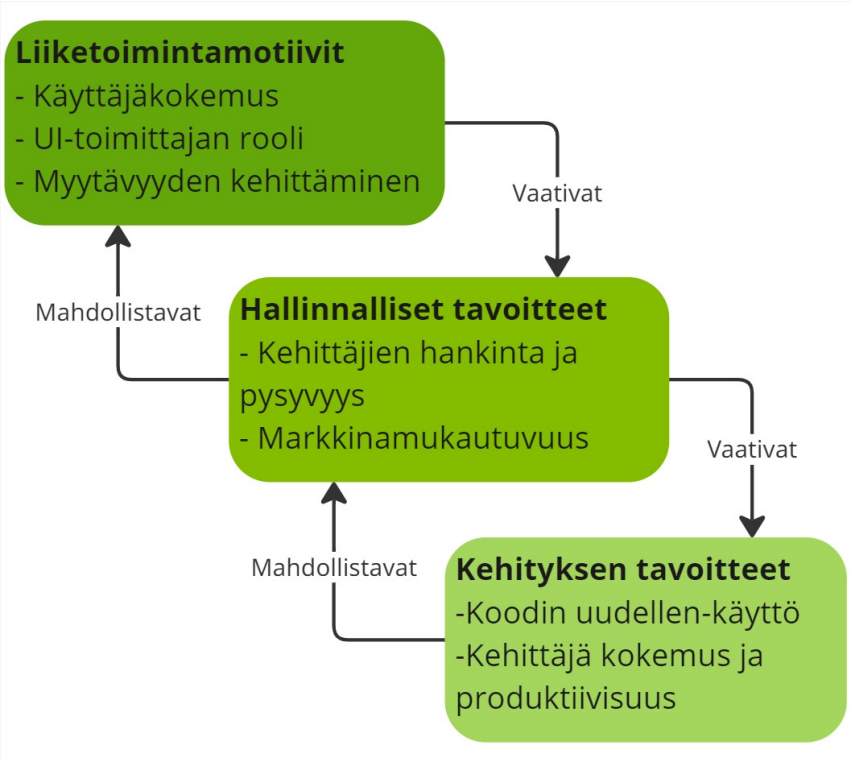
Kaupalliset, hallinnolliset ja kehitystyöhön liittyvät tavoitteet aiheuttavat konkreettisia teknisiä vaatimuksia. Tämä alaluku on siis sidoksissa aiemmin mainittujen tavoitteiden mahdollistajana. Korkean tason tavoitteiden vaatimusten vaikutusta ERA-järjestelmän kontekstissa esitellään seuraavissa kappaleissa. Teknologiavalinnan vaatimuksia ja arkkitehtuuri analyysin lähtökohtia jäsennellään vielä tarkemmin luvussa 5.

Tämän tutkimuksen kannalta tärkeä tekninen vaatimus on uuden käyttöliittymäsovelluksen yhteistyökyvykkyys ERAn nykyisen käyttöliittymän ja muiden järjestelmätoimittajien käyttöliittymäsovellusten kanssa. Erityisesti nykyisen käyttöliittymän tulisi toimia uuden rinnalla saumattomasti joko uudelleenohjauksien tai käyttöliittymäupotusten avulla. Autentikaatioon täytyy riittää yksi kirjautuminen, ja auktorisointiin käytetyt tiedot halutaan välittää sovellusten välillä huomaamattomasti, mutta myös tietoturvallisesti. Latausajat siirtyessä sovelluksesta toiseen eivät voi myöskään olla pitkiä, jotta sovelluksen käyttäjän työskentely pysyy sujuvana.

## 2.5 Koonti tarpeista

Tutkimuksen tarpeet ovat vahvasti sidoksissa toisiinsa, mitä havainnollistaa kuva 2.2. Kuvasta nähdään, kuinka eri kategorioihin luokitellut tavoitteet ja motiivit vaikuttavat toisiinsa, ja alimpana kerroksena kehitystason tavoitteet toimivat perustana liiketoimintamotiivien saavuttamiseksi.

Koska kehityksen tavoitteilla on epäsuoria vaikutuksia sekä hallinnollisten tavoitteiden että liiketoimintatarpeiden saavuttamiseen, voidaan niitä pitää erityisen tärkeinä tutkimuksen kannalta. Koodin huollettavuus ja uudelleenkäyttö ovat kehittäjätasolla erityisesti arkkitehtuuriin liittyviä tarpeita, jotka parantavat myös kehittäjäkokemusta. Käyttöliittymäteknologia on kehittäjän päätyökalu, joten vaihtoehtojen vertailu on olennaista erityisesti kehittäjäkokemuksen kannalta. Tyytyväiset ja tehokkaat kehittäjät mahdollistavat projektin hallinnan jatkuvuuden ja nopean mukautuvuuden markkinatarpeisiin. Lopulta kehitysprosessia koskevat tavoitteet




**Kuva 2.2:** Tutkimuksen motiivit ja tavoitteet tiivistettynä

auttavat mukautumaan käyttäjien tarpeisiin ja tuottamaan kohderyhmää kiinnostavia ominaisuuksia. Rajoitteita tutkimukselle asettaa kuvassa 2.1 jäsennelty hinnan minimointitarve, joka estää teknologian kokonaisvaltaisen vaihtamisen yhdellä tuotantojulkaisulla.

# 3 Teoriaa web-teknologioiden kehittymisestä ja ohjelmistojen uudistamisesta

Teknologioita arvioidessa on ymmärrettävä web-kehityksen erityispiirteet tarkasti, jotta teknologioiden ominaisuudet voidaan arvioida perusteellisesti. Teknologioiden kehittymistä kartoitetaan seuraavassa alaluvussa 3.1 vaiheittain teknologioiden evoluutiota mukaillen.

## 3.1 Web-teknologioiden kehittyminen

Web-ohjelmointia karakterisoiva pääperiaate on yhteensopivuus. Idea lähti liikkeelle akateemisista ja sotilaallisista tarpeista tietokoneiden yhdistämiselle. [42] Web-ohjelmointiin, eli selainsovellusten ja verkkosivujen kehittämiseen, on kehittynyt vastuunjako kolmen eri osa-alueen välille. Erillisiä osa-alueita ovat käyttöliittymän muodostavan sivun rakenne, sen käyttäytyminen sekä käyttöliittymäkomponenttien tyyli ja asettelu. Osa-alueille on määritetty omat standardinsa, mikä mahdollistaa yhteensopivuuden alustasta riippumatta.

JavaScript ja sen laajennettu versio, TypeScript, ovat nykyään valtavirran suosiossa [70]. Suuri osa käyttöliittymien toteutusvastuusta on siirretty selaimelle. Riippuvuuspino on kasvanut suuresti uusien ohjelmistokehysten myötä. Painopiste on jakautumassa osittain takaisin palvelinpuolelle johtuen modernien teknologioiden mahdollisuudesta hyödyntää sekä selaimen että palvelimen hyviä puolia käyttäjäkokemuksen optimointiin [11].

### 3.1.1 Verkkosivujen sisältö ja yhdistäminen

Web-sovelluksiksi kutsutaan ohjelmistoja, joissa käyttöliittymä esitetään selaimessa. Selain tarjoaa standardoidun rajapinnan käyttöliittymään sisällön määrittämiseen, jota kutsutaan . HTML (HyperText Markup Language) on yleisesti selainten ymmärtämä merkintäkieli. Jotta HTML-dokumentin kuvaama sisältö saadaan näkyville käyttäjän selaimeen, täytyy se lähettää palvelimella suoritettavasta sovelluksesta, jota kutsutaan yleisesti back-end-sovellukseksi. Backend-sovellus lähettää selaimessa näytettävän HTML-sisällön verkon yli HTTP (Hyper-Text Transfer Protocol) -viesteillä. HTTP on standardoitu tapa kommunikoida internetin välityksellä, joten eri teknologioilla toteutetut järjestelmät voivat toimia yhteistyössä keskenään HTTP:lle määriteltyjen rajapintojen kautta. [83]. Tämä mahdollistaa tehtävien jaon käyttöliittymän ja liiketoimintalogiikan välillä.

### 3.1.2 Interaktiivisuus ja JavaScript

Pelkkä HTML-sisältö ei riitä nykyaikaisten verkkosivujen rakentamiseen. Pelkillä HTML-rakenteilla saataisiin aikaan hyvin pelkistettyä, staattista Word-dokumentin tyylistä sisältöä. Staattisuus tarkoittaa kyvyttömyyttä reagoida käyttäjältä saatuun syötteeseen, kuten hiiren painalluksiin tai tekstisyötteeseen. Käyttäjälle näytetään juuri se sisältö, mitä backend-sovellus on HTML-rakenteisiin määrittänyt, ja vain hyperlinkit toivat interaktiivisuutta verkkosivuihin [69]. Staattiset verkkosivut olivat kuitenkin yleisesti käytössä, kunnes JavaScript kehitettiin tuomaan dynaamisuutta verkkosivujen sisältöön. Staattisten sivujen esittämiseen vaadittiin palvelimelta vain kyvykkyydet kuvantaa HTML-sisältöä ja toteuttaa internet-kommunikaatiota.

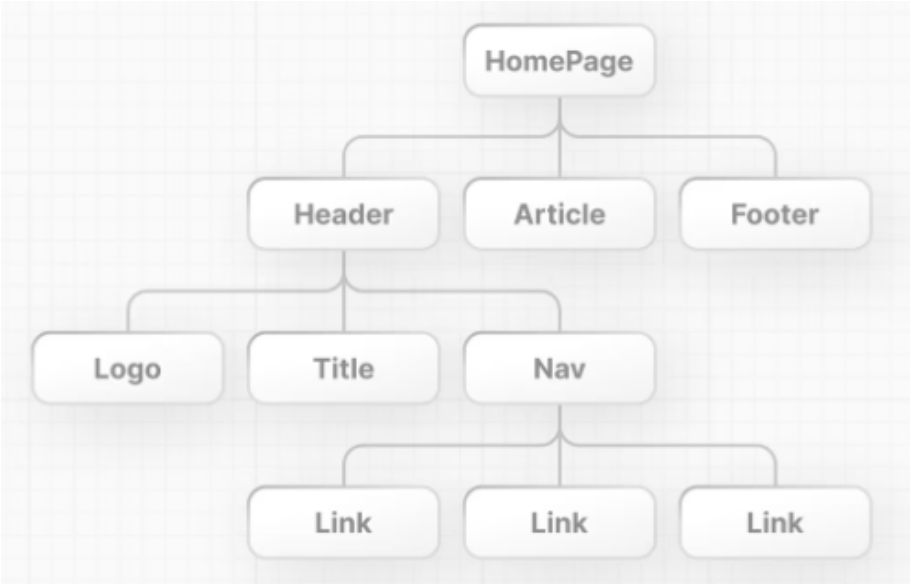
JavaScriptin suosion kasvu loi trendin, jossa uudet sovellukset olivat suurimmaksi osaksi niin-kutsuttuja yhden sivun sovelluksia (engl. Single Page Application, SPA). Tämä tarkoittaa sitä, että käyttäjälle näytettävät komponentit ovat aina saman ylemmän tason HTML-elementin sisällä, vaikka käyttäjä saattaakin saada käyttöliittymästä vaikutelman, että sivu vaihtuu. Yhden sivun sovelluksetkin voivat siis sisältää semanttisesti erillisiä sivuja, mutta sivun vaihtuminen tapahtuu JavaScriptin avulla selaimessa, jolloin vain esitettävä komponenttipuun ”oksa” vaihtuu. Täten sovelluksen sisäinen konteksti voi säilyä navigointien yli komponenttipuun ”rungossa”. Tätä ennen käytetyt staattiset sivut olivat monen sivun sovelluksia (engl. Multi Page Application, MPA), joissa sivulta toiselle navigoitiin hyperlinkkejä painamalla. Hyperlinkit antoivat selaimelle käskyn ladata palvelimelta kokonaan uusi HTML-elementtipuu, jolloin edes puun juurielementti ei pysynyt samana. [66]

JavaScript pohjautuu ECMAScript-standardiin. Tämä standardi sisältää määrittelyn esimerkiksi JavaScriptin syntaksille, tyypeille, virheenkäsittelylle ja moduuleille. ECMAScript-standardi päivittyy vuosittain sisältäen uusia ominaisuuksia, ja selainten tehtävä on rakentaa tuki standardeille, jotta selaimilla pystytään hyödyntämään JavaScriptin ominaisuuksia ennustettavasti. [43] Web-ohjelmoinnissa siis HTML määrittelee sivujen sisällön, JavaScript muokkaa sitä käyttäjän toimien mukaisesti. Lisäksi CSS (Cascading Style Sheets) -kieli vastaa sivujen visuaalisesta ilmeestä. CSS on osa käyttöliittymien toteuttamisen vastuunjakoa, mutta se ei ole tämän tutkimuksen kannalta olennainen kokonaisuus, sillä tyylimääritelmät eivät ole keskeisesti riippuvaisia teknologioista ja arkkitehtuurista.

Toisaalta vastuunjaottelua rikkovat nykyään CSS:n ja HTML:n kyky korvata osittain myös JavaScriptin roolia interaktiivisuuden mahdollistajana. HTML on lisännyt natiiveiksi elementeikseen erilaisia interaktiivisia komponentteja, kuten <details>, <dialog> ja <summary>, joilla voidaan esittää tietoa dialogeissa tai klikkauksesta aukeavissa elementeissä [42]. HTML:ään on myös kehittynyt standardi kustomoitavien HTML-elementtien toteuttamiseksi JavaScriptiä hyödyntäen Web Components -rajapinnalla [9]. Web Components -rajapinnan hyödyntämistä käsitellään tarkemmin luvussa 7. Myös CSS:llä pystytään esimerkiksi asettelemaan HTML-elementtejä klikkauksesta aukeavaksi dropdown-valikoksi ilman JavaScriptiä [15]. Web-ympäristö tarjoaa täten sovellusten rakentamiseen lukuisia mahdollisuuksia, mutta JavaScript-kieleen pohjautuvat ohjelmistokehykset täyttävät vielä oman roolinsa kehittyneiden toiminnallisuuksien rakentamisessa.

### 3.1.3 Ohjelmistokehykset ja työkalut

Tutkimus käsittelee merkittävänä osana uusia teknologiavaihtoehtoja ERAn käyttöliittymäuudistuksen toteuttamiseen. Teknologioilla tekstissä tarkoitetaan web-käyttöliittymien kehitykseen käytettäviä ohjelmistokehyksiä, kuten React.js:ää ja Angularia. Nämä kehykset ovat ajoympäristönsä johdosta pääosin JavaScript-kielen päälle kehitettyjä työkaluja käyttöliittymien rakenteen ja käyttäytymisen määrittämiseen. Suositut web-käyttöliittymäohjelmistokehykset perustuvat komponenttipohjaiseen arkkitehtuuriin. Komponenteista rakennetaan hierarkkisia kokonaisuuksia, joita kutsutaan komponenttipuiksi. Tätä nimitystä havainnollistaa kuva 3.1. Yksittäinen komponentti kapseloi käyttöliittymän toiminnallisuutta, kun komponenttien välille on kehitetty rajatut mekanismit tiedon välittämiseen, ja komponentin tehtävä on läheisesti sidottu käyttöliittymään liitettävään DOM-elementtiin.




**Kuva 3.1:** Visualisoitu esimerkki komponenttipuusta [59].

Kasaustyökalut, kuten Webpack tai Vite, kasaavat useista moduuleista koostuvat sovelluskokonaisuudet erillisistä tiedostoista yhteinäisiksi selaimessa tai palvelinpuolen JavaScript-ympäristössä suoritettaviksi koodinipuiksi (engl. Bundle) [80]. Selainten JavaScript-moottorit eivät suoraan pysty suorittamaan ECMAScript-standardista poikkeavia erikoissyntakseja, kuten JSX:ää, tai JavaScript-kieltä laajentavaa TypeScript-syntaksia. Näitä varten kasaustyökalut käyttävät erillisiä loader-työkaluja syntaksin kääntämiseen puhtaaksi JavaScript-koodiksi käännösvaiheessa [80]. JSX on esimerkki loader-työkalun mahdollistamasta ominaisuudesta, joka mahdollistaa käyttöliittymän ohjelmointiparadigman muuttamisen käskymuotoisista DOM-muokkauksista kuvailevaan elementtimuotoon. TypeScript ja JSX ovat molemmat siis loader-ohjelmien tuomia kehitysvaiheen apumekanismeja, jotka eivät näy selaimelle koodia suoritettaessa.

### 3.1.4 Meta-ohjelmistokehykset

Modernit web-teknologiat, joita kutsutaan myös ohjelmistokehyksiksi, keskittyvät yleisesti käyttöliittymänäkymien ja komponenttien rakentamiseen ja esittämiseen. Tämä jättää avoimeksi teknisiä yksityiskohtia, kuten logiikan, jolla käyttöliittymä välitetään verkon yli palvelimelta käyttäjän selaimeen, koodin suorittamisen optimoinnin, käyttöliittymän tilan hallinnan ja datan kommunikoimisen palvelimen kanssa. Yhdistämällä kehitys- ja kasaustyökalut ja niiden konfiguraatiot ohjelmistokehyksiin on kehitetty tavallisia ohjelmistokehyksiä monipuolisempia työkaluja. Meta-ohjelmistokehykset, joita jatkossa kutsutaan lyhyemmin metakehyksiksi, tuovat rakennetta ja valmiita toiminnallisuuksia myös muihin edellä mainittuihin toiminnallisuuksiin [13]. Esimerkiksi React-teknologialla käyttäjä voi rakentaa käyttöliittymän niin, että komponenttipuun juurikomponentin määrittävä funktio palauttaa pitkässä if-else-ketjussa erilaisia käyttöliittymäsivuja riippuen käyttöliittymän tilasta. Reactille rakennettu metakehys Next.js puolestaan tuo yhtenäisen rajapinnan käyttöliittymäsivujen ja -elementtien esittämiseen App Router -toiminnollaan. App Router linkittää näkymät ja komponentit sovelluskansion tiedostorakenteen perusteella sekä sitoo selaimen osoitepalkissa näkyvän URL-osoitteen tiedostorakenteeseen [51]. Tällöin selaimen peruutusnappia painettaessa sovellus palautuu edelliseen näkymään monen sivun sovelluksen tavoin kuitenkaan lataamatta koko sovellusta uudestaan. Tämä on esimerkki metakehyksen tuomasta toiminnallisuudesta, jonka myötä myös sovelluksen projekti rakentuu järjestelmällisesti käyttöliittymän komponenttisuhteiden mukaan.

Reactin luonteesta ohjelmistokehyksenä on kiistelty verkkosovelluskehittäjien keskuudessa, koska se voidaan nähdä vain kahtena kirjastona, ReactDOM ja React, jotka ladataan käyttäjän selaimeen sovelluksen osana. Puolestaan esimerkiksi Angular kokonaisvaltaisena ohjelmistokehyksenä toteuttaa myös palvelinpuolella ohjelmiston käsittelyä ja toimintaa koskevia tehtäviä, jotka React jättää kehittäjän vastuulle [3]. Täten kehittäjän ei tarvitse määritellä esimerkiksi edellisessä alaluvussa mainittujen kasaustyökalujen konfiguraatioita. Reactin tyylisten selainpuolen ohjelmistokehysten päälle onkin kehittynyt kokonaisvaltaisempia korkean tason ohjelmistokehyksiä, kuten Next.js tai Svelte Kit [51], [71]. Näitä kutsutaan yleisesti meta-ohjelmistokehyksiksi (engl. Meta Framework), jotka vastaavat Angularin tarjoamaa abstraktiotasoa.

## 3.2 Ohjelmistot ja niiden uusiminen

Verkko-ohjelmointi rakentuu lukuisten erityyppisten rajapintojen päälle. Rajapinnat voivat olla sovelluksen ajoympäristöä määrittäviä, kuten itse JavaScript ja selaimen tarjoamat verkkoympäristön rajapinnat (engl. Web APIs) [44], tai abstraktiokerroksia tuovia apuvälineitä, kuten ohjelmistokehykset ja kirjastot. Verkkoyhteydessäkin jokainen protokollakerros tuo oman abstraktiotasonsa, mutta korkeimmalla tasolla HTTP-protokollan muodostama rajapinta riittää usein käyttöliittymän verkkokommunikaation määrittämiseen [83]. Käyttöliittymien ohjelmointiin selain tarjoaa rajapinnan JavaScript-ajoympäristön muodossa, jonka rajapintamäärittely perustuu ECMAScript-standardiin. JavaScriptin pohjalta selain tarjoaa lisää rajapintoja, joista tärkeimpänä DOM (Document Object Model). DOM muo-

dostaa oman rajapintansa JavaScriptin ja selaimessa näytettävän sisällön välille, muodostaen rungon kaikille verkkosivustoille ja web-sovelluksille [43]. JavaScriptiä hyödyntäen on rakennettu lisää valmiita abstraktioita kirjastojen ja ohjelmistokehysten muodossa. Myös yksittäiset käyttäjän määrittelemät luokat ja funktiot määrittelevät käytölleen rajapinnan saatavilla olevien metodien sekä funktioiden parametrien ja paluuarvojen muodossa [28].

Rajapinnat kehittyvät itsenäisesti, jolloin kuluttajan ja tarjoajan sopimukseen tulee ristiriita. Koska ulkopuolisia riippuvuuksia hyödynnetään paljon nykypäivän ohjelmistokehityksessä, on tärkeää, että niitä pystytään päivittämään ilman, että omia ohjelmistoja joudutaan merkittävästi muokkaamaan. Tähän pyritään tuomaan selkeyttä ohjelmistojen semanttisen versioinnin spesifikaatiolla, josta käytetään lyhennettä SemVer [55]. Semanttisessa versioinnissa hyödynnetään ohjelmistoversioiden numeroinnissa universaalia järjestelmää, josta pystytään päättelemään päivityksissä toteutettujen muutosten vaikutusta ja kokoluokkaa. Semanttisen versioinnin järjestelmässä versionumero on kolmi-osainen, ja muuttujilla X, Y ja Z esitettynä se on formaatiltaan X.Y.Z. Numeroista ensimmäinen, eli X, kasvaa, kun päivitys tuo merkittäviä, vanhoja toiminnallisuuksia rikkovia muutoksia ohjelmistoon. Tästä käytetään tässä tutkimuksessa jatkossa nimitystä *pääversio*. Keskimmäistä Y-numeroa kasvatetaan taaksepäin-yhteensopivia uusia toimintoja julkaistaessa. Uuden Y-numeron tuovaa ohjelmistoversiota kutsutaan tässä tutkimuksessa *aliversioksi*. Viimeisellä Z-numerolla ilmoitetaan pienistä bugikorjauksista, jotka eivät tuo ohjelmistoon uutta toiminnallisuutta. [55] Yhteinen standardin mukainen versiointijärjestelmä on useiden ohjelmistojen yhteistyössä tärkeä, sillä pääversioihin perustuviin toimivuuslupauksiin pystytään luottamaan, eivätkä aliversioiden päivitykset aiheuta riippuvuusketjujen hajoamisia.

### 3.2.1 Huollettavuus, tekninen velka ja teknologia-siirtymä

Teoreettisesti tutkimuksen päätavoite on teknisen velan takaisinmaksaminen. Kun nykyiset kehitystyökalut on todettu epäoptimaalisiksi ohjelmistoprojektin tarpeisiin suhteutettuna, teknologian käytön jatkaminen voidaan tulkita teknisen velan kasvamiseksi. Uusia yksittäisiä toiminnallisuuksia on ollut hetkellisesti edullisinta toteuttaa valmiiksi käytössä olevilla työkaluilla, mutta ohjelmiston yleinen huollettavuustaso ei ole päässyt paranemaan uusien toimintojen kehittyessä, kun teknologian heikkouksista ei ole päästy eroon. Teknisen velan syynä on suuri kynnys teknologian vaihtamiselle, kun vanha sovellus sisältää paljon toiminnallisuuksia, jotka olisi toteutettava uudelleen erilaisella työkalustolla. Tutkimuksen tärkeänä pyrkimyksenä onkin kuvainnollisesti ilmaistuna löytää lähtökohdat teknisen velan takaisinmaksulle hiljattain ”osamaksuna”, eli siirtymällä toiminnallisuus kerrallaan uuteen teknologiaan.

Englanninkielinen termi Strangler Pattern kuvaa vanhan ohjelmistojärjestelmän uusimista vaihe vaiheelta ”kuristaen”, eli siirtäen toiminnallisuutta osissa, kunnes koko järjestelmä toimii uudella mallilla [88]. Jatkossa kyseistä toimintamallia kutsutaan suomennettuna kuristusmalliksi. Kuristus-analogia on peräisin köynnöskasveista, jotka hiljattain peittävät puun alleen, lopulta kuristaen alkuperäisen paikalla olleen puun kuoliaaksi. Tällä periaatteella on kehitetty malli vanhojen palvelinohjelmistojen uusimiseen, jossa vanhaa monoliittista järjestelmää läh-

detään kasvattamaan uudella itsenäisellä ohjelmistoarkkitehtuurilla. Samalla luodaan esimerkiksi uusia mikropalveluita vanhan rinnalle, jolloin vanhaa järjestelmää ei tarvitse korvata kerralla kokonaan. [88] Kuristusmalli on täten hallittu ja riskejä minimoiva tapa vähentää vanhan teknologian ja epäselväksi paisuneen arkkitehtuurin aiheuttamaa teknistä velkaa. Verkkokäyttöliittymäsovelluksille kuristusmallista löytyy vähemmän tietoa, mutta sitä mainostetaan käytännön toteutuksessa toimineena teknologiamigraatiotapana kaupallisen yrityksen, Dualboot Partnersin toimesta [53]. Yrityksen dokumentoima kuristusmallin hyödyntäminen mukailee tämän tutkimuksen tavoitteita ja teknologiavaihdoksen lähestymistapaa. Erityisesti tärkeä kuristusmallin tuoma mahdollisuus on jatkaa käyttöliittymäohjelmiston toiminnallista kehittämistä teknologian vaihtamisen ohella.

### 3.2.2 Arkkitehtuuri- ja suunnittelumallit

Tutkimuksen taustalla on halu tuottaa ymmärrettävää, uudelleenkäyttöistä ja toimintavarmaa koodia. Tätä varten on kehitetty erilaisia arkkitehtuuri- ja suunnittelumalleja, jotta hyväksi todettuja ohjelmarakenteita voidaan uudelleenkäyttää ratkaisemaan samantyyppisiä suunnittelutarpeita eri tilanteissa [28]. Arkkitehtuurimalleille on selkeästi dokumentoitu niiden käyttötarkoitukset ja mahdolliset seuraamukset käytöstä, jolloin päätös mallien toteuttamisesta voidaan tehdä perustellusti. Täten mallit ovat kuin tapa siirtää kokeneiden ohjelmistosuunnittelijoiden tietämystä vähemmmän kokeneiden saataville, jotta vahingossa tehtyjä ongelmallisia ohjelmistorakenteita voidaan välttää. Arkkitehtuurimallit esiintyvät käyttöliittymäteknologioden tutkimuksessa ja nykyiseen käyttöliittymäteknologiaan tutustuessa, sillä monet teknologiat tukevat tai jopa vaativat tiettyjen mallien käyttöä. Nykyisen ERA-järjestelmän käyttämä ASP.NET MVC on jo nimensä puolesta selkeä esimerkki Model-View-Controller (MVC) -arkkitehtuurimallista, jossa koko ohjelmiston roolit on karkeasti jaoteltu kolmeen toisistaan erilliseen osaan. Tiedon sisältö eli malli (engl. Model), tiedon käyttäytyminen (engl. Controller) ja tiedon esittäminen (engl. View) ovat luontaisesti erityyppisiä haasteita, joten niiden erottelu luo korkean tason arkkitehtuuriin modulaarisuutta [28].

Pelkkä korkean tason vastuujaottelu ei riitä huollettavan koodin tuottamiseen, sillä yksittäiset toiminnot vaativat lukuisia ohjelmistotason ratkaisuja. Yksittäiset luokat voivat muodostaa hyvinkin sekavia kokonaisuuksia, mikäli ne rakentavat riippuvuuksia toisiinsa hallitsemattomasti [25]. Tätä varten on kehitetty lukuisia matalan tason suunnittelumalleja, jotka voidaan jakaa luomista (engl. creational), rakennetta (engl. structural) ja käyttäytymistä (engl. behavioural) määrittäviin malleihin [28]. Esimerkiksi ohjelmistokehys Angular käyttää rakenteellista Decorator-mallia luokkien luonteen määrittelyyn, ja koko ohjelmistokehys perustuu vahvasti arkkitehtuurimalleihin. ”Component” -Decoratorilla luokalle voidaan määrittää käyttöliittymään kuuluvan komponentin ominaisuuksia, kun taas ”Injectable” -Decoratorilla voidaan määrittää komponenteille tai moduuleille riippuvuudeksi tarkoitettu luokka [3]. Komponenttipohjainen rakenne noudattaa ”Composite” -suunnittelumallia, sillä se kasaa yksittäisiä komponentteja puurakenteeksi, jossa useamman komponentin sisältävää korkeamman tason komponenttia voidaan käsitellä yksittäisenä komponenttina [28]. Riippuvuuksien injektointi on myös suunnittelumallien mukainen tapa eriyttää palvelun käyttäjä palvelun toteutuksesta modulaarisuutta parantaen [3].

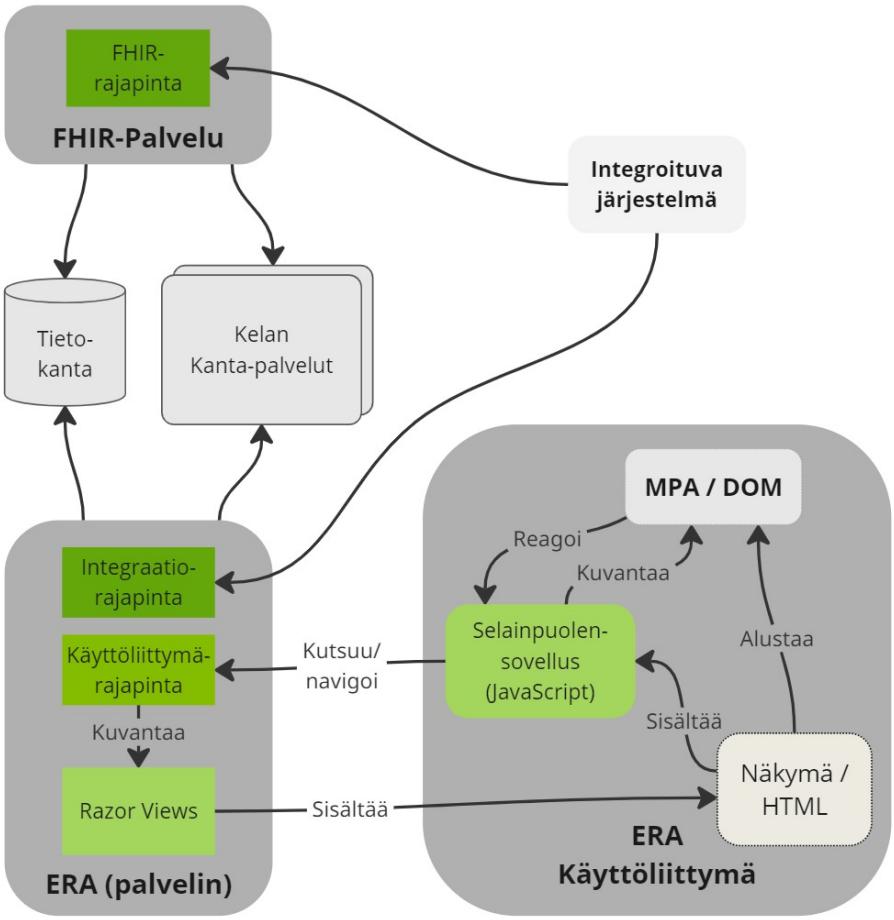
# 4 Järjestelmän taustatiedot

Tutkimus kohdistuu sosiaali- ja terveysalan dataa käsittelevän ERA-järjestelmän käyttöliittymään. Nykyinen järjestelmä vaikuttaa erityisesti tutkimuksen arkkitehtuurisuunnitteluun liityvään osuuteen, sillä se asettaa raamit uuden käyttöliittymän integroitumiselle. Seuraavat alaluvut esittelevät karkeasti ERA-järjestelmän käyttöliittymälle oleellisia sovelluskomponentteja ja niiden rajapintoja. Tutkimuksen teknologia-arviointiin vaikuttaa myös nykyisen käyttöliittymäteknologian yhtäläisyydet ja toiminta-tavat, joten teknologioiden ominaisuudet esitellään pääpiirteittäin. Luvun viimeinen alaluku käsittelee järjestelmän lähtötilanteen suotuisuutta ja mahdollisia haasteita teknologian uusimiselle.

