Sonja Inberg

Low-code ja no-code-kehitys vs. perinteinen ohjelmistokehitys: Hyödyt, haasteet ja käyttökohteet

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

8.5.2025

Tiivistelmä

Tekijä:	Sonja Inberg
Otsikko:	Low-code ja no-code-kehitys vs. perinteinen ohjelmistokehitys: Hyödyt, haasteet ja käyttökohteet
Sivumäärä:	33 sivua
Aika:	8.5.2025
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Tieto- ja viestintätekniikka
Ohjaajat:	Janne Salonen (Tutkintovastaava)

Insinöörityön aiheena oli low-code- ja no-code-kehityksen vertailu perinteiseen ohjelmistokehitykseen. Työn tavoitteena oli selvittää, milloin low-code- tai no-code-kehitys on perusteltu vaihtoehto perinteisten menetelmien sijaan sekä kartoittaa näiden kehitysmallien hyötyjä, haasteita ja soveltuvia käyttökohteita.

Työ toteutettiin kirjallisuuskatsauksena, jossa analysoitiin aiheeseen liittyvää tutkimuskirjallisuutta ja asiantuntijalähteitä. Työssä käsiteltiin ensin ohjelmistokehityksen elinkaarta ja perinteisiä kehitysmalleja, jonka jälkeen tarkasteltiin low-code- ja no-code-menetelmiä, sekä niiden alustoja ja teknisiä erityispiirteitä.

Vertailussa low-code-kehityksen merkittävimmiksi eduiksi nousivat kehityksen nopeus, alhaisemmat kustannukset ja matalampi osaamiskynnys. Haasteina esiin nousivat rajoitettu skaalautuvuus, heikommat mahdollisuudet kustomointiin, riippuvuus yksittäisestä alustasta ja tietoturvariskit.

Low-code-alustat osoittautuivat hyödyllisiksi erityisesti pienissä ja keskisuurissa yrityksissä, joissa resurssit ovat rajalliset. Perinteinen ohjelmistokehitys säilyttää kuitenkin etunsa laajoissa, monimutkaisissa ja korkean turvallisuustason vaativissa projekteissa, joissa tarvitaan täyttä kontrollia teknisiin yksityiskohtiin.

Kehitysmenetelmän valinnan olisi hyvä perustua projektin vaatimuksiin ja organisaation kyvykkyyksiin, ja molemmilla lähestymistavoilla on perusteltu paikkansa nykyaikaisessa ohjelmistokehityksessä.

Avainsanat: low-code, no-code, ohjelmistokehitys

Abstract

Author:	Sonja Inberg
Title:	Low-code and no-code-development vs. traditional software development: Benefits, challenges and use cases
Number of Pages:	33 pages
Date:	8th May 2025
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Professional Major:	Information Technology
Supervisors:	Janne Salonen

The topic of this thesis was the comparison of low-code and no-code development with traditional software development. The objective was to determine when low-code or no-code development is a justified alternative to traditional methods, as well as to examine the benefits, challenges, and suitable use cases of these development models.

The work was conducted as a literature review, analyzing relevant academic publications and expert sources. First, the software development lifecycle and traditional development models were reviewed, followed by description of low-code and no-code approaches, platforms, and technical features.

The comparison highlighted the main advantages of low-code development as faster development speed, lower costs, and a lower technical skill threshold. The challenges identified included limited scalability, reduced customization capabilities, dependency on a specific platform, and security risks.

Low-code platforms proved especially useful for small and medium-sized enterprises with limited resources. However, traditional software development maintains its advantages in large, complex, and high-security projects where full control over technical details is required.

The choice of development method should be based on the specific requirements of the project and the capabilities of the organization, as both approaches have their justified role in modern software development.

Keywords: low-code, no-code, software development

Sisällys

Lyhenteet

SDLC: *Software Development Life Cycle*. Malli, joka kuvaa ohjelmistokehityksen elinkaaren vaiheita.

1 Johdanto

Perinteinen ohjelmistokehitys on monivaiheinen prosessi, joka vaatii usein syvällistä teknistä osaamista ja huomattavan määrän aikaa, ennen kuin alun perin syntyneestä ideasta tulee valmis toteutus markkinoille. Viime vuosina perinteisten menetelmien oheen markkinoille on noussut myös vaihtoehtoisia kehitysmenetelmiä, kuten low-code ja no-code-menetelmät, joiden avulla ohjelmistojen kehitystä voidaan nopeuttaa ja helpottaa niin, että teknisen osaamisen kynnys madaltuu ja ohjelmisto voidaan luoda lyhyemmässä ajassa.

