Kielimallien hyödyntäminen koodimigraatiossa: siirtymä AngularJS:stä Angulariin

Turun yliopisto
Tietotekniikan laitos
TkK-tutkielma
Tietotekniikka
Maaliskuu 2026
Robi Johansson

Turun yliopiston laatujärjestelmän mukaisesti tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck -järjestelmällä.

TURUN YLIOPISTO
Tietotekniikan laitos

Robi Johansson: Kielimallien hyödyntäminen koodimigraatiossa: siirtymä AngularJS:stä Angulariin

TkK-tutkielma, 36 s.
Tietotekniikka
Maaliskuu 2026

Pitkäikäisissä verkkojärjestelmissä tekninen velka, vanhentuneet riippuvuudet ja puutteellinen tyypitys heikentävät ylläpidettävyyttä, turvallisuutta ja muutosten läpimenoaikaa. Erityisesti AngularJS-pohjaisten käyttöliittymäkoodikantojen modernisointi Angulariin ja TypeScriptiin on monissa organisaatioissa väistämätöntä, mutta työ on yhä suurelta osin manuaalista, virhealtista ja vaikeasti skaalautuvaa. Tässä tutkielmassa tarkastellaan, miten suuria kielimalleja (LLM) voidaan hyödyntää AngularJS:stä Angulariin tehtävässä migraatiossa ja JavaScriptistä TypeScriptiin tehtävässä koodimuunnoksessa. Lisäksi jäsennetään LLM-avusteisen migraation keskeiset hyödyt, haitat ja rajoitteet. Työ perustuu kirjallisuuskatsaukseen, jossa analysoidaan kielimallien roolia koodin muunnoksessa, refaktoroinnissa ja modernisoinnin tukitehtävissä, kuten tyyppien luonnostelussa, ohjelmointirajapintojen kartoituksessa sekä testien ja dokumentaation tuottamisessa.
Tulosten perusteella kielimallien keskeisin hyöty on tuottavuuden parantuminen: mallit kykenevät tuottamaan nopeasti ensimmäisiä muunnosversioita ja tukemaan perintökoodin ymmärtämistä, jolloin kehittäjän työ painottuu laadunvarmistukseen ja poikkeusten käsittelyyn. Samalla tunnistetaan merkittäviä riskejä, kuten toiminnallisen vastaavuuden heikkeneminen, sovellusalue- ja ohjelmistokehyskohtaisen tiedon puutteet, ei-deterministinen toiminta sekä automaatioharha. Tutkielman johtopäätös on, että kielimallien paras rooli tässä kontekstissa on puoliksi automatisoidussa migraatioputkessa, jossa mallin tuotokset ankkuroidaan deterministisiin tarkistuksiin (kääntäjä, tyyppitarkistus, koodityylitarkistus, staattinen analyysi) ja testaukseen, ja hyväksyntä sidotaan todistepohjaiseen, riskiperusteiseen katselmointiin.

Asiasanat: laajat kielimallit, ohjelmistomigraatio, AngularJS, Angular, JavaScript, TypeScript, refaktorointi, laadunvarmistus, staattinen analyysi, automaatioharha

UNIVERSITY OF TURKU
Department of Computing

Robi Johansson: Kielimallien hyödyntäminen koodimigraatiossa: siirtymä AngularJS:stä Angulariin

Bachelor's Thesis, 36 p.
Computer Engineering
March 2026

In long-lived web systems, technical debt, outdated dependencies, and insufficient typing weaken maintainability, security, and change lead time. In particular, modernizing AngularJS-based front-end codebases to Angular and TypeScript has become unavoidable in many organizations, yet the work is still largely manual, error-prone, and difficult to scale.

This thesis examines how large language models (LLMs) can be leveraged in migration from AngularJS to Angular and in code conversion from JavaScript to TypeScript. It also structures the key benefits, drawbacks, and limitations of LLM-assisted migration. The study is based on a literature review that analyzes the role of language models in code transformation, refactoring, and modernization support tasks, such as drafting type annotations, mapping APIs, and generating tests and documentation.

The findings indicate that the main benefit of language models is improved productivity: models can quickly produce initial conversion drafts and support understanding of legacy code, shifting the developer's work toward quality assurance and handling exceptions. At the same time, significant risks are identified, including degradation of functional equivalence, gaps in domain- and framework-specific knowledge, non-deterministic behavior, and automation bias. The thesis concludes that the most effective role for language models in this context is within a semi-automated migration pipeline, where model outputs are anchored to deterministic checks (compiler, type checker, linter, static analysis) and testing, and where acceptance is tied to evidence-based, risk-driven review.

Keywords: large language models, software migration, AngularJS, Angular, JavaScript, TypeScript, refactoring, quality assurance, static analysis, automation bias

Sisällys

1 Johdanto **1**

2 Tausta **6**

2.1 Migraatio ohjelmistokehityksessä 6

2.2 Kone- ja syväoppiminen 7

2.3 Tokenisaatio 8

2.4 Laajat kielimallit 10

2.4.1 Kontekstista oppiminen 12

3 Kielimallien hyödyntäminen koodimigraatiossa **14**

3.1 Kielimallien sovelluskohteita 14

3.2 Kielimallien käyttöön liittyvät hyödyt 16

3.3 Kielimallien käyttöön liittyvät rajoitukset 20

3.4 Kielimallien käyttöön liittyvät haitat ja riskit 23

3.5 Oikotieoppimisen vaikutukset koodimigraatioon 26

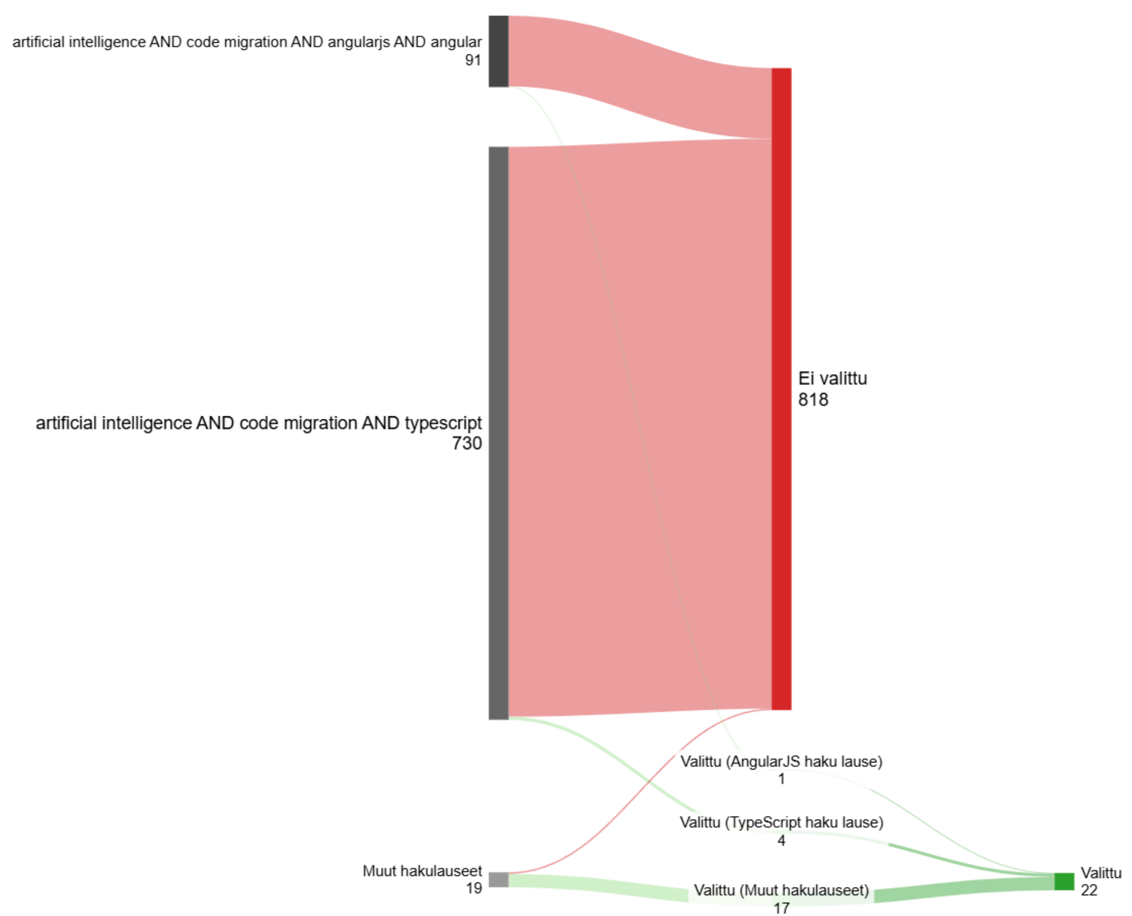
3.6 Kielimallien suorituskykyä parantavat tekijät migraatiossa 28

4 Pohdinta **31**

5 Yhteenveto **34**

Lähdeluettelo **37**

Taulukot



Kuva 1: Hakutermit ja tulokset

1 Johdanto

Tekninen velka on ohjelmistokehityksen kumuloituva kustannus siitä, että nykyhetkessä valitaan nopeampi mutta heikommin ylläpidettävä ratkaisu. Velkaa kertyy esimerkiksi tiukoista aikatauluista, epäyhtenäisistä arkkitehtuurivalinnoista, testauksen ja dokumentaation puutteista, vanhentuneista riippuvuuksista sekä rajallisesta näkyvyydestä tuotantoympäristön tilaan. Velka voi esiintyä koodin, arkkitehtuurin, testauksen, tietomallien ja julkaisuputken tasoilla, ja sen tuottama korko näkyy muutosten hitaampana läpimenoaikana, regressioina, integraatio-ongelmina, haavoittuvuuksina ja kasvavina operointikustannuksina. Pitkään elävissä järjestelmissä velka lukitsee teknologisia valintoja ja heikentää kykyä vastata uusiin vaatimuksiin - erityisesti kun ympäristöt, sääntely ja turvavaatimukset muuttuvat.

Velan hallinta edellyttää sekä teknisiä että prosessuaalisia käytäntöjä: velka tehdään näkyväksi (esim. erillinen velkarekisteri), sitä mitataan ja priorisoidaan riskin ja vaikutuksen mukaan (muutosten läpimenoaika, virhetiheys, MTTR (Mean Time To Repair), riippuvuuksien ikä), ja sen lyhentämiselle varataan säännöllistä kapasiteettia. Keskeisiä keinoja ovat jatkuva refaktorointi, modularisointi ja rajapintojen vakauttaminen, automatisoitu testaus ja staattinen analyysi, riippuvuuksien säännölliset päivitykset, havaittavuuden (observability) vahvistaminen sekä julkaisuputkien modernisointi. Arkkitehtuuritasolla vaiheittainen uudistaminen (esim. strangler pattern) pienentää riskejä verrattuna kaikkien muutosten tekemiseen yhdellä kerralla. Päätösten perusteleminen ja dokumentointi (ADR:t), yhteiset koodikäytännöt ja

oppien jakaminen varmistavat, ettei velka uusiudu heti ja että kompromissit pysyvät tietoisina, hallittuina liiketoimintapäätöksinä.

Monissa tuotantojärjestelmissä on yhä merkittäviä vanhoja fronttipuolen koodipohjia (esim. AngularJS), joiden ylläpidossa ja kehittämisessä korostuvat haasteet kuten tekninen velka, joka heikentää järjestelmän vakaata toimintaa, suuremmat kehityskustannukset, heikentynyt turvallisuus sekä päivitysominaisuuksien puute. Samaan aikaan modernit kehitysalustat ja TypeScript-ekosysteemi tarjoavat paremman työkaluketjun, tyypityksen ja yhteisötuen, minkä vuoksi vanhan koodipohjan modernisaatio uuteen on ennemmin tai myöhemmin välttämätöntä. Modernisaatio on lisäksi kiihtynyt, koska pitkäikäiset järjestelmät ovat organisaatioille kriittisiä (talous, logistiikka, terveydenhuolto), eikä niiden kokonaiskorvaaminen ole käytännössä mahdollista ilman huomattavia riskejä ja kustannuksia. Tekninen velka ja arkkitehtuuriset rajoitteet kuitenkin kasvavat ajan kuluessa, mikä lisää yhteensopimattomuutta nykyaikaisten teknologioiden kanssa ja haastaa integraation, turvallisuuden sekä kapasiteetin kasvattamisen. Vaikka modernisaatiota tukevia ohjeita ja työkaluja on saatavilla, työ on yhä pitkälti manuaalista, virhealtista ja hajanaista. Samaan aikaan laajat kielimallit lupaavat nopeuttaa koodin muunnosta ja tukea kehittäjiä refaktoroinnissa, mutta niiden todellinen hyöty, rajoitteet ja parhaat käyttötavat $legacy \rightarrow modern$ -kontekstissa ovat vielä huonosti jäsentyneet.