ERAn on alunperin kehitetty helpottamaan järjestelmien integroitumista Kelan maanlaajuisiin Kanta-palveluihin. ERA on tuonut lisätoiminnallisuuksia tietojen arkistoinnin päälle, joten ERAn integraatiorajapinnalle on kehittynyt Kanta-palveluiden tietomalleja mukailevat ja laajentavat XML-tietomallit. ERA sisältää myös Kanta-palveluista riippumattoman palvelinohjelmiston, joka toimii standardin mukaisena FHIR (Fast Health Care Interoperability Resources)-rajapintana kliinisen datan käsittelyyn. FHIR-palvelulle ei ole vielä kehitetty käyttöliittymää, sillä se on suunniteltu toteutettavaksi ERAn uudella käyttöliittymäteknologialla. Kuva 4.1 havainnollistaa ERA-järjestelmän keskeisiä sovelluskomponentteja ja niiden vastuualueita. Nuolet kaavioissa kuvastavat suhteita ohjelmistokomponenttien tai teknisten rajapintojen välillä. Nuolen suunta määrittää toimintojen aktiivisen osapuolen. Käyttöliittymän sisäistä vastuunjakoa tarkentavat nuolien toimintoja kuvaavat verbit, joiden on tarkoitus selittää myös teknologioiden yhteistoimintaa.

## 4.1 Palvelinohjelmat ja rajapinnat

Järjestelmä on rakennettu .NET Framework -teknologiaan pohjautuvalla ASP.NET MVC -ohjelmistokehyksellä. Ohjelmistokehys noudattaa nimensä mukaisesti Model-View-Controller -mallia. Ohjelmistokehys reitittää pyynnön polkunsa mukaiseen Controller-metodiin, joka toimii ohjaavana yksikkönä pyynnön käsittelyssä. Liiketoimintadataa eli ERAn tapuksessa kliinistä dataa ohjelmistossa käsittelee Model-luokat, jotka toteuttavat tiedon muokkauksen ja säilömisen pysyvien tietovarantojen avulla. Controller-metodit ovat jakautuneet käyttötarkoitukseltaan kahteen tyyliin. Integraatiorajapintaa palvelevat Controller-metodit eivät hyödynnä MVC-mallin View-osuutta, vaan palauttavat dataa palvelinintegraatioille sopivassa XML-muodossa. Käyttöliittymää palvelevat Controller-metodit palauttavat kullekin näkymälle Razor-syntaksilla valmistellun HTML-rakenteen, eli Razor-näkymän. Razor on näkymätemplaattien kasaamiseen rakennettu syntaksilaajennos C#-kieleen. Razor-näkymät muodostavat MVC-arkkitehtuurin View-osuuden.[4] Käyttöliittymää varten on kehitetty myös JSON-dataa palauttavia Controller-metodeita, jotta koko käyttöliittymän HTML-rakennetta ei tarvitse ladata käyttäjän interaktioiden johdosta uudestaan.




**Kuva 4.1:** Karkea hahmotelma ERA-järjestelmän nykyisestä arkkitehtuurista.

Muuttuvien tarpeiden myötä ERAan on toteutettu datan kommunikoinnin yhtenäistämiseen keskittyvän FHIR-standardin mukaisia kyvykkyyksiä. FHIR on lyhenne sanoista Fast Healthcare Interoperability Resources, ja nimensä mukaan se pyrkii tekemään järjestelmien välisestä terveysdatan kommunikoinnista nopeaa ja yhteensopivaa [21]. ERA tukee FHIR-palvelun R4-versiota. FHIR-palvelu on toteutettu muuta ERA-ohjelmistoa tuoreemmalla .NET 8 -teknologialla, mutta teknologian yksityiskohdat eivät ole käyttöliittymän kannalta merkittäviä.

## 4.2 Käyttöliittymäteknologia

Nykyinen ERA-järjestelmän palvelinpuolella toimiva ohjelmisto on kirjoitettu C# -ohjelmointikielellä. Kieltä käytetään Razor-syntaksia hyödyntäen myös käyttöliittymän HTML-dokumenttien rakentamiseen, mikä erottaa sen moderneista, usein JavaScriptiä hyödyntävistä verkkosovellusteknologioista, kuten useat kappaleessa 6 käsitellyt verkkosovellusten rakennustyökalut. Razor-syntaksilla on määritelty jokaiselle näkymälle pohja, jonka palvelinpuolen ohjelma kasaa selaimen ymmärtämään HTML-muotoon. Valmiiksi rakennettu HTML-näkymä lähetetään selaimeen, jossa JavaScript-koodi ottaa vastuun näkymän muokkaamisesta ja tietojen kommunikoinnista käyttäjän syötteeseen reagoiden. Teknologian käyttäytyminen on siis perinteisten monen sivun sovellusten käyttäytymistä mukailevaa, mut-

ta se sisältää myös modernimpia Ajax-periaatteita, kuten käy ilmi teknologian alkuaikoina julkaistusta Microsoftin K. Smithin artikkelista [69]. Artikkelista käy ilmi teknologian rooli Ajax-tyylisten sovellusten uranuurtajana, mikä selittää ASP.NET MVC:n vanhahtavia piirteitä.

## 4.3 Resurssit ja tietokanta

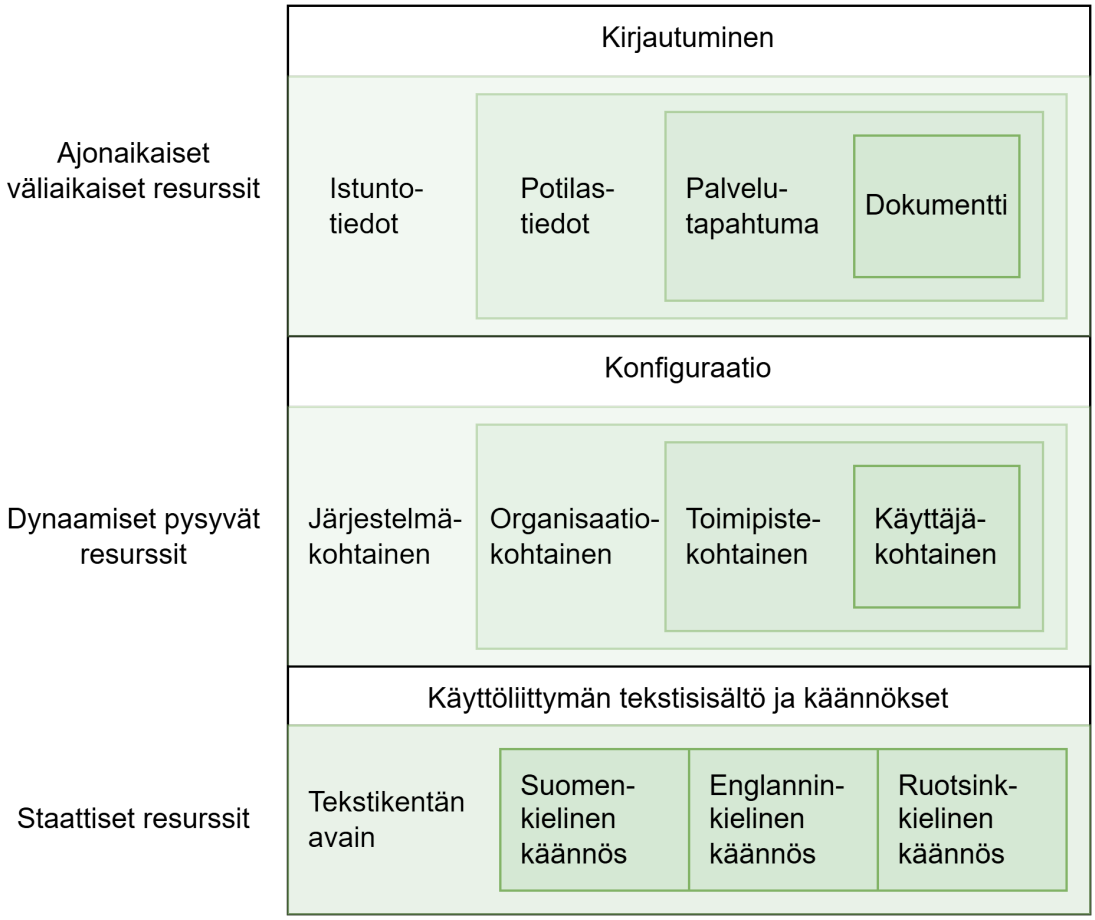
Kliinisen datan pysyvänä säilytyspaikkana ERA käyttää Kanta-palveluita, kuten potilastietovarantoa ja sosiaalihuollon asiakastietovarantoa. ERA sisältää myös tietokannan väliaikaiseen tilan hallintaan. Kanta-palveluiden pysyviä resursseja käsitellään ERAn tietokannassa istuntojen ajan. Täten tilallinen liiketoimintadata on ERAn monia asiakkaita palvelevassa (engl. multi-tenant) sovellusmallissa tallennettuna keskitetysti ja käyttäjäkohtaisesti.

Tiedon säilömisen kannalta käyttöliittymäarkkitehtuurin suunnittelulle olennaista ovat ERAn palvelinsovelluksen toimintaperiaatteen ja liiketoimintatarkoituksen tuomat erityiset ominaispiirteet. Palvelinsovelluksen tallentamat käyttöliittymälle merkittävät tiedot koostuvat käyttäjäkohtaisesta konfiguraatiosta, kontekstitiedoista sekä käyttöliittymän monikielisistä tekstisisällöistä. Näistä lyhytikäisiä ovat kontekstitiedot, jotka sisältävät esimerkiksi asiakaskohtaamiseen liittyviä istuntotietoja. Pitkä-ikäisiä sovelluksen käyttökertojen välillä säilyviä tietoja ovat konfiguraatiotiedot ja käyttöliittymän esittämät tekstisisällöt. Resurssien elinkaarta ja niiden omistussuhteita havainnollistaa kuva 4.2.

## 4.4 Lähtökohdat teknologian uudistamiseen

Käyttöliittymäkoodin rakenne kaipaa vaiheittaista uudistamista järjestelmällisempään suuntaan. jQuery on mahdollistanut monenlaisia tyylejä interaktiivisuuden toteuttamiseen. jQuery itsessään on vain kirjasto, joka keskittyy yksinkertaistamaan käyttäjän interaktioihin reagointiin liittyviä toimintoja, kuten selaintapahtumien hallintaa ja muutosten kuvantamista DOM-rakenteeseen eli käyttäjän näkyville [37]. jQuery ei tuo komponenttipohjaista rakenteellisuutta ja modulaarisuutta käyttöliittymän yksittäisille toiminnoille, ja näkymän elementtien välille voi muodostua monimutkaisia riippuvuuksia. Komponenttipohjaisella ohjelmistokehyksellä käyttöliittymäkomponentin esitysmuoto ja sen toiminnallisuudet säilyvät kapseloituna selaimessakin, jolloin kokonaisuuden hallintaan saadaan kerroksia.

Alaluvun 4.1 esittelemät JSON-dataa käsittelevät rajapintametodit toteuttavat liiketoimintadatan operoinnin lisäksi käyttöliittymään liittyviä toimintoja, kuten käyttäjän syötteen validointia. Rajapintametodit käyttävät kuitenkin sovelluksen Model-osuudesta samoja toimintoja kuin järjestelmäintegraatioille rakennettu XML-rajapinta. Järjestelmäintegraatioille on määritetty selkeä rajapintasopimus, joka mahdollistaa rajatummat tarpeet tiedonkäsittelylle. Käyttöliittymän tuottamaan dataan ei voida luottaa vastaavasti, kun datan validointi sekä käyttäjälle kommunikointi tapahtuvat ajonaikaisesti. Modulaarisuuden puute käyttöliittymän datan hallinnassa on johtanut ydintoiminnallisuuden toisteisuuteen, koska käyttöliittymälle on toteutettu kokonaan erilliset versiot rajapintametodeista.



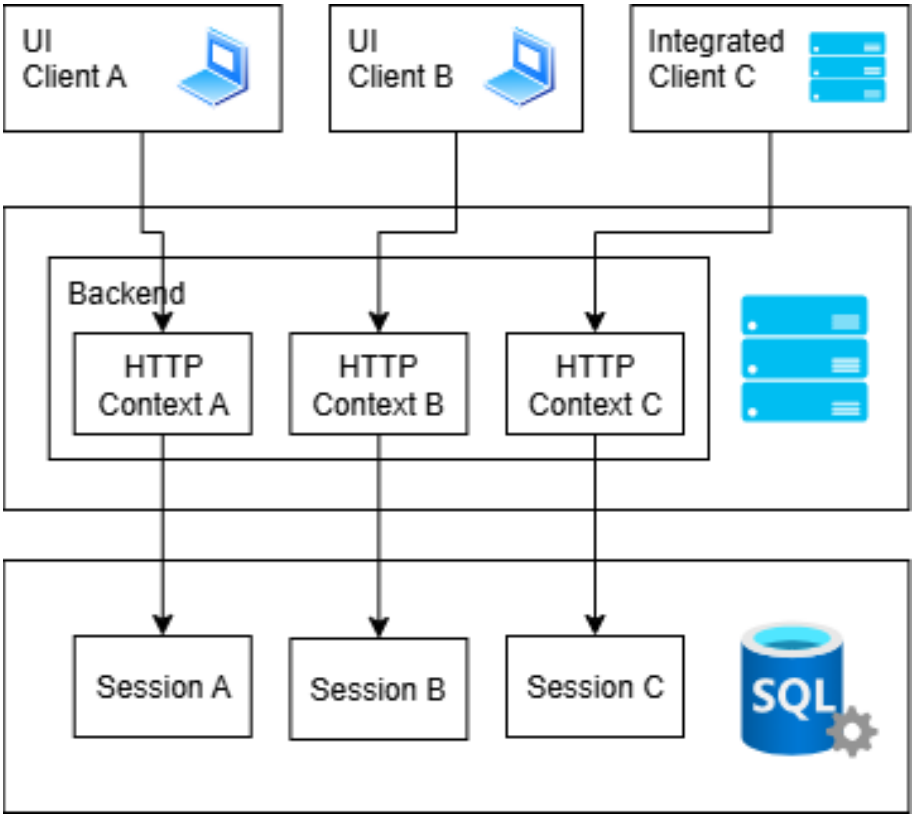

**Kuva 4.2:** ERA-järjestelmän käyttöliitymän kannalta merkittävät resurssit havainnollistettuna

Tämä on mahdollistanut teknisen velan muodostumista, kun liiketoimintalogiikan muutokset täytyy toteuttaa kahteen paikkaan erikseen. Teknologian vaihtaminen uudelle käyttöliittymäpohjalle ja järjestelmärajapintojen päälle mahdollistaa teknisen velkaantumisen pysäyttämisen ja hiljattaisen takaisinmaksun ilman, että koko velkaa täytyy maksaa kerralla.

ERAan rakennettu FHIR-palvelu käyttää samaa tietokantaa muun sovelluksen kanssa, mutta käsittelee kliinistä dataa eri muodossa ja kontekstissa. FHIR-palvelu on toteutettu JSON-formaattia tukevaksi, joten palvelun liiketoimintadata on valmiiksi käyttöliittymälle suotuisassa muodossa. Käyttöliittymäsovelluksen tehtäväksi jää FHIR-resurssien yhdistely käyttötapauksen vaatimaksi kokonaisuudeksi, sillä FHIR-tietomallit sisältävät paljon viitteitä toisiinsa kompositiotyyppisen mallintamistavan vuoksi [21]. Yksittäisiä FHIR-resursseja voivat olla esimerkiksi potilaan perustietoja tallentava ”Patient” ja yksittäisen mittaustuloksen tallentava ”Observation”.

Käyttöliittymän tilanhallintaan liittyvien vaatimusten suhteen ERA tarjoaa suotuisat lähtökohdat rinnakkaisen käyttöliittymän rakentamiselle, sillä istuntotietojen ja väliaikaisen liiketoimintadatan keskitetty tallennus mahdollistaa tilan saavutettavuuden kutsujasta riippumatta. Monen sivun sovelluksena ERAssa näkymien

välille ei ole päässyt muodostumaan monimutkaisia tilanhallinnan riippuvuussuhteita, sillä käyttäjäkohtainen data kootaan näkymämalleihin jo palvelimella. Täten jokainen näkymä säilyttää vain itseensä liittyvää tilallista dataa käyttäjän selaimessa. Kuva 4.3 havainnollistaa multi-tenant -tyyppistä palvelinprosessia, jossa istunnon tilaa hallitaan pyyntöjen yli säilyvän kontekstidatan avulla.




**Kuva 4.3:** ERAn multi-tenant luonteinen käyttäjäkohtainen tilanhallinta havainnollistettuna infrastruktuurin kannalta.

Keskitetty tilanhallinta tukee uuden käyttöliittymäteknologian yhdistämistä vanhaan sovellukseen, sillä sovellusten välisissä siirtymissä ei tarvitse välittää liiketoimintalogiikan kannalta keskeisiä tilallisia tietoja. Kuvan 4.3 esittämään tilanhallintamalliin voi siis tehdä viittauksia samaan tietokantaistuntoon ja järjestelmätietoihin erillisiltä käyttöliittymäsovelluksilta, jolloin konteksti säilyy siirtymien välillä.

# 5 Tutkimusmenetelmät

Tutkimuksessa on ratkaistavana kaksi päätöksenteon kannalta kriittistä pohjakysymystä. Ensimmäinen niistä on uuden käyttöliittymäteknologian valinta, ja toinen on toimivan arkkitehtuuriratkaisun löytäminen. Tietynlaiset arkkitehtuuriratkaisut saattavat sulkea pois joitakin teknologiavaihtoehtoja, ja jotkin teknologiat saattavat sopia toisia paremmin eri arkkitehtuureille. Tämä riippuvuussuhde huomioidaan arkkitehtuurivaihtoehtoja vertaillessa, ja teknologia-arviointiin kirjataan erityisiä huomioita vaikutuksesta arkkitehtuuritulosten tuloksiin.

Siirtymän mahdollistavan arkkitehtuurin suunnittelua varten täytyy ymmärtää vanhan järjestelmän rakenne ja ohjelmistossa käytettyjen teknologioiden perusteet. Osa tästä työstä sisältää siis ASP.NET MVC -ohjelmistokehykseen ja Razor Views -teknologiaan tutustumista. Lisäksi nykyisen järjestelmän toimintaan syvennytään ERA-tiimiläisen perehdyttämänä sekä itsenäisellä dokumentaation, koodipohjan ja käyttöliittymän tutkinnalla. Tässä diplomityössä ERA:n käyttöliittymästä ei kerrota yksityiskohtaisesti, vaan keskitytään käytettyjen teknologioiden yleisiin piirteisiin ja vain pintapuolisesti järjestelmän arkkitehtuuriratkaisuihin. Yksityiskohtien paljastamista vältetään, koska tuote on kaupallinen ja yksityisen yrityksen kehittämä. Lisäksi tuote käsittelee tieturvakriittistä dataa, joten mahdollista hyökkäyspinta-alaa ei tule paljastaa julkisesti.

## 5.1 Kirjallisuuskatsaus teknologioiden soveltuvuudesta tarpeisiin

Teknologiavaihtoehtojen vertailuun määritellään tutkimustarpeiden ja tavoitteiden pohjalta kriteerit, joihin teknologioiden vertailussa halutaan kiinnittää huomiota. Sopivia teknologiavaihtoehtoja kartoitetaan, ja samalla perehdytään niissä esiintyviin konsepteihin. Kirjallisuuskatsauksilla ja teknologiadokumentaatioita tutkimalla hankitaan teknologiavaihtoehdoista yhtenäinen kokonaiskuva, jonka perusteella voidaan paremmin valita potentiaalisia teknologioita tarkempaan vertailuun. Luvussa 6 analysoidaan käytettäväksi harkittuja ohjelmistokehyksiä tai teknologioita yksitellen eri kriteereihin peilaten. Kaikkia kriteereitä ei analysoida jokaiselle tutkimukseen valitulle teknologialle, vaan niiden muista erottavia puolia selvitetään niin kauan, kunnes perusteet teknologian hylkäämiselle tai valinnalle ovat selkeät. Teknologioita karsitaan selvityksen edetessä, ja syvempää tutkimusta jatketaan edelleen harkituille teknologioille.

Teknologia on tulossa suurikokoisen ja tietoturvakriittisen järjestelmän käyttöliittymän tukipilariksi. Tästä huolimatta teknologiaselvityksessä halutaan huomioida myös moderneja uraauurtavia vaihtoehtoja tunnettujen yrityskäytössä suosiota saavuttaneiden teknologioiden, kuten Reactin ja Angularin, lisäksi. Mikäli nuoremman kypsyysasteen teknologian potentiaali nähdään suurena ja tulevaisuuden riskit kohtuullisen pieninä, voi vähemmän suosittu teknologia toimia valikoitua työkaluksi ERAn uuden käyttöliittymäosuuden toteutuksessa. Teknologiaver-

tailun päätavoite on luoda konkreettisia perusteluita teknologiavalinnan tueksi.

Seuraavissa alaluvuissa määritellään tutkimuksen tarpeista johdetut arviointikriteerit, joiden perusteella yksittäisiä teknologioita arvioidaan.

### 5.1.1 Suorityskyky

Suorituskyvyn osa-alueet voidaan jakaa ylimmällä tasolla ajalliseen tehokkuuteen ja resurssien käytön tehokkuuteen. Ajallisesti voidaan vertailla esimerkiksi sivun ensimmäisen avaamisen nopeutta tai käyttöliittymän vasteaikaa. Resurssien kannalta tehokkuutta voidaan mitata muistin käytön, prosessointikuorman ja vaaditun verkkokommunikaation määrän perusteella. Prosessointikuorman sekä muistinkäytön suhteen huomioidaan sekä selaimen että palvelimen puoli. Monesti kuorma saattaa painottua toiseen puoleen riippuen arkkitehtuurista. Selaimen puolella renderöidyissä yhden sivun sovelluksissa taakka keskittyy käyttäjän puolelle, kun taas palvelinpuolen renderöinti kuormittaa enemmän palvelinta. Yleisesti tärkeitä toimintoja tehokkuuden kannalta ovat koodin pilkkominen, laiska lataaminen ja kuvantamisen optimointi. [46]

### 5.1.2 Ekosysteemi ja työkalut

Kehityksen tehokkuuden kannalta tärkeä ominaisuus on valmiiden kirjastokomponenttien ja -toiminnallisuuksien sekä aputyökalujen saatavuus, jolloin kehitystyössä voidaan keskittyä enemmän liiketoimintalogiikan erityisvaatimuksiin eikä yleisesti käytettyjen komponenttien uudelleenkehittämiseen tai matalan tason konfigurointiin. Tärkeää olisi löytää stabiileja avoimen lähdekoodin toteutuksia, jolloin vältyttäisiin käyttömaksuilta ja lisenssivaatimuksilta. Erityisen tärkeä valmiskomponenttityyppi on päivämäärävalitsin ja muut intuitiiviset tiedonsyöttömekanismit. Suoraan kehitettävässä sovelluksessa käytettyjen kirjastojen ja komponenttien lisäksi tärkeitä apuvälineitä ovat kehitystä helpottavat apuohjelmat, kuten modernit kehitystyökalut, joista esimerkkinä toimivat Vite ja Angular CLI [3], [82]. Ekosysteemin sisältämä uusiokäytettävä kirjastotarjonta on ajan säästön kannalta avainasemassa nopeassa mukautuvuudessa markkinatarpeisiin [56].

### 5.1.3 Stabiilisuus ja suosio

ERAn kehitystä halutaan jatkaa vielä vuosikymmeniä eteenpäin, joten uuden teknologian on täytynyt saavuttaa niin vakaa asema ohjelmistoteollisuudessa, että sen tuen ja päivitysten voidaan luottaa jatkuvan pitkällä aikavälillä. Stabiilisuutta ja suosiota mittaavat kriteerit jaetaan teknologian vakaaseen kehittymiseen ja suosioon liittyviin alakriteereihin. Suosiota mitataan kyselytulosten avulla, ja sen oletetaan vaikuttavan välillisesti teknologian tulevaisuuden näkymiin. Mittareina suosiolle käytetään tuloksia State of JS- ja StackOverflow -kehittäjäkyselyistä [26], [70]. Avoimen lähdekoodin yhteisön kasvaessa teknologian kehittymiselle potentiaali kasvaa ohella, ja teknologiaa hyödyntävien merkittävien toimijoiden riippuvuus teknologiasta lisää painetta teknologian säilyvyydelle.

Suosiota konkreettisemmin stabiilisuutta arvioidaan teknologian kehityksestä vastuussa olevan tahon ja sen johtaman kehitysprosessin osalta. Suuren

yrityksen taloudellinen tuki arvioidaan takeeksi teknologian jatkuvuudelle, ja systemaattinen julkaisu- sekä palauteprosessi auttavat ennustamaan teknologian kehittymistä sekä arvioimaan muutostarpeita ja mahdollisuuksia teknologian käytössä. Systemaattisessa kehitysprosessissa on tärkeää ennustettava tuki ja ylläpito käytettävälle teknologialle. Pitkän aikavälin tukea saavia teknologia-versioita merkitään kirjainyhdistelmällä LTS (engl. Long-Term Support), mikä nähdään merkittävänä hyötynä teknologian kannalta.

Työkalujen versioiden päivitykset voivat myös aiheuttaa vaikeuksia suuremman kokoluokan sovellusten ylläpidossa, mikäli vanhoja toimintoja rikkovia päivityksiä julkaistaan. Uuteen versioon päivittäminen on aina investointi, jolloin uusien toimintojen kehittäminen ja vikojen korjaaminen saa vähemmän huomiota. Teknologian muutoshistoriaa tarkastelemalla pyritään selvittämään rikkovien muutosten kokoluokan ja taajuuden osalta sekä järjestelmällinen semanttisen versionnin käyttö. Lisäksi arvioidaan teknologian tarjoamaa tukea muutosten toteuttamiselle ohjeiden ja valmiiden muutostyökalujen osalta.

### 5.1.4 Kehityksen mukavuus vs. oppimisen vaikeus

Teknologian tai ohjelmistokehyksen toimintamallien intuitiivisuus ja kehityksen mielekkyys ovat tärkeitä vertailukohteita rekrytoinnin helpottumisen näkökulmasta. Näihin vaikuttavat erityisesti liiketoimintalogiikan kannalta epäoleellisen koodin tarve, kielen syntaksin helppolukuisuus, staattinen vs. dynaaminen tyypitys ja mentaalimallin selkeys. Tässä projektissa voidaan myös huomioida ERA:ssa käytettyjen työkalujen vaatiman osaamisen hyödyllisyyttä uuden teknologian oppimisessa. Käyttöliittymäkehittäjien on pystyttävä hallitsemaan uusi teknologia nykyisen rinnalla, joten samankaltaisuus ASP.NET MVC:n kanssa voi olla hyödyksi.

### 5.1.5 Hyvien koodauskäytäntöjen tuki ja rakenteet

Tämä kriteeri arvioi teknologian taipumusta ohjaamaan kehittäjää käyttämään vakiintuneita ohjelmiston laatua tukevia suunnittelumalleja ja rakenteita, kuten MVC-arkkitehtuuria tai SOLID-periaatteita. Mikäli ohjelmistokehys tarjoaa vankan rakenteen ohjelmalle, jota kehittäjän on noudatettava, vähenee tarve omille konfiguraatioille, ohjelmistosuunnittelun tyylioppaille ja koodin laadun tarkkailemiselle huonojen arkkitehtuuriratkaisujen välttämiseksi. Esimerkiksi Angular sisältää runsaasti rakenteita käyttöliittymäsovelluksen rungoksi. React taas antaa enemmän vapauksia, mutta vaatii myös enemmän asiantuntemusta huonojen suunnitteluratkaisujen välttämiseksi.

Jotta saavutettaisiin kehitystyön tehostumisen motiiveja, tarvitaan teknologia, jolle löytyy hyvän dokumentaation lisäksi paljon käyttäjiä, jotka yhteisöfoorumeilla osaavat auttaa kehitystyössä esiintyvissä ongelmissa. Harvinaisille teknologioille voi olla vaikeaa saada tukea kokeneilta teknologian käyttäjiltä, ja kehittäjä joutuu turvautumaan viralliseen dokumentaatioon tai jopa maksulliseen tuki-palveluun.

### 5.1.6 Muuta huomioitavaa

Vain teknologiaa tai ohjelmistokehystä itseään koskevien attribuuttien lisäksi on huomioitava liittyminen nykyiseen ASP.NET MVC -sovellukseen ja sopeutuvuus ERAn tarpeisiin. Monet SPA-ohjelmistokehykset vaativat esimerkiksi REST-rajapinnan kytkeytyäkseen ERAan. Osa ratkaisuista ei kuitenkaan rajoitu tähän. Mahdollisuuksia rakentaa ja renderöidä uudella teknologialla näkymiä jo nykyisen palvelinohjelmiston sisällä halutaan tutkia. Tällaisten ratkaisujen hyviä ja huonoja puolia tulee myös vertailla. Erillinen uusi rajapinta ja selaimessa kuvannettava uusi käyttöliittymä kuulostaa alkuun työläämmältä ratkaisulta, mutta saattaa johtaa selkeämpään arkkitehtuuriin pitkällä aikavälillä, kun uusi ja vanha on selkeästi eroteltuna.

Koska ERA keskittyy terveyden ja sosiaalihuollon työntekijöiden palvelemiseen, pystytään tähän käyttötarkoitukseen kohdistamaan erityistä huomiota teknologioita arvioitaessa. ERA käsittelee ja tallentaa lukuisia erilaisia asiakirjamaisia tietoja, joten monessa näkymässä tarvitaan käyttäjältä syötteitä, kuten päivämääriä, lukuja, koodistoarvoja ja vapaata tekstiä. Käyttöliittymän täytyy validoida ja käsitellä nämä syötteet tehokkaasti. Syötekomponenttien tulee olla myös käyttäjäystävällisiä, mikä on hyvä huomioida tätä kriteeriä arvioidessa. Esimerkkinä reseptien kirjaus on paljon käytetty lomakkeiden täyttöä vaativa toiminnallisuus käyttöliittymässä. Reseptiä kirjaavan lääkärin on asiakkaan läsnä ollessa täytettävä asiakkaan ja lääkemääräyksen tiedot käyttöliittymällä. Mahdollisuus toteuttaa tällaiset toiminnot tehokkaasti ja intuitiivisesti on erityisen tärkeä. Keskeisiä toimintoja lomakkeiden käsittelyssä ovat syötteen vastaanottaminen ja validointi sekä lomakkeen tietomallin päivitys ja sen tilan muutosten seuranta [3]. Syötteen vastaanottamiseen tärkeitä komponentteja ovat esimerkiksi tekstikentät, päivämääränvalintakalenterit ja pudotusvalikot. Kehittyneempänä toimintona esimerkiksi vedä-ja-pudota -tuki olisi hyödyllinen paremman käyttäjäkokemuksen luomiseen, ja potilas- tai asiakashistorian esittäminen hyötyisi edistyneistä visualisointityökaluista.

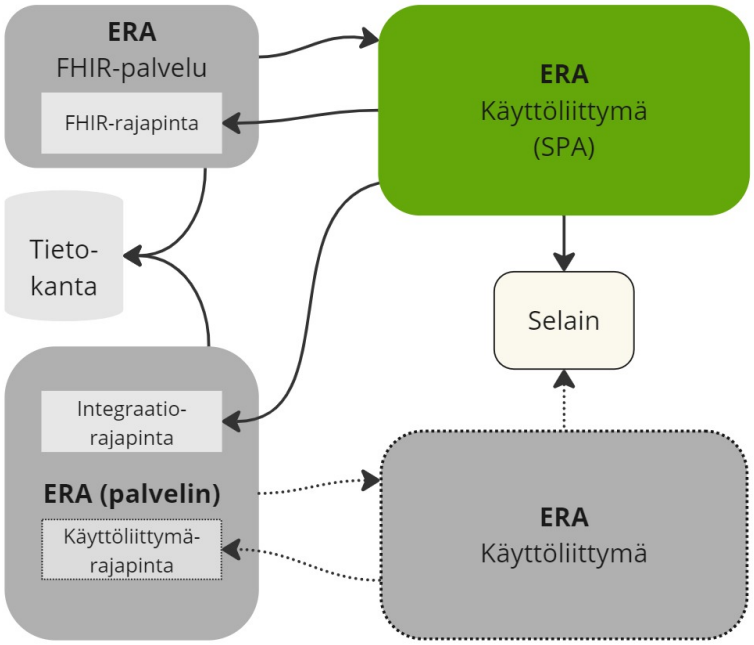
## 5.2 Arkkitehtuurin suunnittelu ja analysointi

Arkkitehtuuriratkaisussa tärkeää on nykyiseen järjestelmään sopeutuvuus sekä ratkaisun tuomat vapaudet ja rajaukset. Ensin tutkimuksessa hahmotellaan arkkitehtuurivaihtoehtoja, joita teknologiaselvityksen yhteydessä on havaittu mahdollisiksi. Tämän jälkeen analysoidaan kutakin arkkitehtuurivaihtoehtoa ja tuodaan esiin yhtäläisyyksiä ja eroavaisuuksia. Kustakin arkkitehtuurivaihtoehdosta analysoidaan niiden hyviä ja huonoja puolia sekä peilataan niitä kappaleessa 2 mainittuihin tarpeisiin. Lisäksi täytyy vielä suhteuttaa kunkin arkkitehtuuriratkaisun hyödyllisyys työmäärään, jolloin arkkitehtuureita voidaan vertailla absoluuttisesta näkökulmasta.