Tässä kirjallisuuskatsauksessa tarkastellaan aiempia tutkimuksia ja kirjallisuutta siitä, milloin low-code tai no-code-kehitys on varteenotettava vaihtoehto perinteisen ohjelmistokehityksen sijaan ja minkälaisia käyttötapauksia niillä on. Lisäksi katsauksessa vertaillaan eri menetelmien etuja ja rajoituksia, muun muassa kehitysnopeuden, kustannusten, skaalautuvuuden ja tietoturvan näkökulmista.

Tutkimuskysymykset ovat seuraavat:

1. Missä tilanteissa low-code tai no-code-kehitys on perusteltu vaihtoehto perinteiseen ohjelmistokehitykseen verrattuna?

2. Miten low-code tai no-code-kehitys eroaa perinteisestä kehityksestä nopeuden, kustannusten ja joustavuuden näkökulmasta?

3. Mitkä ovat low-code tai no-code-kehityksen rajoitteet skaalautuvuuden, integraatioiden ja tietoturvan kannalta?

4. Millaisille käyttäjäryhmille low-code- tai no-code-ratkaisut soveltuvat parhaiten?

2 Ohjelmistokehitysmenetelmät

2.1 Software Development Life Cycle (SDLC)

Software Development Life Cycle (SDLC) kuvaa ohjelmistokehityksen vaiheittaista etenemistä järjestelmällisen prosessin mukaan. Siinä ohjelmiston kehitys ideasta valmiiksi tuotteeksi on jaettu selkeisiin osiin, kuten suunnitteluun, analyysiin, toteutukseen, testaukseen, käyttöönottoon ja ylläpitoon. (Gunnell 2024.) SDLC alkaa, kun konsepti ohjelmistosta muodostuu, ja päättyy kun ohjelmisto lopulta poistuu käytöstä (Aggarwal & Singh 2005: 20). Kuvassa 1 on havainnollistettu SDLC:n vaiheita.

Kuva 1 SDLC:n vaiheet

Suunnittelu: Suunnitteluvaiheessa määritellään projektin laajuus ja tavoitteet, sekä arvioidaan sen toteutettavuus taloudellisesta, toiminnallisesta ja teknisestä näkökulmasta. Suunnitteluun osallistuvat kokeneet tiimin jäsenet hyödyntäen asiakkaiden, myynnin, markkinatutkimusten ja toimialan asiantuntijoiden antamaa tietoa. (Gunnell 2024; SDLC – Overview.)

Analyysi ja määrittelyt: Seuraava vaihe on määritellä ja dokumentoida ohjelmiston tarkat vaatimukset ja varmistaa, että vaatimukset tukevat liiketoiminnan tavoitteita. Näiden vaatimusten tulee saada hyväksyntä asiakkaalta tai markkina-analyytikoilta ennen etenemistä. (Gunnell 2024; SDLC – Overview.)

Arkkitehtuuri: Vaatimusten määrittelyn pohjalta laaditaan tarkka suunnitelma, jossa määritellään ohjelmiston arkkitehtuuri, moduulit, rajapinnat ja tiedonkulku. Usein ehdotetaan useampia eri suunnitteluvaihtoehtoja, jotka arvioidaan sidosryhmien kesken ja valitaan lopullinen suunnitteluratkaisu esimerkiksi riskien, budjetin ja aikataulun perusteella. (Gunnell 2024; SDLC – Overview.)

Toteutus: Toteutusvaiheessa varsinainen ohjelmiston kehittäminen alkaa ja ohjelmakoodi kirjoitetaan suunnitteludokumentaation mukaisesti, noudattaen ohjelmoinnin standardeja ja ohjeita. Ohjelmointikieli valitaan kehityskohteen mukaan. (Gunnell 2024; SDLC – Overview.)

Testaus: Testausvaiheessa varmistetaan, että ohjelmisto toimii suunnitellusti ja täyttää sille asetetut vaatimukset. Kehitystyön aikana suoritetaan erilaisia testauksia, kuten yksikkö-, integraatio-, järjestelmä- ja hyväksymistestejä, joiden avulla havaitaan mahdolliset virheet. Testaus on usein nykyisissä kehitysmalleissa jatkuva prosessi, jota tehdään jokaisen SDLC:n vaiheen aikana. (Gunnell 2024; SDLC – Overview.

Käyttöönotto: Käyttöönottovaiheessa ohjelmisto otetaan käyttöön tuotantoympäristössä. Tämä vaihe sisältää asennuksen, järjestelmän konfiguroinnin ja tarvittavan koulutuksen käyttäjille. Joissain tapauksissa julkaisu voi tapahtua vaiheittain organisaation liiketoimintastrategian mukaisesti, esimerkiksi olemalla

alkuun saatavilla vain rajoitetulle asiakassegmentille. (Gunnell 2024; SDLC – Overview.)