Tässä työssä fokus asetetaan $AngularJS \rightarrow Angular$-migraatioon. Syyt ovat käytännölliset: monissa tuotantoympäristöissä on edelleen huomattavia AngularJS-koodipohjia, kun taas moderni Angular ja TypeScript tarjoavat selkeän kohdealustan, vahvan työkaluketjun ja laajan ekosysteemituen. Näiden vuoksi juuri $AngularJS \rightarrow Angular$ on luonnollinen ja yleinen modernisaatiopolku, mutta työ pysyy silti monilta osin manuaalisena ja virheherkkänä. Vaikka $AngularJS \rightarrow Angular - migraatiota$ tukevia ohjeita ja työkaluja on saatavilla, niiden soveltaminen kriittisiin, pitkään eläneisiin järjestelmiin on haastavaa. Siksi tarkastelen erityisesti, miten

laajat kielimallit voivat käytännössä tukea tätä migraatiota: missä ne nopeuttavat koodin muunnosta, missä ne kompastuvat, ja millaiset käytännöt tuottavat eniten arvoa juuri $AngularJS \rightarrow Angular - kontekstissa.$

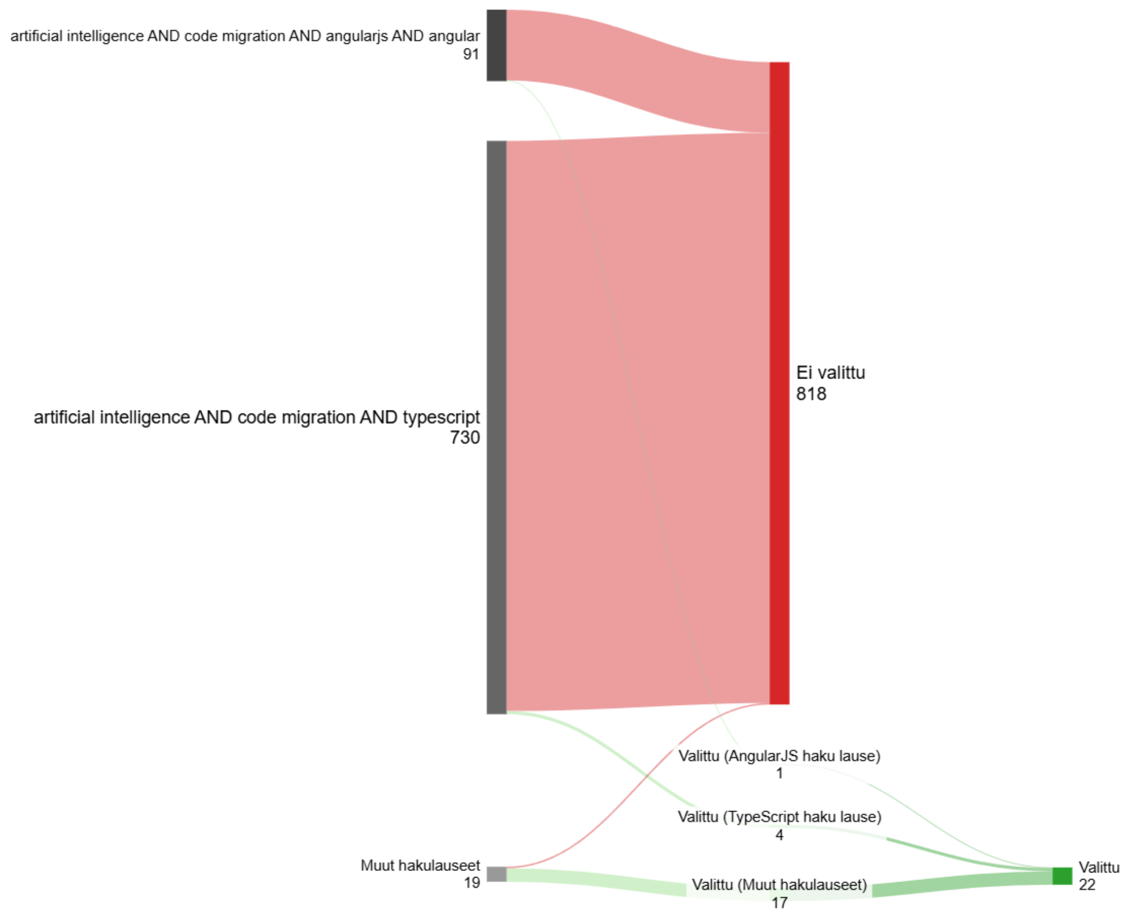
Tämän tutkielman tarkoituksena on selvittää, Miten kielimalleja voidaan hyödyntää $AngularJS \rightarrow Angular$ koodi migraatiossa ja selvittää myös mitä hyötyjä ja haittoja kielimalleissa on kun niitä käytetään koodimigraatiossa ja myös mitä rajoitteita kielimalleilla on migraatiossa ja selvittää millä tekijöillä voidaan parantaa kielimallien suoritus kykyä koodimigraatiossa. Tutkielmassa aihetta tarkastellaan seuraavien tutkimuskysymysten kautta:

TK1: Mitä rajoituksia kielimalleilla on $AngularJS \rightarrow Angular$-migraatioprosessissa?

TK2: Mitä tapoja on parantaa kielimallien suorituskykyä migraatioprosessissa?

TK3: Mitä hyötyjä kielimalleista on koodin migraatioprosessissa?

Tämän tutkielman lähdeaineisto etsittiin Google Scholarista ja käyttämäni hakulauseeni ovat **1.** "artificial intelligence AND code migration AND angularjs AND angular", **2.** "artificial intelligence AND code migration AND typescript"ja satunnaiset hakulauseet. Karsin hakutulokset aluksi myös siten että tulokset hakulaiseelta **1.** ovat vuodelta 2024 ja sitä uudempia. Haku lauseen **2.** kohdalla päädyin rajaamaan hakua vuodelle 2025 ja siitä eteenpäin tulosten suuresta määrästä johtuen. Valtaosa lähteistä on saatu erilaisilla kohdennetuilla hakulauseilla.



Kuva 1.1: Hakutermit ja tulokset

Luku 2 esittelee tutkielman taustan: migraation määritelmän ja rajauksen suhteessa refaktorointiin ja uudelleenkirjoittamiseen, keskeiset AngularJS:n ja Angularin arkkitehtuurin välisiä eroavaisuuksia sekä kone- ja syväoppimisen peruskäsitteet. Lisäksi käsitellään tokenisaatiota ja sen merkitystä koodin käsittelyssä, määritellään laajat kielimallit ominaisuuksineen ja rajoitteineen, ja kuvataan niiden käyttöä migraatiossa. Luvun lopussa kootaan hyödyt ja haitat sekä lyhyt katsaus aiempaan kirjallisuuteen ja työkaluihin. Luku 3 tarkastelee kielimallien suorituskykyä ja rajoitteita migraatiossa sekä tekijöitä, joilla suorituskykyä voidaan parantaa; samalla esitellään arviointiasetelma, käytetyt vertailumenetelmät ja käytetyt mittarit (kään-

nöksen onnistuminen, lint-virheet, testien läpäisy, käytetty aika) sekä oikotieoppiminen kontekstista oppimisessa (engl. Shortcut Learning in In-Context Learning). Luku 4 hahmottelee, miten migraatiota voidaan automatisoida kielimallien avulla esittämällä puoliksi automatisoidun putken luonnoksen, käyttöönoton edellytykset ja keskeiset riskit. Luku 5 kokoaa johtopäätökset ja vastaa tutkimuskysymyksiin.

2 Tausta

2.1 Migraatio ohjelmistokehityksessä

Migraatiolla tarkoitetaan ohjelmiston siirtämistä vanhasta teknologiapinosta uuteen siten, että järjestelmän ulkoinen toiminnallisuus säilyy, mutta ylläpidettävyys, turvallisuus ja suorituskyky paranevat [1], [2]. Se eroaa refaktoroinnista (rakenteen parantaminen ilman teknologian vaihtoa) ja uudelleenkirjoittamisesta (toteutus alusta uudella pinolla)[2]. *AngularJS* → *Angular* -kontekstissa tämä tarkoittaa muun muassa controller/scope-mallin korvaamista komponenteilla, reaktiivisten mallien (RxJS) käyttöönottoa ja TypeScript-tyypityksen vakiinnuttamista, joka toteutetaan yleensä vaiheittain riskien hallitsemiseksi [2].

Empiirisesti migraatio toteutuu useilla tavoilla: suuri osa projekteista siirtyy uuteen teknologiaan äkillisesti (engl. sudden), osa etenee vähitellen (engl. gradual), ja pienessä osassa siirtymä jää keskeneräiseksi [1], [3]. Usein vanhaa ja uutta kieltä ylläpidetään rinnakkain pitkään, ja vain harvat projektit poistavat vanhan kielen kokonaan [3]. Lisäksi tyypillistä on, että migraatio käynnistyy jo varhaisessa vaiheessa projektin elinkaarta, vaikka varsinainen "käännöspiste" (uuden teknologian tiedostojen määrä ylittää vanhan) voi tapahtua paljon myöhemmin [3].

Rinnakkaisvaiheen hallintaan vaikuttavat olennaisesti käännöstyökalun asetukset: käytännössä yleisimpiä ovat ulostulohakemiston määrittely (`outDir`), tiukan tyyppitarkistuksen käyttöönotto (`strict`) ja lähdekarttojen generointi (`sourceMap`)

[1], [3]. Sekakoodipohjaa (JS+TS) tuetaan usein ottamalla `allowJs` käyttöön, ja osa projekteista vaihtelee TypeScriptin kääntäjäasetukset migraation aikana (esim. tiukkuuden kiristäminen askelittain) [3]. Tällainen konfiguraatiostrategia mahdollistaa asteittainen siirtymän, mutta kasvattaa hetkellisesti työkaluketjun kompleksisuutta [3].

Laadun näkökulmasta on havaittu, että migraatio vähentää systemaattisesti muuttujiin ja näkyvyysalueisiin liittyviä ongelmia (esim. `var` → `let/const`) sekä parantaa ylläpidettävyyteen liittyviä käytäntöjä. Toisaalta vanhentuneisiin ohjelmointirajapintoihin (API) liittyvien varoitusten määrä voi kasvaa siirtymävaiheessa [1], [3]. API tarkoittaa sovellusohjelmointirajapintaa: se on ohjelmointikielestä riippumaton "sopimus" (engl. contract), joka määrittelee säännöt sille, miten järjestelmän tarjoamaa toiminnallisuutta voidaan käyttää [4]. Kognitiivinen kompleksisuus on yleinen haaste migraation aikana, etenkin kun koodipohja samaan aikaan kasvaa tai kun tyyppijärjestelmää otetaan käyttöön laajoissa funktioissa [3].

2.2 Kone- ja syväoppiminen

Koneoppiminen (ML) ja syväoppiminen (DL) ovat tekoälyn osa-alueita [5]. ML kokoaa menetelmät, joilla mallit oppivat datasta luokittelemaan, ennustamaan ja klusteroimaan, kun taas DL on ML:n alalaji, joka hyödyntää syviä neuroverkkoja erityisesti monimutkaisen ja suuren datan käsittelyyn [5]. DL suoriutuu usein parhaiten tehtävissä, joissa data on strukturoimatonta (teksti, kuva, puhe), mutta se vaatii tyypillisesti enemmän dataa ja laskentaa, perinteiset ML-mallit ovat usein kevyempiä ja tulkittavampia [5]. Viimeaikainen murros koodiin ja tekstiin liittyvissä tehtävissä perustuu Transformer-arkkitehtuuriin, jossa itsehuomio (engl. self-attention) mallintaa pitkän kantaman riippuvuuksia [6]. Tämän varaan rakennetut laajat kielimallit (LLM:t, esim. BERT/GPT) koulutetaan erittäin suurilla tekstikorpuksilla ja niitä hienosäädetään edelleen tiettyihin tehtäviin, kuten luokitteluun, kysymys–vastaus-

ongelmiin, käännökseen ja tekstin generointiin [5], [6].