Teknologiavaihtoehdot huomioidaan arkkitehtuuripäätöksessä selvittämällä, mitkä teknologiat pystyvät toteuttamaan kunkin arkkitehtuurin ja kuinka hyvin mahdollinen arkkitehtuuri soveltuu teknologian tai ohjelmistokehyksen periaatteisiin ja tarkoituksiin. Arkkitehtuureista identifioidaan teknologioista riippumattomat osuudet, ja keskitytään teknologioiden rajoitteisiin arkkitehtuuriratkaisujen

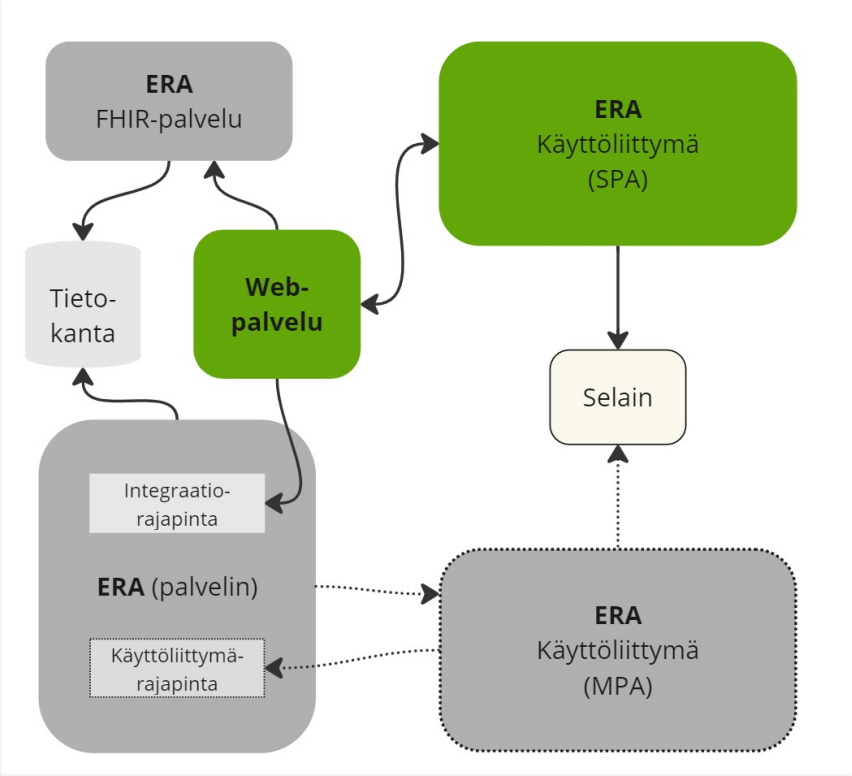
suhteen. Mikäli teknologiavertailussa parhaimmaksi arvioitu teknologia ei sovellu parhaaseen arkkitehtuurivaihtoehtoon, saattaa teknologiaksi valikoitua jokin vähemmän hyödyllisistä teknologioista. Sama voi tapahtua myös toisinpäin, joten arkkitehtuuriratkaisu ja teknologiavertailu ovat vahvasti sidoksissa toisiinsa. Täten lopullisen päätöksen tuottamiseksi täytyy mahdollisesti vielä punnita teknologiavalinnan ja arkkitehtuurivalinnan painoarvoja keskenään.

Arkkitehtuurin suhteen on havaittu kolme karkean tason lähtökohtaa kehitykselle. Liikkeelle lähdetään yksinkertaisesta ratkaisusta, jossa uusi käyttöliittymä pysyy teknisesti täysin erillisenä kokonaisuutenaan, mutta liittyy näkymien uudelleenohjauksilla ja nykyisillä järjestelmäintegraatiorajapinnoilla vanhaan järjestelmään. Tämän vaihtoehdon periaatteita havainnollistaa karkea arkkitehtuurikuva 5.1. Arkkitehtuuri noudattaa yhden sivun sovelluksen periaatteita vanhan monisivuisen sovelluksen rinnalla.




**Kuva 5.1: Vaihtoehto A:** Uusi yhden sivun sovellus nykyisillä integraatiorajapinnalla

Toisen arkkitehtuurihahmotelman käyttöliittymä toimii myös yhden sivun sovelluksena, eikä se poissulje yksinkertaisinta A-arkkitehtuurivaihtoehtoa. Arkkitehtuurissa käyttöliittymän ja dataintegraatiokerroksen välille tulee uusi sovelluskerros, jonka päätavoitteena on yksinkertaistaa käyttöliittymän kehitystä sekä modularisoida tiedon käsittelyä ja esittämistä. Kuvassa 5.2 nähdään erona A-vaihtoehtoon uusi käyttöliittymän tueksi rakennettu palvelinpuolen ohjelma. Kolmas harkittu vaihtoehto C on saanut inspiraatiota ReactJS.NET-teknologiasta, jossa React-komponentteja voidaan upottaa suoraan palvelimella rakennettuihin näkymäpohjiin [61]. Arkkitehtuurimallia analysoidaan kuitenkin teknologiariippumattomasta näkökulmasta, jossa vanha käyttöliittymä korvataan sisältä päin vanhaa muokkaamalla. Tämä eroaa vaihtoehdoista A ja B, joissa prosessi vaatii suurempien kokonaisuuksien siirtämistä, eikä niinkään vanhojen muok-



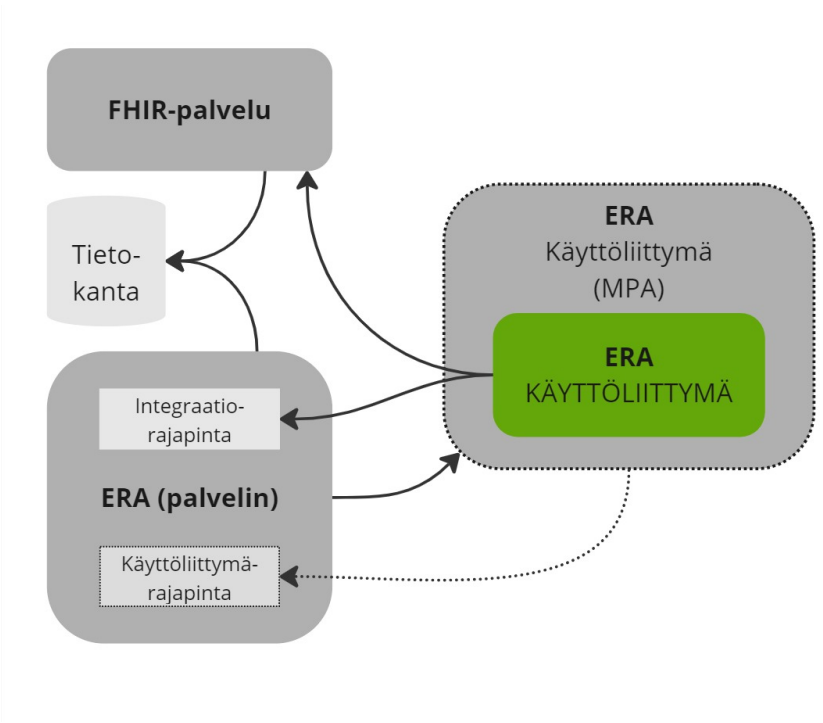

**Kuva 5.2: Vaihtoehto B:** Yhden sivun sovellus uudella palvelinpuolen rajapintakerroksella

kaamista. Käyttöliittymän arkkitehtuuri säilyy monen sivun sovelluksena, kun vanhat näkymät toimivat uuden koodin pohjana. Kuva 5.3 havainnollistaa uuden käyttöliittymän rakentumista vanhan sisälle.

Kuvien arkkitehtuurimallit kuvaavat tilannetta, jossa siirtymä uuteen teknologiaan on kesken. Lopullinen pitkän aikavälin tavoite on päästä vanhasta käyttöliittymäteknologiasta eroon kokonaan, joten siirtymän aikaiset osat on kaavioissa merkitty katkoviivalla. ERA-järjestelmässä ennalta olevat osuudet on värikoodattu harmaan sävyillä, ja uudet osuudet ovat vihreitä. Nuolten merkitys kuvissa on tiedon kulkureittien mallintaminen. Nuolten suunnan määrittää kommunikaatioväylän aktiivinen osapuoli, jota havainnollistaa lähtevä nuoli. Käyttöliittymiin kohdistuvat nuolet kuvaavat käyttöliittymän ohjelmakoodin tai valmiiden näkymädokumenttien lähettämistä. Kaavioiden ”FHIR-palvelu” -komponentti on ERAn uusi .NET 8-teknologialla kehitetty FHIR-standardin mukainen integraatiopalvelu, jonka käsittelemä data on pääosin erillistä vanhemman ERA-sovelluksen tietorakenteista. Arkkitehtuurivaihtoehtoja analysoidaan tarkemmin kappaleessa 7. Arkkitehtuuriratkaisut teknologian käyttöönottoon muistuttavat palvelinohjelmistoje Strangler-patternia arkkitehtuurin tuomana mahdollisuutena [49].

## 5.3 Muutoksen kannattavuuden arviointi

Yksi mahdollinen hyväksyttävä tutkimustulos on myös vanhassa teknologiapinossa pitäytyminen. Mikäli uuden teknologian käyttöönotto ja arkkitehtuurin rakentaminen nähdään kustannuksiin ja vaivaan nähden kannattamattomak-




**Kuva 5.3: Vaihtoehto C:** Vanhan käyttöliittymän sisään rakentuva arkkitehtuurimalli.

si, voidaan tutkimustuloksena perustella syyt kyseiseen päätökseen. Tätä päätöstä varten arvioidaan mahdollisuudet nykyisen käyttöliittymäkehityksen parantamiselle hyödyntäen uusia ohjelmoinnin tyyliohjeita ja mahdollisesti komponenttikirjastoja. Oletuksena tutkimuksen konkreettisena tuloksena on ratkaisu, jota lähdetään kokeilemaan tutkimuksen jälkeisessä PoC-vaiheessa.

# 6 Käyttöliittymäteknologioiden arviointi

Seuraavissa luvuissa arvioidaan mahdollisten teknologiavaihtoehtojen kykyä vastaamaan ERA-järjestelmän ja sen kehityksen tarpeisiin. Soveltuvuutta tutkitaan osiossa 5 tarkennettujen kriteerien avulla. Tavoitteena on tutkia mahdollisimman monipuolisesti ja ennakkoluulottomasti sekä vakiintuneita suosittuja teknologioita, että paljon potentiaalia sisältäviä uudempia teknologiasuuntauksia.

Teknologiavertailu sisältää JavaScript-teknologioista jäsennellyn arvioinnin Angularille, Reactille, Vuelle ja Sveltelle. Tasapuolisen vertailun vuoksi arvioinnissa huomioidaan Reactin metakehys Next.js, Vuen metakehys Nuxt.js ja Svelten metakehys SvelteKit. JavaScript teknologioista erottuvana tutkitaan Blazor-teknologiaa, joka hyödyntää palvelinpuolelta tuttua C# -kieltä ja .NET-ekosysteemiä. Teknologia-arvioinnissa käsitellään lyhyesti myös vähemmän suosittuja teknologioita, jotka hylättiin yksityiskohtaisesta arvioinnista tutkimuksen paisumisen välttämiseksi. Tiivistettyyn arviointiin kuuluvat HTMX, ReactJS.NET, Modern.js ja Elm.

## 6.1 Angular

Angular on Googlen ylläpitämä avoimen lähdekoodin projekti, joka on vapaasti käytettävissä MIT-lisenssillä.[2], [3], Angular on muihin tutkimuksessa vertailtuihin käyttöliittymäkehyksiin verrattuna kokonaisvaltaisempi työkalu, sillä se sisältää ominaisuuksia, jotka ovat muiden teknologioiden erillisten meta-kehysten ominaisuuksia. Näistä esimerkkinä toimii mahdollisuus kuvantaa näkymiä palvelinpuolella Angular Universal-toiminnolla [3].

### 6.1.1 Suorityskyky

Angular tukee palvelinpuolen kuvantamista natiivisti, mikä mahdollistaa staattisten sivujen nopean latautumisen. Angular pyrkii myös optimoimaan DOM-päivitysten tarvetta muutoksentarkastelualgoritmillaan, joka hyödyntää Zone.js -kirjastoa. Angular kertoo optimiointinsa olevan tehokasta, mutta muutoksen tarkastelu voi aiheuttaa suorituskuormaa. Ylimääräisiä muutostarkastuksia voi Angularin ohjeiden mukaan välttää esimerkiksi määrittelemällä erikseen funktioita tai komponenttien alapuita muutostarkasteluvapaaksi [3].

Suuri määrä ominaisuuksia ja abstraktioita tuo kuitenkin haittapuolena kookkaan JavaScript-ohjelman, jonka käyttäjän selain joutuu lataamaan ennen sivun esittämistä. Angularia ja Vue-teknologiaa blogijulkaisussaan [16] vertaileva Dhaduk mainitsee keskikokoisen Angular-sovelluksen kooksi keskimäärin yli 250 ja alle 500 kilotavua. Tämä on merkittävästi suurempi kuin esimerkiksi Vuella rakennettujen sovellusten keskimääräinen koko 50:stä 100:aan kilotavuun. [16] Ensilatauksen hitaus voi olla haitaksi etenkin tilanteessa, jossa Angularilla kehitetty näkymä haluttaisiin näkyviin asiakkaan verkkosovelluksessa, joka on kehitetty eri teknolo-

gialla. Tällainen tilanne voi syntyä luvussa 7 esitetyssä Micro-frontend -tyylisessä arkkitehtuuriratkaisussa. Tässä tapauksessa kahden eri teknologian kirjastokoodi tulisi ladatuksi käyttäjän selaimeen, mikä tekisi näkymästä mahdollisesti raskaan, riippuen sovellusten kokoluokasta.

### 6.1.2 Stabiilisuus ja suosio

Angular on vakiinnuttanut asemansa ohjelmistokehittäjien yhteisössä erityisesti ammattikäytössä. Vuoden 2024 Stack Overflown kehittäjäkyselyssä noin viidennes ammattilaiskehittäjistä vastasi käyttävänsä Angularia merkittävissä määrin [70]. Myös State of JS 2024 -kyselyn tulokset kertovat Angularin olevan suosittu erityisesti suurten yritysten ammattikäytössä, sillä neljännes Angularin ammattikäyttäjistä työskentelee kyselyn mukaan yli tuhannen työntekijän yrityksissä [26].

Angularin myös kertoo panostavansa vakaaseen kehitysprosessiin, joka sisältää suunnitelmallisen vanhentuvien toiminnallisuuksien hallinnan, rikkovien muutosten minimoimisen ja Googlen omassa käytössä testattujen siirtymistyökalujen tuottamisen [3]. Googlen kaltaisen suuren korporaation tuki on merkittävä etu stabiilisuuden kannalta.

Angularin versiointi tuo myös systemaattisuutta sen jatkuvaan käyttöön. Angular toteuttaa semanttista versiointia, mikä helpottaa ohjelmistokehyksen päivittämisen riskien arviointia ja kehittymisen seurantaa [3]. Angular kertoo noudattavansa säännöllistä julkaisurytmiä ja tukevansa pitkän aikavälin tuen versioita 18:n kuukauden ajan, joten teknologian päivittämisen ennakoiva suunnittelu on mahdollista. Angularin kehittäjätiimi kerää myös kehittäjiltä palautetta tuleviin muutossuunnitelmiinsa Request For Comments (RFC) -prosessillaan, mikä auttaa ohjaamaan teknologian kehitystä turvalliseen suuntaan [3].

### 6.1.3 Ekosysteemi ja työkalut

Angular sisältää runsaasti natiivia kirjastotyylistä toiminnallisuutta, kuten animaatiojärjestelmän, joka on rakennettu CSS-toimintojen päälle. Tämä auttaa paremman käyttäjäkokemuksen kehittämisessä, mikä on hyödyllistä kaupallisessa tuotteessa, kuten ERA. [3] Angular Material on Angularin virallinen laaja komponenttikirjasto, joka tarjoaa edistyneitä valmiita komponentteja ja toiminnallisuuksia, kuten päivämääränvalitsin ja ‘vedä-ja-pudota‘-toiminto (engl. Drag and Drop). Angular Material tukee myös teemoja Material Design 3 -spesifikaation mukaisesti. [1], [41] Angular sisältää komentorivityökalun, joka helpottaa web-sovelluksen alustusta, testaamista, kehittämistä ja julkaisemista [3].

Angularin omien kirjastojen ja työkalujen lisäksi Angularille on suosionsa ansiosta kehittynyt laaja kolmannen osapuolen kirjastojen ekosysteemi. Angular on jopa vahvasti kehittynyt riippuvaiseksi RxJS- ja ZoneJS-kirjastoista, joista ensiksi mainittu luo perustan Angularin reaktiivisuustoiminnoille ja toinen toimii käyttöliittymän muutostentarkastelun toteuttajana [3], [64]

Angular ei tue kehitystä helpottavaa HMR-tekniikkaa (Hot Module Replacement) JavaScript-muutosten saumattomaan reaaliaikaiseen päivittämiseen, kun kehitysversio on paikallisesti ajossa. Angularin kehityspalvelinta käytettäessä

vain tyylimuutokset ja templaattien muokkaukset voidaan ladata selaimeen HMR-tekniikkaa hyödyntäen. [3] Tämä kehityksen sujuvuutta parantava toiminnallisuus on yksi harvoista Angularin puutteista verrattuna muihin moderneihin ohjelmistokehyksiin.

### 6.1.4 Kehityksen mukavuus vs. oppimisen vaikeus

Angular on kehittäjien keskuudessa mielipiteitä jakava ohjelmistokehys tiukkoine rakenteineen, jotka saattavat tuntua ylimääräisen monimutkaisilta etenkin aloitteleville kehittäjille. Tämä näkyy vuosia kestäneessä matalassa sijoituksessa State of JS -kyselyn kiinnostavuus-, tyytyväisyys- ja positiivisuus-kategorioissa muihin suosittuihin verkkokäyttöliittymäkehyksiin verrattuna [26]. Vuonna 2024 tyytyväisyys ja positiiviset tuntemukset Angularia kohtaan ovat lähteneet kyselyn mukaan selkeään nousuun, mikä viittaa Angularin uudelleenkehittämän reaktiivisuusmallin suosioon kehittäjien keskuudessa [3]. Angularin Signals -toiminnot muistuttavat myöhemmin arvioitavan kehittäjien keskuudessa pidetyn Svelten reaktiivisuusmallia, jossa ytimekkäästi yhdellä avainsanalla voidaan määritellä muuttujalle muutoksiin automaattisesti reagoivia toimintoja [3], [73]. Tämä mahdollistaa Angulariin kompleksisuutta lisänneen kolmannen osapuolen riippuvuuden, RxJS-kirjaston, käytön välttämisen [64].

Vaikka aloittelijalle teknologian oppiminen voi tuntua monimutkaiselta, kehityksen rutinoituessa Angular ohjaa rakenteellisuudellaan ja suunnittelumalleillaan hyvälaatuisen koodin kirjoittamiseen, mikä tekee pitkällä aikavälillä kehittämisestä mielekkäämpää [28]. Kokeneemmalle tai taitavalle kehittäjälle oppiminen voi olla jopa helpompaa, jos Angularin toimintamallit ovat konsepteina tuttuja.

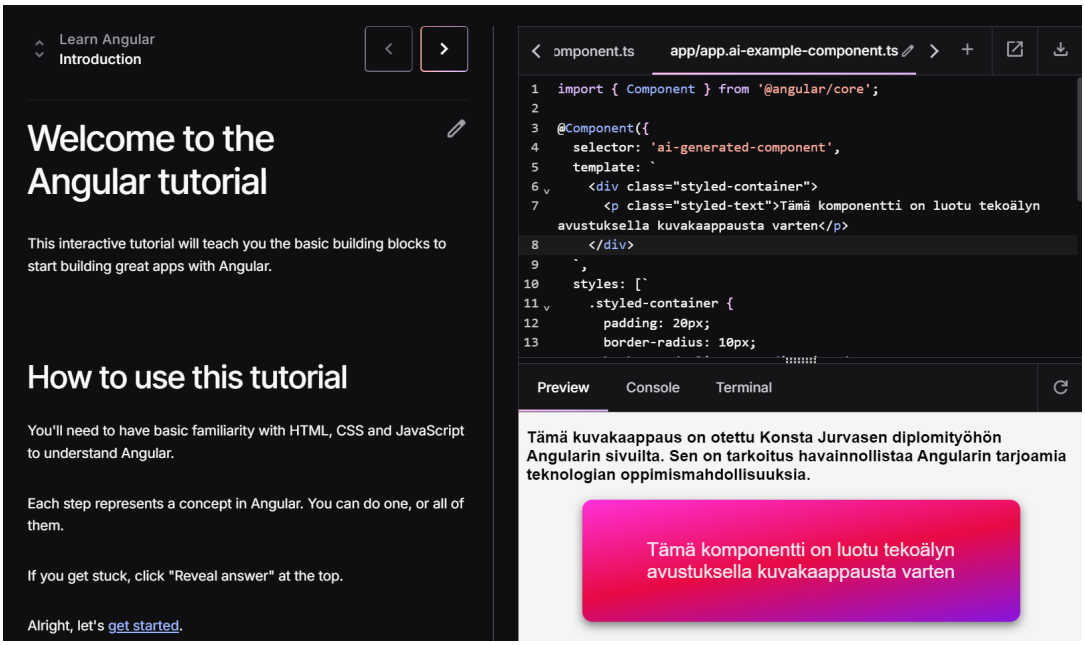
### 6.1.5 Hyvien koodauskäytäntöjen tuki ja rakenteet

Angularilla kehitetyissä sovelluksissa on käytettävä TypeScriptiä, joka pakottaa staattiseen tyypitykseen. Tämä voi auttaa luettavuudessa ja virheiden paikantamisessa, kun kehittäjälle on selkeämpää, minkä tyyppinen tietyn muuttujan kuuluisi olla, ja kääntäjä voi tunnistaa väärän tyypin käytön ennen koodin suoritusta.

Angular ohjelmistokehyksenä antaa ohjelmistolle vankat raamit ja pakottaa hyödyntämään hyväksi todettuja ohjelmointimalleja, kuten riippuvuuksien injektiota (engl. Dependency Injection). Joustavuuden puute voi jakaa mielipiteitä, mutta suunnittelumallien noudattaminen ja toiminnallisuuksien erottelu tekevät yleisesti koodista huollettavampaa [28] Hyvien koodauskäytäntöjen tukemiseksi Angular ylläpitää myös tyyliohjeita dokumentaatiosivuillaan, jotka ohjeistavat esimerkiksi säännöllisiin nimeämiskäytäntöihin ja projektin tiedostorakenteeseen sekä ohjeistavat ohjelmistokehyksen toimintojen optimaaliseen hyödyntämiseen.

Rakenteellisuuden lisäksi Angularille löytyy tukimateriaalia hyvien koodauskäytäntöjen varmistamiseksi ja teknologian opettelua helpottamaan. Angularin viralliset verkkosivut sisältävät runsaasti dokumentaatiota ja ohjeistusta ohjelmistokehyksen ominaisuuksien käyttöön. Dokumentaatio sisältää videoita ja tutoriaalityyppisiä ohjeita yksittäisille ominaisuuksille. Esimerkkejä ominaisuuksien käytöstä on myös annettu kattavasti. Virallisilta verkkosivuilta löytyy myös aloittelijan tutoriaali, jossa verkkosivuille on upotettu kehitysympäristö testailua varten.

Kattavat opiskelumateriaalit auttavat mitigoimaan Angularin huonoa puolta pidemmän oppimiskaaren omaavana ohjelmistokehyksenä. Kuvassa 6.1 havainnollistetaan tutoriaaliympäristöä. Angularin sivustolta löytyy myös yksityiskohtaiset tyyliohjeet koodauskäytäntöihin, jotka ovat suositeltuja, mutta eivät ohjelmistokehyksen pakottamia. Sivustoilla annetaan pääosin ohjeita, joista kannattaa poiketa vain hyvillä perusteluilla. [3] Angularin kaltaiselle suositulle teknologialle on myös helppo löytää kolmannen osapuolen apua esimerkiksi Youtubesta.




**Kuva 6.1:** Kuvakaappaus Angularin omasta tutoriaaliympäristöstä, jossa loin AI:n avustuksella Angularin väreihin tyylitellyn komponentin.

### 6.1.6 Muuta huomioitavaa

Lomakekäsittely on Angularissa mahdollista sekä reaktiivisesti että mallipohjaisesti (ks. Docs -> Forms). Mallipohjainen lomakekäsittely soveltuisi hyvin yksinkertaisiin lomakkeisiin, mutta reaktiiviset lomakkeet ovat skaalautuvampia, testattavampia ja uudelleenkäyttöä tukevia. [3]

## 6.2 React

React.js, eli lyhyemmin React, ei ole itsessään varsinainen ohjelmistokehys, vaan JavaScript-kirjasto, joka sisältää runsaasti toiminnallisuutta käyttöliittymien rakentamisen perustaksi. React jättää paljon suunnittelupäätöksiä käyttäjän vastuulle, mikä luo kehittämiselle sekä mahdollisuuksia että uhkia. Kokeneet kehittäjät voivat toteuttaa hyväksi toteamiaan suunnittelumalleja vapaasti, mutta aloittelija saattaa käyttää Reactia vastoin sen periaatteita ja luoda vaikeasti huollettavaa koodia. [59]

Tavallisia JavaScript-toimintoja suoritetaan käskymuotoisesti eli imperatiivisesti rivi kerrallaan. DOM-mallin kanssa täytyy kommunikoida vaihe vaiheelta sisällön

päivittämistä varten JavaScript-käskyillä, kuten `document.getElementById` ja `document.createTextNode`. React tekee vastaavien toiminnallisuuksien määrittelystä tehokkaampaa hyödyntämällä deklaratiivistä ohjelmointiperiaatetta, jossa näkymien sisältö määritellään HTML:ää mukailevalla ja komponenttien asettelua havainnollistavalla tyylillä. Tähän React hyödyntää JavaScript-syntaksin JSX-laajennosta, jonka avulla näkymä voidaan esittää JavaScriptin seassa `render`-metodin tai komponenttifunktion paluuarvona. [59]

React itsessään on teknisesti vain käyttöliittymien rakentamiseen keskittyvä kirjastokokonaisuus, joten teknologian arvioinnissa huomioidaan Reactin kokonaisvaltaisempi metakehys. Next.js, lyhyemmin Next, on Reactin päälle rakennettu korkeamman tason metakehys. Next.js konfiguroi ohjelman palvelurakenteen valmiiksi, jolloin kehittäjän ei tarvitse välittää Viten tai Webpackin kaltaisiin kasaustyökaluihin liittyvistä yksityiskohdista. Next.js:n olennaisiin ominaisuuksiin kuuluu myös reititys ja komponenttikohtaisen kuvantamisvastuun kontrollointi selaimen ja palvelimen välillä. Next.js tuo siis Reactin päälle rakennetta ja lisäominaisuuksia. Next ei kuitenkaan poissulje Reactin tarjoamia vapauksia, vaan sen tuomia toiminnallisuuksia ja optimoituja ratkaisuja voi ottaa käyttöön halutuissa määrin. [59]

### 6.2.1 Suorityskyky

Selaimen prosessointikuorman osalta React pyrkii tehostamaan suorituskykyä virtuaalisen DOM-rakenteensa avulla. React optimoi vaadittujen uudelleenrenderöintien määrää kutsumalla render-funktiota vain silloin, kun komponentin esittämiselle kriittinen data muuttuu. Hienojakoinen optimointi voi kuitenkin vaatia käyttäjältä omaakin työtä, sillä React renderöi automaattisesti komponentin kaikki jälkeläiset, eli sisemmät komponentit uudelleen, mikäli komponentin tila tai tilaan vaikuttavat parametrit muuttuvat. [58] Tämä ei ole aina tarpeellista, joten kehittäjän on täytynyt hyödyntää lisäominaisuuksia, kuten `useMemo` -*Hook*-toimintoa kertomaan Reactille mahdollisuudesta ohittaa uudelleenkuvantaminen tai funktioiden suorittaminen. Tähän on kuitenkin kehitetty korjaus, sillä React on julkaissut staattisesti koodia analysoivan kääntäjän, joka optimoi kuvantamista automaattisesti. Kääntäjä poistaa kehittäjän vastuuta kuvantamismäärän minimoimiseksi, ja pystyy myös tarkempaan optimointiin kuin manuaaliset *Hook*-toiminnot [39].

Reactin metakehys, Next.js, on rakentanut Reactin päälle valmiiksi konfiguroituja toimintoja, jotka tarjoavat kehittäjälle mahdollisuuksia optimoida käyttöliittymäsovelluksen suorituskykyä hienojakoisesti käännösprosessin, palvelinprosessin ja selainprosessin välillä. Next kuvantaa komponentit oletuksena palvelimen puolella, jolloin näkymän alkutila latautuu käyttäjän näytölle tavallisia SPA-sovelluksia nopeammin, parantaen sovelluksen käyttömukavuutta. Next tarjoaa kolme erilaista strategiaa palvelinpuolen kuvantamiseen, jotka ovat staattinen ja dynaaminen kuvantaminen sekä valmiiksi kuvannettujen komponenttien ”striimaus” (engl. streaming). Striimausta voi käyttää myös muun datan lataamisessa, mikä voi tehdä suuria datamääriä lataavan sivun käyttäjäkokemuksesta sujuvampaa. [50], [51]

### 6.2.2 Stabiilisuus ja suosio

Reactin suosio ja hallitseva markkina-asema ovat ehdottomia valttikortteja yrityskäytössä. React on käyttäjämäärien suhteen selkeästi suosituin web-käyttöliittymien kehitykseen käytetty teknologia, kun StackOverflown vuoden 2024 kehittäjäkyselyn mukaan sitä käytti jopa 41,6 prosenttia kyselyyn vastanneista ohjelmistoalan ammattilaisista. Kyselyssä seuraavaksi suosituimpia käyttöliittymäohjelmointiin tarkoitettuja ohjelmistokehyksiä ovat Angular ja Next.js, joilla molemmilla on käyttäjiä alle puolet Reactiin verrattuna. [70] State of JS 2024 -kysely paljastaa, että jopa 82 prosenttia vastaajista on joskus käyttänyt Reactia, ja 99 prosenttia vastaajista tietää teknologian [26]. Reactin suuren käyttäjämäärän ansiosta lukuisilla kehittäjillä on merkittäviä riippuvuuksia teknologiaan, mikä tukee avoimen lähdekoodin yhteisön tehokkuutta ja kiinnostusta teknologian kehittämiseen ja ylläpitoon. Täten Reactia voidaan pitää turvallisena valintana myös yrityskäyttöön.

Reactin kehitystä hallinnoi suuri teknologiayritys Meta, mikä tuo Reactin asemalle vakautta. Tätä havainnollistaa esimerkiksi Meta:n tapa tuoda React-kääntäjä markkinoille uutena ominaisuutena. Kääntäjä on vielä tämän tutkimuksen kirjoitushetkellä virallisesti beta-testausvaiheessa, mutta Meta on jo testannut sitä tuotantokäytössä omissa paljon käytetyissä sivustoissaan, kuten Instagramissa. [58] Metan julkaisuprosessi osoittaa merkkejä kypsän teknologian tavoista tuoda uusia ominaisuuksia tuotteeseensa tarkoin harkitusti.

Reactin kehityksessä on myös vakiintunut RFC-palautekäytäntö, jossa uusille ominaisuuksille voi tehdä pyyntöjä, mutta myös Meta:n ominaisuusehdotuksille voi antaa palautetta [60]. Tämä mahdollistaa yritysten ja yksittäisten kehittäjien mielipiteiden huomioimisen ja kehityssuunnan pysymisen käyttäjien mieleisenä, mikä myös osaltaan vahvistaa teknologian luotettavuutta.

#### Metakehyksen stabiilisuus

Stack Overflown kehittäjäkyselyn mukaan vuonna 2024 Next.js on merkittävässä käytössä jo useammalla kehittäjällä kuin pitkäjänteistä suosiota saavuttanut Angular. Tämä on positiivinen merkki teknologian jatkuvuutta ja tulevaisuuden näkymiä ajatellen, vaikka ammattilaisten keskuudessa kyselyn mukaan Angular on yhä vajaan prosenttiyksikön verran suositumpi vastausten määrään suhteutettuna. [70] Tätä eroavaisuutta ammattilaiskehittäjien ja harrastelijoiden välillä selittää Angularin erikoistuminen ammattikäyttöön sekä uusien teknologioiden käyttöönottokynnys, joka on pienempi yksittäisten kehittäjien omassa käytössä. Täten Nextin nopeasti kasvanut suosio on erittäin lupaavaa. Metakehysten joukossa Next.js on selkeästi suosituin State of JS 2025 -kyselyn mukaan 54 prosentin käyttöasteella [27].

Teknologian stabiilisuutta lisää sen kehityksen taustalla toimiva pilvipalveluyritys Vercel, joka tarjoaa myös natiivin tuen Next.js-käyttöliittymien julkaisuun ja on vastuussa Next.js-teknologian kehityksestä. [50] Next.js noudattaa Reactin ja Angularin tavoin systemaattista semanttista versiointia ja pääosin tasaista päivityssykliä, jossa merkittävät toiminnallisuuksia muuttavat päivitykset tapahtuvat noin vuoden välein. Pitkän aikavälin tukea saa aktiivisesti uusin merkittävä ver-

sio, ja ylläpidollisia päivityksiä luvataan kahdeksi vuodeksi päivitysten julkaisusta [51].