Ylläpito: Käyttöönoton jälkeen järjestelmä vaatii jatkuvaa tukemista ja päivittämistä. Ylläpitovaihe sisältää esimerkiksi virheiden korjaamista, uusien ominaisuuksien lisäämistä ja parannuksia, jotka perustuvat käyttäjiltä saatuun palautteeseen. Tavoitteena on varmistaa, että ohjelmisto pysyy toimivana ja ajantasaisena pitkällä aikavälillä. (Gunnell 2024; SDLC – Overview.)

SDLC auttaa hallitsemaan kehitysprosessin kokonaisuutta tukemalla riskien ennakointia, ohjelmiston laadun varmistamista, tiimien välistä yhteistyötä sekä kustannusten ja aikataulujen arvioimista. Erilaiset ohjelmistokehitysmallit, kuten vesiputousmalli (Waterfall), ketterä malli (Agile) ja spiraalimalli (Spiral model), tarjoavat vaihtoehtoisia tapoja toteuttaa SDLC:n vaiheet käytännössä projektin tarpeiden mukaan. (Gunnell 2024; Stoica 2013.)

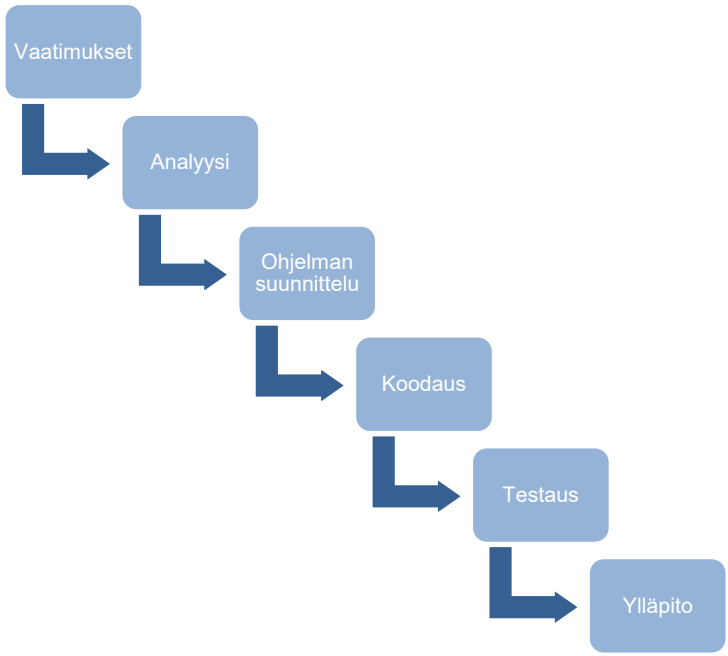
Ohjelmistokehitysmallit voidaan jakaa perinteisiin menetelmiin ja ketteriin menetelmiin. Perinteiset mallit, kuten vesiputousmalli ja spiraalimalli, etenevät ennalta suunniteltujen vaiheiden mukaisesti: kehitys alkaa vaatimusten keräämisellä ja kattavalla dokumentoinnilla, minkä jälkeen siirrytään ohjelmiston arkkitehtuurin ja korkean tason suunnitteluun. Perinteisissä malleissa muutosten tekeminen kesken projektin ei suju yleensä kovin helposti, minkä vuoksi niiden kaltainen prosessikeskeinen lähestymistapa koetaan usein rajoittavaksi. Tämä on johtanut ketterien menetelmien syntyyn, jotka painottavat kehitystyössä joustavuutta ja nopeaa reagointia muutoksiin. Ketterissä menetelmissä korostuu myös ihmisten välinen yhteistyö ja sopeutuminen uusiin vaatimuksiin yhteistyössä asiakkaiden kanssa. (Awad 2005.)

2.1.1 Vesiputousmalli

Vesiputousmalli on perinteisistä ohjelmistokehitysmalleista tunnetuin ja sen esitteli ensimmäisessä muodossaan Winston W. Royce vuonna 1970. Vesiputoustermiä käytetään yleisesti lineaarisista metodeista ohjelmistokehityksessä (Awad 2005). Vesiputousmalli on lineaarinen, vaiheittainen kehitysprosessi,

jossa jokaisen vaiheen tulee valmistua ennen seuraavaan vaiheeseen siirtymistä. Vaiheita ovat järjestelmän ja ohjelmiston vaatimukset, analyysi, ohjelman suunnittelu, koodaus, testaus ja ylläpito. Vesiputousmallissa oletetaan, että projektin vaatimukset pysyvät muuttumattomina kehitysprosessin aikana, mikä on yksi mallin keskeisistä rajoituksista. Tämä oletus on usein epärealistinen, sillä ohjelmistovaatimukset voivat kehittyä ja muuttua projektin edetessä. (Royce 1970; Awad 2005; Aggarwal & Singh 2005: 21–22.)