ML:ää on viime vuosina otettu laajasti käyttöön, koska se kykenee hyödyntämään erilaisia reaalimaailman datalähteitä [5]. Oikean menetelmän valinta riippuu pitkälti käytettävissä olevan datan luonteesta [5]. Data voidaan jäsentää karkeasti neljään tyyppiin: strukturoitu (esim. taulukkomuotoinen finanssidata), strukturoimaton (tekstit, kuvat, videot), puolistrukturoitu (esim. JSON, lokit) ja aikasarjadata (aikaan indeksoidut havainnot) [5]. Vastaavasti ML-lähestymistavat jaotellaan tyypillisesti valvottuun (engl. supervised), valvomattomaan (engl. unsupervised), puolivalvottuun (engl. semi-supervised) ja vahvistusoppimiseen (engl. reinforcement learning) [5].

Käytännössä näiden välillä ei ole yksiselitteistä yksi-yhteen-vastaavuutta, mutta tietyt yhdistelmät ovat yleisiä [5]. Jäsennelty data (kuten taloudelliset tietueet) soveltuu usein valvottuun oppimiseen, jossa tavoitteena on luokittelu tai regressio (esim. luottoriskin arviointi) [5]. Strukturoimaton data, kuten kuvat ja videot, toimii hyvin DL-malleilla (konvoluutioverkot, transformerit) ja hyötyy myös valvomattomista tai itseohjautuvista esikoulutusmenetelmistä, kuten esitysten oppimisesta [5], [6]. Aikasarjadatassa yleisiä tehtäviä ovat ennustaminen ja poikkeamien havaitseminen (ARIMA, LSTM/transformer-pohjaiset mallit), kun taas puolistrukturoitu data yhdistää usein useita tekniikoita [5], [6]. Vahvistusoppiminen on luonteva valinta tilanteisiin, joissa tehdään peräkkäisiä päätöksiä ja oppiminen perustuu palkkioihin [5].

2.3 Tokenisaatio

Tokenisaatio on tekstin esikäsittelyn perusvaihe, jossa merkkivirta jaetaan sanoiksi tai tarkemmin tokeneiksi [7]. Ilman tokeneiden tunnistamista on hankalaa muodostaa korkeamman tason piirteitä tai tehdä myöhempiä vaiheita, kuten normalisointia, lemmatisointia ja kielimallintamista [7]. Token on yksittäinen esiintymä, kun taas

tyyppi (engl. type) viittaa tokenin uniikkiin muotoon, esimerkiksi lauseessa voi olla kaksi tokenia *"the"*, mutta vain yksi tyyppi *"the"* [7].

Ihmiselle tokenisointi on usein intuitiivista, mutta tietokoneelle se on sääntöpohjainen ja sovellusriippuvainen tehtävä [7]. Osa merkeistä toimii lähes aina erottimina esim. välilyönti, tabulaattori, rivinvaihto eikä niitä lasketa tokeneiksi [7]. Jotkin merkit ovat yleensä erottimia ja voivat samalla muodostaa oman tokeninsa esim. sulkeet, lainausmerkit, huutomerkki, kysymysmerkki [7]. Toiset merkit ovat kontekstiriippuvaisia: piste, pilkku ja kaksoispiste voivat kuulua lukuun (3.14, 1,000:50), lyhenteeseen esim. "Dr." tai merkitä virkkeen loppua [7]. Yleinen käytäntö epäselvissä tapauksissa on käsitellä piste sekä sanan katkaisijana että omana tokeninaan, jotta myöhemmissä vaiheissa voidaan päättää, onko kyseessä lauseen raja vai lyhenne [7].

Apostrofillä ja yhdysmerkillä on useita käyttötapoja, jotka vaikuttavat tokenisointiin [7]. Apostrofi säilytetään yleensä osana tokenia, kun sitä ympäröivät eierottimet (esim. "isn't", "D'Angelo") [7]. Jos apostrofia edeltää sana ja sitä seuraa yksikäsitteinen lopetusmerkki, tulkinta voi olla joko sisäisen lainauksen päättyminen tai omistusmuoto ("Tess' ") [7]. Yhdysmerkki toimii usein erottimena ja tokenina etenkin kaksoismerkkinä ("–"), mutta numeroiden välissä se voi olla osa rakennetta esim. puhelinnumero "555-1212" tai vähennysmerkki [7]. Käytännön ratkaisu on valita yhtenäinen periaate ja dokumentoida poikkeukset sovelluskohtaisesti.

Paras tulos saadaan räätälöimällä tokenisaattori käsiteltävälle aineistolle (esim. XML/JSON-lokit, uutisartikkelit, sosiaalisen median teksti), koska yleissääntöjen jälkeen tarvitaan lähes aina sovellusaluekohtaisia korjauksia [7]. Tokenisaatio on lisäksi kieliriippuvaista [7]: englannissa sanavälit helpottavat pilkkomista, kun taas kiinassa ja japanissa tarvitaan segmentointia ilman välilyöntejä. Suomi on morfologisesti rikas ja agglutinoiva kieli, jolloin pitkät yhdyssanat ja taivutusmuodot korostavat päätösten vaikutusta myöhempään analyysiin (esim. lemmatisointi ja sanaston koko).

Modernissa NLP:ssä käytetään usein myös alikomponenttisia menetelmiä (subword/BPE, WordPiece), joissa harvinaiset tai yhdyssanoiksi kasvavat muodot pilkotaan osiksi [6]. Tämä pienentää sanaston kokoa ja käsittelee sujuvammin sanojen uusia muotoja, mutta siirtää tulkintaa myöhempiin mallikerroksiin.

2.4 Laajat kielimallit

Laajat kielimallit (LLM:t) ovat transformer-arkkitehtuuriin perustuvia neuroverkkoja, jotka on opetettu erittäin suurilla tekstimäärillä ennustamaan seuraavaa tokenia aiemman kontekstin perusteella [6]. "Laaja" viittaa sekä parametrimäärään (miljardeja) että koulutusdatan ja laskennan kokoon [6]. Tämän mittaluokan ansiosta mallit kykenevät tuottamaan sujuvaa tekstiä ja yleistämään moniin tehtäviin ilman tehtäväkohtaista opetusta [6].

LLM:n perusrakenne koostuu sanastoon koodatuista tokeneista, upotusvektoreista sekä itsehuomioon (engl. self-attention) perustuvista lohkoista [6]. Kontekstilistan järjestystä mallinnetaan positioenkoodauksilla (esim. relatiiviset tai rotaatiopohjaiset menetelmät), ja esikoulutus tehdään tyypillisesti autoregressiivisella kielimallinnuksella, vaihtoehtoisesti käytetään peittämiseen (engl. masking) tai yhdistettyihin tavoitteisiin nojaavia menetelmiä [6]. Mallien suorituskyky noudattaa skaalautuvuussääntöjä: datan, parametrien ja laskennan kasvaessa häviö paranee ennustettavasti [6].

Käytännössä LLM:t otetaan käyttöön useamman vaiheen kautta [6]. Ensin suoritetaan pre-training laajalla datamäärällä, minkä jälkeen mallia hienosäädetään (engl. fine-tuning) tehtävä- tai ohjemuotoisella datalla (engl. instruction-tuning), jotta se noudattaa pyyntöjä ja toimii zero-shotissa paremmin [6]. Lisäksi mallia kohdistetaan (engl. alignment) ihmispalautteella esimerkiksi RLHF-menetelmällä (*reinforcement learning from human feedback*), jossa vakiintunut putki koostuu kolmesta vaiheesta: (i) supervised fine-tuning (SFT) demonstroinneilla, (ii) reward modelling

(RM), jossa malli opetetaan ennustamaan ihmisten preferenssejä vastauspareista, ja (iii) vahvistusoppimisvaihe (RL), jossa SFT-mallia optimoidaan RM:n antaman palkkion mukaan tyypillisesti on-policy -algoritmilla kuten PPO [8]. Vaihtoehtoisesti kohdistus voidaan tehdä suoralla preferenssioptimoinnilla, kuten DPO-menetelmällä (*direct preference optimization*), joka optimoi mallia suoraan preferenssipareista ilman erillistä RL-vaihetta [9]. Lopullinen hyödyntäminen perustuu kehotteiden muotoiluun: zero-shot tarkoittaa tehtävän ratkaisemista ilman esimerkkejä pelkän ohjeen perusteella, kun taas few-shot (engl. in-context learning) sisällyttää kehotteeseen muutamia syöte–vastaus-esimerkkejä [6], [10]. Chain-of-thought tekniikan hyödyntäminen kehotteessa puolestaan ohjaa mallia tuottamaan väliaskelia eli eksplisiittisen päättelyketjun, mikä voi parantaa erityisesti monivaiheista päättelyä vaativia tehtäviä [6], [11]. Monimutkaisia tehtäviä varten voidaan lisäksi rakentaa monikierroksisia agentteja ja työkalukutsuja [6].

Laajojen kielimallien keskeisiä kyvykkyyksiä ovat mm. yleistaitoinen tekstin ymmärtäminen ja tuottaminen, päättely ja tehtäväsuunnittelu, kontekstissa oppiminen (in-context learning) sekä soveltaminen koodiin, hakuun ja useisiin muihin sovellusalueisiin: kehitys on laajentunut myös multimodaalisiin malleihin (teksti + kuva/ääni) [6]. Samalla keskeisiä haasteita ovat kustannukset ja viive, datan ja mallin hallusinaatiot, vinoumat sekä turvallisuus [6]. Tehokkuutta parannetaan mm. parametritehokkaalla hienosäädöllä (parameter-efficient fine-tuning, PEFT: päivitetään vain pieni osa parametreista tai lisämoduuleja), karsinnalla (pruning: poistetaan osia mallista), kvantisoinnilla (quantization: pienennetään painojen/aktivaatioiden esitystarkkuutta) ja distillaatiolla (knowledge distillation: pienempi malli opetetaan suuremman mallin avulla) sekä pidentämällä konteksti-ikkunaa ja käyttämällä tehokkaampia huomioalgoritmeja (engl. efficient attention) [6].

LLM:ien hyödyllisyyttä määrittää myös se, miten malli viestii epävarmuutensa ja miten käyttäjä sen tulkitsee [12]. Käytännössä pitkät, vakuuttavat selitykset

voivat lisätä yli-luottamusta ilman että vastausten oikeellisuus paranee [12]. Siksi on suositeltavaa (i) näyttää ennusteen varmuus joko numeerisesti tai verbaalisina luokitteluina (esim. epävarma / melko varma / varma), (ii) aloittaa selitys nimenomaan tällä epävarmuusviestillä ja (iii) sovittaa selityksen tyyli mallin arvioituun varmuuteen: matalan varmuuden tilanteissa varovainen, korkean varmuuden tilanteissa suorempi [12]. Tämä parantaa käyttäjän kalibraatiota ja vähentää virheellistä luottamusta ilman muutoksia mallin arkkitehtuuriin [12].

Vastausten luotettavuutta voidaan parantaa hakuavusteisella generoinnilla (engl. retri-eval-augmented generation, RAG), jossa malli hakee kontekstiin lähdetekstejä ennen tuottoa, sekä työkalukutsuilla (tietokannat, laskimet, API:t) [6]. Näin generointi ankkuroidaan ulkoiseen tietoon ja tulokset ovat helpommin jäljitettävissä [6]. Tuotantokäytössä hyödynnetään lisäksi tehokkuustekniikoita, kuten kvantisointia ja karsintaa (mallin keventäminen), avain-arvo-välimuistia (engl. KV-cache) [6] ja spekulatiivista dekoodausta viiveen pienentämiseksi sekä riittävän pitkää kontekstiikkunaa tai dokumenttien paloittelua [6]. Suunnittelussa on hyvä määrittää selkeä vaatimustaso (tarkkuus, viive, kustannus) ja suojakaiteet (sisällön suodatus, käyttöoikeudet, audit-loki), jotta malli on sekä käyttökelpoinen että turvallinen [6].

2.4.1 Kontekstista oppiminen

Kontekstista oppimisella (engl. in-context learning, ICL) tarkoitetaan sitä, että laaja kielimalli pystyy omaksumaan tehtävänannon ja sille annetut demonstraatioesimerkit ilman erillistä hienosäätöä [13]. Käytännössä tehtävä esitetään kehotteena, joka sisältää ohjeen sekä yhden tai useamman esimerkkiparin, ja malli päättelee toivotun vastaustavan annetun kontekstin perusteella ilman parametrien päivittämistä [13].

ICL on keskeinen mekanismi, jonka ansiosta laajoja kielimalleja voidaan soveltaa uusiin tehtäviin ilman tehtäväkohtaista koulutusta. Samalla menetelmä altistuu oikotieoppimiselle, jossa malli hyödyntää pinnallisia mutta tilastollisesti vahvo-

ja vihjeitä yleistettävän ratkaisuperiaatteen sijaan [13]. Tämän seurauksena malli voi toimia hyvin tutuissa asetelmissa, mutta suorituskyky heikentyä, kun syötteen rakenne, tyyli tai esitystapa muuttuu.