### 6.2.3 Ekosysteemi ja työkalut

React-teknologian sallimat vapaudet ja suuri suosio ovat mahdollistaneet monipuolisen ekosysteemin kehittymisen. Reactin päälle on rakennettu paljon kolmannen osapuolen työkaluja ja ratkaisuja, jotka tuovat ohjelmistokehysten ominaisuuksia saataville. Tällaisista hyviä esimerkkejä ovat tilanhallintatyökalut, kuten Redux, Recoil ja Jotai, tai Reactin päälle rakennettu kokonainen meta-ohjelmistokehys Next.js, jota analysoidaan myöhemmin erikseen.

Uuden sovelluksen konfigurointiin ja kehityksen aloittamiseen React tarjoaa Create React App -komentorivityökalun. Create React App konfiguroi uudelle sovellukselle automaattisesti kasaus- ja kehitystyökalut, kuten Webpack-bundlerin ja Babel-loaderin [14]. Create React App -työkalu ei toisaalta ole merkittävä etu, sillä sen tuki on julkaistu lopetetuksi [40]. Build-työkalut, kuten Vite ja Rsbuild, sisältävät mallipohjia useille teknologioille, ja uusien projektien alustus onnistuu ilman erillistä riippuvuutta työkaluun, kuten Create React App [63], [82]. Täten React on todennut kasaustyökalut tai meta-ohjelmistokehykset lisäominaisuuksineen paremmaksi tavaksi rakentaa sovelluksia valmiille konfiguraatiopohjille [40]. Esimerkiksi Rsbuild väittää olevansa modernisoitu, tehokkaampi versio Create React App-työkalusta, ja se tukee useita ohjelmistokehyksiä plugin-lisäosien avulla [63].

Teknologiasta riippumaton Technology Radar -blogi arvioi vuoden 2025 ensimmäisessä julkaisussaan Reactin lomakekomponentteihin erikoistuneen React Hook Form -kirjaston kypsäksi, helposti integroitavaksi ja tehokkaaksi apuvälineeksi React-kehityksessä [79]. Kirjasto vaikuttaa sopivan hyvin ERAn vaatimiin tietojenkirjaustoimintoihin, jotka vaativat datan validointia. Saman blogin harkinnanvaraisesti suosittelema shadcn tarjoaa lukuisia valmiita komponenttipohjia kopioitavina koodipohjina. Tämä mahdollistaa komponenttien täyden kustomoinnin ja laajentamisen, mutta komponenteista tulee samalla integroidessa itse-huollettavia. Kirjaston komponenttitarjonta sisältää ERA:ssa vaadittuja päivämäärävalitsimia ja datan visualisointikomponentteja [68]. Harkinnanvaraisen suosituksen Technology Radar -blogi antaa myös kattavalle Material UI -komponenttikirjastolle [79].

Komponenttikirjastoista löytyy myös uniikkeja implementointitapoja hyödyntäviä ratkaisuja, kuten tyylin ja toiminnallisuuden erottavat ”Headless UI” -komponentit [32]. Headless UI tarjoaa itse-tyyliteltävät komponentit, tuoden näkymien rakentamiseen vaadittavat perustoiminnallisuudet, kuten alasvetovalikot, tekstikentät ja valintaruudut keittäjän käytettäviksi.

### 6.2.4 Kehityksen mukavuus vs. oppimisen vaikeus

Reactin mentaalimalli sisältää mahdollisia kompastuskiviä aloittelijoille. Funktiokomponenttien mentaalimallista ei selviä eksplisiittisesti funktion suorituskertojen väliset suhteet. React sisältää piilotettuja sääntöjä, jotka aiheuttavat funktion uudelleenajamisen, eli komponentin kuvantamisen. Funktion syntaksista ei myöskään selkeästi erotu tilalliset arvot verrattuna tavallisiin, jokaisella suoritusker-

ralla uudelleen alustettaviin muuttujiin tai funktioihin. Komponentin luomiseen ja päivittämiseen käytetään samaa funktiota, mutta React abstrahoi eroavaisuudet funktion käyttäytymisessä. Nämä ominaisuudet, yhdistettynä asynkroniseen JavaScript ohjelmointiin, voivat aiheuttaa ennalta-arvaamatonta käyttäytymistä ja ylimääräisiä kuvantamiskertoja, jos kehittäjä ei ymmärrä Reactin abstrahoimaa toiminnallisuutta. [52] React ei ole turhaan saavuttanut suosiotaan, sillä oikein käytettynä se on tehokas työkalu. Ongelmana on lukuisat mahdollisuudet teknologian epäoptimaaliseen käyttöön ilman selkeitä virheilmoituksia tai huomautuksia teknologian osalta. Tätä voi toki paikata kolmannen osapuolen työkaluilla, kuten `eslint-plugin-react-hooks` -lisäosalla koodia analysoivaan ESLint työkaluun [52].

Aiemmin suorituskykyä analysoitaessa mainittu Reactille vuonna 2025 julkaistu oma kääntäjäohjelma tuo etuja myös kehitysmukavuuden kannalta [39]. Kääntäjä osaa optimoida React-koodia staattisen koodin analyysin avulla, jolloin kehittäjän ei tarvitse hyödyntää manuaaliseen optimointiin tarkoitettuja *hook*-toimintoja, kuten `useMemo()` ja `useCallback()`. [58] Kyseiset funktiot aiheuttavat ylimääräistä monimutkaisuutta käyttöliittymäkehitykseen, sillä ne keskittyvät Reactin sisäisen toiminnan ohjaamiseen, eikä komponenteille tärkeän liiketoimintalogiikan suorittamiseen. Kääntäjän julkaiseminen osoittaa Reactin ylläpitäjien aktiivista halukkuutta kompensoida teknologian epäkohtia, mikä lieventää epäkohtien merkittävyyttä tulevaisuuden kannalta.

### 6.2.5 Hyvien koodauskäytäntöjen tuki ja rakenteet

Komponenttipohjaisuus Reactissa kapseloi yksittäisten käyttöliittymäosuuksien toiminnallisuutta. Täten granulaarisesti rakennettu komponenttipuu voi olla vastuualueita erotteleva ja huollettavuuden kannalta toimiva kokonaisuus. Komponenteille tieto valuu esi-isiltään props-attribuuttien kautta, jolloin ylemmän tason komponentit voivat toimia ikään kuin ohjaajina toisiinsa liittyville alakomponenteille ilman alakomponenttien tietoa toisistaan [58]. Props-ketjujen kasvattaminen voi kuitenkin johtaa toiminnallisuuksien verkostoitumiseen ja huollettavuuden heikentymiseen. Pitkien props-ketjujen välttämiseksi React tarjoaa globaalin datakontekstin, jolla tietoa voi välittää siistimmin eri komponenttipuun kerrosten välillä. Monimutkaisia tilanhallintatarpeita täydentävät aiemmin mainitut kolmannen osapuolen tilanhallintatyökalut, jotka tuovat tilan hallintaan systemaattisuutta ja eksplisiittisyyttä.

React ei tuo komponenttipohjaisen suunnittelumallin lisäksi pakollisia koodausta ohjaavia rakenteita ja jättää täten paljon vapauksia kehittäjälle. Hyviä koodauskäytäntöjä painotetaan virallisen dokumentaation *"Rules of React"* -ohjeiden avulla. Näillä suosituksilla React-kehittäjää ohjataan ymmärrettävämpiin ja laadukkaampiin koodauskäytäntöihin, joilla React- ja React-DOM-kirjastojen hyödyt saadaan esiin. [58] Reactin "säännöt" painottavat React-koodin säilyttämistä deklaratiivisena, jolloin React abstrahoi imperatiivisen koodin, kuten render-funktion kutsumisen. React on kehittänyt myös lisäosan staattiselle koodianalysityökalulle ESLint:lle. Lisäosan avulla ESLint osaa ilmoittaa koodista löytyvistä Reactin periaatteiden rikkomuksista [58]. Tämä ei ole kuitenkaan täydellinen tapa ohjata hyviin koodauskäytäntöihin, sillä sen hyödyntäminen vaatii ESLint-työkalun käytön kol-

mannen osapuolen riippuvuutena kehityksessä. Lisäksi React tarjoaa dokumentaatiossaan osion nimeltä *"Thinking in React"*, jossa kerrotaan suositellut työvaiheet React-käyttöliittymän suunnitteluun [58]. Yhteenvetona Reactin kattavat lisämateriaalit ja työkalut korvaavat hyvin Reactin jättämien vapauksien vaaroja, mutta sudenkuoppiin astumisen välttäminen on silti pääosin kehittäjän vastuulla, toisin kuin esimerkiksi kokonaisissa ohjelmistokehyksissä, kuten Angular, joka pakottaa ohjelmistot tiukkaan MVC-malliinsa [3].

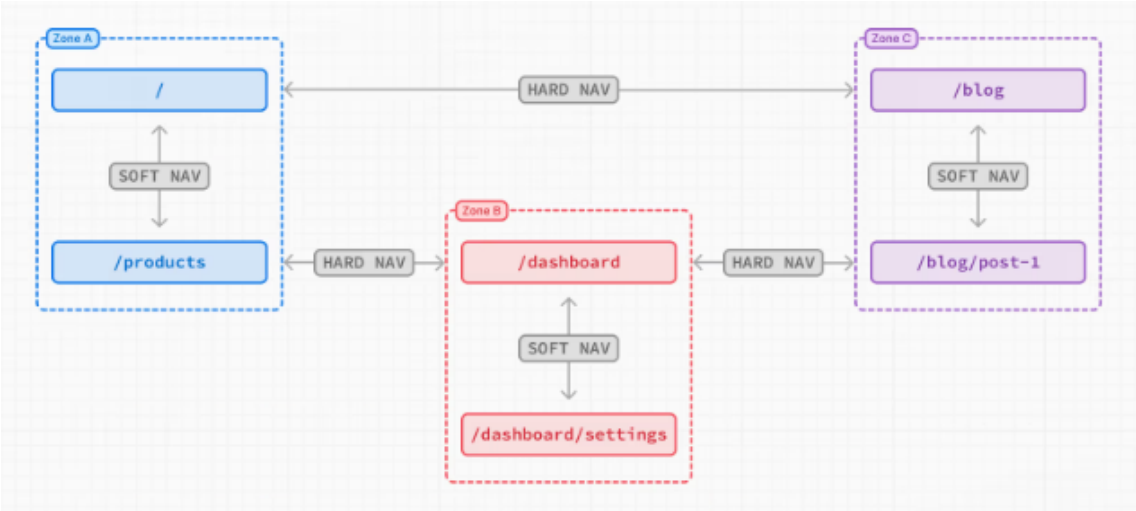
Reactin metakehyksen, Next.js:n omat tukimateriaalit sisältävät Reactin ja Next.js:n suhdetta alustavan "React Foundations" -osuuden sekä tärkeimpiä ominaisuuksia opettavat "App Router" "Pages Router" ja "SEO" (hakukoneoptimointi, engl. Search Engine Optimization) -osuudet. Jokainen tutoriaaliosuus sisältää useita lukuja, joissa sovelluksen rakentumista selitetään vaihe vaiheelta esimerkin avulla. Lukujen lopussa dokumentaatio testaa vaiheen ydinsisällön oppimista ja yhteenvetää opitun sisällön. [51] Täten Next.js:n perusteiden oppiminen on tehty helpoksi, etenkin jos React on kehittäjälle ennestään tuttu teknologia. Lisäksi Next.js:n emoyhtiö, Vercel, tarjoaa kehittäjälle laajemman kattauksen yksityiskohtaisia oppaita liittyen tarkempiin käyttötapauksiin ja yleisiin web-sovellusten konsepteihin, kuten evästeisiin ja kyberturvallisuuteen [81].

### 6.2.6 Muuta huomioitavaa

ERAssa tärkeää ominaisuutta, lomakekäsittelyä, varten React ja React-DOM täydentävät natiivia HTML:n form-elementtiä esimerkiksi useFormStatus ja useOptimistic -hookeilla, jotka voivat parantaa käyttäjäkokemusta lomakkeiden lähettämiseen liittyen. [58]

Next.js tarjoaa puolestaan oman mahdollisuutensa Micro-frontend tyyliseen käyttöliittymän pilkkomiseen itsenäisiksi sovelluksikseen, joista käytetään nimitystä Zone. Nextin Multi-Zones -tekniikalla sivuja voidaan sitoa yhteen, myös eri teknologioilla toteutetuista sovelluksista, mikä voi olla hyödyllistä ERAn tulevaisuuden arkkitehtuurissa, mikäli "Strangler Pattern" -tyylistä menetelmää käytetään käyttöliittymän uudistamiseen. Eri Zoneille voidaan määrittää reititykset proxyn avulla tai suoraan yhdestä Next.js Zonesta käsin Nextin konfiguraatio-tiedostossa. Zonen sisäiset reititykset hyödyntävät Next.js:n optimoituja navigointitapoja, mutta navigointi erillisten Zonejen välillä johtaa sivun uudelleenlataamiseen. Tätä havainnollistaa kuva 6.2, jossa "HARD NAV" tarkoittaa uudelleenlatausta navigoinnin yhteydessä ja "SOFT NAV" optimoitua Next.js:n sisäistä navigaatiota. [51]

Nextin tuomista merkittävimmistä eduista toinen, eli hakukoneoptimointi ei suoraan osu tavoitteisiimme ERA-järjestelmän kehityksessä. Tietylle asiakasryhmälle rajattuna kaupallisena järjestelmänä hakukoneoptimoinnista ei ole lainkaan hyötyä ERAn käyttöliittymässä, sillä sitä käytetään ammattilaisten työkaluna, jonne navigoidaan etukäteen tiedostetulla URL-osoitteella, eikä hakukoneiden kautta. Toinen merkittävä etu, suorituskyvyn optimointimahdollisuus, on puolestaan hyvinkin relevantti, sillä ERA sisältää näkymiä, joissa suurten tietomäärien lataaminen ja esittäminen vaatii runsaasti tietoliikennettä ja renderöintikapasiteettia.




**Kuva 6.2:** Multi-Zones -tekniikan navigointia ja reititystä havainnollistava kaavio [51]

.

## 6.3 Vue.js

Vue on Reactin ja Angularin rinnalle luontaisesti noussut suosittu käyttöliittymäkehys. Luontaista Vuen noususta on tehnyt suuren korporaation tuen puuttuminen, sillä Vue on noussut kartalle innokkaan yhteisönsä ja yksittäisten sponsorien avustamana [84]. Suosiota puoltavat suuret käyttöasteet Stack Overflow ja State Of JS -teknologiakyselyiden mukaan [26], [70]. Vue on Angularista inspiroituneena lähtenyt kehittämään sen hyvien ominaisuuksien pohjalta kevyempää ja aloittelijaystävällisempää käyttöliittymäteknologiaa, mikä on osoittautunut tarpeelliseksi teknologiaprofiiliksi web-käyttöliittymien kehityksessä [86].

### 6.3.1 Suorituskyky

Selaimessa ajettavan JavaScript-bundlen koon optimointi on yksi Vuen vahvuuksista, minkä ansiosta sitä voidaan pitää kevyenä ohjelmistokehyksenä. Vue osaa ladata tarvitsemansa kirjastotoiminnot ajonaikana dynaamisesti, ja erillisille sivuille ladataan ainoastaan kyseiseen näkymään sisältyvät riippuvuudet. Suurempien verkkosovellusten kanssa latausajat hidastuvat vaadittujen riippuvuuksien määrän kasvaessa, mikä heikentää Vuen suorituskykyarviota. Kuten React, Vue hyödyntää renderöintiin virtuaalista DOM-rakennetta, jolloin turhia sivun uudelleenlatauksia voidaan välttää. Tämä tehostaa verkkosovelluksen responsiivisuutta. [16]

### 6.3.2 Stabiilisuus ja suosio

Vue.js eroaa stabiilisuuden näkökulmasta kahdesta muusta valtavirran ohjelmistokehyksestä, Reactista ja Angularista, sillä se on täysin yhteisönsä ja kuukausittaisista sponsoreistaan taloudellisesti riippuvaisen ydintiiminsä varassa [84]. Kehitysmalli tuo teknologian kehittymiseen epävarmuutta verrattuna suuryritysten antamaan omistuspohjaiseen tukeen. Vue.js on kuitenkin saavuttanut suuren suo-

sion ja sen mukana laajan yhteisön. State of JS 2024 -kyselyn mukaan Vue on lähes koko verkkosovelluskehitysyhteisön tietoisuudessa, ja 51 prosenttia kyselyyn vastanneista kertoi käyttävänsä Vue-teknologiaa, mikä tekee siitä käyttöasteensa perusteella toiseksi suosituimman verkkokehyksen Reactin jälkeen [26]. Kysely erottelee myös Vuen olevan toiseksi suosituin ammattikäytössä ja yritysten kokoluokan olevan React-ammattilaisten kanssa samalla tasolla. Stack Overflown 2024 kyselytulokset vahvistavat, että Vue on ollut merkittävässä käytössä 17 prosentilla vastaajista [70]. Suurten Vue.js:stä riippuvien käyttäjämäärien ansiosta ohjelmistokehystä voidaan pitää yrityskäyttöön luotettavana, mitä puoltaa riippumattoman Technology Radar -blogin viimeisin vahva suositus Vuen käyttöönotosta [79].

Vue ei lupaa Reactin tai Angularin tavoin säännöllisiä päivityksiä, mutta antaa uusia ominaisuuksia sisältäville päivityksille aikaväliarvioksi kolmesta kuuteen kuukautta. Vue noudattaa semanttista versiointia kahta poikkeustapausta lukuun ottamatta. Poikkeuksia aiheuttavat TypeScript-päivityksistä propagoituvat semanttisen versioinnin rikkomukset sekä käännetyn koodin yhteensopivuus, jonka toisaalta kerrotaan olevan ongelma vain kirjastojen tuottajille. [84] Vanhaa toiminnallisuutta rikkovia major-julkaisuja Vue on tehnyt harvoin, sillä tämänhetkinen uusin major-versio on teknologian pitkähkön elinkaaren kolmas. Vue tarjoaa myös virallista tukimateriaalia rikkovia muutoksia sisältäviin päivityksiin, ja viimeisin vanhentunut major-versio sai 18 kuukautta pitkän aikavälin tukea sekä asteittaisen siirtymisen työkaluja [87].

### 6.3.3 Ekosysteemi ja työkalut

Vuelle on suosionsa ja laajan ekosysteeminsä johdosta kehittynyt kattava valikoima kirjastotarjontaa, niin komponenttikirjastojen kuin erityisten valmistoiminnallisuuksien osalta. Kuratoitu Awesome Vue.js -lista Vuelle kehitetyistä työkaluista ja apumateriaaleista sisältää esimerkiksi Googlen Material Design -järjestelmän mukaisen Vuetify-komponenttikirjaston [7].

Kuten Reactille, komponenttitarjonnasta Vuelle löytyy myös uniikkeja käyttötapoja hyödyntäviä ratkaisuja, kuten tyylin ja toiminnallisuuden erottavat Headless UI -komponentit tai osaksi omaa koodia kopioitavat shadcn-komponenttipohjat, jotka Awesome Vue.js -listaus mainitsee epävirallisiksi yhteisöversioiksi Reactin alkuperäisestä shadcn-valikoimasta [7], [32], [68].

### 6.3.4 Kehityksen mukavuus vs. oppimisen vaikeus

Vue-teknologiaa pidetään muihin suosittuihin ohjelmistokehyksiin verrattuna helposti opittavana. Reactiin verrattuna Vue:n mentaalimalli on yksinkertaisempi. Vuen oletustyyli komponenttien määrittelyyn ei ole funktio, vaan tiedosto, joka erottelee komponentin elementtirakenteen, toiminnallisuuden ja tyylimäärittelyt. [84], [86] Rakenteen määrittelevä osuus muistuttaa tyyliltään muita templaattisyntakseja, kuten Reactin JSX:ää. Toiminnallisuuden määrittelevässä osuudessa esimerkiksi komponentin elinkaareen liittyvät toiminnallisuudet määritetään niille kuvaavasti nimetyillä rajapintafunktioilla, ja komponentin alustusosio voidaan jakaa erillisen `script-setup`-osion alle. Syntaksin eksplisiittisyyden ja

näkyvämmän mentaalimallin ansiosta Vue on aloittelijaystävällisempää kuin React. Tyylien määrittelyn yhdistäminen komponenttitiedostoon auttaa välttämään komponenttien välisiä tyyliriippuvuuksia ja globaalien tyyliluokkien päällekkäisyyksiä, kun tyylimäärittelyjen vaikutusalue voidaan rajata yksittäiseen komponenttiin. [84]

Angulariin verrattuna Vue sisältää merkittävästi vähemmän opeteltavia toiminnallisuuksia, ja Vue mahdollistaa laajalti käyttäjän oman ohjelmointilogiikan hyödyntämisen. Vue osaltaan muistuttaa Angularia templaattisyntaksin erityisten attribuuttien, kuten `v-if` ja `v-for` -direktiivien osalta. Näille löytyy lähes identtiset vastineet Angularista, minkä selittää Vuen syntyminen Angularia edeltäneen AngularJS-teknologian inspiroimana. [86]

Vuen tapa määritellä kokonainen komponentti yhdessä tiedostossa tukee komponenttipohjaisen arkkitehtuurin hyviä puolia modulaarisuuden ja yksinkertaisuuden kannalta [79], [84]. Vuen opittavuus ja intuitiivinen mentaalimalli näkyvät myös käyttäjätyytyväisyyskyselyssä State of JS 2024, joka sijoittaa Vuen kärkikolmikkoon verkkokäyttöliittymien aiheuttamassa tyytyväisyydessä ja positiivisissa tuntemuksissa [26]. Yhtenäisen komponenttimäärittelyn tuomien kognitiivisten etujen lisäksi Vuen sisäänrakennettu yhden sivun komponenttien käännösprosessi mahdollistaa automaattisesti kehitystä sujuvoittavan HMR-toiminnallisuuden.

### 6.3.5 Hyvien koodauskäytäntöjen tuki ja rakenteet

Reactin tavoin Vue ei ole kokonaisvaltainen, tietynlaista rakennetta vaativa ohjelmistokehys, vaan se keskittyy käyttäjälle näkyvän toiminnallisuuden toteuttamiseen, jättäen kehittäjälle paljon vapauksia muun toteutuksen osalta [86]. Kuten Reactissa, haittapuolena on runsaat vapaudet kehittää mahdollisesti sekava ohjelmistokokonaisuus, minkä riski kasvaa sovelluksen koon laajentuessa. Kehittäjän vastuulle jäävät esimerkiksi reititys ja sovelluksen laajuinen tilanhallinta [86].

Vuen dokumentaatio sisältää Style Guide -osion, joka ”Rules of React” -oppaan tavoin kompensoi ohjelmistokehyksen vähäisiä rakenteellisuusvaatimuksia. [85] Vuen virallinen komentorivityökalu, Vue CLI, tuo puolestaan apukomentoja hyväksi todettujen perusrakenteiden pystyttämiseen. Lähestymistavan tarkoituksena on sallia yksityiskohtainen konfiguroitavuus, mutta mahdollistaa myös perustarpeiden toteuttaminen valmiiksi määritetyllä oletusrakenteella [86]. Vue-teknologialle on tarjolla myös kolmannen osapuolen maksullista koulutusta ja pitkän aikavälin erikoistukea vanhoille versioille [84], [87].

### 6.3.6 Muuta huomioitavaa

Vuen progressiivinen luonne tuo mahdollisuuksia ERAn käyttöliittymän uudistamiseen. Vue.js-kirjasto on integroitavissa ERAssa käytetyn jQuery-kirjaston tapaisesti yksittäisiin palvelinpuolella kuvannettuihin näkymiin, sillä Vue sallii jopa useita Vue-sovellusinstansseja samassa näkymässä [84]. Vuelle on myös toteutettu erillinen kevytversio petite-vue, joka on erityisesti tarkoitettu teknologian progressiiviseen käyttöönottoon [54]. Tällaista lähestymistapaa hyödyntävän arkkitehtuurivaihtoehdon ominaisuuksia analysoidaan luvussa 7.

## 6.4 Svelte

Svelte on tuore ja nopeasti suosiota kerännyt ohjelmistokehys, joka haluaa erottua perinteisistä JavaScript-ohjelmistokehyksistä keveydellään ja kehittäjäkokemuksellaan. Tämän tavoitteen se pyrkii saavuttamaan kääntäjän ja oman syntaksinsa avulla. Kun HTML:ää ja JavaScriptiä laajentava syntaksi saadaan käännettyä natiiviksi JavaScriptiksi, selaimelle ei tarvitse erikseen lähettää ylimääräistä ohjelmistokehyksen koodia, kuten React-DOM -kirjastoa. Täten selaimen ja käyttöliittymäkoodin väliltä häviää yksi abstraktiokerros, mikä vähentää selaimen prosessointikuormaa ja selaimelle lähetettävän koodi-bundlen kokoa. Svelten kääntyminen natiiviksi JavaScriptiksi auttaa myös yksittäisten komponenttien yhteentoimivuudessa muiden teknologioiden kanssa. Selaimet osaavat tulkita komponentit sellaisinaan, jolloin myös koodin pilkkominen bundleihin tuo enemmän hyötyjä. [31]

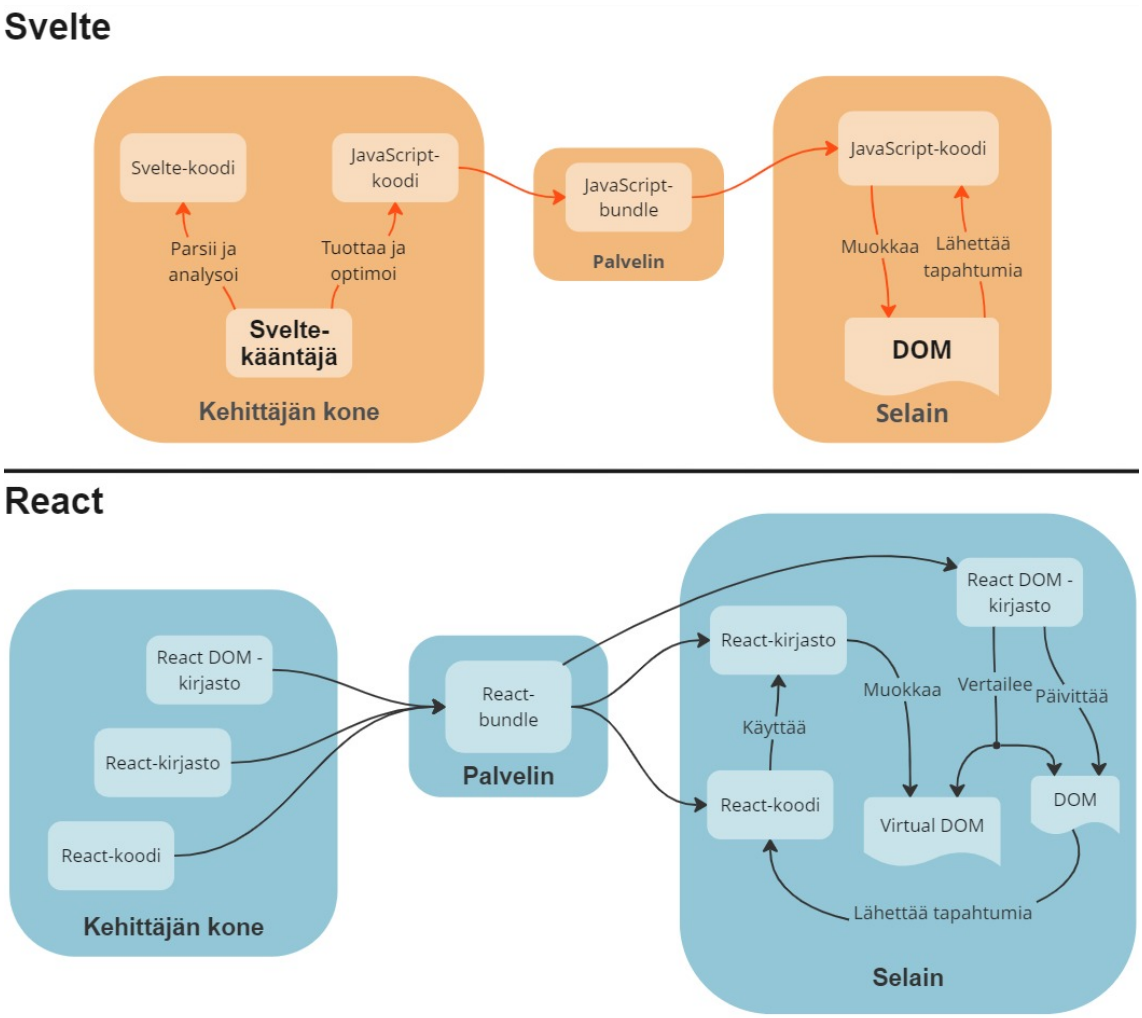
### 6.4.1 Suorityskyky

Svelten kehittäjä, Rich Harris, väittää teknologiansa olevan nopeimpien vaihtoehtojen joukossa [31]. Tämän mahdollistaa Svelten tapa kääntää selaimeen lähetettävä sovellus natiiviksi JavaScript-koodiksi, jolloin ohjelmistokehyksen rakenteet ovat integroituneina sovelluksen rakenteeseen, eikä erillisinä moduuleina. Täten Svelte ei tarvitse esimerkiksi virtuaalista DOM-rakennetta ja sen vertailukoodia. Tätä eroavaisuutta havainnollistaa kuva 6.3. Harrisin blogikirjoitus kertoo yksittäisen pienehkön Svelte-sovelluksen olevan yli kymmenen kertaa pienikokoisempi kuin pelkkä Reactin oma kirjastokoodi, joka selaimelle lähetetään. Tämän pitäisi kirjoituksen mukaan nopeuttaa sovelluksen latautumista suunnilleen samassa suhteessa. Ajonaikaista tehokkuutta ja sovelluksen pientä kokoa puolustavat Chromella ajetun JavaScript-teknologioiden vertailun tulokset, joissa Svelte on tässä tutkimuksessa vertailluista teknologioista selkeästi kevyin ja nopein [38].

### 6.4.2 Stabiilisuus ja suosio

Stack Overflown kehittäjäkyselyt osoittavat Svelten olevan yksi kiinnostavimmista teknologioista kehittäjien näkökulmasta [70]. State of JS 2024 -kysely osoittaa, että Svelte on kehittäjien tietoisuudessa noussut lähes jokaisen tietoisuuteen, kun 96 prosenttia vastaajista tiesi teknologiasta [26]. Tämä antaa teknologialle lisää tulevaisuuden potentiaalia nuoresta iästään huolimatta. Stack Overflown kyselyssä on kuitenkin huomioitava, että vastaajat ovat yksittäisiä kehittäjiä, joiden näkökulma ei täysin vastaa yrityksen intressejä. Kyselystä ilmenee Svelten suosio ja trendikkyys, mutta yrityskäyttöön arvioitavana Svelten stabiilisuutta on myös tutkittava. State of JS 2024 -kysely kertoo 11:n prosentin vastaajista käyttävän Svelteä ammattilaiskontekstissa, kun taas Stack Overflown kyselyn ammattilaiskehittäjistä 6 prosenttia on käyttänyt Svelteä merkittävissä määrin [26], [70]. Täten Svelte on saavuttanut jo merkittävän jalansijan myös ammattikäytössä, mutta jää silti jälkeen suuresta kolmikosta, jonka muodostaa React, Vue ja Angular.

Sveltestä on tämän tekstin kirjoitushetkelläjulkaistu jo viides merkittäviä muutoksia tuova versio vuonna 2016 tapahtuneen ensijulkaisunsa jälkeen [73]. Kehittyminen on siis ollut hyvin nopeaa, ja entisiä ominaisuuksia rikkovia muutoksia on




**Kuva 6.3:** Reactin ja Svelten konseptuaalisia eroja havainnollistava kuva hahmoteltuna teknologioiden dokumentaation pohjalta [31], [58]

tuotu Svelteen. Tämä hankaloittaa teknologialla rakennetun yritystason sovelluksen ylläpitoa, ja päivitystahti voi viitata tapauksessamme liian nuoreen teknologian kypsyysasteeseen. Versiopäivityksen käyttöönotolle on luotu kattava opas Svelten dokumentaatioon [73] ja myös syntaksipäivityksissä avustava työkalu on kehitetty. Tämä lieventää tiheiden versiojulkaisujen haittoja, sillä Svelten kehittäjät selkeästi haluavat minimoida vanhoille käyttäjille aiheutetut muutostarpeet. Siitä huolimatta muutos vaatii työpanosta ja uusien ominaisuuksien sekä syntaksin opettelua Svelten käyttäjiltä. Svelte 5 tukee kuitenkin Svelte 4:n syntaksia, joten muutoksille annetaan teknologian käyttäjille aikavaraa [73]. Toinen haittapuoli merkittävässä syntaksimuutoksessa on myös aiemmin tuotettujen koodausoppaiden vanheneminen. Uudella syntaksilla rakennettujen ohjeiden löytäminen on haastavaa, ja migraatio-oppaan avulla vanhojen tukimateriaalien tulkitseminen vaatii lisätyötä.

Svelten stabiiliutta parantaa myös Next.js -teknologiaa tukevan Vercel-suuryrityksen taloudellinen tuki Svelten kehityksessä [71]. Sveltellä on myös palautteen ja toimintoehdotusten keräämisprosessi eli RFC, mutta se ei ole samalla tasolla aktiivinen, kuin esimerkiksi Reactilla [76].