Vesiputousmallin suurin haaste on sen rakenteessa, jossa toimivaa ohjelmistoa ei ole saatavilla ennen kuin kehitysprosessi on valmis, ja testaus suoritetaan vasta kehitysprosessin lopussa. Testauksessa mahdollisesti löydetyt ongelmat voivat tarkoittaa merkittäviäkin muutoksia koodiin, mikä johtaa usein projektin viivästymiseen ja kustannusten kohoamiseen. (Aggarwal & Singh 2005: 23; Royce 1970.) Royce (1970) huomauttaa, että tämä rakenne houkutteleekin epäonnistumisia, sillä se tarjoaa vain rajallisesti joustavuutta käsitellä ennalta arvaamattomia ongelmia. Tämän vuoksi vesiputousmalli ei aina sovellu projekteihin, joissa vaaditaan joustavuutta ja kykyä reagoida muutoksiin kehityksen aikana. (Royce 1970.) Kuva 2 havainnollistaa vesiputousmallin vaiheita.



Kuva 2 Vesiputousmalli

2.1.2 Spiraalimalli

Spiraalimalli on joustava ja iteratiivinen lähestymistapa ohjelmistokehitykseen. Siinä on valittu perinteisistä malleista, esimerkiksi vesiputousmallista, hyvät puolet ja yhdistelty niistä spiraalimalli, joka jakaa kehitystyön sykleihin kuvaten spiraalin kierroksia. Jokainen sykli sisältää neljä päävaihetta:

1. Tavoitteiden asettaminen, vaihtoehtojen etsiminen ja rajoitteiden tunnistaminen

2. Riskianalyysi ja riskien hallinta

3. Kehitys ja testaus

4. Asiakkaan arviointi ja seuraavan vaiheen suunnittelu

Spiraalimallin jokaisessa syklissä tarkastellaan huolellisesti riskejä, arvioidaan vaihtoehtoisia ratkaisuja ja tehdään näiden perusteella päätöksiä projektin etenemisestä. Spiraalirakenne mahdollistaa sen, että projektia voidaan muuttaa tarpeen mukaan, jos olosuhteet muuttuvat ja täten virheitä voidaan tunnistaa ja korjata aikaisessa vaiheessa. (Boehm 1986; Aggarwal & Singh 2005: 27.)

Keskeisiä piirteitä spiraalimallissa ovat riskienhallinnan voimakas painotus, iteratiivinen lähestymistapa ohjelmiston kehitykseen, asiakkaan aktiivinen osallistuminen jokaisessa vaiheessa sekä mahdollisuus palata prosessissa taaksepäin ja tarkistaa ovatko ratkaisut asiakkaan toiveiden mukaisia ennen lopullista toteutusta. Boehmin (1986) mukaan spiraalimalli sopii parhaiten suuriin ja monimutkaisiin projekteihin, jossa riskit ja muutosten mahdollisuus ovat merkittäviä. (Boehm 1986.)

2.1.3 Ketterät menetelmät

Ketterät menetelmät muodostuivat vastauksena nopeasti muuttuvaan liiketoiminta- ja teknologiaympäristöön, jossa perinteinen ohjelmistokehitys ja sen tiukkaan määritellyt prosessit ei enää kyennyt vastaamaan muutostarpeisiin.

Cockburn ja Highsmith (2001a) korostavat, että ketterien menetelmien tarkoitus on hyväksyä muutos jo lähtökohtaisesti osaksi kehitysprosessia, ja keskittyä vähentämään muutosten aiheuttamia kustannuksia koko ohjelmiston elinkaaren ajan. Ketterässä kehityksessä painotetaan nopeaa palautetta, yksinkertaisten ratkaisujen luomista, jatkuvaa laadun parantamista ja tehokasta testausta, jotta muutoksiin reagoiminen olisi tehokasta. Tämä voidaan saavuttaa esimerkiksi tuomalla käyttäjäasiantuntijat osaksi tiimiä, jotta nopean palautteen saaminen toteutuisi. (Cockburn & Highsmith 2001a; Beck ym. 2001.)

Toisin kuin perinteisissä menetelmissä, joissa tarkat ja laajat prosessikuvaukset, työkalut ja sopimukset ovat tärkeässä roolissa, ketterissä menetelmissä korostetaan toimivaa ohjelmistoa ja ihmisten välistä vuorovaikutusta (Cockburn & Highsmith 2001a). Agile Manifesto – ketterän kehityksen julistus – kiteyttää tämän ajatustavan arvottamalla yksilöt ja vuorovaikutuksen korkeammalle kuin prosessit ja työkalut, toimivan ohjelmiston tärkeämmäksi kuin laajan dokumentaation, asiakasyhteistyön olennaisemmaksi kuin sopimusneuvottelut, sekä muutoksiin reagoinnin tärkeämmäksi kuin alkuperäisen suunnitelman tiukan noudattamisen (Beck ym. 2001).