ICL-yhteydessä oikotiet voidaan jakaa instinktiivisiin ja kontekstista omaksuttuihin oikoteihin [13]. Ensiksi mainitut heijastavat mallin koulutusvaiheen vinoumia ja kehotteen muotoilun vaikutuksia, kuten vanilla-label bias, context-label bias ja domain-label bias. Jälkimmäiset syntyvät, kun malli poimii demonstraatioista näennäisiä korrelaatioita ja käyttää niitä tehtävärakenteen sijaan, tällaisia ovat esimerkiksi lexicon-, concept-, position- ja text style -oikotiet sekä group dynamics [13].

Oikotieoppimisen syntyyn vaikuttavat sekä mallin ominaisuudet että kehotteen sisältö. Erityisesti vinoutuneet demonstraatiot voivat ohjata mallia hyödyntämään epäolennaisia signaaleja, eikä ilmiö välttämättä lievenny mallia skaalaamalla, sillä suurempien mallien on havaittu monissa asetelmissa olevan taipuvaisia hyödyntämään oikoteitä sekä mallin sisäisten että kontekstista opittujen vinoumien kautta [13].

3 Kielimallien hyödyntäminen koodimigraatiossa

3.1 Kielimallien sovelluskohteita

Kielimalleja voidaan hyödyntää ohjelmistomigraatiossa sekä itse koodin muunnoksessa että laajemmin legacy-järjestelmien modernisoinnin tukena [2], [14]. Perinteisesti migraatio, kuten siirtyminen yhdestä ohjelmointikielestä tai teknologiasta toiseen, on perustunut manuaaliseen refaktorointiin, sääntöpohjaisiin työkaluihin tai AST-tason muunnoksiin [2]. Nämä lähestymistavat ovat kuitenkin työläitä, vaikeasti skaalautuvia ja usein herkkiä koodipohjan erityispiirteille [2], [15]. Uudet laajat kielimallit (LLM:t) tuovat tähän prosessiin "älykkään käännöskoneen", joka pystyy samaan aikaan tulkitsemaan koodin rakennetta, dokumentaatiota ja nimeämiskäytäntöjä sekä tuottamaan kohdekieleen tyypitetyn, luettavan version [2], [16].

Yleisemmässä legacy-koodin modernisoinnissa kielimalleja voidaan hyödyntää osana AI-pohjaisia refaktorointiputkia [2], [15]. Koneoppimismallit voivat analysoida suuria koodikantoja, tunnistaa toistuvia rakenteita, teknistä velkaa ja koodihajuja sekä ehdottaa tai generoida korvaavia toteutuksia säilyttäen alkuperäisen toiminnallisuuden [2], [15]. Sama teknologia voidaan valjastaa myös varsinaiseen migraatioon: esimerkiksi legacy-kielestä (kuten COBOL) voidaan tuottaa modernia Java-koodia yhdistämällä luonnollisen kielen käsittelyä ja koodin semanttista mallinnusta [15].

Näissä tapauksissa kielimalli toimii eräänlaisena kaksikielisenä kääntäjänä, joka tuntee sekä lähde- että kohdekielen idiomit ja kirjastot [15].

JavaScript–TypeScript-migraatio voidaan nähdä tästä näkökulmasta erityisen luontevana käyttökohteena [3], [16]. TypeScript on JavaScriptin tyypitetty superset, joten suuri osa syntaksista on yhteistä, mutta lisäksi tarvitaan tyyppipäätelmää, SAPUI5-kaltaisten frameworkien tuntemusta ja kykyä tunnistaa käyttöön sopivat rajapintatyypit [16]. GPT-4:ää voidaan hyödyntää nimenomaan SAPUI5-sovellusten migraatiossa JavaScriptistä TypeScriptiin siten, että kielimalli saa syötteenä alkuperäisen koodin sekä kontekstia (esimerkiksi controller-rakenteet, datamallit ja bindaukset) ja tuottaa vastaavan TS/TSX-toteutuksen, jossa on lisätty tyyppimääritykset sekä framework-spesifit annotaatiot [16].

Kielimallien käyttö ei rajoitu yksittäisten tiedostojen muuntamiseen, vaan ne voidaan kytkeä osaksi automaattista migraatioputkea [2], [14]. Työkalu voidaan toteuttaa esimerkiksi SAP Business Technology Platformin (BTP) AI Core -palvelun päälle siten, että GPT-4-malli julkaistaan generative AI hubin kautta ja sitä kutsutaan skriptipohjaisella työkalulla, joka käy läpi projektin lähdekoodia ja korvaa .js/.jsx-tiedostot .ts/.tsx-versioilla [16]. Tällainen lähestymistapa yhdistää kielimallin joustavuuden yritysympäristön vaatimuksiin (NDA:t, sisäiset repositoriot, automaattiset build-prosessit) ja mahdollistaa sen, että migraatio voidaan tehdä hallitusti vaiheittain ilman, että koodi tarvitsee viedä ulkoisiin pilvipalveluihin [16].

Käytännön tasolla kielimallia hyödynnetään migraatiossa tyypillisesti kolmella tavalla [2], [14]. Ensinnäkin se toimii “älykkäänä lähdekoodimuuntimena”, joka muuntaa yksittäisiä funktioita ja luokkia JavaScriptistä TypeScriptiin, lisää tyyppiparametreja ja ehdottaa sopivia rajapintoja esimerkiksi UI5:n malleille ja kontrolleille [16]. Toiseksi malli toimii avustettuna kehitystyökaluna: kehittäjä voi tarkentaa kehotetta, pyytää perusteluja tyyppivalinnoille ja tehdä korjauksia vuorovaikutteisesti [15], [16]. Kolmanneksi kielimalli voidaan yhdistää muihin AI-työkaluihin, ku-

ten staattiseen analyysiin tai testigenerointiin, jolloin migraation yhteydessä tuotetaan myös uusia yksikkötestejä ja varmistetaan, ettei olemassa oleva toiminnallisuus rikkoudu [2], [14], [17].

Tutkimusten perusteella kielimallipohjainen migraatio tuo merkittäviä etuja erityisesti tuottavuuden ja ylläpidettävyyden kannalta [2], [14]. AI-avusteinen refaktorointi vähentää manuaalisen työn määrää, parantaa suorituskykyä ja auttaa pienentämään teknistä velkaa, kun toistuvat optimoinnit ja koodipuhdistukset voidaan automatisoida [2], [15]. Käytännön kokemusten perusteella GPT-4-pohjainen työkalu nopeuttaa siirtymää merkittävästi verrattuna täysin manuaaliseen tai pelkästään "chat-ikkunassa" tehtävään, tiedosto kerrallaan etenevään migraatioon, ja parantaa samalla lopputuloksen tyypitettyä laatua [16].

Hyötyjen rinnalla kielimalleihin perustuva migraatio tuo myös uusia rajoitteita ja riskejä [15], [17]. On huomattava, että mallit eivät aina ymmärrä sovelluslogiikan liiketoimintakontekstia, mikä voi johtaa hienovaraisiin regressioihin, vaikka koodi olisi syntaktisesti ja tyypillisesti oikein [15], [17]. Lisäksi mallit voivat tuottaa liiankin itsevarmoja, mutta virheellisiä tyyppipäätelmiä tai käyttää framework-ominaisuuksia epätyypillisellä tavalla, mikä kasvattaa jälkikäteisen koodikatselmoinnin tarvetta [15], [16]. Luottamukseen liittyvät haasteet, selitettävyyden puute, laskentakustannukset ja mahdolliset tietosuoja- ja IP-riskit rajaavat osaltaan sitä, miten aggressiivisesti migraatiota voidaan automatisoida [15], [16].

3.2 Kielimallien käyttöön liittyvät hyödyt

Ohjelmistojen elinkaari on usein huomattavasti pidempi kuin yksittäisten teknologioiden tai ohjelmointikielten. Monissa organisaatioissa on edelleen käytössä järjestelmiä, jotka on rakennettu vuosikymmeniä sitten kielillä ja alustoilla, joiden osaajia on yhä vähemmän ja joiden ympärille rakentunut ekosysteemi on pitkälti kadonnut. Tällaiset niin sanotut legacy-järjestelmät ovat työläitä ylläpitää, laajentaa

ja integroida nykyisiin pilvipohjaisiin ratkaisuihin. Koodimigraatiolla tarkoitetaan prosessia, jossa olemassa oleva ratkaisu siirretään uuteen teknologiaan esimerkiksi vaihtamalla ohjelmointikieltä, modernisoimalla arkkitehtuuria tai uudistamalla merkittäviä osia sovelluspinosta. Perinteisesti koodimigraatio on tarkoittanut pitkälti manuaalista työtä, jossa kehittäjät tulkitsevat vanhaa koodia ja kirjoittavat sen uudelleen uusilla kielillä ja työkaluilla. Tämä on hidasta, kallista ja altista inhimillisille virheille [15].

Viime vuosina laajat kielimallit ovat nousseet keskeiseksi tutkimus- ja kehityskohteeksi, ja erityisesti ohjelmakoodiin erikoistuneet mallit ovat herättäneet kiinnostusta koodimigraation tukena. Nykyaikaiset, transformer-arkkitehtuuriin perustuvat mallit kykenevät sekä lukemaan että tuottamaan koodia useilla eri kielillä. Ne tunnistavat koodista rakenteellisia ja semanttisia kuvioita ja pystyvät tuottamaan niiden pohjalta muunnelmia, kuten refaktorointiehdotuksia tai käännöksiä toiseen kieleen. Koodin refaktorointikykyä on myös mitattu empiirisesti esimerkiksi (i) syntaktisen kelvollisuuden ja toiminnallisen vastaavuuden kautta (unit-testit) sekä (ii) rakenteellisten ohjelmistomittarien avulla (esim. LOC, cyclomatic complexity ja Halstead-mittarit) [18]. Lisäksi harvaesimerkkisessä asetelmassa (engl. few-shot) on havaittu, että malli tuottaa usein yksinkertaisemman version, jolloin syklomaattinen kompleksisuus ja koodirivien määrä (LOC) pienenevät keskimäärin [19]. Toisaalta refaktoroinnin vaikutusta on arvioitu myös staattisten analyysien varoitus- ja virheilmoituksilla (esim. viestit/rivi), vaikka tällöin toiminnallista oikeellisuutta ei välttämättä erikseen validoida [20]. Tämän kehityksen myötä koodimigraatiota ei tarvitse enää nähdä pelkkänä “vanhan koodin uudelleenkirjoittamisena”, vaan prosessina, jossa merkittävä osa työstä voidaan osin automatisoida ja jossa ihmisen rooli siirtyy yhä enemmän valvontaan, suunnitteluun ja laadun arviointiin [5], [6].

Yksi keskeinen haaste koodimigraatiossa on legacy-järjestelmän ymmärtäminen. Pitkän elinkaaren aikana kertynyt tekninen velka, puutteellinen tai vanhentunut

dokumentaatio sekä tilapäisiksi tarkoitetut "pikakorjaukset" johtavat helposti siihen, että järjestelmän todellinen liiketoimintalogiikka ja riippuvuudet ovat epäselviä jopa kokeneillekin kehittäjille. Kielimallit tarjoavat tähän uudenlaisia työkaluja. Ne voivat tiivistää laajoja kooditiedostoja, kuvata funktioiden ja luokkien toimintaa luonnollisella kielellä sekä ehdottaa nimeämis- ja rakenneparannuksia. Tällaiset ominaisuudet auttavat sekä yksittäistä kehittäjää perehtymään järjestelmään että koko tiimiä muodostamaan yhteistä käsitystä järjestelmän rakenteesta. Kun koodin lukeminen ja hahmottaminen tehostuu, myös migraation suunnittelu muuttuu realistisemmaksi ja riskit helpommin hallittaviksi [6], [7].

Toinen keskeinen hyöty liittyy varsinaiseen käännökseen kielestä toiseen. Kielimallit voivat toimia eräänlaisena "käännösparina" eri ohjelmointikielten välillä: ne voivat tuottaa esimerkiksi Python-versioita vanhasta Java-koodista tai muuntaa COBOL-toteutuksia modernimmille kielille. Automaatio ei ole täydellistä eikä poista manuaalisen tarkastuksen tarvetta, mutta voi merkittävästi keventää raskainta, mekaanista vaihetta [2]. Kun peruslogiikka ja rakenne on mallin avulla siirretty uuteen kieleen, kehittäjän ei tarvitse aloittaa työtään tyhjästä, vaan voi keskittyä tuloksen tarkistamiseen, hienosäätöön ja optimointiin [2].