### 6.4.3 Ekosysteemi ja työkalut

Nuorena teknologiana Svelten ekosysteemi ei yllä suuren kolmikon, eli Reactin, Angularin ja Vuen tasolle, mutta Sveltelle on kehitetty runsaasti kirjastoja tärkeimpiin aputoimintoihin, kuten reititykseen ja komponenttipohjiksi [77]. Komponenttikirjastoista esimerkiksi Technology Radar -blogin harkinnanvaraisesti suosittelema, alunperin Reactille kehitetty, shadcn tarjoaa lukuisia valmiita komponenttipohjia ”copy-paste” -tyylillä integroitavaksi myös Sveltellä kehitettynä [77], [79]. Tämä mahdollistaa komponenttien täyden kustomoinnin ja laajentamisen, mutta komponenteista tulee samalla integroinnin jälkeen itse huollettavia. Kirjaston komponenttitarjonta sisältää ERA:ssa vaadittuja päivämääränvalitsimia ja datanvisualisointikomponentteja [68].

### 6.4.4 Kehityksen mukavuus vs. oppimisen vaikeus

Svelten kehittäjän, Rich Harrisin, mukaan kehityksen mukavuus on pääprioriteetti Svelten filosofiassa [75]. Vaikka käyttämisen mielekkyys ei vaikuta ensikuulemalta tärkeimmältä ominaisuudelta web-teknologialle, voi se välillisesti tuoda merkittäviä hyötyjä, kuten paremmat tulevaisuuden näkymät ja ekosysteemin kasvaminen. Rich Harris kertoo kirjoituksessaan Svelten syntaksin rakentuneen enemmän HTML:n kuin JavaScriptin ympärille [75]. Koska HTML on kuvainnollinen eli deklaratiivinen kieli, on se jo luonteeltaan intuitiivisempi käyttöliittymien rakentamiseen, kuin käskymuotoinen eli imperatiivinen JavaScript [59]. Käyttöliittymänäkymä nähdään selaimessa kerrallaan kokonaisena, eikä sitä tarvitse katsella osio kerrallaan rakentuvana kokoelmana. Täten HTML-tyyliin rakennettu koodi vastaa enemmän selaimessa nähdyn käyttöliittymän rakennetta kuin esimerkiksi Reactin funktiokomponentit. Yleensä ohjelmakoodissa funktio ajetaan kerran ja sisällä määritetyt muuttujat ”elävät” yhden suorituskerran ajan. Reactissa funktiolla rakennettu komponentti voi puolestaan määrittää esimerkiksi suorituskertojen yli säilyviä muuttujia *useState* -hookilla, mikä aiheuttaa jo perinpohjaisesti ristiriitaa ajattelumalleissa [58]. Myös Svelte vaatii ”script”osuuden muuttujien ja niiden välisten suhteiden sekä toiminnallisuuden määrittämiseen, mutta scriptien sisällöt muistuttavat enemmän reaktiivista ajattelumallia, eikä vain kerran ajettavaa funktiota, joka jättää jälkeensä sivuvaikutuksia. Täten Svelte pystyy syntaksillaan ja kääntäjällään luomaan JavaScriptin päälle abstraktion, joka on ohjelmoijalle intuitiivisempi ja vähemmän altis virheille.

Svelte näyttää kyselytulosten perusteella onnistuneen hyvin tavoitteessaan olla koodaajille mielekäs ja ytimekäs ohjelmistokehys. State of JS 2024 -kyselyn mukaan Svelte on verkkokäyttöliittymäteknologioiden kärkikaksikossa mielenkiintoisuuden, tyytyväisyyden ja positiivisten tuntemusten kategorioissa [26]. Tämä viittaa teknologian hyvään kehittäjäkokemukseen ja intuitiivisuuteen. Tulosta puoltaa myös Stack Overflown vuoden 2024 kehittäjäkysely, jossa 73 prosenttia Svelten käyttäjistä on kiinnostuneita jatkamaan käyttöä [70].

Intuitiivisuutta tavoitellakseen Svelte toteuttaa tiedon reaktiivisuuden määrittelyjä kompaktisti JavaScriptin toiminnallisuutta laajentamalla. Aiemmissa Svelten versioissa muuttujien käytöstä päätellen Svelte määritteli muuttujille reaktiivisuustoimintoja, kuten käyttöliittymässä esitetyn arvon automaattisen sitomisen ”let”

-avainsanalla määriteltyyn muuttujaan tai reaktiivisen johdannaismuuttujan määrittämisen dollarimerkillä tyylillä `$: doubledValue = 2 * value;` [72], [74]. Svelten viides versio on tehnyt reaktiivisuuden määrittelystä eksplisiittisempää vaatimalla dollarimerkin yhteyteen muuttujan käyttäytymistä tarkentavan avainsanan. Näistä dollarimerkki-avainsana-yhdistelmistä Svelte käyttää nimitystä ”rune”Ėsimerkiksi aiemmin mainituille sidonnaismuuttujille on määritelty `$state` -niminen ”rune”funktio ja vastaavasti johdannaismuuttujille `$derived`-rune [73]. Tällä syntaksimuutoksella Svelte tekee koodistaan ymmärrettävämpää sekä kehittäjälle että kääntäjälleen.

Svelten ollessa nuori teknologia, voi sille olla haastavaa löytää osaavia kehittäjiä. Kehittäjätyytyväisyys, teknologian intuitiivisuus ja laaja mielenkiinto teknologiaa kohtaan kuitenkin korvaavat kokeneiden kehittäjien puutetta, ja todennäköisesti helpottavat Svelte-kehittäjien rekrytointia.

### 6.4.5 Hyvien koodauskäytäntöjen tuki ja rakenteet

Rakenteeltaan Svelte tuo selkeät työkalut UI-näkymien rakentamiseen ja ytimekkäillä intuitiivisuutta tavoittelevilla reaktiivisuustoiminnoillaan auttaa pitämään mentaalimallin yksinkertaisena. Svelte tuo myös sisäänrakennetun tilanhallintaratkaisun store-toimintonsa avulla, mikä auttaa monimutkaisten tilanhallintatarpeiden toteuttamisessa. Käyttöliittymää laajemman tason rakenteisiin Svelte tarjoaa valmista toiminnallisuutta SvelteKit-metakehyksellään.

Svelten kliininen reaktiivisuusmalli ja natiivi tilanhallintaratkaisu tukevat järjestelmällisyyttä ja pystyvät pitämään myös monimutkaisen käyttöliittymän huollettavana. Svelte 5:n Runes-toiminnot auttavat myös jäsentelemään käyttöliittymän toiminnallisuutta, eikä käyttäjän tarvitse opetella dollarimerkin ”taikavaikutuksia” Svelte 4:n ja sitä aiempien versioiden tavoin [71].

Svelte dokumentoi ominaisuutensa sivustollaan jaoteltuna johdantoon, ohjeellisiin ominaisuuskuvauksiin, luokittelemattomiin ohjeisiin sekä täsmällisempiin referenssimääritelmiin. [71] Selkeästi luokitellut ohjeet auttavat teknologiaan tutustumisessa.

## 6.5 Blazor

Microsoftin Blazor on kuin vastavirtaan uiva kala muiden JavaScript-ohjelmistokehysten joukossa. Se tuo palvelinpuolen ohjelmistoissa käytetyn C#-kielen työkaluksi selainohjelmistojen kirjoittamiseen, kun taas perinteisesti selainohjelmissa käytetty JavaScript on yleistynyt palvelinpuolella ajoympäristöjen, kuten Node.js:n myötä. C#:n käyttö selaimessa on mahdollista Blazorin sisältämän .NET-ajoympäristön avulla, joka on rakennettu WebAssembly-kielellä. [10] WebAssembly on JavaScriptiin verrattuna huomattavasti matalamman tason kieli, joka muistuttaa perinteistä assembly-kieltä [45]. Täten WebAssembly on pohjimiltaan JavaScriptiä tehokkaampaa. Blazor tukee myös JavaScriptin hyödyntämistä selainpuolen interaktiivisuudessa, mikä täydentää Blazorin .NET-ajoympäristön kyvykkyyksiä [10].

### 6.5.1 Suorityskyky

Blazorin suorituskykyyn vaikuttavat sen tarjoamat neljä erilaista kuvantamismoodia, jotka ovat Static Server, Interactive Server, Interactive WebAssembly ja Interactive Auto [10]. Static Server kuvantaa sivut staattiseksi HTML-rakenteeksi jo palvelimella, eikä siten tue interaktiivisuutta. Muut kuvantamismoodit ovat interaktiivisia, mutta kahdella eri periaatteella.

- Interactive WebAssembly muistuttaa näistä eniten muita ohjelmistokehyksiä, kuten Reactia, jossa ohjelmistokehyksen koodi ja käyttöliittymäsovellus lähetetään selaimeen ajettavaksi ensilatauksella. WebAssembly-kielellä rakennettu Blazorin .NET-ajoympäristö mahdollistaa käyttäjän syötteeseen reagoinnin C#-kielellä kirjoitetulla koodilla. [10]
- Interactive Server tuo täysin erilaisen lähestymistavan interaktiivisuuden luomiseen. Se ei lähetä käyttöliittymäsovellusta tai .NET-ajoympäristöä käyttäjän selaimeen lainkaan, vaan ajaa ne palvelimen puolella. Interaktiivisuuden mahdollistamiseksi Interactive Server -moodi luo WebSocket-yhteyden käyttäjän selaimen kanssa, jolloin internet-yhteys välittää käyttäjän syötteen palvelimelle ja palvelin palauttaa selaimelle vain DOM-rakenteen ja siihen luodut muutokset. [10]
- Joustavuutta Interactive Server ja Interactive WebAssembly -moodien välille tuo Interactive Auto -moodi, joka pyrkii piilottamaan kahden muun interaktiivisuusmoodin suorituskykyheikkoudet käyttäen käyttöliittymän ensilataukseen Interactive Server -moodia ja siirtäen vastuun WebAssembly-ajoympäristölle vasta kun se on ladattu käyttäjän selaimeen. Vastuunjako mahdollistaa nopean ensilatauksen ja responsiiviset käyttöliittymämuutokset. [10]

Molemmilla interaktiivisuusperiaatteilla on haittapuolensa suorituskyvyn näkökulmasta. Web-käyttöliittymäteknologioiden benchmark-tulokset paljastavat, että Blazorin Interactive WebAssembly -malli on suorituskyvyltään keskimäärin vertailun huonoin responsiivisuuden, sovelluksen koon ja selaimen muistinkäytön suhteen [8]. Tämä viittaa siihen, että ajonaikainen c#-koodin tulkitseminen WebAssembly-kielelle hidastaa koodin suoritusta merkittävästi WebAssemblyn teoreettisesta nopeudesta huolimatta, ja .NET-ajoympäristö on raskas.

Interactive Server -moodissa komponenttien tila on tallessa palvelimella ja muutokset DOM-rakenteeseen välitetään kaksisuuntaisen WebSocket-yhteyden kautta. Tämä lisää viivettä käyttäjän interaktioihin reagoimiseen, mutta vähentää merkittävästi selaimeen ladattavan sivun kokoa. Blazor Serverillä .NET:n luontaisessa ajoympäristössä koodin suorittaminen on nopeampaa, mutta käyttäjäkohtaisen tilan säilytys sekä WebSocket-yhteyden ylläpito tuo mahdollisia skaalautuvuusongelmia suurilla käyttäjämäärillä. [10]

### 6.5.2 Stabiilisuus ja suosio

Blazor hyötyy Microsoftin tuesta, ja lupaa tarjoavansa stabiilit, helppokäyttöiset ja monipuoliset työkalut käyttöliittymien rakentamiseen [10]. StackOverflown kehittäjäkyselyn mukaan Blazor on runsaassa käytössä viidellä prosentilla vastaajis-

ta, sekä ammattilaisten että kaikkien kehittäjien keskuudessa. Täten Blazor asettuu käyttäjämäärissään tässä tutkimuksissa arvioitujen Svelten ja HTMX:n väliin. Sveltellä on vajaa kaksi prosenttiyksikköä enemmän vakituisia käyttäjiä kuin Blazorilla. HTMX:ää puolestaan käyttää laajamittaisesti noin kaksi prosenttia vähemmän vastaajia kuin Blazoria. [70] Täten Blazor on vielä melko tuoreiden teknologioiden joukossa suosionsa kannalta, mutta Microsoftin tuki lisää teknologian säilyvyyden luotettavuutta.

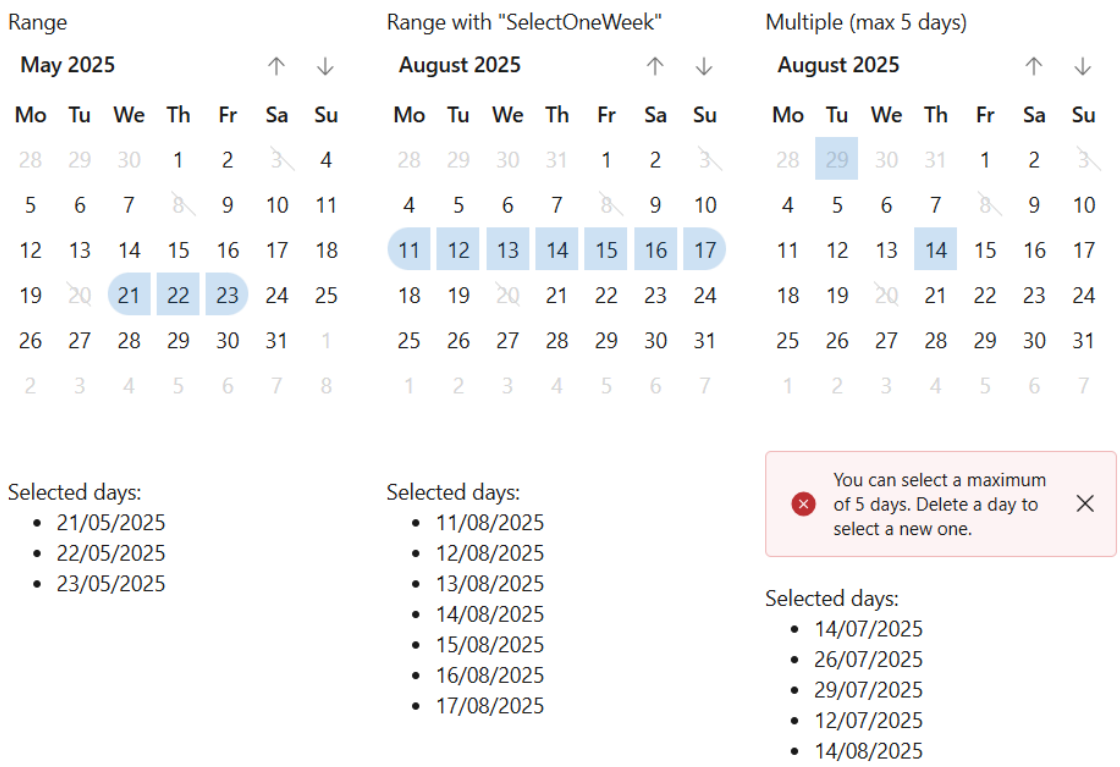
Toisaalta Blazor luottaa Web Assembly -teknologiaan, joka ei ole yhtä vakaalla pohjalla kuin JavaScript. Lisäksi Microsoftilta ei löydy lupauksia pitkän ajan tuesta Blazorille, mikä tuo epävarmuutta teknologian käyttöönottoon. Pitkän ajan tuen ASP.NET Core 10 -version aikaisessa kehittäjätiimin roadmapissä Blazor kuitenkin mainitaan, mikä viittaa Microsoftin panostavan teknologian kehittämisen jatkamiseen [17]. Roadmap sisältää uusia ominaisuuksia ja parannuksia, joten kehitys vaikuttaa aktiiviselta. Blazorin tilanne jättää silti epävarmaksi teknologian pitkän aikavälin jatkuvuuden, johtuen pienehköstä suosiostaan ja kehityslupausten puuttumisesta.

### 6.5.3 Ekosysteemi ja työkalut

Yhteensopivuus .NET-ekosysteemin kanssa tuo jo itsessään Blazorille runsaasti valmista toiminnallisuutta, jota ERAnkin palvelinohjelmistossa on hyödynnetty. Toisaalta Blazorin selainpuolen .NET-ajoympäristö tukee vain osajoukkoa .NET:n API-tarjonnasta, ja täysi tarjonta on saatavilla vain Server- tai Hybrid-hostausmalleilla [10]. JavaScript-yhteensopivuus tuo myös JavaScriptin rikkaan ekosysteemin Blazor-sovellusten saataville, joten kokonaisuudessaan saatavilla oleva ekosysteemi on hyvin laaja.

Microsoft tarjoaa myös oman Fluent-tyylijärjestelmänsä mukaisen Fluent UI Blazor -komponenttikirjaston ilmaiseksi MIT-lisenssillä [23]. Kirjaston dokumentaation mukaan se mahdollistaa tyylillisesti yhdenmukaisten valmiskomponenttien hyödyntämisen Blazorin Razor-näkymissä [22]. Käyttöliittymään voi tuoda yhtenäisyyttä myös kirjaston dokumentaation mainitsemalla Design token -järjestelmällä, joka sisältää valmiiksi määriteltyjä väri- ja kokoarvoja esimerkiksi kirjasimille ja elementeille eri käyttötarkoituksiin. Esimerkiksi ERAlle tärkeinä pidetyt päivämääränvalitsimet vaativat vain muutamia koodirivejä toimiakseen useilla erilaisilla päivämäärän valintatavoilla, kuten kirjaston dokumentaatiosivulta otetussa kuvakaappauksessa 6.4 nähdään. Dokumentaation sisältämät esimerkkikoodit kuvakaappauksen kalentereille paljastavat, että komponentin parametrisoidut valmistoiminnallisuudet auttavat pitämään komponenttien käytön ytimekkäänä.

Kaupallisena komponenttien tarjoajana Radzen erottuu massasta kattavalla valikoimallaan korkean tason valmiskomponentteja, kuten tiedon taulukointiin tarkoitettu DataGrid ja tiedon visualisointiin soveltuvat graafikomponentit [57]. Radzen tarjoaa maksullisena myös jopa Visual Studioon kehitystä helpottavan lisäosan tai kokonaisen kehitysympäristön, Radzen Blazor Studion. Radzen tuo siis mahdollisuuksia Blazorin tehokkaaseen hyödyntämiseen, mutta ei merkittävästi nosta teknologian ekosysteemin arvoa evaluoinnissa ERAn käyttöön.




**Kuva 6.4:** Esimerkkejä päivämäärävalitsimista Fluent UI Blazor -kirjastoa hyödyntäen [22]

Blazor-komponenttien yksikkötestauksessa Technology Radar -blogi suosittelee käyttämään bUnit -kirjastoa [79]. Kyseisen kirjaston kerrotaan tuovan yksittäisten komponenttien testaukselle ympäristön, joka ei vaadi muuta sovellusta ympärilleen.

Aiemmin mainittujen komponenttikirjastojen lisäksi .NET:n kehitystiimin tekninen johtaja Adrien Torris listaa ”Awesome Blazor” -Github-sivulla komponenttikirjastoja, kuten Material Design -tyylijärjestelmän mukaiset MudBlazor, MatBlazor ja Material.Blazor, sekä lukuisia muita kirjastoja ja työkaluja [10]. Kokonaisuudessaan Blazorin ekosysteemi vaikuttaa riittävältä ERAn tarpeisiin.

### 6.5.4 Kehityksen mukavuus vs. oppimisen vaikeus

Blazor on muiden yleisten web-teknologioiden tavoin komponenttipohjainen ohjelmistokehys ja sisältää täten yhtäläisyyksiä esimerkiksi Reactin kanssa. Reactia vastaavia perusteellisia kyvykkyyksiä ovat komponenttien sisäistäminen (engl.nesting) ja uudelleenkäyttö. Blazorin komponentteja rakennetaan ERA:stakin tutulla Razor-syntaksilla, mikä helpottaa kehittäjien siirtymää nykyisestä teknologiasta Blazoriin. [10] Komponenttipohjaisuus voi tuoda intuitiivisuutta käyttöliittymien rakentamiseen suositun Reactin tai muiden komponenttipohjaisten teknologioiden oppineille, mutta nykyisen ERA-järjestelmän käyttämään ASP.NET MVC:hen verrattuna näkymät rakentuvat erilaisella periaatteella. Blazorin käyttämää C#-kieltä pidetään StackOverflown vuoden 2024 kyselytuloksiin perustuen houkuttelevampana kuin muissa käyttöliittymäteknologioissa

käytettyä JavaScriptiä [70].

Käyttöliittymäkoodin debuggaus Blazorin Interactive Server -moodilla onnistuu kätevästi esimerkiksi Visual Studiossa, koska koodi ajetaan täysin palvelimen puolella [10]. Täten erillisiä kehitystyökaluja ei tarvita selaimessa suoritettavan koodin debuggaukseen, mikä sujuvoittaa kehitysprosessia.

### 6.5.5 Hyvien koodauskäytäntöjen tuki ja rakenteet

Microsoftin omien Blazor-dokumentaatioiden lisäksi Awesome Blazor -sivusto listaa kattavan määrän Blazorille saatavaa tukimateriaalia, kuten ohjeartikkeleita ja tutoriaaleja [6].

### 6.5.6 Muuta huomioitavaa

Blazor on tämän tutkimuksen kannalta erityisen mielenkiintoinen teknologia, koska myös ERA on rakennettu .NET-ohjelmistokehyksellä. Tämä tuo merkittäviä synergiaetuja niin nykyiseen sovellukseen kuin sen kehittäjillekin. Sama (rajoitettu) .NET-ekosysteemi toimii Blazorissa ja nykyisessä .NET-sovelluksessa. Kehittäjät voivat käyttää tuntemaansa C#-kieltä ja hyväksi todettuja kirjastoja. Nykyisen palvelinpuolen ohjelmiston logiikkaa voi myös suoraan hyödyntää Blazorissa. [10]

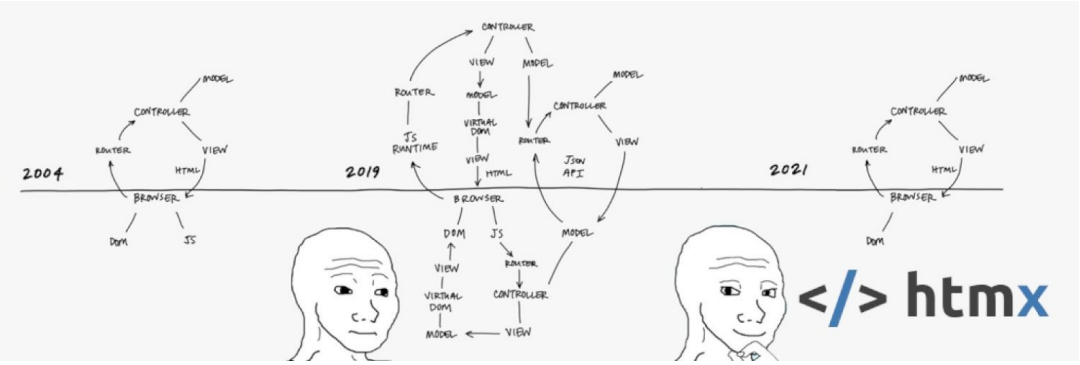
## 6.6 Muut teknologiavaihtoehdot

Tässä kappaleessa käydään läpi vielä muita teknologiavaihtoehtoja, jotka omaavat potentiaalia tai mielenkiintoisia ominaisuuksia, mutta tietystä selkeästä syystä voidaan sulkea pois syvemmästä vertailusta.

### 6.6.1 HTMX

HTMX on kevyt HTML:n toiminnallisuutta laajentava ”Domain Specific Language” (DSL), joka sopii yksinkertaisiin käyttöliittymiin ilman kokonaista verkkoohjelmistokehystä. Perinteinen JavaScript ja jQuery ovat kykeneväisyyksiltään hyvinkin samankaltaisia HTMX:n kanssa, joten vain osa ERAn tavoitteista täyttyisi; jQurystä päästäisiin eroon, mutta kehittyneempien komponenttien ja toiminnallisuuksien käyttömahdollisuudet eivät juuri paranisi. HTMX voi olla hyvinkin yksinkertaista ja luettavaa HTML:stä tutun syntaksin ja vahvojen abstraktioiden ansiosta. Tätä havainnollistaa kuva 6.5 Lisäksi JavaScriptiä ei ole pakko syrjäyttää kokonaan, vaan HTMX:ään voi lisätä toiminnallisuutta esimerkiksi kustomoiduilla eventtien kuuntelijoilla, kuten koodiesimerkissä 6.1 [29]

HTMX vähentää tarvetta erilliselle JavaScript-koodille sisällyttämällä toiminnallisuutta suoraan HTML-elementtien attribuutteihin. Yksi attribuutti voi määrittää HTML-elementin sisällön toteuttaen AJAX-kutsun taustalla. AJAX-kutsujen lisäksi HTMX:n avulla voi toteuttaa esimerkiksi CSS-siirtymiä ja kuunnella WebSocket-yhteyksiä. [33]




**Kuva 6.5:** Htmx:n periaatteita havainnollistava meemi [33].

```
1 document.body.addEventListener('htmx:afterSwap', function(event) {
2   console.log('Content has been swapped!');
3 });
```

**Ohjelma 6.1:** Esimerkki HTMX:n tarjoamasta mahdollisuudesta sisällyttää JavaScript-koodia. [29]

Teknologioiden kypsyysastetta arvioiva Technology Radar -blogi on arvioinut HTMX:n olevan harkittavissa yrityskäyttöön, mutta luokitus kehottaa arvioimaan käyttöönottoa harkitusti [79]. Blogi mainitsee HTMX:n tuovan vanhojen monen sivun sovellusten tyylistä yksinkertaisuutta verkkokäyttöliittymien kehitykseen, kuten myös kuva 6.5 havainnollistaa. Technology Radar huomauttaa myös teknologian rajoitetusta kokoluokasta, jonka takia se voidaan todeta toiminnallisuuksiltaan liian vajaaksi ERAn vaatimuksiin nähden.

### 6.6.2 ReactJS.NET

ReactJS.NET on Facebookin kehittämä integrointityökalu, joka mahdollistaa React-komponenttien hyödyntämisen suoraan ASP.NET-sovelluksen palvelimella renderöitynä. [61]

ReactJS.NETin viimeisin päivitys on julkaistu kesäkuussa 2020. Täten sen ylläpito on lopetettu Facebookin toimesta, ja työkalua täytyisi kehittää ja ylläpitää itse, mikäli se otetaan käyttöön. Projektin GitHub-repositorionistaakaan ei löydy aktiivisesti ylläpidettyjä kolmannen osapuolen fork-toteutuksia, jotka jatkaisivat työkalun ylläpitoa. [62] Tämä ratkaisu toimiikin inspiraationa kappaleessa 7 esitettyyn teknologiaratkaisuun, jossa modernin web-teknologian komponentteja renderöidään suoraan ERAn palvelimella.

### 6.6.3 Modern.js

Modern.js on Next.js:n tavoin Reactin päälle rakennettu ohjelmistokehys. Modern.js on luonteeltaan progressiivinen, eli se tuo valmiita lisäominaisuuksia täysin vapaaehtoisena lisänä tavalliseen React-kehitykseen. Modern.js eroaa Nextistä myös suhtautumiseltaan palvelinpuolen renderöintiin, sillä Next käyttää palvelinpuolen renderöintiä oletuksena jokaiselle komponentille, toisin kuin

Modern.js. Modern.js profiloituukin ylipäätään progressiiviseksi ohjelmistokehykseksi, eli se tukee Reactin luontaista toimintaa ja tuo vapaaehtoisia lisätoimintoja Reactin päälle. [34] Tämä lähtökohta auttaa välttämään ongelmat myös Module Federation -tekniikkaa ajatellen ja Modern.js on todetukin kyseistä tekniikkaa tukevaksi ohjelmistokehykseksi tekniikan kehittäjän Zack Jacksonin puolesta [36]. Modern.js tukee myös tämän tutkimuksen kannalta mielenkiintoista arkkitehtuurikaavaa käyttöliittymää palvelevalle palvelinohjelmistolle, josta käytetään lyhennettä BFF (engl. Backends for Frontends).

### 6.6.4 Elm

Elm on Svelten tavoin oma kielensä, joka muutetaan JavaScriptiksi sovelluksen käännösprosessissa. Teknologialla on oma syntaksinsa, johon ohjelmistokehyksen toiminta perustuu. Syntaksi on hyvin erilaista verrattuna JavaScriptiin tai esimerkiksi Svelteen. Elm pohjautuu funktionaaliseen ohjelmointiparadigmaan ja lupaa havaitsevansa kaikki virheet käännösprosessissa, jolloin ajon aikana ei voi tulla yllätyksellisiä poikkeustilanteita. Elm mainitsee myös vahvuudekseen kuvainnolliset virheviestit. Myös tyyppimerkintöjen käyttö on tuettua, mikä tuo kielelle TypeScriptin hyviä puolia ja parantaa virheviestejä entisestään. [19]

Mielenkiintoinen ominaisuus Elm-teknologiassa on JavaScript-yhteensopivuus, joka mahdollistaa teknologian osittaisen käyttöönoton[19]. Mahdollisuus sisällyttää JavaScriptiä Elm-sovellukseen tuo teknologialle progressiivisen ohjelmistokehyksen etuja, vaikka Elm onkin kokonaisvaltainen ohjelmistokehys itsessään. JavaScript-yhteensopivuus ei kuitenkaan ole täydellisen kattava, sillä Elm ei sisällä vastineita (FFI, engl. Foreign Function Interface) kaikille JavaScript-toiminnoille, vaan rajoittuu kehittämiinsä yhteensopivuustoimintoihin säilyttääkseen ennaltaarvattavat ominaisuutensa [19]. Yhteensopivuustoiminnot mahdollistavat puuttuvien toimintojen täydentämistä JavaScript-ekosysteemistä, mutta sovelluksen rakentaminen osittain Elm-teknologialla ja osittain JavaScriptillä vaikuttaa potentiaalisesti sekavalta suurikokoisia web-sovelluksia rakennettaessa.

Elm luottaa vahvasti omaan ekosysteemiinsä ja natiiveihin paketteihinsa ja välttää täten riskejä haavoittuvuuksille kolmansien osapuolten kirjastoista [19]. Elm:n kaltaiselle vähemmän tunnetulle teknologialle tämä voi olla myös heikkous, kun valmiita kirjastoratkaisuja on tarjolla huomattavasti vähemmän verrattuna suosittuihin teknologioihin, kuten Reactiin tai Angulariin. Stack Overflow:n teknologiakyselyissä Elm on web-teknologioiden käyttäjämäärissä täpärästi mukana listalla, kun sen kanssa on päätoimisesti työskennellyt vain 0,6 prosenttia kyselyyn vastanneista kehittäjistä [70]. Elmin rajallinen suosio, pieni ekosysteemi ja yritystuen puute ovatkin kriittisiä ominaisuuksia, joiden takia teknologia on liian riskialtis ERA:ssa käytettäväksi. Elm-teknologialle olisi vaikea löytää osaajia, ja teknologian kehitys näyttää jo nyt hyvin passiiviselta, jos ei jopa lopetetulta, teknologian GitHub-repositorioiden perusteella [18].

# 7 Arkkitehtuurianalyysi

Pääperiaatteena ERAn uuden käyttöliittymän rakentamisessa on syrjäyttää vanha järjestelmä hiljalleen ilman merkittävää yksittäistä refaktorointia vaativaa harppausta. Tavoitteena on ohjata kehitystä yksittäinen käyttöliittymätarve kerrallaan kohti uutta teknologiaratkaisua ja tilaisuuden tullen refaktoroida vanhoja näkymiä uuden sovelluksen puolelle. Kun uutta käyttöliittymää on rakennettu uudelleenkäytettäviä komponentteja hyödyntäen, voidaan valmista toiminnallisuutta käyttää vanhojen näkymien korvaamiseen uudella teknologialla, jolloin kynnys muutokselle pienenee. Tässä luvussa analysoidaan mahdollisia arkkitehtuuriratkaisuja uudistetulle, kahta eri käyttöliittymäteknologiaa hyödyntävälle ERAn käyttöliittymäsovellukselle. Lähestymistapa mukailee palvelinsovelluksille kehitettyä teknologiamigraation kuristusmallia (eng. Strangler pattern) [88].

Lähtöasetelma arkkitehtuurisuunnitelmalle sisältää micro-frontend-periaatteen mukaisia piirteitä, sillä micro-frontend-teknologiat yhdistelevät yksittäisiä web-sovelluksia yhtenäiseksi käyttöliittymäksi [78]. Täten tuleva ERAn arkkitehtuuri on micro-frontend-arkkitehtuurien erikoistapaus, jossa erillisiä sovelluksia on kaksi. Arkkitehtuurin kannalta käyttöliittymät voivat silti olla omia erillisiä sovelluksiaan, jolloin käyttöliittymien yhdistyminen tapahtuu käyttäjän näkökulmasta selaimen uudelleenohjauksilla. Kaksi arvioitavista arkkitehtuuriratkaisuista noudattaa tätä periaatetta, jolloin käyttäjälle näyttäytyy vertikaalijaon mukainen mikro-frontend-käyttöliittymä. Kolmas käyttöliittymäarkkitehtuuri rakentuu nykyisen sisälle, jolloin käyttöliittymä noudattaa micro-frontendien horisontaalijakoa.