Ketterä kehitys perustuu vahvasti ihmisten osaamiseen ja tiimien väliseen yhteistyöhön. Teknisen taidon lisäksi ketteryys vaatii myös kykyä sopeutua muuttuviin tilanteisiin sekä kykyä kommunikoida tehokkaasti. Menestyvät ketterät tiimit ovat itseorganisoituvia, sietävät epävarmuutta ja tekevät päätöksiä nopeasti yhdessä. (Cockburn & Highsmith 2001b.) Agile Manifeston taustalla onkin ajatus siitä, että ohjelmistokehityksen ydin on ihmisissä, yhteistyössä ja kyvyssä sopeutua muutoksiin, ei yksityiskohtaisesti määritellyissä prosesseissa (Beck ym. 2001).

Ketteriä menetelmiä, jotka noudattavat samaa yllä kuvattua ajattelutapaa ja viitekehystä, on monia, esimerkiksi Extreme Programming (XP), FDD (Feature-Driven Development) ja Kanban (Awad 2005). Yksi tunnetuimmista menetelmistä on Scrum. Se on iteratiivinen ja inkrementaalinen menetelmä, joka on suunniteltu vastaamaan kehitysprosessin ennakoimattomuuteen. Scrum perustuu 1–4 viikkoa kestäviin sprintteihin, joiden aikana tiimit työskentelevät

jatkuvasti työtään arvioiden, jotta voivat tarvittaessa tehdä muutoksia sen mukaan, kun tarve vaatii. Muita Scrum-menetelmää noudattavien projektien olennaisia piirteitä ovat esimerkiksi pienet tiimit, säännölliset arvioinnit, yhteistyö sekä joustava aikataulu. (Schwaber 1997.)

2.2 Perinteinen ohjelmistokehitys

Perinteinen ohjelmistokehitys on monivaiheinen ja dynaaminen prosessi, joka perustuu usein koodin käsin kirjoittamiseen ja perinteisten ohjelmointikielten käyttöön, kuten Java, Python ja C#. Tämä lähestymistapa korostaa ammattitaitoa ja käsityöläisyyttä, sillä ohjelmointi vaatii paljon taitoa ja jatkuvaa osaamisen kehittämistä. Koodarin rooli onkin perinteisessä ohjelmistokehityksessä keskeinen. (Martin 2021.) On tärkeää myös huomioida konteksti, jossa ohjelmointi tapahtuu, kuten laitteistot ja ympäröivät prosessit, jotka voivat vaikuttaa ohjelmiston toimintaan ja kehitykseen (Sommerville 1998).

Ohjelmistokehittäminen on itsessään usein monimutkainen ja aikaa vievä prosessi. Aggarwalin ja Singhin (2005: 3–4) mukaan yksi sen suurimmista haasteista on vaatimusten määrittely, sillä oikeanlaisten vaatimusten kerääminen ja ymmärtäminen on perusta koko kehitykselle. Myös testaus on erittäin kriittinen osa prosessia, sillä virheet ohjelmiston logiikassa voivat olla paljon haitallisempia kuin koodissa olevat virheet, jotka usein ovat helposti korjattavissa. Ohjelmistokehittämisen tavoitteena on aina tuottaa ohjelmisto, joka täyttää määritellyt vaatimukset, on laadukas ja pysyy budjetin ja aikarajoitteiden puitteissa. (Aggarwal & Singh 2005: 3–5.) Koko kehitysprosessin läpi tulee kulkea huolellisesti ja tarkasti, jotta lopputuote on toimiva ja luotettava.

Ajan myötä ohjelmisto stabiloituu ja tulee luotettavammaksi, ja sen alttius häiriöille vähenee. Ympäristön muutokset tai uudet vaatimukset voivat kuitenkin johtaa ohjelmiston vanhentumiseen. Tällöin ohjelmisto voi poistua käytöstä, jos se ei enää vastaa nykypäivän tarpeita tai vaatimuksia. (Aggarwal & Singh 2005: 10.) Tämän vuoksi ohjelmistokehityksessä onkin tärkeää jatkuvasti arvioida ja päivittää ohjelmistoa, jotta se pysyy relevanttina ja käyttökelpoisena.