Kielimallien hyöty ei kuitenkaan rajoitu pelkästään syntaksitasoisiin muunnoksiin. Laajemmin tarkasteltuna ne voivat tukea koodin laadun, turvallisuuden ja ylläpidettävyyden parantamista osana migraatioprosessia. Monet nykyiset työkalut hyödyntävät koneoppimista tunnistaakseen koodista toistuvia virhemalleja, koodihajuja ja mahdollisia haavoittuvuuksia. Kun migraation yhteydessä joka tapauksessa käydään läpi suuria osia järjestelmästä, voidaan samassa yhteydessä automatisoidusti osoittaa kohdat, joissa esimerkiksi resurssien hallinta, virheenkäsittely tai tietoturvakäytännöt ovat puutteellisia. Näin migraatiosta tulee tilaisuus paitsi päivittää teknologiaa myös nostaa koko järjestelmän laatutasoa [5], [15].

Koodimigraatiossa korostuvat myös osaamiseen ja hiljaiseen tietoon liittyvät ky-

symykset. Monet legacy-järjestelmät on toteutettu teknologioilla, joiden syvällistä osaamista on enää harvoilla. Esimerkiksi finanssialalla COBOL on yhä kriittinen monissa ydinjärjestelmissä: erään johtavan hollantilaisen pankin tapaustutkimuksessa COBOL toimii pankin IT-toimintojen selkärankana, ja sitä käytetään keskeisissä prosesseissa, kuten maksujen ja transaktioiden käsittelyssä. Samalla COBOL-osaajapula ja eläköityminen aiheuttavat merkittäviä tiedonsiirron ja ylläpidon haasteita [21]. Lisäksi alkuperäiset kehittäjät eivät usein ole enää organisaation käytettävissä.

Kielimallit voivat toimia eräänlaisena sillanrakentajana eri kehittäjä- ja teknologiasukupolvien välillä. Uudempiin kieliin tottuneet kehittäjät voivat pyytää mallia selittämään vanhan koodin toimintaa tai tuottamaan vastaavan ratkaisun tutummalla kielellä. Tällainen "virtuaalinen pariohjelmointi" madaltaa kynnystä osallistua migraatioprojektiin ilman syvää asiantuntemusta jokaisesta historiallisesta teknologiasta. Samalla se tukee oppimista: kehittäjä näkee rinnakkain vanhan ja uuden ratkaisun ja oppii, miten sama algoritmi tai liiketoimintasääntö voidaan ilmaista eri kielissä [22].

Organisaation näkökulmasta kielimallien hyödyntäminen koodimigraatiossa kytkeytyy myös resursointiin ja projektinhallintaan [1]. Koska migraatio on perinteisesti ollut työvoimavaltaista, suuria modernisointihankkeita on usein lykätty kustannus- ja riskisyistä. Jos kielimallit pystyvät pienentämään manuaalisen työn osuutta, ne voivat parantaa tällaisen hankkeen kustannus–hyöty-suhdetta ja tehdä siitä strategisesti houkuttelevamman. Tämä voi edesauttaa sitä, että kriittisiä järjestelmiä uskalletaan päivittää ajoissa sen sijaan, että niiden annetaan kertyä tekniseksi veloiksi. Samalla on muistettava, että myöskään kielimallit eivät ole ilmaisia: niiden käyttöönotto, mahdollinen räätälöinti ja integrointi olemassa oleviin kehitysputkiin vaativat omia investointejaan ja uutta osaamista [5], [6].

3.3 Kielimallien käyttöön liittyvät rajoitukset

Kielimallien hyödyntämiseen koodimigraatiossa liittyy useita suorituskykyyn vaikuttavia rajoituksia, jotka on syytä tunnistaa erityisesti liiketoimintakriittisissä järjestelmissä. Vaikka LLM-mallit ovat osoittautuneet tehokkaiksi koodin generoinnissa ja käännöksessä, ne on koulutettu yleiskäyttöiseen luonnollisen kielen ja ohjelmakoodin käsittelyyn, eivätkä ne optimoi suoritustaan yksittäisen organisaation migraatiotavoitteiden, arkkitehtuurin tai koodistandardien mukaan [5], [6]. Tämä näkyy erityisesti siinä, miten mallit tasapainottavat syntaktisen oikeellisuuden, tyyppiturvallisuuden ja alkuperäisen liiketoimintalogiikan säilymisen välillä.

Tässä luvussa esitetyt rajoitteet on koottu kirjallisuudesta, mutta niiden ryhmittely, keskinäinen painotus ja migraatiokontekstiin sidotut vaikutusarviot ovat tämän työn tekijän synteesiä. Toisin sanoen viitteet tukevat yksittäisiä ilmiöitä, kun taas kokonaisjäsennys ja käytännön tulkinnat perustuvat omaan pohdintaan.

Ensimmäinen keskeinen rajoitus liittyy semanttiseen säilyvyyteen. Sekä automaattista koodikäännöstä että AI-avusteista refaktorointia käsittelevä tutkimus korostaa, että mallit onnistuvat usein tuottamaan syntaktisesti oikeaa kohdekoodia, mutta eivät välttämättä säilytä kaikkia liiketoimintasääntöjä tai reunaehtoja muuttumattomina [2], [15]. AI-pohjaisissa refaktorointityökaluissa on havaittu tapauksia, joissa malli optimoi esimerkiksi ehtolauseita tai silmukoita tavalla, joka muuttaa hienovaraisesti laskennan lopputulosta tai virheenkäsittelyä, vaikka koodi läpäisee kääntäjän ja yksinkertaiset testit. Tämän vuoksi LLM-pohjainen migraatio vaatii tyypillisesti manuaalista tarkastusta ja regressiotestausta, mikä rajoittaa saavutettavissa olevaa automaatioastetta.

Toinen rajoite koskee framework- ja domain-spesifistä osaamista. Legacy-järjestelmät hyödyntävät usein organisaatiokohtaisia kirjastoja ja kehysratkaisuja, joiden esimerkkejä on vain vähän mallien koulutusdatassa. Sekä $JavaScript \rightarrow TypeScript$-migraatiota että yleisemmin web-sovellusten komponenttimigraatiota käsittelevät

tutkimukset osoittavat, että automaattiset työkalut kompastuvat erityisesti dynaamisiin kieliominaisuuksiin, epäselviin tyyppirajoihin ja framework-kohtaisiin konventioihin [3]. SAPUI5-TypeScript-migraatiota GPT-4:llä tarkastellut työ kuvaa käytännön rajoitteita: malli tarvitsee usein tarkkaan suunniteltuja kehotteita noudattaakseen UI5:n nimeämis- ja arkkitehtuurikäytäntöjä ja tekee virheitä monimutkaisessa tyyppipäättelyssä esimerkiksi syvälle sisäkkäisten objektien tai kehyskohtaisen metadatan osalta. Samalla tavoin laajemmat katsaukset AI- ja ML-pohjaiseen komponenttimigraatioon korostavat, että monet nykyiset työkalut ovat vahvasti kieli- tai framework-riippuvaisia, eikä niiden suorituskyky yleisty suoraan uusiin ympäristöihin [5], [15].

Kolmantena rajoitteena voidaan pitää mallien kestävyyttä ja yleistettävyyttä. LLM:t ovat alttiita niin kutsutulle oikotieoppimiselle (shortcut learning), jossa malli nojaa helppoihin, mutta hauraisiin päätössääntöihin sen sijaan, että oppisi tehtävän kannalta olennaisen rakenteen [13]. Kontekstista oppimisen yhteydessä malli voi esimerkiksi päätellä migraatiostrategian yksittäisten esimerkkien pintapiirteiden (tiedostonimi, kommenttityyli, option järjestys) perusteella ja epäonnistua, kun koodipohjan rakenne poikkeaa hieman annetuista esimerkeistä. Tämä heikentää migraation ennustettavuutta: yksittäiset tiedostot voivat kääntyä lähes täydellisesti, kun taas hyvin samankaltaiset moduulit vaativat runsaasti käsityötä. Vastaavaa ilmiötä on havaittu myös yleisissä ohjelmakoodiin erikoistuneissa malleissa, joissa suoritustaso heilahtelee voimakkaasti riippuen syötteen pienistäkin muutoksista [6].

Myös mallien epävarmuuden hallinta ja luottamuksen kalibrointi rajoittavat niiden käyttöä täysin automaattisissa migraatioissa. Tutkimus LLM:ien kalibraatiosta osoittaa, että käyttäjät yliarvioivat usein mallin vastauksen oikeellisuuden, erityisesti silloin, kun malli tuottaa pitkiä ja vakuuttavia selityksiä [12]. Lisäksi AI-avusteisen refaktoroinnin kirjallisuudessa korostetaan luottamukseen ja selitettävyyteen liittyviä haasteita: mallit koetaan “mustina laatikkoina”, joiden tekemiä muutoksia ei

hyväksytä ilman ylimääräisiä perusteluja ja koodikatselmointia [15], [22]. Tämä näkyy suoraan suorituskyvyssä: vaikka malli pystyisi kääntämään suuren määrän tiedostoja, todellinen läpimeno riippuu siitä, kuinka paljon aikaa kehittäjät käyttävät muutosten tarkistamiseen ja korjaamiseen.

Neljäs rajoite liittyy skaalautuvuuteen ja laskentakustannuksiin. Laajat enterprise-koodikannat koostuvat helposti miljoonista koodiriveistä ja tuhansista tiedostoista, joita ei voida syöttää mallille yhtenä kokonaisuutena konteksti-ikkunan rajoitusten vuoksi. Koodi joudutaan pilkkomaan osiin, mikä vaikeuttaa pitkien riippuvuuksien ja arkkitehtuuristen rakenteiden (esim. laajat palvelukerroksen rajapinnat tai yhteiset datamallit) huomioimista. Lisäksi syväoppimismalleihin perustuvat refaktorointityökalut ovat laskennallisesti raskaita, mikä voi rajoittaa niiden käyttöä CI-putkessa tai interaktiivisissa työkaluissa [5], [6]. Aiemmissa tutkimuksissa on todettu, että suurten monoliittisten koodikantojen analysointi ja muuntaminen AI-malleilla voi kestää tunteja tai päiviä, mikä tekee reaaliaikaisesta optimoinnista epäkäytännöllistä [2], [15].

Lopuksi myös migraation laadun arviointiin liittyy rajoitteita. Web-sovellusten komponenttien migraatiota kartoittavat katsaukset korostavat, että yhtenäiset vertailumittarit ja metriikat ovat vasta kehittymässä: käytössä on joukko syntaktisia, rakenteellisia ja testipohjaisia mittareita, mutta standardoitua tapaa mitata esimerkiksi semanttista ekvivalenssia tai kokonaisvaltaista koodilaadun muutosta ei ole vakiintunut [6]. JavaScript–TypeScript-migraatiota laajasti analysoinut tutkimus osoittaa, että vaikka tyyppiturvallisuuteen liittyvät ongelmat usein vähenevät, migraatio voi samanaikaisesti lisätä kognitiivista kompleksisuutta ja tuoda uusia varoituksia esimerkiksi deprekaatioista [3]. Tämä viittaa siihen, että kielimallien suorituskykyä ei voida arvioida pelkästään “onnistuneiden käännösten” määrällä, vaan tarvitaan monimuotoisempaa, sekä teknisiä että inhimillisiä tekijöitä huomioivaa arviointia.

3.4 Kielimallien käyttöön liittyvät haitat ja riskit

Kielimalleihin perustuvan koodimigraation hyödyt eivät tule ilmaiseksi, vaan menetelmä tuo mukanaan myös uusia rajoitteita ja riskejä, jotka on huomioitava erityisesti liiketoimintakriittisissä järjestelmissä [1]. Vaikka laajat kielimallit kykenevät tuottamaan syntaktisesti oikeaa ja usein hyvin tyypitettyä koodia useilla kielillä, niiden tuotos ei ole determinististä, eikä se perustu formaaleihin takuisiin vaan tilastolliseen päättelyyn laajoista harjoitusaineistoista [5], [6]. Tämän vuoksi LLM-pohjainen migraatio ei korvaa perinteisiä testaus- ja verifiointimenetelmiä, vaan pikemminkin lisää tarvetta järjestelmälliselle laadunvarmistukselle.