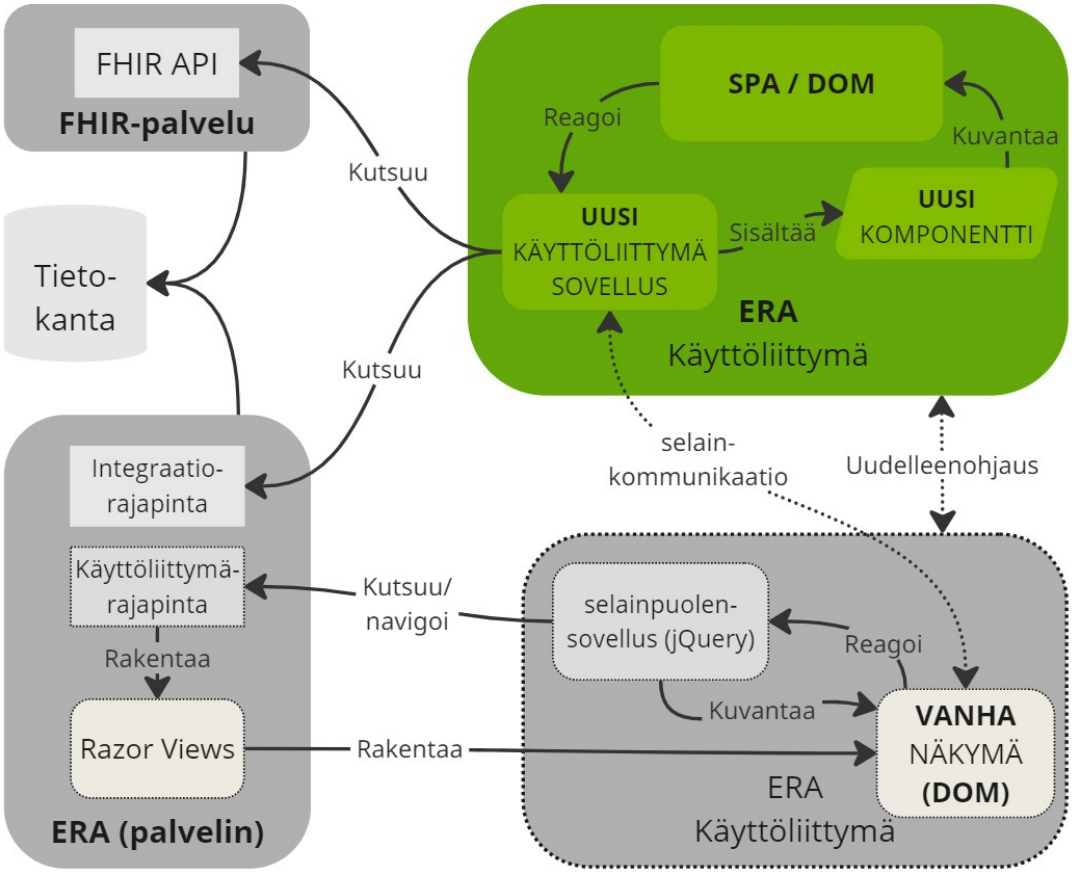
Datan hallinnan osalta sovellukset voivat synkronoitua nykyisellä lähtökohdalla tietokannan ja ERAn rajapintojen avulla tai selaimen tiedon tallennusrajapinnoilla, kuten evästeet ja localStorage. Uusi arkkitehtuuriratkaisu muovaa mahdollisuuksia käyttöliittymän tietojen synkronointiin, joten eri arkkitehtuurivaihtoehtojen dataintegraatiotyyli on ensimmäinen kriittinen arviointikohde erillisille arkkitehtuuriratkaisuille. Arkkitehtuurien mahdollistamien tiedonvälityskeinojen vaikutuksia arvioidaan erityisesti koodin huollettavuuteen ja kehityksen tehostumiseen keskittyen. Nämä ominaisuudet vaikuttavat tutkimuksen **kehitykseen** ja sen **hallintaan** liittyviin **tarpeisiin**. Toinen arvionnissa huomioitava lähtökohta on arkkitehtuurien tuomat tekniset mahdollisuudet esimerkiksi suorituskyvyn optimointiin tai käyttöliittymän kaupalliseen jakeluun. Nämä mahdollisuudet vastaavat pääosin tutkimuksen **kaupallisiin tarpeisiin**. Esimerkiksi erillinen JavaScript-sovelluskerros käyttöliittymän ja ERAn palvelinsovelluksen välillä mahdollistaa optimointia tiedon lataamiseen ja käyttöliittymän esittämiseen, mutta myös komponenttien jakamiseen muihin sovelluksiin esimerkiksi Module Federation-tekniikan avulla [78].

Arkkitehtuurivaihtoehtojen pelkistetyt versiot esitetään luvun 5 kuvissa 5.1 (A-malli), 5.2 (B-malli), ja 5.3 (C-malli). Näihin verrattavana ERAn nykyinen arkkitehtuuri hahmotellaan samaan tyyliin kuvassa 4.1. Seuraavat alaluvut esittävät kuvista yksityiskohtaisemmat versiot, jotka kuvastavat vastuunjakoa arkkitehtuurimallien eri osien välillä.

## 7.1 Malli A: Uusi käyttöliittymä yhden sivun sovelluksena

Malli A lisää uuden käyttöliittymäsovelluksen erillisenä yhden sivun sovelluksena, jota jaellaan nykyiseltä palvelinsovellukselta. Mallissa käyttöliittymäkoodilla ei ole täten vaaraa sekoittua palvelinpuolen ohjelmaan, koska se suoritetaan fyysisesti eri sijainneissa. Kuva 7.1 havainnollistaa käyttöliittymäteknologian suoraa integroitumista kahteen erilliseen palvelinsovellukseen.

Uuden selaimen puolelle sijoittuvan SPA-käyttöliittymän periaatteena on erottaa datan esityskerros liiketoimintalogiikasta ja hyödyntää nykyisiä järjestelmäintegraatiorajapintoja. SPA keskustelee sekä vanhojen XML-pohjaisten integraatiorajapintojen että uusien FHIR-rajapintojen kanssa. Näin saadaan nopea, näkyvä parannus käytettävyyteen ja komponenttien uudelleenkäyttöön ilman uuden palvelinsovelluksen rakentamista.




**Kuva 7.1: Vaihtoehto A:** Uusi yhden sivun sovellus nykyisillä integraatiorajapinnalla.

### 7.1.1 Hyödyt

Erillinen selainpuolen sovellus erottelee palvelinpuolen rajapintasovelluksen käyttöliittymäsovelluksesta. Täten sovellukset ovat kooditasolla erillisiä kokonaisuuk-

sia, mutta ne toimivat saman prosessin palvelemana. Erilliset teknologiat poistavat mahdollisuuden käyttöliittymäkoodin sekoittumisesta rajapintatoimintoihin, mutta ratkaisu mahdollistaa keskitetyn kehittäjäkokemuksen yhden repositorion alla.

Ratkaisun toteuttamiseen löytyy virallista Microsoftin tukimateriaalia, ja kynnys aloittamiselle on pienempi kuin seuraavassa alaluvussa 7.2 arvioidulla B-arkkitehtuurilla. Arkkitehtuurimalli ei tarvitse erillistä palveliohjelmistokomponenttia, jonka toteutuksen yksityiskohdista täytyy huolehtia.

Yhden sivun sovelluksena käyttöliittymä pysyy palvelimella staattisena JavaScript-tiedostona, jolloin nykyinen ERA-infrastruktuuri voi säilyä ennallaan. Microsoftin Visual Studio 2022 -kehitysympäristö tarjoaa mallipohjia .NET 8 -palvelinsovellukseen liitettäville JavaScript- ja TypeScript-käyttöliittymille [5]. Nämä toimivat yksinkertaisena esimerkkinä ja pohjana modernin yhden sivun sovelluksen lisäämiselle vanhan käyttöliittymän rinnalle. ERAn palvelinohjelmiston .NET-projektiin täytyy vain konfiguroida uudelle käyttöliittymälle määritettävän osoitteen mukainen reititys ja kehityspalvelin. Nämä toiminnallisuudet onnistuvat .NET-teknologiaan tukeutuen [5].

### 7.1.2 Haitat

Haittana A-mallin arkkitehtuurissa on erillisten integraatiorajapintojen käytöstä aiheutuva monimutkaisuus käyttöliittymäsovelluksessa. Täysin modulaarisessa arkkitehtuurissa käyttöliittymäsovellus keskittyy vain tiedon esittämiseen ja muokkausten vastaanottamiseen, mutta A-arkkitehtuuriratkaisussa tiedon välityksen yksityiskohdat ERAn oman XML-rajapinnan ja uuden FHIR-rajapinnan välillä vaativat ylimääräistä orkestrointia. Tämä asettaa huollettavuuden kannalta vaatimuksia käyttöliittymäsovelluksen sisäiselle arkkitehtuurille, mikä lisää riskiä teknisen velan uudelleensyntymiselle.

Uusien komponenttien julkaiseminen yhden sivun sovelluksessa ei onnistu ilman koko näkymän pohjan rakentamista. Täten teknologiasiirtymän aloittaminen vaatii sovelluksen sisäistä arkkitehtuurisuunnittelua ja täten suuren kehitysaskeleen. Yksittäisiä elementtejä ei myöskään voi julkaista väliaikaisiksi parannuksiksi vanhaan käyttöliittymään. Nämä ominaisuudet vaikeuttavat teknologiasiirtymän vaiheittaista toteuttamista.

### 7.1.3 Mahdollisuudet

A-mallia täydentävänä mahdollisuutena nähdään väliaikainen vanhan käyttöliittymän ehostaminen uuden käyttöliittymän komponenttien avulla. Kuten luvun 2 business-tarpeissa huomioitiin, ERAn käyttöliittymän osittainen jakaminen nähdään Atostekilla myös liiketoiminnan kannalta tärkeänä mahdollisuutena asteittaiseen käyttöliittymätoimittajan rooliin sopeutumiseen.

ERAn käyttöliittymän yksittäisiä näkymiä jaetaan jo nykyään asiakkaiden sovelluksiin. Tämä tehdään HTML-standardiin kuuluvien iFrame-elementtien avulla [65], jotka ikään kuin upottavat selaimen sisälle toisen selainnäkymän. Sovellukset ajetaan silti erillisissä konteksteissaan, joten yhteistyö iFrame-elementin ja ulomman

sovelluksen välillä on rajoitettua. Teknisenä vaihtoehtona iFrame-upotuksille Module Federation kykenee käyttöliittymien upottamiseen JavaScript-moduulien tasolla, mikä mahdollistaa yhdin sivun sovelluksen osittaisen jakelun muihin sovelluksiin [47].

#### Module Federation

Module Federation on JavaScript-sovelluksen integrointitekniikka, joka mahdollistaa micro-frontend-periaatteen toteuttamisen dynaamisesti käyttöliittymän ajon aikana. Module federation-tekniikan avulla käyttöliittymäkoodin moduuleita, kuten kirjastoja ja komponentteja, voidaan ladata juuri tarpeen hetkellä toisesta käyttöliittymäsovelluksesta. Käyttöliittymien yhdistyminen dynaamisesti ajonaikana mahdollistaa käyttöliittymien osien päivittämisen itsenäisesti, mikä auttaa erityisesti useiden tiimien osuuksia sisältävän käyttöliittymän julkaisuprosesseissa [47].

Module Federation eroaa muista micro-frontend-tekniikoista siinä, että se yhdistää sovellukset JavaScript-ohjelman tasolla, jolloin toisen sovelluksen ohjelmakoodi latautuu riippuvuuksineen JavaScript-moduulina pääsovellukseen. Pääsovellusta, joka hyödyntää moduuleita ulkoisista sovelluksista, kutsutaan tekniikassa englanniksi nimellä ”Host”. Ulkoisia sovelluksia tai niiden osia kutsutaan nimellä ”Remote”. Host-sovellukseen ladataan Remote-moduuli ajonaikana aivan kuten se olisi JavaScript-tiedostoon ladattu kolmannen osapuolen kirjastokomponentti. Erona tavalliseen kirjastokomponenttiin, Remote-moduuli ladataan Module Federation -konfiguraation mukaisesti erikseen tuotannossa olevasta sovelluksesta, kun taas kirjastokomponentti niputetaan osaksi Host-sovellusta jo sen kasausprosessissa tai ladataan Host-sovelluksen omalta palvelimelta dynaamisesti. [35] Module Federation mahdollistaa myös kaksisuuntaiset riippuvuudet, eli esimerkiksi Host-sovelluksen sisältämä nappi-komponentti voi olla osana remote-moduulia, joka ladataan ajonaikana Host-sovellukseen [48].

Module Federation -tekniikan kehittäminen aloitettiin vuonna 2017 [47]. Tekniikan nuori ikä herättää epäilyksiä riittävästä tekniikan kypsyysasteesta ja vakaudesta. ERAan ei haluta sisällyttää kriittisiä arkkitehtuuriominaisuuksia, jotka saattavat vanhentua muutaman vuoden kuluessa. Hypoteesia vahvistaa Module Federationia tukevien meta-ohjelmistokehysten vajaa tarjonta [35]. Koska Module Federation on matalan tason konfiguraatiota koskeva tekniikka, meta-ohjelmistokehykset toteuttavat päällekkäisiä toimintoja moduulien lataamiseen ja paketointiin liittyen. Täten Module Federation voi olla vielä liian nuori teknologia tehokkaaseen hyödyntämiseen kehityksessä. Näitä epäilyksiä vahvistaa esimerkiksi Next.js-teknologian vanhentunut tuki Module Federationille [36]. Vaikka Module Federation ei ole itsessään tiettyyn teknologiaan sitoutunut tekniikka, korkean tason metakehykset vaativat ylläpitäjiltään yhteistyöhalukkuutta ja työpanosta toimiakseen sujuvasti Module Federationin kanssa. [36]. Tämän tutkimuksen teknologiavalinnalla on täten vaikutuksia Module Federationin käyttömahdollisuuksiin. Toisaalta Module Federation 2.0:n dokumentaatiosivusto, module-federation.io, esittää Showcase-sivullaan hyvinkin tunnettujen yritysten sivustoja, kuten Epic Games, BestBuy ja Adidas, jotka luottavat Module Federation -tekniikkaan sivustoillaan [47]. Teknologian esiintyminen suuren kokoluokan

tuotantokäytössä lisää tekniikan luotettavuutta yrityskäytössä.

Module Federation -ratkaisun haittapuolena on sen vaatima matalan tason konfiguraatio, joka on toteutettu valmiiksi esimerkiksi meta-ohjelmistokehyksillä, kuten Next.js [51]. Konfiguroinnin vaatima lisätyö ei ole ainoa ratkaisun heikkous. Itse rakennettu matalan tason konfiguraatio voi olla virhealtista ja kehittäjälle vaikeaselkoista. Moduulien hienojakoinen käsittely vaatii syvällistä ymmärrystä JavaScript-moduulien käsittelystä ja käytetystä bundler-työkalusta. Yhden sivun sovelluksen moduuleita jaettaessa Module Federation voi kuitenkin olla kohtuullisen yksinkertainen ja hyödyllinen ratkaisu.

#### Web Component -rajapinta

Web Component -rajapinta toimii ikään kuin standardin mukaisena selaimiin integroituna kevyenä web-käyttöliittymäteknologiana. Web Component-rajapinta mahdollistaa komponenttien toteuttamisen kustomoituna HTML-elementtinä [9]. Web Component -rajapintaa tukee tärkeimmät selainversiot, kuten kaikki Angularia tukevat selaimet [3].

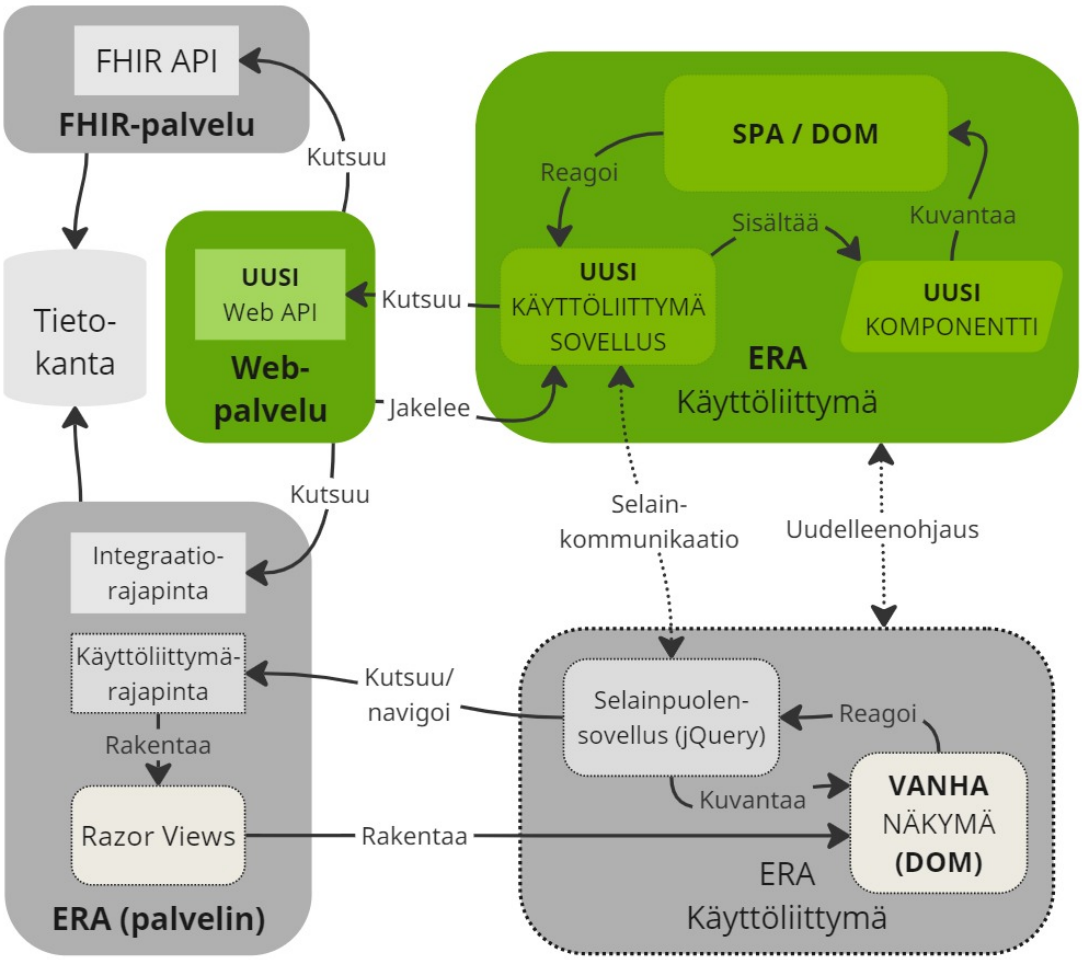
Web Component -rajapinnan uniikki mahdollisuus on tehdä käyttöliittymäkomponenteista itsenäisiä kokonaisuuksia, joita voi käsitellä html-elementtien tavoin. Täten spesifikaatio abstrahoi elementin toteuttamiseksi vaaditun JavaScript-teknologian yhtenäisen rajapinnan alle, mahdollistaen eri teknologioilla rakennettujen komponenttien yhdistelyn saumattomasti selainympäristöön nojaten.

## 7.2 Malli B: Uusi palvelinrajapinta ja uusi yhden sivun sovellus

Kaaviossa 7.2 esitetään hahmotelma kokonaisen uuden web-palvelukerroksen lisäävästä arkkitehtuuriratkaisusta. Tässä tapauksessa uudella käyttöliittymällä on erillinen web-sovelluksen palvelemiseen keskittyvä ohjelma, jolla voidaan hallinnoida datan käsittelyyn keskittyvän integraatiorajapinnan kutsumista ja mahdollisesti myös käyttöliittymän kuvantamista. Erillinen uusi palvelinohjelmisto toteuttaisi käyttöliittymää palvelevan rajapinnan yhtenäistämisen ”Facade” -tyylisesti. Palvelinohjelmisto voi toimia käänteisen välityspalvelimen tavoin, abstrahoiden eroavaisuudet kommunikoinnissa vanhan xml-integraatiorajapinnan ja uuden FHIR-rajapinnan kanssa. Uusi palvelinohjelmisto voi toteuttaa FHIR-standardin mukaista kommunikaatiota taustalla sekä toimia adapterina XML-muotoisen datan kommunikointiin JSON-muodossa uuden käyttöliittymän kanssa. Tällöin käyttöliittymän ei tarvitse olla tietoinen sosiaali- ja terveysalan datan käsittelyn teknisistä vaatimuksista.

### 7.2.1 Hyödyt

Tämä lähestymistapa lisää erittelyä palvelinpuolen tehtävien vastuujakoon, sillä erillinen moduuli keskittyy käyttöliittymän palveluun, ja toinen toimii liiketoimintadatan käsittelijänä. Arkkitehtuurimallin merkittävänä etuna on lisääntynyt modulaarisuus, koska tehtäväjako on selkeämpi.




**Kuva 7.2: Vaihtoehto B:** Yhden sivun sovellus uudella palvelinpuolen rajapintakerroksella.

Erillinen web-rajapintapalvelu poistaa heti teknologiasiirtymän alkuvaiheessa uudelta käyttöliittymältä vastuuta käyttöliittymälle epäolennaisista toiminnoista. Eri tietomallien yhdistelyä ei tarvitse kehittää jokaiselle näkymälle uudelleen, kun se tapahtuu keskitetysti palvelinsovelluksessa.

### 7.2.2 Haitat

Ratkaisu tuo lisää abstraktiota käyttöliittymän rakentamiseen, mutta abstraktio ei synny itsestään. Jotta saavutetaan arkkitehtuurin tavoittelema modulaarisuusetu, täytyy käyttöliittymälle kehittää uusi yhteinäinen tietomalli, joka palvelee dataa käyttöliittymäystävällisessä muodossa. Tietomallin suunnittelu vaatii erillistä työtä, jotta se mukautuu sekä liiketoimintadatan sisältöön keskittyviin FHIR-tietomalleihin, että myös ERAn omaan järjestelmärajapintaan, joka sisältää CDA R2-määritelmää mukailevien tietokenttien lisäksi tekniseen toteutukseen liittyviä tietomalleja.

### 7.2.3 Mahdollisuudet

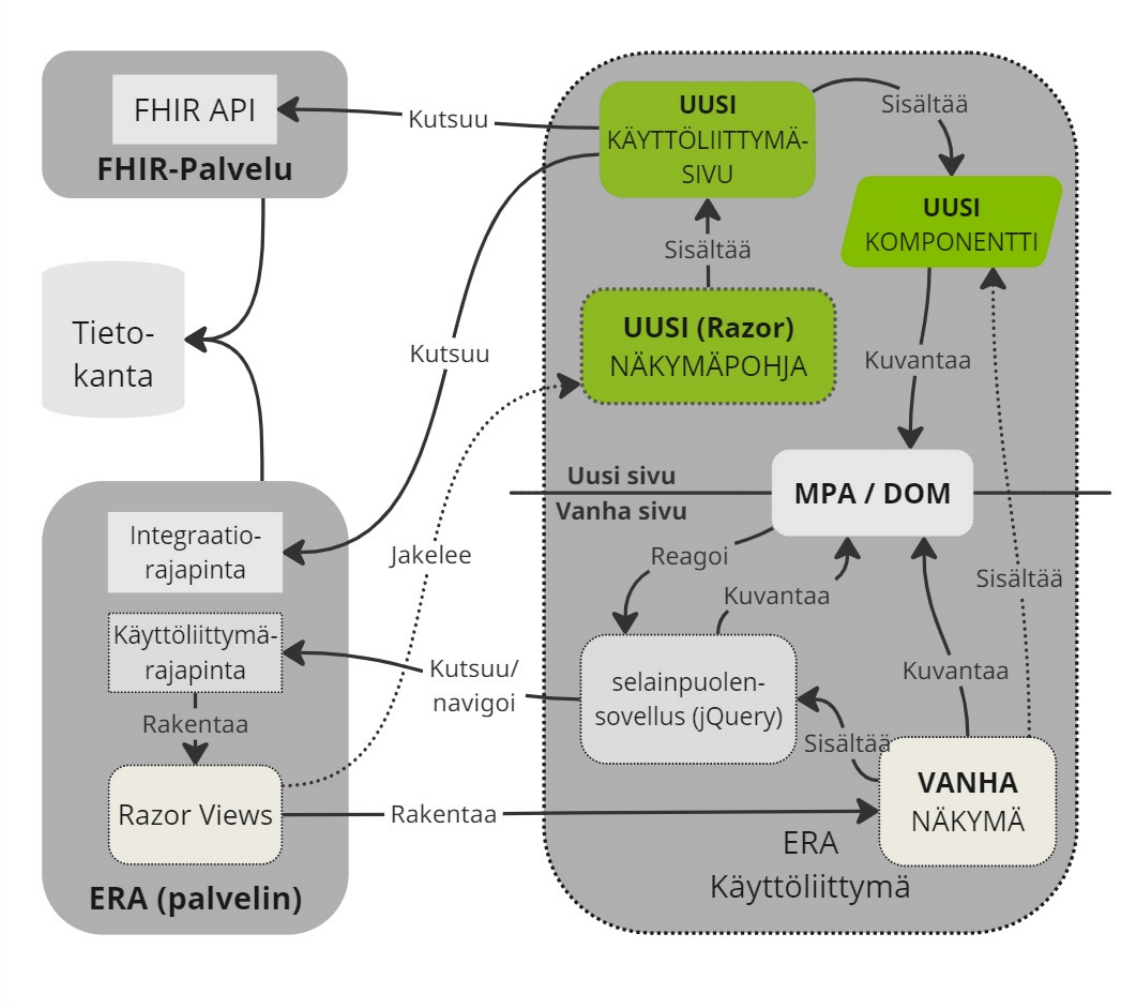
B-mallin arkkitehtuurin erillinen palvelinpuolen ohjelmistokomponentti tuo mahdollisuuden esimerkiksi GraphQL-arkkitehtuurin rakentamiseen. Käyttöliittymää palveleva ohjelmistokomponentti voi toimia GraphQL-palvelimen roolissa, jolloin käyttöliittymäsovellus pystyy hienojakoiseen datan kommunikointiin GraphQL-client -roolissa. GraphQL mahdollistaa vain tarvittavan tiedon välittämisen, kuten pelkän potilaan ikätiedon lataamisen ilman, että täytyy ladata koko potilaan tietorakenne käyttöliittymään. [12] GraphQl-teknologia ei ole itsessään korkean tason arkkitehtuuriratkaisu, joten sitä ei käsitellä tutkimuksessa laajemmin. GraphQL nähdään mahdollisen etuna, jonka B-malli mahdollistaa.

B-arkkitehtuurin uusi web-palvelu voi olla esimerkiksi JavaScript-sovellus, jolloin palvelu tarjoaa samalla ajoympäristön palvelinpuolen kuvantamista toteuttavalle käyttöliittymälle [11]. Toisaalta tiedon esittämiseen liittyvän koodin tuominen palvelinpuolelle palauttaa samalla nykyisen ERA-sovelluksen riskin käyttöliittymäkoodin ja rajapintatoiminnallisuuden yhteen kasvamisesta. Tämän välttämiseksi JavaScript-palvelussa täytyisi varmistaa selkeästi eroteltu ohjelman sisäinen arkkitehtuuri. JavaScript-koodin täytyisi myös olla isomorfista, eli suoritettavissa sekä palvelimella, selaimella [11]. Palvelinpuolen kuvantamisen tärkeimmät edut ovat käyttäjäkokemuksen parantuminen ja hakukoneoptimoint, jotka eivät ole ERAn teknologiasiirtymän kannalta keskeisiä. Täten palvelinpuolen kuvantamista ei huomioida arkkitehtuurisuunnittelussa tarkemmin, vaan se nähdään tulevaisuuden mahdollisuutena.

## 7.3 Malli C: Käyttöliittymän rakentaminen nykyisen sisälle

Kolmas tutkimuksessa analysoitu käyttöliittymän uusimistapa muistuttaa vaihtoehdoista eniten tavallista ohjelmiston refaktoroimista, sillä siinä muutokset ja lisäykset tehdään nykyiseen palvelinohjelmistoon sisäisesti. Arkkitehtuurivaihtoehdon tutkimista perustelee hypoteesi, jonka mukaan vaadittu työmäärä teknologiasiirtymän aloittamiselle on pienin, kun käyttöliittymäsovelluksen runko on jo olemassa eikä uuden sovelluksen pystyttämistä vaadita. Kaaviossa 7.3 esitellään hahmotelma esimerkkiarkkitehtuurista, jossa uutta käyttöliittymää hyödynnetään osana nykyistä käyttöliittymää.

Vanhan pohjan päälle rakentamisessa ja asteittaisessa käyttöliittymäkoodin korvaamisessa ASP.NET MVC:n ja Razor-näkymien toimintaperiaatteet rajoittavat uuden käyttöliittymän integrointimahdollisuuksia. Palvelinpuolen .NET-ympäristö ei tue sellaisenaan JavaScript-käyttöliittymäkoodin suorittamista, joten uusi käyttöliittymä on integroitava joko selaimessa tai esimerkiksi palvelinpuolen erillisellä JavaScript-ajoympäristöllä, kuten Node.js:llä [59]. Yksinkertaisimmillaan uuden käyttöliittymän sovelluskoodi ja sen vaatimat ohjelmistokehys-riippuvuudet voidaan liittää Razor-näkymän tuottamaan HTML-lomakkeeseen, jolloin selain suorittaa uuden käyttöliittymäosuuden kuvantamisen. Malli-C:tä tutkitaan oletuksella selainpuolen transkluusiolla toteutettuna.




**Kuva 7.3: Vaihtoehto C:** Vanhan käyttöliittymän sisään rakentuva arkkitehtuurimalli.

### 7.3.1 Hyödyt

Vanhan käyttöliittymän sisälle rakentuva uusi käyttöliittymämalli C tarjoaa pienen kynnyksen teknologian uusimisen aloitukselle. Kehitystyön aloitus vaatii teknisesti ainoastaan uuden käyttöliittymäteknologian asentamisen vanhan käyttöliittymän saataville. Tällöin uuden teknologian käyttönotto on nopeaa vanhoja käyttöliittymiä ehostamalla, kun toiminnallisuutta voi korvata pieni määrä kerrallaan.

Malli C mahdollistaa myös uudelleenohjauksien välttämisen käyttöliittymien välillä, jolloin uusi ja vanha sovellus yhdistyvät saumattomasti. Käyttöliittymän tilallisten tietojen synkronointia toisen sovelluksen kanssa ei tarvitse ratkaista kun sovellukset pysyvät konkreettisesti samana.

### 7.3.2 Haitat

Ratkaisussa nähdään kuitenkin selkeänä haittapuolena uuden ja vanhan sotkeutuminen samoihin näkymiin, joissa säilyy ongelma käyttöliittymäkoodin sekoittumisesta liiketoimintalogiikkaan. Tutkimuksen tarpeissa ydinasemassa mainittu koodin huollettavuus ei täten parane toivotun tehokkaasti. Toisteisen logiikan

välttämiseksi arkkitehtuuriratkaisussa pyritään erottamaan uuden käyttöliittymän Ajax-kutsut nykyisistä ERAn käyttöliittymärajapinnoista. Mikäli vanhaa käyttöliittymää palvelevien rajapintametodien päälle rakennetaan uutta logiikkaa, voi vanhaa käyttöliittymärajapintaa joutua jopa jatkokehittämään. Vanhasta käyttöliittymärajapinnasta halutaan irtaantua teknisen velan takaisinmaksamiseksi, joten kyseisen rajapinnan kehittäminen ei ole tuottavaa tavoitteen kannalta.

Jokainen näkymä joutuu lataamaan yhteiset riippuvuudet uudelleen, kuten käyttöliittymän ohjelmistokehyksen, kun käyttöliittymä säilyy monen sivun sovelluksena. Eri näkymien välillä siirtyessä koko sivun HTML-runko vaihtuu ja yleisesti tarvittua JavaScript-koodia täytyy ladata selaimeen uudelleen [66]. Tämä ei ole optimaalinen tapa hyödyntää yhden sivun sovelluksiin suunniteltuja JavaScript-teknologioita.

Lopullinen irtaantuminen vanhasta teknologiasta tulee olemaan arkkitehtuuriratkaisussa suurempi kehitysaskel verrattuna muihin arkkitehtuurivaihtoehtoihin. Nykyiset Razor-näkymät rakentuisivat uusien näkymien pohjiksi, jolloin uusi ”kuorisovellus” käyttöliittymälle vaaditaan lopulta teknologiasiirtymän loppuvaiheessa. Täten ratkaisu säilyttää teknistä velkaa muita ratkaisuja pidempään.

### 7.3.3 Mahdollisuudet

C-mallin arkkitehtuuriin antoi inspiraatiota ReactJS.NET-teknologia, joka mahdollistaa React-komponenttien kuvantamisen ja yhdistämisen ASP.NET MVC:n näkymiin palvelimella [61]. Kyseinen teknologia on kuitenkin vanhentunut, mutta sen toimintaperiaatetta olisi mahdollista hyödyntää esimerkiksi muokkaamalla teknologian avointa lähdekoodia pohjana omalle toteutukselle.