2.2.1 Työkalut ohjelmistokehityksessä

Ohjelmistokehityksen avuksi on olemassa monia työkaluja, joista tässä esitellään kaksi. Yksi suosittu työkalu perinteisessä ohjelmistokehityksessä on esimerkiksi IntelliJ Platform, joka on avoimen lähdekoodin ohjelmistokehitysympäristö ja -alusta, joka mahdollistaa laajennusten ja työkalujen kehittämisen (The IntelliJ Platform 2025). Kurbatova ym. (2021) esittelevät neljä pääasiallista käyttötarkoitusta IntelliJ Platformille. Nämä ovat ohjelmistodatan louhinta (data mining), koneoppimismallien ajaminen, refaktorointiehdotusten tekeminen ja tiedon visualisointi (Kurbatova ym. 2021).

Toinen suosittu työkalu on Microsoftin kehittämä Visual Studio Code. Se on ilmainen koodieditori, joka tukee monia eri alustoja ja laitteistoja. Ohjelma tukee muun muassa Git-versionhallintaa, sekä automaattista koodin täydennystä, syntaksin korostusta ja virheiden jäljitystä usealle eri ohjelmointikielelle. Visual Studio Code on myös laajasti mukautettavissa omiin tarpeisiin ja siihen on saatavilla useita eri laajennuksia. (Visual Studio Code documentation.)

2.2.2 Ohjelmointikielet

Python on yksi suosituimmista ohjelmointikielistä, ja sen suosio johtuu monista sen tarjoamista eduista. Se on tehokas kieli, sillä se vaatii vähemmän koodirivejä kuin monet muut ohjelmointikielet, mikä tekee ohjelmoinnista nopeampaa ja vähemmän virhealtista. Pythonin syntaksi on myös helppo ja selkeä, mikä helpottaa koodin lukemista ja virheiden etsimistä. Myös koodin laajentaminen on syntaksin ansiosta helpompaa, kuin monissa muissa kielissä. (Matthes 2023: xxxvi.) Tämä tekee Pythonista houkuttelevan kielen niin aloittelijoille kuin kokeneillekin kehittäjille.

IEE Spectrum esitteli ohjelmointikielten suosion nykytilaa ja kehityssuuntia. Python säilytti johtoasemansa vuoden 2024 suosituimpana ohjelmointikielenä, erityisesti tekoälyn ja opetuskäytön ansiosta. Työmarkkinoilla SQL oli kielistä suosituin, mikä on seurausta nykyisistä verkko- ja pilvipohjaisista järjestelmäarkkitehtuureista, joissa tietokannat ovat ohjelmistossa käsiteltävän datan

pääasiallinen säilytyspaikka. Muita suosittuja ohjelmointikieliä vuonna 2024 olivat Java, JavaScript, C++ ja TypeScript. (Cass 2024.)

Pereira ym. (2021) vertailivat tutkimuksessaan ohjelmointikielten energiatehokkuutta mittaamalla niiden energian-, ajan- ja muistinkäyttöä. Tutkimuksesta kävi ilmi, ettei suoraa yhteyttä suoritusajan ja energiatehokkuuden välillä ollut. Nopeimmat kielet eivät aina olleet energiatehokkaimpia, ja muistinkäyttö vaikutti merkittävästi energian kulutukseen. Käännetyt kielet, kuten C, C++ ja Rust olivat suoritusaikatehokkuudeltaan keskimäärin parhaita. Sen sijaan tulkattavat kielet, kuten Python ja Ruby, kuluttivat enemmän energiaa ja aikaa. (Pereira ym. 2021.) Tämä osoittaa, että ohjelmointikielen valinta riippuu käytettävissä olevista resursseista, kuten ajasta, energiasta ja muistista, ja ohjelmistokehittäjän on otettava huomioon nämä tekijät valitessaan kieltä tietyille tehtäville.

2.3 Low-code ja no-code-kehitys

2.3.1 Low-code ja no-code määritelmät

Forrester Research esitteli termin ”low-code” ensimmäisen kerran vuonna 2014. Tutkimuslaitoksen mukaan yritykset suosivat low-code-ratkaisuja, koska ne mahdollistavat nopean ja jatkuvan kehityksen sekä tarjoavat hyvät edellytykset kokeiluun ja oppimiseen kehitysprosessin aikana. (Richardson ym. 2014, viit. Luo ym. 2021 mukaan.) Bock ja Frank (2021a) kuitenkin huomauttivat, että ”low-code” ei heidän tutkimuksensa perusteella muodosta teknologisilta ominaisuuksiltaan yhtenäistä kokonaisuutta, minkä vuoksi termin käyttö tieteellisessä kontekstissa on heidän mukaansa kyseenalaista.