Yksi keskeinen haitta liittyy siirtyvän koodin semanttiseen oikeellisuuteen [1]. Kielimalli voi tuottaa koodia, joka on syntaktisesti virheetöntä ja tyyppijärjestelmän mielestä johdonmukaista, mutta joka muuttaa hienovaraisesti alkuperäistä liiketoimintalogiikkaa. Tämä riski korostuu tilanteissa, joissa lähdekoodi sisältää poikkeuskäsittelyä, rinnakkaisuutta tai monimutkaisia reunaehtoja, joita malli ei kontekstin perusteella täysin ymmärrä. LLM:t oppivat tyypillisiä ratkaisutapoja ja idiomeja koulutusdatastaan, eivät yksittäisen järjestelmän “todellista” spesifikaatiota, mikä voi johtaa regressioihin erityisesti harvinaisten tai organisaatiokohtaisten rakenteiden kohdalla [6]. Myös automaattista koodikäännöstä käsittelevä tutkimus korostaa, että vaikka syntaksitasoinen käännös voidaan pitkälti automatisoida, semanttinen ekvivalenssi joudutaan edelleen varmistamaan testeillä ja manuaalisella katselmoinnilla [2].

Mallien taipumus niin sanottuun oikotieoppimiseen muodostaa toisen merkittävän rajoitteen. Oikotieoppimisella tarkoitetaan sitä, että malli oppii ratkaisemaan tehtäviä hyödyntämällä pinnallisia mutta tilastollisesti vahvoja vihjeitä sen sijaan, että se omaksuisi yleistettävän ratkaisuperiaatteen [13]. Koodimigraatiossa tämä voi ilmetä siten, että malli oppii mekaanisen vastaavuuden tiettyjen API-kutsujen, luokkarakenteiden tai nimeämiskäytäntöjen välille, vaikka se ei tosiasiassa hallitsisi

muunnoksen taustalla olevia periaatteita. Tämän seurauksena malli voi tuottaa uskottavan lopputuloksen tavanomaisissa tapauksissa, mutta epäonnistua erikoistilanteissa tai organisaatiokohtaisissa laajennuksissa. Oikotieoppiminen heikentää siten mallin häiriönsietokykyä ja vaikeuttaa sen luotettavaa arviointia, koska kehittäjän on vaikea ennakoida, milloin malli yleistää opitun muunnossäännön uuteen tilanteeseen ja milloin se nojaa vain pintatason korrelaatioihin.

Luottamukseen liittyvät ongelmat syventävät tätä haastetta. Ihmisillä on taipumus yliarvioida kielimallien antamien vastausten luotettavuus erityisesti silloin, kun vaste on pitkältä ja hyvin perustellulta vaikuttavaa selitystekstiä sisältävä [12]. Tämä kalibrointivaje näkyy koodimigraatiossa esimerkiksi niin, että kehittäjä hyväksyy mallin ehdottaman TypeScript-version suoraan tarkastamatta ehdotettuja muutoksia, koska koodi kompiloituu ja malli on lisäksi tuottanut vakuuttavan selityksen tyyppivalinnoille. Selitys ei kuitenkaan ole tae siitä, että mallin esittämä ratkaisuehdotus olisi oikein, se on vain osa samaa generatiivista prosessia. Tästä seuraa automaatioharha (engl. automation bias): riski, että tarkastukseen käytettyä aikaa vähennetään juuri niissä kohdissa, joissa tarkkuutta eniten tarvittaisiin [12].

Selitettävyyden ja jäljitettävyyden puute vaikeuttaa myös virheiden korjaamista. Perinteisissä, sääntöpohjaisissa migraatiotyökaluissa käännössäännöt ovat eksplisiittisiä, ja kehittäjä voi jäljittää virheen tiettyyn muunnossääntöön. Kielimallipohjaisessa järjestelmässä yksittäisen muunnosratkaisun taustalla olevia perusteita on vaikea jäljittää, koska ne eivät perustu eksplisiittisiin muunnossääntöihin vaan mallin sisäiseen päättelyyn. Tämä hankaloittaa sekä tyypillisten virhemallien tunnistamista että korjaavien toimenpiteiden systematisointia, ongelmat korjataan usein tapauskohtaisesti, mikä voi pitkällä aikavälillä lisätä ylläpitokustannuksia [15].

Toinen keskeinen haitta liittyy mallien rajoittuneeseen toimialakohtaiseen tietoon [1]. LLM:t on yleensä koulutettu pääosin avoimen lähdekoodin ja julkisten lähteiden perusteella, jolloin niiden tuntemus suljetuista yritissovelluskehyksistä, si-

säisistä kirjastoista tai organisaation omista koodaustyyleistä on väistämättä puutteellinen [6]. Tämä näkyy esimerkiksi SAPUI5- tai muiden enterprise-kehysten migraatiossa siten, että malli suosii yleisiä JavaScript/TypeScript-idiomeja, mutta ei välttämättä noudata framework-kohtaisia suosituksia, annotaatioita tai suorituskykyrajoitteita. Myös JavaScript–TypeScript-migraatiota analysoineet empiiriset tutkimukset osoittavat, että migraatiossa syntyy uusia haasteita, kuten kognitiivisen kompleksisuuden kasvu ja deprekoituihin rajapintoihin liittyvien ongelmien lisääntyminen [3]. LLM-pohjainen muunnos voi pahimmillaan vahvistaa näitä ilmiöitä, jos se nojaa vanhentuneisiin tai epätyypillisiin esimerkkeihin.

Tietosuojaan ja immateriaalioikeuksiin liittyvät riskit rajaavat omalta osaltaan kielimallien käyttöä migraatiossa. Jotta malli voisi analysoida ja muuntaa koodia, lähdekoodi on tyypillisesti lähetettävä mallipalveluun, joka pyörii joko julkisessa pilvessä tai kolmannen osapuolen infrastruktuurissa. Tämä voi olla ristiriidassa NDA-sopimusten, toimialakohtaisten säännösten tai organisaation sisäisten turvallisuuskäytäntöjen kanssa, erityisesti jos koodi sisältää liikesalaisuuksia tai asiakasdataa [15]. Vaikka erilliset paikalliset ratkaisut ja suljetut mallit lieventävät näitä huolia, ne lisäävät toisaalta käyttöönoton kustannuksia ja teknistä kompleksisuutta. Lisäksi avoimen koulutusdatan käyttö herättää epävarmuutta siitä, onko malli oppinut lisenssiehdoiltaan rajoitettua koodia ja voiko se siten tuottaa vastaavaa sisältöä myöhemmin [7].

Laskenta- ja resurssikustannukset muodostavat oman haittaluokkansa. Suuret mallit ovat laskennallisesti raskaita, ja laajojen koodikantojen migraatio voi vaatia huomattavia määriä API-kutsuja tai erillisiä GPU-resursseja, mikä näkyy suoraan kustannuksina [5], [6]. Lisäksi käytettävissä olevan kontekstin rajallisen pituuden vuoksi, kuinka suuri osa järjestelmästä voidaan kerralla huomioida: malli näkee vain osan projektista ja voi tehdä tyyppipäätelmiä tai refaktorointiratkaisuja, jotka eivät skaalaudu koko koodipohjaan. Tämä johtaa helposti tilanteeseen, jossa samankal-

taiset rakenteet siirretään hieman eri tavoin riippuen siitä, mihin paikalliseen kontekstiin ne sattuvat.

Kun merkittävä osa migraatiotyöstä siirretään mallin tehtäväksi, kehittäjien paine ymmärtää legacy-järjestelmän syvällisiä mekanismeja voi pienentyä, mikä pitkällä aikavälillä rapauttaa hiljaista tietoa järjestelmästä [22]. Tämä voi olla ongelmallista erityisesti tilanteissa, joissa migraation jälkeenkin tarvitaan kykyä diagnosoida tuotantoympäristössä ilmeneviä virheitä tai optimoida suorituskykyä. Lisäksi AI-avusteisen työskentelyn käytettävyys- ja vuorovaikutussuunnittelu on vielä kehittyvä alue: jos työkalujen käyttöliittymät ja palautemekanismit eivät tue läpinäkyvää yhteistyötä ihmisen ja mallin välillä, lisääntyy riski väärinymmärryksille ja virheiden huomaamatta jäämiselle.

Lopuksi on huomattava, että AI-pohjainen migraatio voi synnyttää uudenlaisia riippuvuuksia ja lukkiutumista tiettyihin toimittajiin tai malleihin [1]. Jos migraatioputki on rakennettu vahvasti yhden LLM-palvelun ympärille, mallin lisenssiehtojen, hinnoittelun tai saatavuuden muutokset voivat vaikuttaa suoraan organisaation kykyyn ylläpitää järjestelmiä ja jatkaa niiden migraatiota. Lisäksi malli itsessään vanhenee: uudet kieliversiot, framework-päivitykset ja kirjastot eivät näy koulutusdatassa, jolloin migraation laatu heikkenee ajan myötä, ellei mallia päivitetä tai korvata uudella [6].

3.5 Oikotieoppimisen vaikutukset koodimigraatioon

Koodimigraatiossa kontekstista oppiminen näkyy tyypillisesti siten, että mallille annetaan pieni joukko esimerkkimuunnoksia, kuten $JavaScript \rightarrow TypeScript$ -käännöksiä, ja sitä pyydetään soveltamaan samaa muunnoslogiikkaa uuteen lähdekoodiin. Tässä asetelmassa oikotieoppiminen voi johtaa siihen, että malli tuottaa pintapuolisesti uskottavan muunnoksen, joka näyttää oikealta ja saattaa jopa kääntyä, mutta ei säilytä alkuperäisen järjestelmän semantiikkaa, arkkitehtonisia oletuk-

sia tai organisaatiokohtaisia reunaehtoja [13].

Käytännössä oikotieoppiminen voi ilmetä siten, että malli alkaa suosia tiettyjä tyyppimerkintöjen muotoja, import-rakenteita tai refaktorointityylejä pelkän kehoteformaatin ja esimerkkien esitysjärjestyksen perusteella [13]. Tällöin pienetkin muutokset ohjeistuksessa tai demonstraatioiden järjestyksessä voivat johtaa yllättäviin ja epäjohdonmukaisiin muunnoksiin. Migraation lopputulos ei siis riipu ainoastaan lähdekoodin rakenteesta, vaan myös siitä, millaisia vihjeitä malli sattuu poimimaan syötteestä.

Lisäksi tietyt nimeämiskonventiot, kommenttityyli, tiedostorakenne tai toistuvat koodifragmentit voivat alkaa toimia signaaleina, joiden perusteella malli valitsee migraatiostrategian varsinaisen tehtävärakenteen sijaan [13]. Jos demonstraatiot ovat vinoutuneita, esimerkiksi kaikki esimerkit nojaavat samaan arkkitehtuurimalliin tai samanlaiseen tyyppipäättelyyn, malli voi yleistää saman enemmistöratkaisun myös sellaisiin kohtiin, joissa siitä pitäisi poiketa. Seurauksena voi olla muunnos, joka toimii tavanomaisissa tapauksissa mutta epäonnistuu poikkeustilanteissa tai projektikohtaisissa erikoisratkaisuissa.

Koodimigraation kannalta tämä tarkoittaa, että ICL-pohjainen migraatioputki voi olla erityisen herkkä esimerkkivalinnoille ja kehotteen muotoilulle. Sama malli voi tuottaa huomattavasti erilaisia ratkaisuja pienilläkin syötemuutoksilla, vaikka varsinainen migraatiotavoite pysyisi samana. Tämän vuoksi migraation onnistumista ei tulisi arvioida vain sen perusteella, toimiiko tuotettu koodi, vaan myös sen perusteella, kuinka hyvin muunnos säilyttää semantiikan, kestää kontekstin ja syötteen luonnollista vaihtelua sekä noudattaa johdonmukaisesti migraatioperiaatteita koko koodikannassa [13].

3.6 Kielimallien suorituskykyä parantavat tekijät migraatiossa

Kielimallien käytännön suorituskyky koodimigraatiossa määräytyy pitkälti sen perusteella, miten migraatioputki on rakennettu: mallin tuottama koodi on vain yksi osa kokonaisuutta, jossa korostuvat tehtävän tarkka määrittely, projektikontekstin saatavuus sekä laadunvarmistuksen automaatio. Tehtävän tarkka määrittely tarkoittaa käytännössä sitä, että mallille annetaan yksiselitteiset muunnossäännöt ja hyväksymiskriteerit (mitä saa muuttaa ja mitä ei, millaista kohdekoodia tavoitellaan ja missä muodossa tulos tuotetaan). Projektikontekstin saatavuudella viitataan siihen, että mallin päätöksenteon tueksi tuodaan relevantit riippuvuudet ja projektisäännöt (esim. tyyppimäärittelyt, yhteiset datamallit, arkkitehtuurirajoitteet ja koodityyli), jotta ratkaisut pysyvät johdonmukaisina koko koodikannassa. Laadunvarmistuksen automaatio puolestaan ankkuroidaan deterministisiin tarkistuksiin, kuten kääntäjän, tyyppitarkistimen, lintterien ja testien tuottamaan palautteeseen, jolloin semanttiset poikkeamat ja regressiot havaitaan varhaisessa vaiheessa. Pelkkä "paremman mallin" valinta ei tyypillisesti riitä, jos migraatiossa ei ole mekanismeja, joilla ohjataan tuotosta organisaation tavoitteiden, standardien ja teknisten reunaehtojen mukaiseksi sekä havaitaan semanttiset poikkeamat varhaisessa vaiheessa [2], [6].