## 7.4 Sovellusten sisäiset arkkitehtuuriratkaisut

Valittu korkean tason arkkitehtuuri vaikuttaa sovellusten sisäisiin toimintoihin, kuten käyttöliittymän tilan hallintaan ja reititykseen. Reititys tapahtuu yhden sivun sovelluksissa virtuaalisesti JavaScript-koodin ohjaamana, kun taas ERAn kaltaisessa monen sivun sovelluksessa yksittäiset reitit vastaavat erillisiä html-dokumentteja, jotka korvaavat edellisen käyttöliittymässä esitetyn näkymän [4], [58].

### 7.4.1 Tietojen synkronointi ja tilan hallinta

ERA toteuttaa asiakastietojen käsittelyä sosiaali- ja terveydenhuollossa yhdelle potilaalle kerrallaan. ERA ylläpitää asiakaskohtaista istuntotietoa tietokannassaan, ja nykyinen integraatiorajapinta sisältää jo valmiudet istuntokohtaisten tietojen käsittelyyn. Tätä varten uusi käyttöliittymä vaatii vain kyvykkyyden käsitellä XML-muotoisia tietorakenteita JavaScript-sovelluksessa. ERA-järjestelmään on myös kehittymässä JSON-muotoisen FHIR-datan käsittelyä vaativa rajapinta, joten tiedonhallinnan haasteet liittyvät palvelimen ja käyttöliittymän väliseen kommunikointiin, eikä niinkään tiedon tallentamiseen.

Käyttöliittymän tilanhallinnan kannalta nykyisen järjestelmän vaatimukset eivät ole monimutkaiset, sillä ERAn monen sivun sovelluksen käyttöliittymätyyli erottelee näkymät itsenäisiksi kokonaisuuksiksi. Näkymien välisiä monimutkaisia tilanhallintatarpeita ei ole siis päässyt kehittymään. Näkymien välisiksi globaaleiksi käyttöliittymän tilanhallintatarpeiksi tunnistetaan autentikaatiotietojen synkronointi sekä käyttäjän kielivalinnan ja muiden käyttäjäpreferenssien tallentaminen.

HTML-standardi tarjoaa mahdollisuuden säilyttää dataa selaimen Web Storage -rajapintojen avulla [65]. Web Storage ja evästeet ovat mahdollisia tilan säilytyspaikkoja selaimen sisällä. Nämä tilan säilyvyysratkaisut ovat kuitenkin lähdekohtaisia, jolloin molempien käyttöliittymien täytyy toimia saman domain-nimen alla. Tämä olisi mahdollista toteuttaa ERAn käänteisellä välityspalvelimella, joka ohjaisi käyttäjän sovelluspolun perusteella oikeaan käyttöliittymäsovellukseen. Täten käyttöliittymien osoitteet voisivat olla esimerkiksi ”era.osoite.fi/main” vanhalle ja ”era.osoite.fi/new” uudelle. Esimerkkiosoitteet ovat keksittyjä.

### 7.4.2 Reititys ja navigointi

Nykyinen ERA-sovellus käyttää monen sivun sovelluksen periaatetta, jossa käyttöliittymän reititys on palvelinpuolen ohjelmiston vastuulla. Kun käyttäjä navigoi näkymästä toiseen, käyttöliittymä lähettää pyynnön uuden näkymän osoitteeseen, joka reitittyy palvelimella kyseistä osoitetta vastaavaan Controlleriin ja rajapintametodiin. Rajapintametodi palauttaa uuden käyttöliittymänäkymän HTML-muodossa, joka sisältää myös uuden sivukohtaisen JavaScript-osuuden. Monen sivun sovelluksesta on mahdollista navigoida pois joko suoraan selaimen uudelleenohjauksella tai palvelinohjelmistolta tulevalla Redirect-vastauksella. Kolmas vaihtoehto on palauttaa kokonainen yhden sivun sovellus tietystä rajapintametodista, jolloin palvelinpuolen ohjelmiston täytyy asettaa kohdereitti parametriksi käyttöliittymän lataamiselle. Tällöin selainpuolen sovellus ottaa hallinnan sisäisestä reitityksestään ja voi käyttää samoja keinoja toiseen sovellukseen palaamiseen.

## 7.5 Arkkitehtuurin vaikutukset teknologiavalintaan

Yhden sivun sovelluksen sisältävät mallit A ja B soveltuvat pääosin universaalisti kaikkien tutkimuksessa vertailtujen JavaScript-kehysten kanssa. Angular, React, Vue ja Svelte voivat nojautua täysin selaimen JavaScript-ympäristöön, joten ne eivät aseta välttämättömiä vaatimuksia palvelinpuolen ympäristölle. Tilannetta muuttaa mahdollinen tarve käyttää Angularin tai muiden teknologioiden metakehysten sisältämiä palvelinpuolen kuvantamismahdollisuuksia suorituskyvyn optimointiin.

C-mallin mukainen vanhan käyttöliittymän sisään rakennettu uusi käyttöliittymä toimisi luontaisena ratkaisuna Blazor-teknologiaan yhdistettynä. JavaScript-kehyksistä sovellustyyliin sopii parhaiten Vue.js:n petite-vue -erityismuoto, joka on erityisesti progressiiviseen parannukseen tarkoitettu versio Vue-teknologiasta [54].

Yhden sivun sovelluksen komponenttien micro-frontend -tyylistä jakelua mahdollistava Web Component -yhteensopivuus löytyy Sveltestä, Vuesta ja Angularista. Svelte-komponentit voidaan kääntää Web Component -rajapinnan mukaisiksi ”custom elementeiksi” [71]. Vue-komponentit voidaan rakentaa Web Component APIn mukaisiksi [84]. Angular Elements toteuttaa käyttöliittymäkomponentit myös Web Component APIa noudattaen. [3].

## 7.6 Koonti

Arkkitehtuurianalyysissä vertailtiin kolmea eri tasolla integroituvaa mallia, jotka eivät kuitenkaan ole täysin toisiaan poissulkevia. Uuden yksittäisen yhden sivun sovelluksen lisäävä A-arkkitehtuurin etuina nähdään käyttöliittymän eriytymisen lisäksi toteutuksen yksinkertaisuus. Merkittävänä haittapuolena A-mallissa on palvelinpuolen rajapintojen hallinnan monimutkaisuus käyttöliittymäsovelluksessa. B-mallin web-rajapintakerroksen lisäämisessä haittapuolena on lisääntynyt työmäärä projektin aloituksessa, mutta se voidaan nähdä investointina huollettavuuteen, joka tuottaa tulevaisuuden säästöjä työmäärässä. C-mallin rakentuminen vanhan käyttöliittymän pohjalle ja erottelun puuttuminen ovat arkkitehtuuriratkaisun kriittisiä heikkouksia, mutta käyttöliittymien yhdistelyn micro-frontend-periaatetta voidaan hyödyntää pienien siirtymäaskeleiden saavuttamiseksi.

# 8 Analyysi

Teknologiavertailun tulokset konkretisoidaan määrittämällä teknologiodien tuomille hyödyille arvosanat arviointikriteereittäin. Arvosanojen avulla havainnollistetaan teknologioiden selkeimpiä etuja ja heikkouksia, jotta teknologioiden ominaisuuksia voidaan vertailla tutkimuksen tavoitteiden kannalta. Teknologioiden ominaisuuksien visuaalisena havainnollistuksena teknologioita profiloidaan teknologioiden arvioinnin kohdistamiseksi eri käyttötapauksiin sopivaksi. Arkkitehtuurimallit eivät ole toisiaan poissulkevia, joten ratkaisuja ei vastaavasti vertailla keskenään, vaan pyritään poimimaan jokaisesta tärkeimmät hyödyt lopulliseen arkkitehtuuriratkaisuun. Arkkitehtuuriratkaisun ja teknologia-arvioinnin yhteistulosten pohjalta perustellaan teknologiavalinta ja arkkitehtuurimalli ERAn käyttöliittymäuudistuksen PoC-toteutukseen.

## 8.1 Teknologiavertailun koonti

Taulukossa 8.1 teknologioiden ominaisuudet on arvosteltu asteikolla -” (ei merkittävää eroa nykyiseen), ”+” (etu nykyiseen verrattuna), ”++” (selkeä etu myös muihin vaihtoehtoihin nähden).

| **Verkkokehys** | ***React*** | ***Vue*** | ***Angular*** | ***Svelte*** | ***Blazor*** |
|---|---|---|---|---|---|
| **Suorituskyky** | + | + | + | ++ | = |
| **Stabiilius ja suosio** | ++ | ++ | ++ | + | + |
| **Ekosysteemi** | ++ | ++ | ++ | + | + |
| **Rakenteet ja skaalautuvuus** | + | + | ++ | + | + |
| **Kehittämiskokemus** | + | + | + | ++ | + |
| **Erityiset ominaisuudet** | = | + | = | = | ++ |
| **Summa** | +7 | +8 | +8 | +7 | +6 |

**Taulukko 8.1:** Teknologiavertailun koontitaulukko

Seuraavissa alaluvuissa kutakin koontitaulukon kategoriaa selitetään arvosanoihin vaikuttavien arviointikriteerien osalta ja perustellaan kunkin teknologian saamat arvosanat kategorioittain.

### 8.1.1 Suorituskyky

Suorituskykyvertailussa merkittävin etu on tutkimuksen pohjalta Svelte-teknologialla. Svelte erottuu muista vertailukohteista selaimessa ajettavan koodin tehokkuuden ja pienen koon ansiosta. Svelte tekee ohjelmistokehyksen abstraktiomäärän vaatiman prosessointityön käännöksen aikana ja optimoi koodia valmiiksi, jolloin sovellusta käytettäessä ohjelmistossa on mahdollisimman vähän ylimääräisiä kerroksia. Vähäinen koodin määrä käyttöliittymän

toiminnallisuuteen suhteutettuna nopeuttaa koodin suorittamista ja lataamista käyttäjän selaimelle. SvelteKit-metakehyksen tuomat mahdollisuudet prosessointitaakan ja tiedon lataamisen optimointiin ansaitsevat Sveltelle absoluuttisen suorituskykyetunsa tukemana kaksinkertaisen ”++-arvosanan vertailussa.

Blazor puolestaan on suorituskyvyllisesti vertailukohteista heikoin. Blazor vaatii kompromissin suorituskyvyn eri osa-alueiden välillä. Blazor WebAssembly aiheuttaa hitaan ensilataamisen ja suuren sovelluskoon selaimessa, mutta riittävän responsiivisuuden käyttöliittymän interaktioissa. Blazor Server puolestaan pystyy nopeaan ensilataukseen, mutta interaktiivisuuteen aiheutuu viivettä verkkoliikenteestä, ja käyttöliittymän tilan sekä verkkoyhteyden ylläpito aiheuttaa taakkaa palvelimelle. Blazor pyrkii tarjoamaan valmiin kompromissin Blazor Auto -renderöintimallillaan, mutta tämä aiheuttaa kompleksisuutta käyttöliittymän rakentamiseen, sillä komponenttien on kyettävä toimimaan sekä palvelin- että selainympäristössä. Vaikka Blazor kykenee käyttäjälle kohtalaiseen suorituskykyyn, jokainen renderöintimalli sisältää huomattavia haittapuolia, joten Blazor ansaitsee vertailussa -” -arvosanan suorituskyvyn osalta.

React, Vue ja Angular puolestaan tarjoavat vakaan ja riittävän suorituskyvyn sovelluskoon ja responsiivisuuden osalta. Vue on kahteen verrokkiinsa verrattuna kevyin ja tehokkain, mutta Nuxt ei aivan yllä Reactin Next-metakehyksen ja Angularin suorituskyvyn optimointikyvykkyyksien tasolle. Täten kaikki kolme mainittua teknologiaa saavat yksittäisen ”+” -arvosanan suorituskyvyn osalta.

### 8.1.2 Stabiilisuus ja suosio

Stabiilisuuden ja suosion kannalta edukseen erottuvat erityisesti jo pidemmällä aikavälillä vakiintuneet React ja Angular. Näillä teknologioilla on runsaasti avoimen lähdekoodin yhteisötukea ja lisäksi suuri kaupallinen yritys tuomassa taloudellista vakautta ja luotettavuutta teknologian kehittymiseen ja säilyvyyteen. Molemmilla on myös vakiintunut ja ennustettava toimintojen julkaisuprosessi. React ja Angular erottuvat edukseen muihin arvioituihin teknologioihin verrattuna, joten ne saavat ”++” -arvosanan. Vue ei aivan yllä samalle tasolle epäsäännöllisemmästä julkaisuprosessistaan ja suuren omistusyhtiön puutteestaan johtuen, mutta teknologiaa voidaan pitää kypsänä ja turvallisena työkaluna yrityskäyttöön, joten arvioinnin kannalta se ansaitsee myös ”++” -arvosanan.

Taulukko 8.2 kokoaa stabiilisuuteen vaikuttavia ominaisuuksia teknologioittain. Taulukosta on huomattavissa, että kaikki arvioiduista teknologioista sisältävät vähintään valtaosan teknologian kehittymistä stabiloivista ominaisuuksista. Stabiilisuudesta ja suosiosta ”+” -arvosanan saaneet Svelte ja Blazor sisältävät pieniä epävarmuustekijöitä, kuten Blazorilta puuttuvan ja Sveltellä käyttämättömäksi jääneen kehittäjäpalautteen keräämiseen tarkoitetun RFC-prosessin.

### 8.1.3 Ekosysteemi

React, Vue ja Angular sisältävät kattavimmat ekosysteemit vertailukohteista, mikä ansaitsee teknologioille täyden ”++” -arvosanan. Teknologioiden laaja suosio on mahdollistanut lukuisten avoimen lähdekoodin kirjastoratkaisujen syntymisen, niin komponenttitarjonnan kuin myös edistyneempien valmistoimintojen osalta.

| Teknologia | Semver | LTS | RFC | Taloudellinen tuki | Suosio |
|---|---|---|---|---|---|
| **Angular** | Kyllä | Kyllä | Kyllä | Kyllä (Google) | Kyllä |
| **React** | Kyllä | Kyllä | Kyllä | Kyllä (Meta, Vercel) | Kyllä |
| **Vue** | Kyllä | Kyllä | Kyllä | Kyllä (sponsorit) | Kyllä |
| **Svelte** | Kyllä | Ei | Osittainen | Kyllä (Vercel) | Kyllä |
| **Blazor** | Kyllä | Kyllä | Ei | Kyllä (Microsoft) | Osittainen |

**Taulukko 8.2:** Teknologioiden stabiilisuuden jäsentelevä koontitaulukko

Komponenttikirjastoista löytyy vaihtoehtoja tyyliltään erilaisiin peruskomponenttitarpeisiin, ja datan visualisointiin tarvittavia edistyneempiä komponenttiratkaisuja on myös tarjolla.

Myös Svelte ja Blazor sisältävät tärkeimmät komponenttikirjastot sekä työkalut käyttöliittymien rakentamiseen. Korkean tason komponenttiratkaisuja löytyy myös esimerkiksi datan visualisointiin, mutta vaihtoehdot ovat vähäisemmät. Tämä johtaa arvosanaan ”+”.

### 8.1.4 Rakenteellisuus ja skaalautuvuus

Vahvoine rakennevaatimuksineen ja mielipiteitä jakavine periaatteineen Angular on vaihtoehdoista tehokkain ohjelmistojen rakenteen ohjaamisessa vakiintuneisiin suunnittelumalleihin, kuten MVC-arkkitehtuuriin ja järjestelmälliseen komponenttihierarkiaan. Tämä mahdollistaa myös monimutkaisten käyttöliittymänäkymien rakentamisen huollettavasti ja systemaattisesti.

Myös muut ohjelmistokehykset sisältävät tukea rakenteellisuuden lisäämiseen metakehystensä kautta, lukuun ottamatta Blazoria. Tuki ei ole Angulariin verrattuna yhtä vahvasti teknologiaan sisäänrakennettua, joten systemaattinen ja rakenteellinen ohjelmistosuunnittelu vaatii kehitysoppaita tai teknologian syvällistä tuntemusta. Reactilla, Angularilla ja Vuella on suosionsa ansiosta saatavilla kattava määrä kolmannen osapuolen kehitysoppaita ja tutoriaaleja. Myös Sveltelle ja Blazorille löytyy ohjemateriaalia erityisesti teknologioiden omista dokumentaatioista, joten rakenteiden noudattamiseen on riittävät edellytykset. Täten Angularia lukuun ottamatta kaikki teknologiat saavat arvosanan ”+”.

### 8.1.5 Kehittämiskokemus ja tuottavuus

Tämä koontikategoria sisältää teknologian syntaksin tehokkuuden ja intuitiivisuuden tuomat hyödyt, joiden vastapainona arvioidaan teknologian oppimisen vaikeutta. Svelten pääperiaatteena on kehittämiskokemuksen priorisointi ja kyselytulokset osoittavat tavoitteen toteutuneen tehokkaasti muihin teknologioihin verrattuna. Ohjelmistokehittäjät ovat kiinnostuneita teknologiasta ja Svelteä käyttäneet haluavat usein jatkaa sen käyttöä. Intuitiivisuutta korostava ytimekäs syntaksi kykenee tuottamaan myös luettavaa ja huollettavaa koodia, tehostaen produktiivisuutta. Svelte voidaan luokitella täten kehittämiskokemukseltaan arvioinnin parhaaseen ”++” -luokkaan.

Arvosanan ”+” saaneet teknologiat parantavat kehittämiskokemusta verrattuna nykyiseen teknologiaratkaisuun, mutta hyvien ominaisuuksien rinnalla on myös selkeitä haittapuolia Svelten loppuunsa hiottuun käyttäjäkokemukseen nähden. Reactin kehittämiskokemusta heikentää sekavuus imperatiivisen ja deklaratiivisen koodin välillä, joten oikeaoppinen kehitys vaatii ohjelmistokehyksen renderöintilogiikan ymmärtämistä. React abstrahoi käyttäjältä Hookkien toimintaa ja uudelleenrenderöintilogiikkaa, mikä saattaa aiheuttaa ymmärtämättömälle kehittäjälle ongelmia. Heikkoutta lieventää Reactin laaja oppineisuus kehittäjäyhteisössä ja kehittäjien mieltymys Reactin antamiin vapauksiin. Angular voi tuntua kehittäjälle verboosilta ja monimutkaiselta, mutta järjestelmällisyys ja vakiintuneet rakennusperiaatteet voivat vähentää suunnittelutaakkaa ja tehdä ohjelmoinnista produktiivisempaa. Vuen yhdellä tiedostolla rakennettavat komponentit niputtavat loogisen kokonaisuuden yhtenäiseksi, ja ohjelmistokehyksen toiminnot on jaettu vue-tiedostojen sisällä selkeästi. Kehittäminen on välimuoto Angularia ja Reactia, joten Vuen arvosana on samassa luokassa.

### 8.1.6 Erityiset hyödyt tai haitat

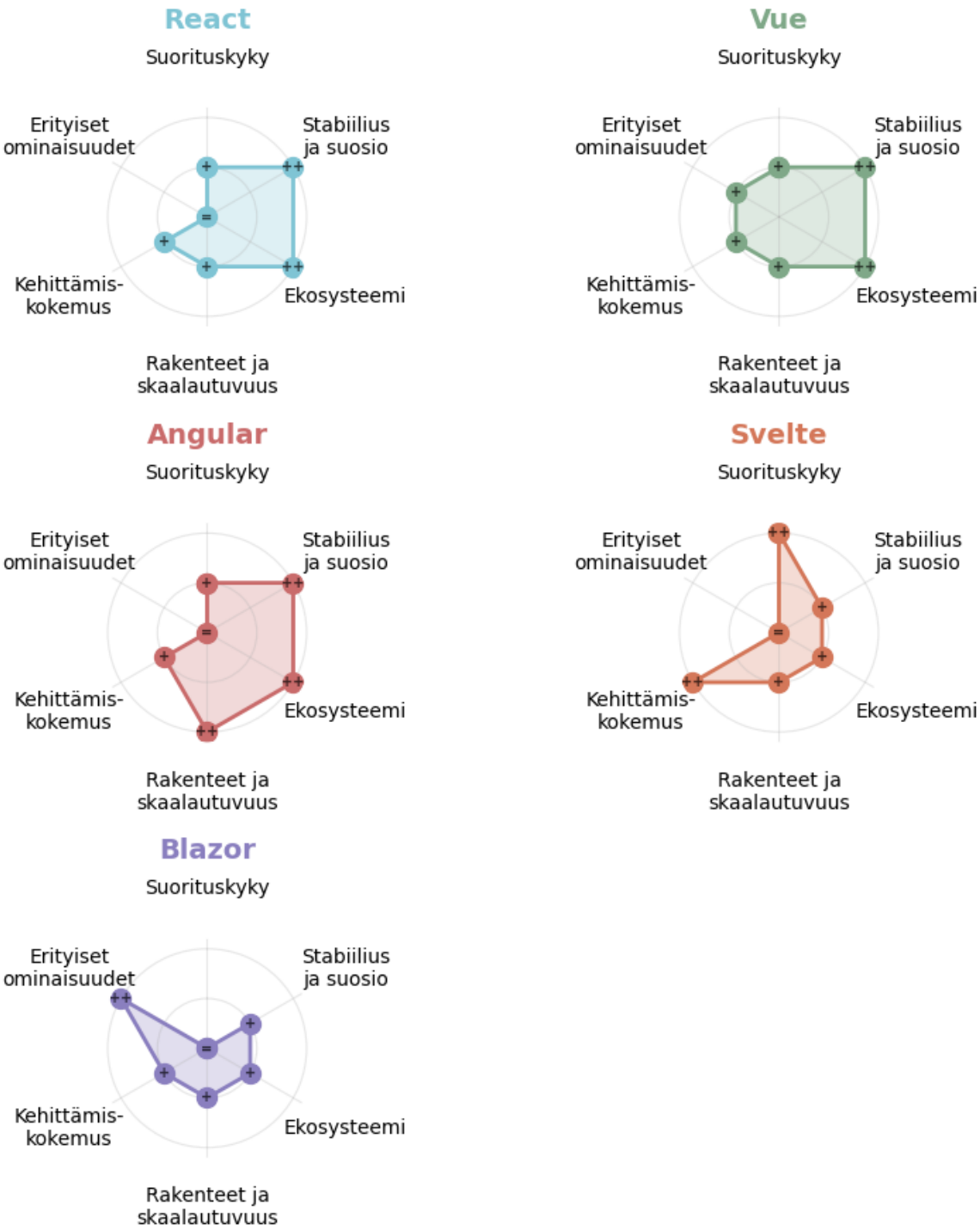
Jäsenneltyjen arviointikriteerien lisäksi Blazorille ja Vuelle on syytä huomioida arvioinnissa niiden erityiset ominaisuudet, jotka tuovat tutkimuksen taustatilanne huomioituna selkeää etua vertailukohteisiinsa nähden. Blazorin valinta ERA:n uudeksi käyttöliittymäteknologiaksi aiheuttaisi erityistä synergiaetua kehittäjille tutun kielen ja mahdollisesti kopioitavien toiminnallisuuksien ansiosta.

Vue tuo vaiheittaiseen teknologiavaihdokseen lisää mahdollisuuksia progressiivisen luonteensa ansiosta, sillä teknologiaa voidaan lisätä sivuille tarvittaessa komponentti kerrallaan, jolloin on mahdollista rakentaa uutta käyttöjärjestelmäkokonaisuutta ja samalla paikata vanhoja näkymiä Vuella rakennetuilla komponenteilla. Tällöin myös Vuen ekosysteemin komponenttikirjastoja voidaan hyödyntää vanhoissa näkymissä.

## 8.2 Teknologiavertailun analyysi

Kuva 8.1 havainnollistaa teknologioiden ominaisuusprofiileja tutkimuksessa käytettyjen kriteerien pohjalta. Tutkimustuloksista ei voi absoluuttisesti määrittää parasta teknologiaa, sillä kullakin teknologialla on omat uniikit hyötynsä, jotka voivat tehdä mistä tahansa arvioidusta teknologiasta muita paremman teknologialle asetetut tarpeet huomioiden. Teknologia-arvio antaa tutkimuksessa huomioitujen kriteerien pohjalta kehyksen teknologian päätöksentekoon. Esimerkiksi kaaviosta voidaan havaita Svelten ja Angularin loistavan pääosin vastakkaisissa kategorioissa. Kaavio osoittaa Svelten parhaiksi puoliksi suorituskyvyn ja kehittämiskokemuksen, kun taas Angular loistaa vakaudellaan, monipuolisella ekosysteemillään ja skaalautuvuutta tukevilla vahvoilla rakenteillaan. Kaavio osoittaa myös Reactin ja Vuen olevan kokonaisvaltaisesti vahvoja teknologiavaihtoehtoja, joten epävarmoihin tarpeisiin nähden ne ovat kaikki hyviä ratkaisuja. Päätöksen voi yksinkertaistaa kysymykseen: Halutaanko priorisoida Angularin rakenteellisuutta, Vuen progressiivisuutta, Reactin yleispätevyyttä vai Svelten kehittäjäystävällisyyttä? Blazorin kirkkain ominaisuus on puolestaan C#-kielen ja .NET-ympäristön tuo-

mat synergiaedut kehittäjien kykyjen tai aiempien kyseisiä teknologioita käyttävien ohjelmistoratkaisujen pohjalta.




**Kuva 8.1:** Teknologia-arvioinnin tulokset ominaisuusprofiileina

Kaavio 8.1 on generoitu liitteessä A esitetyllä Python-ohjelmalla.

## 8.3 Teknologioiden karsinta analyysin perusteella

Teknologia-arvioinnin perusteella Vue ja Angular ovat kokonaisvaltaisesti potentiaalisimmat vaihtoehdot ERAn uudeksi käyttöliittymäteknologiaksi. Arviointikriteereiden pisteytys tuottaa teknologioille saman arvosanan ”+8” joten päätöstä varten on arvioitava teknologioiden konkreettisimpia eroavaisuuksia ja niiden priorisointia. Angular tuo vankemman rakenteen ja tuen laadukkaan koodin kirjoittamiselle, kun taas Vue tarjoaa hyödyllisen mahdollisuuden yksittäisten tarpeiden paikkaamiseen vanhoihin näkymiin integroiduilla Vue-komponenteilla.

Lähelle Angularin ja Vuen tasoa yltävät ”+7” -arvosanoilla Svelte ja React, joten kyseisten teknologioiden valinta voi olla perusteltua, mikäli ne tuovat etuja arkkitehtuuriratkaisun ja tutkimuksen ERAan kohdistettujen tarpeiden pohjalta. Näistä erityisesti Svelte:n kehittäjäkokemushyödyt ovat relevantteja tarpeisiin nähden.

## 8.4 Arkkitehtuurimallien vertailu

Kappaleessa 7 kuvatuista arkkitehtuuriratkaisuista löydettiin jokaisesta hyviä puolia, jotka pyritään yhdistämään lopulliseen, teknologiasiirtymän mahdollistavaan arkkitehtuuriin. Malli A:n tavoitteena on saada uusi käyttöliittymä mahdollisimman nopeasti markkinoille ja aloittaa käyttöliittymän rakentaminen kokonaan vanhasta käyttöliittymästä erilliseksi. Mallin haittapuolena selain joutuu kohtaamaan useiden rajapintojen käsittelyn monimutkaisuuden. Malli on kompromissi teknisen velan takaisinmaksun suhteen, sillä sen aloittaminen vaatii enemmän työtä kuin C-mallin suora integrointi, mutta vähemmän työtä kuin B-mallin Web-rajapintakerros.

Tutkimusta motivoineisiin hallinnollisiin tarpeisiin suurimman hyödyn tuo ratkaisu B, jossa erillinen web-rajapintakerros auttaa poistamaan teknistä velkaa teknologiasiirtymän alkuvaiheesta lähtien, ja käyttöliittymäsovellus rakentuu yhtenäisen rajapinnan päälle. Arkkitehtuurimalli B vastaa tehokkaimmin myös tutkimuksen kehittäjänäkökulman tavoitteisiin. Modulaarinen ja vastuunjaoltaan selkeä arkkitehtuuri on kehittäjälle intuitiivinen.

Kaupallisten tarpeiden tehokkaaseen saavuttamiseen tutkimus tuotti ratkaisun arkkitehtuurimallia C ulkoisesti mukaillen. Tekninen toteutus tulee eroamaan C-mallin staattisesta komponenttikohtaisesta integraatiosta, mutta loppukäyttäjän näkökulmasta uuden käyttöliittymän osia voidaan yhdistellä vanhan käyttöliittymän näkymiin. Vanha sovellus tulee käyttämään micro-frontend -teknologioita uusien käyttöliittymäkomponenttien väliaikaista käyttöä varten. Tämä mahdollistaa asteittaisen käyttöliittymäkehityksen ja pienemmät kynnykset vaiheittaiseen teknologiasiirtymään.

Johtopäätöksenä uusi sovellus kehitetään itsenäisenä kokonaisuutena, mutta ratkaisuille A ja B tutkituilla mahdollisilla moduulien integrointitavoilla, Web Component- ja Module Federation-tekniikoilla, voidaan saavuttaa teknologiariippumaton komponenttien jakaminen myös uuden yhden sivun sovelluksen ulkopuolelle. Tämä auttaa kaupallisesti sekä markkinatarpeisiin mukautuvuu-

den että liiketoiminnan laajenemismahdollisuuksien saavuttamisessa. Vanhoja näkymiä voidaan parantaa siirtämättä koko näkymää uuteen teknologiaan, jolloin uuden käyttöliittymätoteutuksen julkaiseminen on mahdollista nopeasti ja pienin askelin. Uusia käyttöliittymäkomponentteja pystytään samalla integroimaan teknologiariippumattomasti myös asiakkaiden järjestelmiin, mikä tuo liiketoimintamahdollisuuksia käyttöliittymän osittaiseen myyntiin.

Siirtymä tullaan pääosin tekemään liiketoimintatarpeita mukaillen, mutta myös uusia toiminnallisuuksia on suunniteltu toteutettavaksi kokonaan uudessa käyttöliittymässä. Uuden käyttöliittymäohjelman kehitys tehdään uusiokäyttömielessä tavoitellen vanhan toiminnallisuuden korvaamista, mutta myös kestävän pohjan rakentamista tulevaisuuden käyttöliittymäkehitykselle.

## 8.5 Päätös teknologiavertailun ja arkkitehtuurianalyysin pohjalta

Teknologiavertailun jäljiltä parhaimmat arvosanat saivat Angular ja Vue. Lähelle samaa arvosanaa jäivät myös React ja Svelte. Lopullista päätöstä varten käydään läpi ominaisuuksien painokertoimia ja arvosana-asteikkoa tarkentavia yksityiskohtia valittuun arkkitehtuuriratkaisuun ja tutkimuksen tarpeisiin nähden.

Angular on turvallinen valinta uudeksi teknologiaratkaisuksi ERA:n käyttöliittymätoteutukselle ja se sopii erityisesti tilanteeseen, jossa käyttöliittymäkehityksessä keskitytään uuden yhden sivun sovelluksen rakentamiseen. Vankat raamit määrittävänä teknologiana Angular ei sovellu toivotulla tavalla micro-frontend -tyyliseen vanhan sovelluksen ehostamiseen ja käyttöliittymäosuuksien ulkoiseen jakamiseen. Angular on useaan Razor-näkymään kopioitavuuden kannalta raskaampi, kuin muut vaihtoehdot, eikä se tue erityisesti kiinnostavaa Module Federation -teknologiaa [3], [8] Hallinnollisten tarpeiden vaatimukset ketterästä ja asteittaisesta siirtymästä eivät täten ole Angularin vahvuuksia. Erityisesti PoC-toteutukseen voidaan nähdä sopivampana vaihtoehtona React, Vue tai Svelte, mutta Angular toimii hyvänä varavaihtoehtona, mikäli kokeilussa löytyy yllättäviä heikkouksia muista teknologioista.

Vue on myös suosionsa ja ekosysteeminsä osalta turvallinen valinta käyttöliittymäteknologiaksi, mutta Vuen rakenteellisuus ja hyvien koodauskäytäntöjen tuki sisältävät puutteita erityisesti yrityskäyttöön sopeutuvan meta-kehyksen puuttumisen johdosta. Täten tutkimuksen kehittäjänäkökulman tarpeet eivät täyty Vue-teknologialla optimaalisesti. Monipuolinen metakehys tai sellaista vastaavat toiminnot tuovat ERAan kaivattua rakennetta ja valmista toiminnallisuutta, joita Vue ei tarjoa riittävällä tasolla [13].

Svelte on noussut nopeasti suosiossa merkittävälle tasolle ja loistaa erityisesti käyttäjäkokemuksellaan ja suorituskyvyllään. Svelten metakehys SvelteKit on Vuen metakehystä monipuolisempi ja ERAn tarpeisiin sopivampi, joten Svelte vastaa selkeästi paremmin tutkimuksen kehittäjätarpeisiin. Hallinnollisiin tarpeisiin liittyen haasteena voi olla kokeneiden käyttäjien löytäminen, mutta helposti opittavana ja mielenkiintoa keränneenä teknologiana motivoituneita kehittäjiä todennäköisesti löytyy. Svelten suorituskykyedut mahdollistavat liiketoimintamotiivei-

hin liittyen sulavan käyttäjäkokemuksen.

React on myös asemansa ansiosta turvallinen valinta teknologiaratkaisuksi. Reactilla on potentiaalia vastata parhaiten tutkimuksen liiketoimintamotiiveiihin runsaan ekosysteeminsä ansiosta, mutta kehittäjäkokemuksen ja opittavuuden epäkohdat nähdään merkittävinä heikkouksina tutkimuksen kehittäjää koskeviin tarpeisiin peilaten.