Luo ym. (2021) tutkimuksen mukaan low-code-kehitystä kuvataan usein prosessiksi, jossa ohjelmointi vaatii vain vähäistä vaivannäköä. Käyttöliittymä on usein graafinen ja hyödyntää visuaalisia abstraktioita ja toimii usein ”vedä ja pudota” tyylillä, eikä vaadi juurikaan käsin koodaamista tai vaatii sitä vain vähän. Low-code-kehitystä voikin kuvailla visuaaliseksi ohjelmoinniksi, jossa voidaan käyttää valmiiksi suunniteltuja pohjia ohjelmiston luomiseen. (Luo ym. 2021; The Definitive Guide to Low-Code Development; Waszkowski 2019.)

Low-code-kehitykseen on olemassa useita eri alustoja ja työkaluja, joista voidaan valita sopiva kehityksessä olevaan projektiin. Sopivaa alustaa valitessa tulee ottaa huomioon esimerkiksi kehitysprojektin budjetti, halutaanko mahdollisuus päästä käsiksi lähdekoodiin sekä se, mitä ohjelmointikieltä kyseinen alusta tukee. Lisäksi on tärkeää tietää, tarjoaako valittu alusta riittävästi toteutusyksiköitä sovelluksen eri osien – kuten käyttöliittymän, taustalogiikan ja tietovaraston – kehittämiseen low-code-menetelmällä. (Luo ym. 2021.)

No-code-kehityksellä tarkoitetaan kehitystä, jossa käsin koodausta ei tarvita lainkaan. Se sopii käyttäjille, joilla ei ole lainkaan kokemusta koodauksesta, ja on usein yksinkertainen, visuaalinen ympäristö rajallisella kustomoinnilla ja integraatioilla (The Definitive Guide to Low-Code Development). Monissa tutkimuksissa no-code-kehitys rinnastetaan kuitenkin yhteen low-code-kehityksen kanssa (Käss ym. 2022; Guthardt ym. 2024), joten myös tässä työssä käytetään jatkossa low-code-termiä laajassa merkityksessä viittaamaan sekä low-code- että no-code-kehitykseen.

2.3.2 Low-code-alustat

Low-code-kehitykseen on saatavilla useita alustoja, joista käyttäjä voi valita mieleisensä. Mendix on yksi suosituista alustoista low-code-kehitykseen. Se tarjoaa mahdollisuuden sovellusten luomiseen, testaamiseen, julkaisemiseen ja skaalaamiseen pilvessä. Mendixillä on laaja tuki integraatioille ja saatavilla on myös AI-avustetut kehitystyökalut. (Mendix Platform.)

Samankaltainen alusta on myös Microsoft PowerApps, joka on suunnattu erityisesti Microsoftin ekosysteemissä toimiville organisaatioille. Se on integroitunut tiiviisti muihin Microsoftin pilvipalveluihin, kuten Office 365, SharePoint ja Dynamics 365. Myös PowerApps sisältää AI-avusteista kehitystä Microsoftin Copilotin avulla. (What is Power Apps? 2024.)

Kolmas suosittu low-code-alusta on OutSystems, jonka vahvuuksia ovat muun muassa sen nopea kehitysaika, monikanavainen tuki ja laaja integraatiokyky. Alustan tarjoama visuaalinen kieli tarjoaa kehittäjille mahdollisuuden mallintaa ja

suunnitella sovellusten eri kerroksia abstraktiotasolla. Kuten muutkin low-code-alustat, myös OutSystems tarjoaa tekoälyavusteista kehitystä sisäänrakennetun AI-ratkaisun avulla. (The AI-powered low-code platform.)

Sahay ym. (2020) tutkimuksen mukaan low-code-alustojen haasteena on muun muassa niiden yhteensopivuus toistensa kanssa. Low-code-alustat ovat usein yksinoikeudellisia ja suljettuja, jonka vuoksi artefaktien ja toteutusten uudelleenkäyttö muilla alustoilla on hankalaa. Lisäksi tällaisilla alustoilla laajennettavuus ei ole aina helppoa tai mahdollista, vaan voi vaatia laajaakin koodaamista. (Sahay ym. 2020.)