Seuraavassa jäsennän viisi käytännön suorituskykyä parantavaa tekijää migraatioputken suunnitteluun: (1) tehtävän rajaaminen ja kehotteiden standardointi [2], [13], (2) projektikontekstin rikastaminen [2], [6], (3) deterministiset tarkistukset ja palautesilmukat, kuten kääntäjä-, analyysi- ja testipalautteen hyödyntäminen, [2], (4) migraation vaiheistus ja arviointimittarit [2], [6] sekä (5) ihmisen ja mallin yhteistyötä tukeva prosessi ja käyttöliittymä [12], [22]. Tekijöiden ryhmittely ja painotus migraatiokontekstissa on tämän työn synteesiä.

Ensimmäinen keskeinen parantava tekijä on migraatiotehtävän eksplisiittinen

rajaaminen ja kehotteiden standardointi. Migraatiolle kannattaa määritellä selkeä "muunnossopimus": mitä saa refaktoroida (esim. vain tyyppimerkinnät ja importit) ja mitä ei (liiketoimintalogiikan muutos), millaista kohdekoodin idiomatiikkaa tavoitellaan sekä missä muodossa ulostulo tuotetaan (esim. vain diff tai vain tiedoston sisältö). Lisäksi few-shot-esimerkkien valinnassa tulee korostaa monipuolisuutta: jos demonstraatiot ovat liian homogeenisia, malli voi yleistää pintapiirteitä ja päätyä oikotieoppimiseen, mikä heikentää ennustettavuutta erityisesti poikkeustapauksissa [6], [13]. Tämän vuoksi esimerkkijoukkoon kannattaa sisällyttää myös esimerkkejä "rajatapauksista" ja vastakkaisia tapauksia (counterexamples), jotta mallin päätösperusteet eivät lukkiudu yksittäisiin laukaiseviin piirteisiin [13].

Toinen tekijä liittyy kontekstin rikastamiseen, koska yksittäinen tiedosto ei yleensä sisällä kaikkea sitä tietoa, jonka perusteella voidaan tehdä johdonmukaisia päätöksiä koko koodikannan tasolla. Migraatiossa tämä tarkoittaa käytännössä sitä, että mallille tuodaan saataville projektin kannalta keskeinen tieto: tyyppimäärittelyt, jaetut datamallit, arkkitehtuurirajoitteet, lint-säännöt, koodityyli sekä tarvittaessa sisäiset API-kuvaukset. Kontekstin augmentointi (esim. hakemalla relevantit artefaktit ennen generointia ja lisäämällä ne kehotteeseen) voi vähentää "paikallisen kontekstin" aiheuttamaa vaihtelua ja parantaa ratkaisujen yhdenmukaisuutta laajoissa migraatioissa, joissa konteksti-ikkuna ei riitä koko järjestelmän kerralla käsittelyyn [6].

Kolmas suorituskykyä parantava tekijä on työkaluketjun ja palautesilmukoiden systemaattinen hyödyntäminen. LLM-tuotokset kannattaa ankkuroida deterministisiin tarkistuksiin: kääntäjä, TypeScriptin tyyppitarkistus, lintterit ja staattinen analyysi toimivat "objektiivisina tuomareina", jotka paljastavat nopeasti virheelliset tyyppipäätelmät, puuttuvat importit ja epäjohdonmukaiset rajapinnat. Lisäksi testaus (yksikkö- ja regressiotestit) vähentää riskiä, että migraatio onnistuu syntaktisesti mutta muuttaa semantiikkaa [2], [15]. Erityisesti $JavaScript \rightarrow TypeScript$

-migraatiossa havaittu varoitusten ja ongelmien siirtymä (osa ongelmista vähenee, mutta esimerkiksi deprekaatioihin ja kompleksisuuteen liittyvät haasteet voivat kasvaa) tukee sitä, että pelkkä "kääntyy" ei riitä hyväksymiskriteeriksi, vaan laadunvarmistuksen tulee kattaa myös ylläpidettävyys- ja evoluutiovaikutukset [3].

Neljäs tekijä on migraation vaiheistus ja arviointimittareiden valinta. Suorituskyky käytännössä tarkoittaa läpimenoa: kuinka suuri osa tuotoksista voidaan hyväksyä pienellä korjaustyöllä ja kuinka paljon migraatio synnyttää uutta teknistä velkaa. Tämän vuoksi migraatio kannattaa pilkkoa vaiheisiin (esim. ensin "tyyppikuoret" ja rajapinnat, sitten moduulikohtainen tarkennus), ja edistymistä seurata useammalla mittarilla kuin vain käännösasteella: esimerkiksi varoitusten luokittelu, kognitiivinen kompleksisuus, deprekaatioihin liittyvät löydökset sekä koodihajujen kehitys antavat paremman kuvan todellisesta laadusta [3]. Laajemmassa LLM-kirjallisuudessa korostuu myös vertailuanalyysien ja arvioinnin rooli, koska ilman toistettavaa arviointia migraatioputken muutosten (kehotteet, konteksti, työkalut) vaikutusta on vaikea todentaa [6].

Viides parantava tekijä liittyy ihmisen ja mallin yhteistyöhön. Migraatiotyössä kehittäjän ajankäyttö kohdistuu usein tarkastukseen ja korjauksiin, joten käyttöliittymä- ja prosessiratkaisut, jotka tukevat läpinäkyvää tarkistamista (esim. pienet, rajatut diffit, selkeät hyväksymiskriteerit ja automaattisesti koottu todistusaineisto kääntäjä-/testituloksista), parantavat kokonaisläpimenoa [22]. Samalla on tärkeää hallita luottamuksen kalibrointia: pitkät, vakuuttavat selitykset voivat nostaa käyttäjän luottamusta ilman, että todellinen oikeellisuus paranee, jolloin tarkastusta tehdään liian kevyesti juuri riskialttiissa kohdissa [12]. Käytännön suorituskykyä parantaa siis se, että prosessi ohjaa arvioimaan tuotosta todisteiden (käännös- ja testitulokset) eikä selitystekstin pituuden perusteella [12].

4 Pohdinta

Kielimallien käyttö migraation automatisoinnissa voidaan nähdä ennen kaikkea tuottavuus- ja riskienhallintakysymyksenä: tavoitteena on vähentää manuaalisen käännöstyön työmäärää ja inhimillisiä virheitä, mutta samalla säilyttää järjestelmän semantiikka, arkkitehtuuriehdot ja tuotannollinen reunaehtokelpoisuus. Kirjallisuus koodikäännöksestä ja AI-avusteisesta refaktoroinnista korostaa, että automatisointi voi nopeuttaa siirtymiä kielestä tai teknologiasta toiseen erityisesti silloin, kun muunnokset ovat toistuvia ja muutoskohteet ovat selkeästi rajattuja (esim. syntaksin, tyyppimerkintöjen ja mekaanisten rakenteiden päivitykset) [2], [15]. Käytännössä tämä tarkoittaa, että kielimallit soveltuvat hyvin migraation "rutiinikerrokseen", jossa suuri määrä samankaltaisia muunnoksia voidaan tuottaa nopeasti ja yhtenäisesti, kunhan tuotokset ankkuroidaan automaattisiin tarkistuksiin ja hyväksymiskriteereihin.

Täysin automaattinen "kerran läpi ja valmiiksi" migraatio on kuitenkin harvoin realistinen tavoite, koska migraatiossa syntyvät virheet ovat usein semanttisia ja ilmenevät vasta integraatio- tai regressiotesteissä. Lisäksi laajan järjestelmän kokonaiskonteksti ei yleensä mahdu mallin konteksti-ikkunaan, jolloin päätökset voivat jäädä paikallisiksi ja epäjohdonmukaisiksi [6]. Tämän vuoksi automatisointi kannattaa hahmottaa ensisijaisesti semi-automaationa: malli tuottaa ehdotuksia ja muunnoksia, mutta hyväksyntä edellyttää käännöksen, staattisen analyysin ja testien läpäisyä sekä kriittisten muutosten katselmointia. Sama periaate näkyy myös

AI-refaktorointia käsittelevässä kirjallisuudessa, jossa korostetaan ihmisen ja mallin yhteistyötä sekä determinististen tarkistusten (kääntäjä/lintterit/testit) roolia tuotantokelpoisuuden varmistamisessa [15].

Migraation automatisoinnin kannalta keskeinen havainto on, että migraatiostrategia vaikuttaa lopputuloksen laatuun riippumatta siitä, tehdäänkö muunnoksia käsin vai mallin avustamana. JavaScript–TypeScript-migraatiota tarkasteleva tutkimus osoittaa, että projekteissa esiintyy useita strategioita, kuten äkillinen ja vaiheittainen siirtymä sekä osittainen käyttöönotto, ja että niiden vaikutus laatuun on kaksijakoinen: osa ongelmaluokista, kuten scopeen ja käyttämättömään koodiin liittyvät ongelmat, vähenee, mutta samalla korostuvat esimerkiksi deprekaatioihin ja kognitiiviseen kompleksisuuteen liittyvät haasteet [3]. Tämä tukee ajatusta, että automatisoinnin onnistumista ei tule arvioida vain sen perusteella, kuinka moni tiedosto kääntyy onnistuneesti, vaan myös sen perusteella, millaisia uusia laatu- ja ylläpitovaikutuksia migraatio synnyttää. Käytännössä tämä puoltaa vaiheittaista lähestymistapaa, jossa ensin automatisoidaan matalariskiset ja helposti validoitavat muutokset ja vasta sen jälkeen edetään epäselvempiin ja semanttisesti herkempiin kohtiin [3].

Automatisointia rajoittavat myös kielimallien yleistettävyyteen liittyvät ilmiöt, joista oikotieoppiminen on migraation kannalta erityisen relevantti. Kontekstista oppimisen tilanteissa malli voi omaksua hauraan päätössäännön annetusta esimerkkijoukosta ja tuottaa näennäisesti oikeita, mutta poikkeustapauksissa rikkoutuvia muunnoksia [13]. Tämä korostaa kahta käytännön johtopäätöstä: (i) migraation ohjeistus ja esimerkit on suunniteltava siten, että ne kattavat myös rajatapaukset ja vaihtoehtoiset rakenteet, ja (ii) migraatioputkessa on oltava systemaattiset “jarrut” (käännös-, lint- ja testigatet), jotka katkaisevat automaation, kun poikkeama havaitaan. Ilman näitä mekanismeja automatisointi voi siirtää virheitä huomaamatta eteenpäin ja kasvattaa myöhempää korjauskustannusta.

Olennaista on myös luottamuksen ja vastuun jakautuminen: kielimalli voi tuottaa vakuuttavia perusteluja, jotka lisäävät käyttäjän luottamusta, vaikka todellinen oikeellisuus ei paranisi. Kalibraatiotutkimus osoittaa, että erityisesti pitkät selitykset voivat kasvattaa käyttäjän varmuutta riippumatta vastausten tosiasiallisesta laadusta [12]. Migraatiokontekstissa tämä tarkoittaa, että automaatiota käyttävä organisaatio tarvitsee eksplisiittiset käytännöt, jotka pakottavat arvioimaan muutoksia todisteiden (kääntäjä-, testitulokset, regressiohavaintoja) eikä selitystekstin perusteella. Muutoin riskinä on automaatioharha: tarkastusresursseja kohdistetaan vähemmän juuri niihin kohtiin, joissa semanttiset virheet ovat todennäköisimpiä [12].

Koska migraatio on myös yhteistyö- ja prosessiongelma, työkalujen käytettävyys ja vuorovaikutusmalli vaikuttavat suoraan siihen, kuinka paljon automaatio todellisuudessa säästää aikaa. Ihmisen ja tekoälyn yhteistyötä käsittelevä kirjallisuus tukee mallia, jossa muutokset esitetään pieninä ja rajattuina muutosjoukkoina, katselmointipolku on selkeä ja palautemekanismit mahdollistavat iteroinnin (esim. “korjaa nämä kolme tyyppivirhettä”, “noudata tätä nimeämissääntöä”) [22]. Tällaisessa asetelmassa automatisointi voi parantaa migraation etenemisnopeutta, koska kehittäjän työ siirtyy mekaanisesta kirjoittamisesta laadunvarmistukseen ja poikkeusten käsittelyyn, mikä on myös migraation kokonaisriskin hallinnan kannalta järkevämpää.