Arkkitehtuurisuunnitelman pohjalta Svelte sopii parhaiten PoC-toteutukseen, sillä se tukee monipuolisesti micro-frontend tekniikoita, kuten Module Federationia. Lisäksi B-mallin arkkitehtuurin web-rajapinnan tuomat mahdollisuudet suorituskyvyn optimointiin säilyvät Svelten meta-kehyksen johdosta. Angular toimii turvallisena varavaihtoehtona, sillä PoC-toteutuksen jälkeen siihen voi turvautua pienellä riskillä, mikäli Svelten kokeilussa paljastuu haasteita.

# 9 Jatkokehitys ja pohdinta

Tämä tutkimus toimii esiselvityksenä ERAn käyttöliittymäuudistukselle. Kehitys jatkuu mahdollisesti pienimuotoisella lisätutkimuksella erityisesti Atostekia kiinnostavalle Flutter-teknologialle. Yleisesti teknologiavertailua voi jatkokehittää kokonaisvaltaisemmaksi huomioimalla lisänäkökulmia uusilla arviointikriteereillä. Esimerkiksi testattaavuus voi olla merkittävä ominaisuus teknologiavalinnalle käyttötarkoituksesta riippuen. Atostekin projektin jatkuvuuden kannalta teknologian ja arkkitehtuurikonseptien toimivuutta testaava PoC-toteutus on tärkeä seuraava askel tutkimuksen tulosten vahvistamiseksi. Mikäli lopullinen päätös uuden teknologian käyttöönotosta tehdään, tulee tämän tutkimuksen tarpeiden pohjalta suunnitella selkeät suuntaviivat uuden kehityksen jatkamiselle koodauskäytäntöjen, käyttäjäkokemuksen ja sovelluksen sisäisten rakenteiden osalta.

## 9.1 Teknologioiden lisätutkimus

Tutkimus osoittautui helposti paisuvaksi kokonaisuudeksi markkinoiden valtavan teknologiatarjonnan vuoksi, ja kaikki potentiaaliset teknologiavaihtoehdot eivät mahtuneet mukaan tutkimukseen. Mahdollisesti merkittävimpänä teknologiavaihtoehtona tutkimuksesta jäi pois Flutter. Kyseinen teknologia on erityisesti Atostekille mielenkiintoinen, sillä yrityksen muissa projekteissa on tutkimuksen aikana laajenevissa määrin käytetty Flutteria. Täten yrityksen sisällä olisi jo Flutterille osaajia, ja heitä voisi mahdollisesti hyödyntää ERAn käyttöliittymäuudistuksessa. Flutter on alustariippumaton avoimen lähdekoodin ohjelmistokehys, ja Googlen tukemana sen markkina-asema on rakentunut vakaaksi [24].

## 9.2 Arvioinnin syventäminen ja jatkotutkimukset

Teknologia-arviota voi myös kohdistaa käyttötarkoitukseen lisäämällä arviointikriteereille tarpeisiin pohjautuvat painoarvot teknologiapäätökseen. Esimerkiksi kokeilumaisissa projekteissa kehittäjäkokemuksen vaikutus voidaan priorisoida arvoltaan kaksinkertaiseksi, ja stabiilisuuden vaikutus puolittaa, kun tärkeintä on saada rakennettua toimiva käyttöliittymä mahdollisimman nopeasti. Näin teknologia-arviointi voi toimia pohjana päätöksenteolle myös muissa projekteissa, ja arviointi on täydennettävissä esimerkiksi yksityiskohtaisemmalla projektille tärkeällä kriteerillä.

Laajennuksena teknologia-arviointiin on hyvä huomioida myös ohjelmistokehysten testattavuus. Testattavuus on merkittävä kriteeri ERAn kaltaisissa suurissa projekteissa, joilla on erillinen testaustiimi ja tiukat testausvaatimukset. Teknologian tulee sopeutua nykyiseen testiympäristöön mahdollisimman pienellä lisävaivalla. Yleisesti teknologian toimintojen testattavuus on arvioitava yksikkötestien, integraatiotestien ja järjestelmätestien kannalta. Testattavuuteen ohjelmis-

tokehysten kannalta eniten vaikuttavat tekijät ovat saatavilla olevat työkalut, debuggausmahdollisuudet manuaalitestaukseen ja mahdollisuudet kattavaan automaattitestaukseen.

Arkkitehtuurisuunnittelussa on hyödyllistä punnita tarkemmin GraphQL-teknologian tuomia mahdollisuuksia palvelinpuolen rajapintojen yhtenäistämiseksi. GraphQL:n käyttöönoton kannattavuutta on myös arvioitava vaaditun lisätyön kannalta suhteutettuna sen tuomiin hyötyihin.

## 9.3 Proof-of-Concept, teknologian kokeilu

Valinnan selkeytyessä toteutetaan pienimuotoinen näkymä ”Proof-of-Concept” (PoC) -tyylillä. Uudella käyttöliittymäratkaisulla rakennetaan yksinkertainen näkymä liitettynä vanhaan käyttöliittymään, ja arvioidaan käyttöönoton vaativuutta sekä ratkaisun toimivuutta. PoC-toteutuksen on tarkoitus paljastaa kirjallisessa tutkimuksessa mahdollisesti huomaamatta jääneet haasteet uuden teknologian käyttöönotossa ja varmistaa teknologian soveltuvuus ERA-järjestelmään. PoC-kokeilulla testataan myös arkkitehtuurisuunnitelmia ja selvitetään tarvetta syvemmälle suunnittelulle tai suunnitelmien muutokseen. PoC-toteutusta tullaan tarkastelemaan myös korkealla tasolla havaittujen hyötyjen kannalta ja pohditaan, mitkä tarpeet täyttyivät tai tulevat täyttymään uudella käyttöliittymän teknologialla ja kehitystavalla.

## 9.4 Uuden teknologian käyttöönotto ja tyyliohjeet

Uutta käyttöliittymäpohjaa määrittäessä on tarjolla mahdollisuus asettaa raamit ja suuntaviivat kehitykselle, jotta käyttöliittymän yksittäisten kehittäjien ei tarvitse ratkaista toistuvia sovellusarkkitehtuuriin liittyviä haasteita itse. Tällä arkkitehtuurin tyylisuunnat määrittelevällä jatkotutkimuksella pyritään helpottamaan esimerkiksi koodin katselmointia. Katselmoijalla on tukenaan lista ominaisuuksia, jotka kehittäjän koodista on tarkastettava ennen uuden tai muokatun koodin päästämistä osaksi sovellusta.

Tyyliohjeilla listataan uuden teknologian mahdollistamat käytänteet, jotka koetaan hyödyllisiksi systemaattisesti käytettyinä. Teknologiakohtaisten käytänteiden lisäksi jatkotutkimuksessa pyritään määrittämään myös yleistä koodin rakennetta koskevat suunnittelumallit, joihin käyttöliittymän rakenne perustetaan. Myös käyttötapauskohtaisten suunnittelumallien käytöstä on syytä määrittää tarkistuslista, josta kehittäjä voi etsiä uuteen toiminnallisuuteen sopivaa mallia. Kun käyttöliittymä tulee käsittelemään ERAn ydinjäjestelmän tietomallien ohella myös FHIR-palvelun erimuotoisia resursseja, täytyy web-rajapinnalle määrittää systemaattinen tapa yhdistää tietomallit tilanteissa.

### 9.4.1 Meta-kehyksen käyttöönotto

Mikäli PoC-toteutuksen jälkeen Svelte pysyy teknologiavalintana, täytyy vielä uudelleenarvioida Svelten metakehyksen, SvelteKit:n käyttöönoton kannattavuutta. Metakehyksen osalta tulee esimerkiksi arvioida palvelinpuolen kuvantamisen käyttöönottamisen tuomia suorituskyky- ja käyttäjäkokemusetuja suhteutettuna tekniikan aiheuttamaan kasvavaan monimutkaisuuteen kahdessa erilaisessa ympäristössä, suoritettavan käyttöliittymäkoodin takia.

## 9.5 Käyttäjäkokemussuunnittelu

ERA on sosiaali- ja terveydenhuollon ammattilaisten työkalu, jota käytetään reaaliaikaisen palvelutapahtuman yhteydessä. Täten ERAlle on tärkeää käytön sujuvuus, jotta järjestelmä ei aiheuta hidastuksia tai muita häiriöitä ammattilaisen työntekoon, vaan toimisi intuitiivisesti tukena sekä uuden tiedon tallentamiseen että asiakkaan aiempien tietojen tutkimiseen ja analysointiin. Yksi käytön sujuvuuteen vaikuttava kriteeri on teknologian ja arkkitehtuurin suorituskyky, jota tässä tutkimuksessa on aiemmissa luvuissa analysoitu. Toinen merkittävä tekijä käytön sujuvoittamisessa on käyttäjäkokemussuunnittelu, jolla tutkimusten mukaan voi olla merkittävä vaikutus käyttöliittymän käytön tehokkuuteen [20]. Uutta käyttöliittymää suunnitellessa on täten kriittistä tehdä myös käyttäjäkokemustutkimusta ja suunnittelua siihen erikoistuneen ammattilaisen toimesta, jotta uudesta käyttöliittymäteknologiasta ja arkkitehtuurista pystytään valjastamaan maksimaalinen hyöty käyttäjätyytyväisyyden kasvattamiseksi.

## 9.6 Reflektio

Työ on vaatinut web-kehityksen ja siihen liittyvien konseptien tutkimista erittäin laaja-alaisesti. Työ on käsitellyt Web-käyttöliittymäteknologioiden sisäisiä toimintamalleja, mutta niiden ohella myös teknologioiden käytön vaikutuksia ja vaatimuksia tuotekehityksessä. Olen halunnut ymmärtää teknologioita ja tekniikoita perusteellisesti, mikä on johtanut keskittymisen ajautumiseen tutkimuksen edistymisen kannalta epäideaaliin suuntaan. Aiemmin opiskeluaikana mieleen muodostuneisiin abstraktioihin perustuneet oletukset ovat häirinneet minua epävarmuudellaan, joten olen pureutunut abstraktioiden alle, kuten selaimen sisäiseen toimintaan ohjelmistoympäristönä. Täten työn prosessi on ollut hyvin opettavainen. Olen tutkimusta tehdessäni kehittänyt ymmärrystäni selaimen toimintojen sisäisestä vastuunjaosta ja yhteistyöstä palvelinpuolen ohjelmistojen kanssa, sekä ohjelmistokehysten rooleista web-sovellusten toiminnassa. Korkealla tasolla olen oppinut haasteiden kautta priorisoimaan ohjelmistoprosessin kannalta kriittisiä asioita ja huomaamaan tilanteita, joissa ajaudun yksityiskohtien perään.

# 10 Yhteenveto

Tutkimuksessa päädyttiin valitsemaan uuden käyttöliittymän PoC-kokeiluun uusi käyttöliittymäteknologia sekä teknologian nykyjärjestelmään integroiva arkkitehtuurisuunnitelma. Teknologiavalinnaksi valikoituneen Svelten opittavuutta ja intuitiivisuutta päästään kokeilemaan puhtaalta pöydältä PoC-vaiheessa. Tutkimuksessa vertailtujen arkkitehtuurihahmotelmien kompromissi päätyi lopulliseksi ratkaisuksi teknologian käyttöönotolle, kun PoC-käyttöliittymä rakennetaan yhden sivun sovelluksena minimalistisen web-rajapinnan päälle, mutta komponenttien yhdistämistä kokeillaan myös micro-frontend-tyylisesti vanhaan käyttöliittymään.

Tutkimuksessa toteutettiin web-käyttöliittymien kehittämiseen tarkoitettujen teknologioiden arviointia määriteltyjen arviointikriteerien pohjalta. Teknologiavertailun tuloksena ei saavutettu absoluuttista valintaa parhaaksi teknologiaksi, sillä suosituimmilla moderneilla web-teknologioilla on jokaisella omat vahvuutensa. Sen sijaan saavutettiin teknologioita profiloivat arvosanataulukot, jotka toimivat tukena lopullisen teknologiavalinnan tekemiseen käyttökohteen prioriteettien pohjalta.

ERAn käyttötapauksessa valinnan tuottamiseksi oli palattava tutkimusta motivoineisiin tarpeisiin sekä arkkitehtuuriratkaisun synergiaan teknologian ominaisuuksien kanssa. Pelkän teknologia-analyysin pohjalta parhaimmiksi teknologioiksi osoittautuivat kenttätestattu Angular ja siitä inspiroitunut kevyempi ohjelmistokehys Vue. Lähelle samaa arvosanaa pääsivät JavaScript-kehyksistä React ja Svelte. Näistä neljästä teknologiavaihtoehdosta arkkitehtuuriratkaisuun sopii parhaiten Svelte ja Vue. Vuen arviointiasteikossa havaittujen epätarkkuuksien johdosta Svelten voitiin todeta olevan kokonaisvaltaisesti tutkimuksen tarpeisiin parhaiten vastaava teknologia.

Angular ja Svelte loistavat omilla, lähes vastakkaisilla osa-alueillaan. Angular on vakiinnuttanut asemansa liiketoimintatarkoituksiin luotettavana teknologiana, joka kehittyy tasaisesti ja ennustettavasti. Svelte puolestaan on tuore ja trendikäs haastaja, joka kehittyy nopeasti, mutta tulevaisuuden näkymät ovat epävarmemmat kuin Angularilla. Angular on ominaisuuksiltaan runsas, mutta selaimelle Svelteä raskaampi. Svelte toteuttaa Angularia vastaavia toiminnallisuuksia SvelteKit-metakehyksen avulla, mutta se saavuttaa erityisen hyvän suorituskyvyn käännösprosessinsa ansiosta.

Uuden teknologian integroivia arkkitehtuuriratkaisuja analysoitiin kolmesta eritasolla integroituvasta lähtökohdasta, joista jokaisesta saatiin vaikutteita lopulliseen arkkitehtuurisuunnitelmaan. Ensimmäinen yksinkertainen arkkitehtuurivaihtoehto A lisää uuden yhdensivunsovelluksen olemassa olevien rajapintojen päälle. Toinen arkkitehtuurivaihtoehto B lisää A-mallin käyttöliittymäsovellukselle kohdistetun palvelinsovelluksen. Kolmas arkkitehtuurivaihtoehto C rakentuu vanhan käyttöliittymän sisälle pala palalta vaatimatta vanhaan runkoon nojaten. A-mallin yhden sivun sovelluksen periaatteella saadaan puhdas pohja uuden käyttöliittymän rakentamiselle ja teknisen velan takaisinmaksulle.

B-mallin Web-rajapintakerroksella saavutetaan selkeyttä järjestelmäarkkitehtuuriin kehittäjien ja pitkän aikavälin huollettavuuden näkökulmasta. Käyttöliittymä näyttäytyy palvelinohjelmistojen näkökulmasta uuden järjestelmäintegraation tavoin, ja käyttöliittymälogiikan vastuunjako on selkeää. Micro-frontend-tekniikoilla mahdollistetaan samanaikaisesti sekä C-mallin tyyliset pienet siirtymäaskeleet teknologian vaihdoksessa, että liiketoiminnallisten jakelumahdollisuuksien laajentaminen, kun uuden käyttöliittymän komponentteja voidaan uudelleenkäyttää käyttöliittymiä yhdistelevien kuorikomponenttien avulla.

Yhdessä modernin ja suorituskykyisen teknologian kanssa modulaarinen ja muovautuva arkkitehtuuriratkaisu mahdollistavat kunnianhimoisen kehityssuunnan ERAn käyttöliittymäuudistukselle, jolla voidaan saavuttaa tutkimuksen tarpeita monipuolisesti ja tehokkaasti. Mikäli PoC-vaiheessa huomataan Sveltessä uusia puutteita, tai teknologian hyödyt osoittautuvat todettua pienemmiksi, voidaan turvautua yksinkertaiseen B-mallin arkkitehtuuriratkaisuun ja turvallisen Angular-teknologian käyttöönottoon.

# Lähteet


[1] *Angular Material, web page*, Google. url: https://material.angular.io/.
[2] *Angular, Github README*, Google. url: https://github.com/angular/angular.
[3] *Angular, v18, Documentation and Reference*, Google. url: https://angular.dev/.
[4] *ASP.NET MVC Views*, Microsoft. url: https://learn.microsoft.com/en-us/aspnet/mvc/overview/older-versions-1/views/.
[5] *Overview of Single Page Apps (SPAs) in ASP.NET Core*, Microsoft. url: https://learn.microsoft.com/en-us/aspnet/core/client-side/spa/intro?view=aspnetcore-8.0.
[6] A. Torris, *Awesome Blazor, resource collection*. url: https://github.com/AdrienTorris/awesome-blazor.
[7] *Awesome Vue, resource collection*. url: https://github.com/vuejs/awesome-vue.
[8] *JavaScript framework Benchmark results for Chrome 138*. url: https://krausest.github.io/js-framework-benchmark/2025/table_chrome_138.0.7204.50.html.
[9] E. Bidelman, "Custom Elements v1 - Reusable Web Components", *web.dev*, 2017. url: https://web.dev/articles/custom-elements-v1.
[10] *Blazor documentation*. url: https://learn.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-9.0.
[11] A. Boduch ja R. Derks, *React and React Native*, eng, Third edition. Birmingham, England ; Packt, 2020, luku 13, Server-Side Rendering, ISBN: 9781839212437. url: https://learning.oreilly.com/library/view/react-and-react/9781805127307/Chapter_13.xhtml.
[12] A. Boduch ja R. Derks, *React and React Native*, eng, Third edition. Birmingham, England ; Packt, 2020, luku 11, Fetching Data from a Server, ISBN: 9781839212437. url: https://learning.oreilly.com/library/view/react-and-react/9781805127307/Chapter_11.xhtml.
[13] D. Chingleput Sathyakumar, "The Marko-Run Meta Framework", eng, teoksessa *Practical Marko*, United States: Apress L. P, 2025, s. 341–418, ISBN: 9798868814822.
[14] *Create React App documentation*, Facebook, Inc. url: https://create-react-app.dev/.
[15] *CSS Tutorial*, W3Schools. url: https://www.w3schools.com/Css/.
[16] H. Dhaduk, "Angular vs Vue: Which Framework to Choose in 2024?", *Simform blog*, 2024.
[17] *ASP.NET Core Roadmap for .NET 10*, Microsoft. url: https://github.com/dotnet/aspnetcore/issues/59443.
[18] *Elm Github Repositories*. url: https://github.com/elm.
[19] *Elm, Official guide*. url: https://guide.elm-lang.org/.
[20] J. M. Ferreira, F. D. Rodríguez, A. Santos, O. Dieste, S. T. Acuña ja N. Juristo, "Impact of Usability Mechanisms: A Family of Experiments on Efficiency, Effectiveness and User Satisfaction", *IEEE Transactions on Software Engi-*

*neering*, vol. 49, nro 1, s. 251–267, tammikuu 2023, ISSN: 2326-3881. DOI: 10.1109/tse.2022.3149586. url: http://dx.doi.org/10.1109/TSE.2022.3149586.

[21] *Hl7 FHIR R4 Overview*. url: https://hl7.org/fhir/R4/overview.html.

[22] *Fluent UI Blazor documentation*, Microsoft. url: https://www.fluentui-blazor.net/.

[23] *Fluent UI Blazor Github*, Microsoft. url: https://github.com/microsoft/fluentui-blazor/.

[24] *Flutter documentation*. url: https://flutter.dev/.

[25] N. Ford, R. Parsons, P. Kua ja P. Sadalage, *Building Evolutionary Architectures*, eng, 2. painos. Sebastopol: O'Reilly Media, Incorporated, 2023, ISBN: 9781492097549.

[26] *State of JS 2024, Front-end Frameworks survey*. url: https://2024.stateofjs.com/en-US/libraries/front-end-frameworks/.

[27] *State of JS 2024, Meta-Frameworks survey*. url: https://2024.stateofjs.com/en-US/libraries/meta-frameworks/.

[28] E. Gamma, R. Helm, R. Johnson ja J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley Professional Computing Series), eng, 1st edition. Hoboken: Pearson Education, Limited, 1994, ISBN: 0201633612.

[29] Y. Ganbar, "HTMX vs React: A First Look and Comparison", *builder.io*, elokuu 2024.

[30] J. E. Hannay, *Benefit/Cost-Driven Software Development : With Benefit Points and Size Points* (Simula SpringerBriefs on Computing ; v.8), eng. Cham: Springer International Publishing AG, 2021, ISBN: 3-030-74218-0.

[31] R. Harris, "Frameworks without the framework: why didn't we think of this sooner?", *Svelte blog*, 2016. url: https://svelte.dev/blog/frameworks-without-the-framework.

[32] *Headless UI component library*. url: https://headlessui.com/.

[33] *HTMX*, htmx.org. url: https://htmx.org/.

[34] *Introduction to Modern.js*, ByteDance. url: https://modernjs.dev/en/guides/get-started/introduction.

[35] Z. Jackson, "Understanding Module Federation: Deep dive", *Medium*, 2023. url: https://scriptedalchemy.medium.com/understanding-webpack-module-federation-a-deep-dive-efe5c55bf366.

[36] Z. Jackson, *Deprecation Notice for nextjs-mf*. url: https://github.com/module-federation/core/issues/3153.

[37] *jQuery dokumentaatio*. url: https://jquery.com/.

[38] *JavaScript framework Benchmark results for Chrome 138*. url: https://krausest.github.io/js-framework-benchmark/2024/table_chrome_131.0.6778.85.html.

[39] J. S. Lauren Tan ja M. Zhang, "React Compiler v1.0", *React blog*, 2025. url: https://react.dev/blog/2025/10/07/react-compiler-1.

[40] R. H. M. Carroll, "Sunsetting Create React App", *React blog*, 2024. url: https://react.dev/blog/2025/02/14/sunsetting-create-react-app.

[41] *Material Design 3, design system*, Google. url: https://m3.material.io/.

[42] *mdn web docs, HTML Reference*, Mozilla. url: https://developer.mozilla.org/en-US/docs/Web/HTML/.

[43] *mdn web docs, JavaScript Reference*, Mozilla. url: https://developer.mozilla.org/en-US/docs/Web/JavaScript/.

[44] *mdn web docs, Web APIs*, Mozilla. url: https://developer.mozilla.org/en-US/docs/Web/API.

[45] *mdn web docs, WebAssembly*, Mozilla. url: https://developer.mozilla.org/en-US/docs/WebAssembly/Guides/Concepts.

[46] R. V. Melissa Ada, "Performance", *Web Almanac By HTTP Archive*, lokakuu 2022. url: https://almanac.httparchive.org/en/2022/performance.

[47] *Module Federation Documentation*, Module Federation core team. url: https://module-federation.io/.

[48] *Webpack Documentation: Module Federation*. url: https://webpack.js.org/concepts/module-federation/.

[49] S. Newman, *Monolith to microservices : evolutionary patterns to transform your monolith*, eng, First edition. Beijing: O'Reilly, 2020, ISBN: 1-4920-4779-1.

[50] *Next with Vercel, documentation*, Vercel. url: https://vercel.com/docs/frameworks/nextjs.

[51] *Next.js documentation*, Vercel, Inc. url: https://nextjs.org/docs.

[52] O. Parlapiano, "A visual guide to React Mental models, part 2: useState, useEffect and lifecycles", *Obed Parlapiano, blog*, 2025.

[53] D. partners, *Frontend Strangler pattern implementation experiences*. url: https://dualbootpartners.com/insights/strangler-fig-pattern/.

[54] *Github repository, petite-vue*. url: https://github.com/vuejs/petite-vue.

[55] T. Preston-Werner, *Semantic Versioning Specification*. url: https://semver.org/.

[56] A. Prytulenets, "The Best Tech Stack for Web App Development in 2024", *Fively*, tammikuu 2024. url: https://5ly.co/blog/best-web-app-tech-stack/.

[57] *Radzen Blazor documentation*, Radzen. url: https://blazor.radzen.com/.

[58] *React documentation*, Meta. url: https://react.dev/.

[59] *React Foundations course for Next.js*, Vercel, Inc. url: https://nextjs.org/learn/react-foundations/.

[60] *React Request for Comments -Process*, Meta. url: https://github.com/reactjs/rfcs.

[61] *ReactJS.NET Docs*, Facebook, Inc. url: https://reactjs.net/.

[62] *ReactJS.NET Github repository*, Facebook, Inc. url: https://github.com/reactjs/React.NET.

[63] *Rsbuild, Getting Started Guide*. url: https://rsbuild.dev/guide/start/index.

[64] *Angular: The RxJS library*. url: https://v17.angular.io/guide/rx-library.

[65] S. Sarris, *HTML5 Unleashed* (Unleashed), eng, 1st edition. Pearson Education, 2013, ISBN: 9780133151329. url: https://andor.tuni.fi/permalink/358FIN_TAMPO/1j3mh4m/alma9910688740205973.

[66] E. A. Scott, *SPA Design and Architecture: Understanding single-page web applications*, eng, 1. painos. New York: Manning, 2015, ISBN: 9781617292439.

[67] *Sertifiointi ja olennaiset vaatimukset*, Kanta. url: https://www.kanta.fi/jarjestelmakehittajat/sertifiointi-ja-olennaiset-vaatimukset.

[68] *shadcn komponenttikirjasto*. url: https://ui.shadcn.com/docs.

[69] K. Smith, "Simplifying Ajax-style Web development", eng, *Computer (Long Beach, Calif.)*, vol. 39, nro 5, s. 98–101, 2006, ISSN: 0018-9162.

[70] *Stack Overflow Annual Developer Survey*, Stack Exchange Inc. url: https://survey.stackoverflow.co/.

[71] *Svelte*. url: https://svelte.dev/.

[72] *Svelte 4: Zero To Mastery*, Dev Community. url: https://dev.to/admaletsas/svelte-4-zero-to-mastery-165f.

[73] *Svelte 5 migration guide*, Svelte. url: https://svelte.dev/docs/svelte/v5-migration-guide.

[74] R. Harris, *Svelte blog, Write less code*. url: https://svelte.dev/blog/write-less-code.

[75] *Svelte philosophy, Github-keskustelu, Rich Harris*. url: https://github.com/sveltejs/svelte/discussions/10085.

[76] *Svelte RFCs*. url: https://github.com/sveltejs/rfcs.

[77] *Svelte Society community and ecosystem*. url: https://www.sveltesociety.dev/.

[78] D. Taibi ja L. Mezzalira, ”Micro-Frontends: Principles, Implementations, and Pitfalls”, *SIGSOFT Softw. Eng. Notes*, vol. 47, nro 4, s. 25–29, syyskuu 2022, ISSN: 0163-5948. DOI: 10.1145/3561846.3561853. url: https://doi-org.libproxy.tuni.fi/10.1145/3561846.3561853.

[79] *Technology Radar, Languages & Frameworks*, Thoughtworks, 2025. url: https://www.thoughtworks.com/radar/languages-and-frameworks.

[80] *Webpack Documentation*. url: https://webpack.js.org/concepts/.

[81] *Vercel Guides*, Vercel, Inc. url: https://nextjs.org/docs.

[82] *Vite, Getting Started Guide*. url: https://vite.dev/guide/.

[83] C. Wong, *HTTP Pocket Reference: Hypertext Transfer Protocol*, eng, 1. painos. Sebastopol: O’Reilly Media, Incorporated, 2000, ISBN: 9781565928626.

[84] *Vue 3 Documentation*. url: https://vuejs.org/guide/introduction.html.

[85] *Vue Style Guide*. url: https://vuejs.org/style-guide/.

[86] *Vue 2 Documentation*. url: https://v2.vuejs.org/v2/guide/comparison.html.

[87] *Vue 2 LTS, EOL & Extended Support*. url: https://v2.vuejs.org/lts/.

[88] J. W. Yoder ja P. Merson, ”Strangler patterns”, teoksessa *Proceedings of the 27th Conference on Pattern Languages of Programs*, sarja PLoP ’20, Virtual Event: The Hillside Group, 2022, ISBN: 9781941652169.

# Liite A: Kaavioiden generointiin käytetty ohjelma

Teknologia-arvioinnin yhteenvedossa käytetty teknologiaprofiileita havainnollistava kaaviokokoelma 8.1 tuotettiin Python-ohjelmalla, joka on jaettu kolmeen osaan koodipätkissä A.1, A.2 ja A.3.

```
import matplotlib.pyplot as plt
import numpy as np
from math import pi
from scipy.interpolate import make_interp_spline

CATEGORIES = ['Suorituskyky',
              'Stabiilius\nja␣suosio',
              'Ekosysteemi',
              'Rakenteet␣ja\nskaalautuvuus',
              'Kehittämis-\nkokemus',
              'Erityiset\nominaisuudet']

N_CATEGORIES = len(CATEGORIES)

TECHNOLOGIES = {
    'React': [1, 2, 2, 1, 1, 0],
    'Vue': [1, 2, 2, 1, 1, 1],
    'Angular': [1, 2, 2, 2, 1, 0],
    'Svelte': [2, 1, 1, 1, 2, 0],
    'Blazor': [0, 1, 1, 1, 1, 2]
}

TECHNOLOGY_COLOURS = {
    'React': '#7FC4D4',    # Desaturated cyan
    'Vue': '#7FA888',      # Desaturated green
    'Angular': '#C96B6B',  # Desaturated red
    'Svelte': '#D47759',   # Desaturated orange
    'Blazor': '#8A7FBF'    # Desaturated purple
}

def getMarker(value):
    if value == 2:
        return "++"
    if value == 1:
        return "+"
    return "="
```

**Ohjelma A.1:** Osa 1: Teknologiayhteenvedon kuvan 8.1 tuottanut Python-ohjelma

```
def plotRadarCharts():
    angles = [n / float(N_CATEGORIES) * 2 * pi
              for n in range(N_CATEGORIES)]
    angles += angles[:1]
    _, axes = plt.subplots(3, 2, figsize=(8, 11),
                           subplot_kw=dict(projection='polar'))
    axes = axes.flatten()

    for idx, (framework, values) in
            enumerate(TECHNOLOGIES.items()):
        ax = axes[idx]
        # Close the radar chart "circle"
        plot_values = values + [values[0]]
        ax.plot(angles, plot_values, 'o-', linewidth=2,
                color=TECHNOLOGY_COLOURS[framework],
                markersize=10)
        ax.fill(angles, plot_values, alpha=0.25,
                color=TECHNOLOGY_COLOURS[framework])
        for angle, value in zip(angles[:-1], values):
            ax.text(angle, value, getMarker(value),
                    ha='center', va='center', fontsize=7,
                    fontweight='bold', color='black',
                    alpha=0.7)
        # Fix axis to go in the right order and start at 12
o'clock
        ax.set_theta_offset(pi / 2)
        ax.set_theta_direction(-1)
        # Draw axis lines for each angle and label
        ax.set_xticks(angles[:-1])
        ax.set_xticklabels(CATEGORIES, size=10)
        ax.tick_params(axis='x', pad=15)
        # Set y-axis limits and labels
        ax.set_ylim(0, 2.4)
        ax.set_yticks([0, 1, 2])
        ax.set_yticklabels([])
        ax.grid(True, alpha=0.3)
        ax.spines['polar'].set_alpha(0.0)
        ax.set_title(framework, size=13, weight='bold', pad=10,
                     color=TECHNOLOGY_COLOURS[framework])

    # Remove the empty subplot (bottom right)
    axes[-1].set_visible(False)
    plt.tight_layout()
    plt.subplots_adjust(hspace=1.1, wspace=0.1)
    plt.show()
```

**Ohjelma A.2:** Osa 2: Teknologiayhteenvedon kuvan 8.1 tuottanut Python-ohjelma

```
def plotAreaCharts():
    _, axes = plt.subplots(5, 1, figsize=(8, 11), sharex=True)
    x = np.arange(N_CATEGORIES)
    for idx, (framework, values) in
            enumerate(TECHNOLOGIES.items()):
        ax = axes[idx]
        x_smooth = np.linspace(-0.5, N_CATEGORIES - 0.5, 300)
        spl = make_interp_spline(x, values, k=2)
        y_smooth = spl(x_smooth)
        # Create area chart with solid fill
        ax.fill_between(x_smooth, 0, y_smooth, alpha=1.0,
            color=TECHNOLOGY_COLOURS[framework])
        ax.plot(x_smooth, y_smooth, linewidth=3, color='white')
        # Set y-axis limits and labels
        ax.set_ylabel(framework, fontsize=14,
            fontweight='bold',
            color=TECHNOLOGY_COLOURS[framework])
        ax.set_ylim(0, 2.5)
        ax.set_yticks([0, 1, 2])
        ax.grid(True, alpha=0.3, axis='y')
        # Remove top and right spines
        ax.spines['top'].set_visible(False)
        ax.spines['right'].set_visible(False)
        ax.set_xlim(-0.02, N_CATEGORIES - 1 + 0.02)

    # Set x-axis labels on bottom chart
    axes[-1].set_xticks(x)
    axes[-1].set_xticklabels(CATEGORIES, fontsize=10,
    fontweight='bold')
    plt.tight_layout()
    plt.show()

if __name__ == "__main__":
    plotRadarCharts()
    plotAreaCharts()
```

**Ohjelma A.3:** Osa 3: Teknologiayhteenvedon kuvan 8.1 tuottanut Python-ohjelma