Yhteenvetona voidaan todeta, että kielimalleja voidaan hyödyntää migraation automatisoinnissa, kun sen käyttö rajataan tarkoituksenmukaisesti: Kielimallin käyttö migraatiossa nopeuttaa erityisesti toistuvia muunnoksia, mutta tuotantokäyttö edellyttää puoliautomaattista prosessia, ja katselmointi ehkäisee automaatioharhaa [3], [12], [13], [15]. Automaatio ei korvaa migraatio-osaamista, vaan siirtää sen painopisteen semanttisesti kriittisiin kohtiin sekä laadun ja ylläpidettävyyden varmistamiseen [6].

5 Yhteenveto

Tässä tutkielmassa tarkasteltiin, miten suuria kielimalleja voidaan hyödyntää koodimigraatiossa legacy-frontendin modernisoinnissa. Kontekstina oli erityisesti siirtymä AngularJS:stä Angulariin sekä siihen liittyvä JavaScriptin muuntaminen TypeScriptiksi. Lähtökohtana oli, että pitkäikäisissä järjestelmissä tekninen velka heikentää muutosten läpimenoa, turvallisuutta ja ylläpidettävyyttä, jolloin modernisaatio on usein lopulta välttämätön. Samalla migraatio on työläs ja riskialtis prosessi, jossa onnistuminen edellyttää automaation, laadunvarmistuksen ja riskienhallinnan yhteispeliä.

Tutkielma toteutettiin kirjallisuuskatsauksena. Lähteitä haettiin kohdennetuilla hakulauseilla ja valintaa painotettiin uudempiin julkaisuihin, jotta tarkastelu vastaisi nopeasti kehittyvää kielimallikenttää. Työ jäsennettiin kolmen tutkimuskysymyksen kautta: (**TK1**) Mitä rajoituksia kielimalleilla on *AngularJS* → *Angular* -migraatioprosessissa, (**TK2**) Mitä tapoja on parantaa kielimallien suorituskykyä migraatiossa ja (**TK3**) Mitä hyötyjä kielimalleista on koodin migraatioprosessissa.

TK1: Keskeisin rajoite liittyy semanttisen säilyvyyden varmistamiseen: syntaktisesti uskottava kohdekoodi ei takaa, että liiketoimintalogiikka, virheenkäsittely ja reunatapaukset säilyvät muuttumattomina. Haastetta korostaa se, että *AngularJS* → *Angular* -siirtymä on usein arkkitehtuurinen (komponenttipohjaisuus, elinkaarimallit ja reaktiivisemmat rakenteet), jolloin tuotantokelpoisuus edellyttää regressiotestausta ja manuaalista tarkastusta, eikä “yhdellä

ajolla" -automaatio ole useimmiten realistinen.

Toinen rajoite koskee framework- ja domain-spesifistä tietoa: organisaation omat kirjastot, arkkitehtuurisäännöt ja koodaustyyli voivat jäädä mallilta vajaasti huomioiduiksi, mikä lisää epäjohdonmukaisuutta ja korjaustyötä. Lisäksi oikotieoppiminen ja pintapiirteisiin nojaaminen heikentävät ennustettavuutta, ja luottamuksen kalibrointi altistaa automaatioharhalle. Myös skaalautuvuus ja kustannukset rajaavat käyttöä, koska laaja koodikanta joudutaan pilkkomaan osiin ja onnistumista on vaikea arvioida pelkkien käännösten määrällä.

TK2: Suorituskyky paranee ennen kaikkea migraatioputken suunnittelulla: tehtävä rajataan ja kehotteet standardoidaan ("muunnossopimus"), esimerkit valitaan monipuolisiksi ja myös rajatapauksia sisältäviksi, ja mallille tuodaan projektikohtainen konteksti (tyypit, datamallit, säännöt ja arkkitehtuuriperiaatteet). Kriittisin tekijä on palautesilmukka, jossa tuotokset ankkuroidaan deterministisiin tarkistuksiin (kääntäjä, tyyppitarkistus, lintterit, staattinen analyysi) ja testaukseen, ja migraatio toteutetaan iteratiivisesti ja vaiheistetusti. Lisäksi katselmointi suunnitellaan todistepohjaiseksi (käännös- ja testitulokset), jotta automaatioharha vähenee.

TK3: Keskeisin hyöty on tuottavuus: mallit automatisoivat mekaanisia muunnoksia ja tuottavat nopeasti ensimmäisen version, jolloin kehittäjän työ painottuu laadunvarmistukseen ja poikkeusten käsittelyyn. Lisäksi mallit tukevat legacy-koodin ymmärtämistä (selitykset, tiivistykset, riippuvuudet), mikä nopeuttaa suunnittelua ja pienentää perehdytyskynnystä. Migraatio voi samalla toimia laatua parantavana ikkunana, kun tyypitys ja modernit työkalut otetaan hallitusti käyttöön.

Kokonaisuutena tutkielman johtopäätös on, että kielimallien tarkoituksenmukaisin käyttötapa AngularJS:stä Angulariin siirtymisen kontekstissa on puoliksi auto-

matisoitu migraatioputki. Malli tuottaa muunnoksia ja tukee analyysiä, mutta hyväksyntä sidotaan automaattisiin tarkistuksiin ja testaukseen sekä riskiperusteiseen katselmointiin. Tällöin voidaan saavuttaa merkittävä tuottavuushyöty ilman, että semanttiset riskit, epäjohdonmukaisuudet ja automaatioharha kasvavat hallitsemattomiksi.

Lähdeluettelo

[1] P. Garcia et al., *On Migrating Framework Components in Web Applications: A State-of-the-Art and State-of-the-Practice Review*, syyskuu 2025. DOI: 10.2139/ssrn.5413487. url: https://ssrn.com/abstract=5413487.

[2] A. Das et al., ”Automated Code Translation: Enhancing Efficiency and Accuracy Across Programming Languages”, heinäkuu 2025. DOI: 10.2991/978-94-6463-787-8_55. url: https://www.atlantis-press.com/proceedings/raisd-25/126013742.

[3] ”Understanding JavaScript to TypeScript Migration: Strategies and Quality Assessment”, tammikuu 2025. DOI: 10.2139/ssrn.5614285. url: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=5614285.

[4] S. Preibisch, *API Development: A Practical Guide for Business Implementation Success*. Berkeley, CA: Apress, 2018, ISBN: 978-1-4842-4139-4 978-1-4842-4140-0. DOI: 10.1007/978-1-4842-4140-0. viitattu 4. helmikuuta 2026.

[5] K. Razzaq ja M. Shah, ”Machine Learning and Deep Learning Paradigms: From Techniques to Practical Applications and Research Frontiers”, *Computers*, vol. 14, nro 3, s. 93, maaliskuu 2025, ISSN: 2073-431X. DOI: 10.3390/computers14030093. viitattu 28. lokakuuta 2025.

[6] H. Naveed et al., *A Comprehensive Overview of Large Language Models*, Issue: arXiv:2307.06435 _eprint: 2307.06435, lokakuu 2024. DOI: 10.48550/arXiv.

2307.06435. viitattu 4. marraskuuta 2025. url: https://dl.acm.org/doi/full/10.1145/3744746.

[7] S. M. Weiss, N. Indurkhya ja T. Zhang, *Fundamentals of Predictive Text Mining* (Texts in Computer Science). London: Springer London, 2010, ISBN: 978-1-84996-225-4 978-1-84996-226-1. DOI: 10.1007/978-1-84996-226-1. viitattu 4. marraskuuta 2025.

[8] R. Kirk et al., *Understanding the Effects of RLHF on LLM Generalisation and Diversity*, Issue: arXiv:2310.06452 _eprint: 2310.06452, helmikuu 2024. DOI: 10.48550/arXiv.2310.06452. viitattu 4. helmikuuta 2026. url: https://arxiv.org/abs/2310.06452.

[9] R. Rafailov, A. Sharma, E. Mitchell, S. Ermon, C. D. Manning ja C. Finn, ”Direct Preference Optimization: Your Language Model Is Secretly a Reward Model”, 2023. url: https://proceedings.neurips.cc/paper_files/paper/2023/hash/a85b405ed65c6477a4fe8302b5e06ce7-Abstract-Conference.html.

[10] T. B. Brown et al., ”Language Models Are Few-Shot Learners”, 2020. url: https://proceedings.neurips.cc/paper_files/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html?utm_source=transaction&utm_medium=email&utm_campaign=linkedin_newsletter.

[11] J. Wei et al., *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*, Issue: arXiv:2201.11903 _eprint: 2201.11903, tammikuu 2023. DOI: 10.48550/arXiv.2201.11903. viitattu 5. helmikuuta 2026. url: https://proceedings.neurips.cc/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html?utm_source=chatgpt.com.

[12] M. Steyvers et al., ”What Large Language Models Know and What People Think They Know”, *Nature Machine Intelligence*, vol. 7, nro 2, s. 221–231,

tammikuu 2025, ISSN: 2522-5839. DOI: 10.1038/s42256-024-00976-7. viitattu 4. marraskuuta 2025.

[13] R. Song, Y. Li, L. Shi, F. Giunchiglia ja H. Xu, *Shortcut Learning in In-Context Learning: A Survey*, Issue: arXiv:2411.02018 _eprint: 2411.02018, marraskuu 2024. DOI: 10.48550/arXiv.2411.02018. viitattu 21. lokakuuta 2025. url: https://arxiv.org/abs/2411.02018.

[14] S. Nikolov et al., ”How Is Google Using AI for Internal Code Migrations?”, teoksessa *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, Ottawa, ON, Canada: IEEE, huhtikuu 2025, s. 481–492, ISBN: 979-8-3315-3685-5. DOI: 10.1109/ICSE-SEIP66354.2025.00048. viitattu 21. tammikuuta 2026.

[15] S. Podduturi, ”AI-Driven Code Optimization: Leveraging ML to Refactor Legacy Codebases”, *Open Access*, vol. 6, nro 1, tammikuu 2025. url: https://najer.org/najer/article/view/115.

[16] Y. Ding, ”Advancing Code Intelligence with Language Models”, 2026. url: https://academiccommons.columbia.edu/doi/10.7916/pys4-4n42.

[17] C. Gao, X. Hu, S. Gao, X. Xia ja Z. Jin, ”The Current Challenges of Software Engineering in the Era of Large Language Models”, *ACM Transactions on Software Engineering and Methodology*, vol. 34, nro 5, s. 1–30, kesäkuu 2025, ISSN: 1049-331X, 1557-7392. DOI: 10.1145/3712005. viitattu 21. tammikuuta 2026.

[18] J. Cordeiro, S. Noei ja Y. Zou, *An Empirical Study on the Code Refactoring Capability of Large Language Models*, Issue: arXiv:2411.02320 _eprint: 2411.02320, marraskuu 2024. DOI: 10.48550/arXiv.2411.02320. viitattu 5. helmikuuta 2026.

[19] A. Shirafuji, Y. Oda, J. Suzuki, M. Morishita ja Y. Watanobe, ”Refactoring Programs Using Large Language Models with Few-Shot Examples”, teoksessa *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*, _eprint: 2311.11690, joulukuu 2023, s. 151–160. DOI: 10.1109/APSEC60848.2023.00025. viitattu 5. helmikuuta 2026.

[20] R. A. Poldrack, T. Lu ja G. Beguš, *AI-assisted Coding: Experiments with GPT-4*, Issue: arXiv:2304.13187 _eprint: 2304.13187, huhtikuu 2023. DOI: 10.48550/arXiv.2304.13187. viitattu 5. helmikuuta 2026. url: https://arxiv.org/abs/2304.13187.

[21] H. Y. Kang ja D. S. Weon, ”Empirical Study on Digital Core-Banking System”, *International Journal of Internet, Broadcasting and Communication*, vol. 9, nro 2, s. 48–57, toukokuu 2017, Publisher: DOI: 10.7236/IJIBC.2017.9.2.48. viitattu 5. helmikuuta 2026.

[22] T. Z. Ahram, W. Karwowski ja P.-L. Rau, *Human-Computer Interaction & Emerging Technologies*. AHFE International, heinäkuu 2025, ISBN: 978-1-964867-71-7.