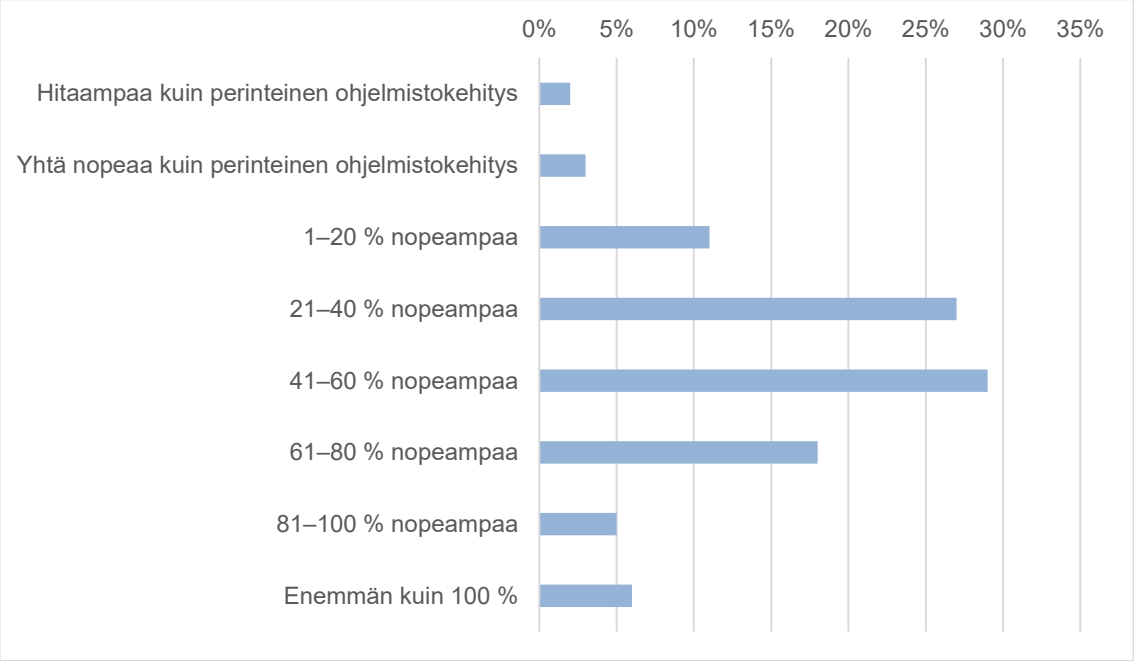
3 Vertaileva analyysi

3.1 Kehitysnopeus ja kustannukset

Low-code-kehitys mahdollistaa nopeamman kehityksen sekä sovellusten tuomisen markkinoille nopeammin kuin perinteisellä ohjelmistokehityksellä. Kehitys on myös usein kustannuksiltaan matalampaa. Tuottavuuden paraneminen low-code-kehityksessä selittyy osin sillä, että se keventää toistuvien ja rutiininomaisten tehtävien työmäärää. (Bock & Frank 2021a; Luo ym. 2021.) Osa low-code-alustoista voi kuitenkin olla kalliita hankkia suurelle määrälle työntekijöitä, mikä voi tehdä siitä kalliimman tavan kehittää ohjelmistoja (Luo ym. 2021). Tutkimusten mukaan pienet ja keskisuuret yritykset ovatkin kiinnostuneita hyödyntämään low-code-alustoja kehitysprosessien tehostamiseen, mutta samalla ne kokevat korkeiden koulutuskustannusten riskin merkittävänä haasteena (Guthard ym. 2024).

Käss ym. (2020) nostavat esille low-code-kehityksen ajureina ohjelmistokehitysprosessin suorituskykymetriikoiden parantumisen, ohjelmistokehityksen esteiden vähentymisen ja kehittämisen helpottumisen. Kehityksen nopeus onkin yksi low-code-kehityksen ylivoimaisista eduista perinteiseen ohjelmistokehitykseen verrattuna.

Creatio julkaisi vuonna 2021 tutkimuksen, jossa yli tuhat IT- ja liiketoiminta-alan toimijaa vastasi kysymyksiin low-code-kehityksestä. Yksi tutkimuksen kysymyksistä koski sitä, kuinka paljon nopeammaksi vastaajat arvelivat low-code-kehityksen verrattuna perinteiseen ohjelmistokehitykseen. Kuvassa 3 on esitetty, miten vastaukset kysymykseen jakautuivat. Suurin osa vastaajista oli sitä mieltä, että low-code-kehitys on 41–60 % nopeampaa. (The state of low-code/no-code 2021.)



Kuva 3 Low-code-kehityksen nopeus verrattuna perinteiseen ohjelmistokehitykseen

3.2 Joustavuus ja skaalautuvuus

Low-code-alustat ovat pääsääntöisesti pilvipohjaisia, mikä mahdollistaa pilvipohjaisten sovellusten kehittämisen ja hyödyntämisen tehokkaasti (Khorram ym. 2020). Kuitenkin niiden käytössä on haasteita, kuten joustamattomuus ja mukautettavuuden puute, jotka voivat rajoittaa niiden soveltuvuutta eri tarpeisiin. Esimerkiksi skaalauksen mahdollisuus ei välttämättä ole low-code-alustoilla yhtä hyvä, kuin perinteisessä ohjelmistokehityksessä (Käss ym. 2022). Lisäksi low-code-alustoilla kehitys on usein rajoitettu siihen mitä alusta tarjoaa, kun taas perinteisellä ohjelmistokehityksellä käsin koodaamalla on mahdollista saavuttaa mitä tahansa, millä tahansa valitsemillaan teknologioilla ja työkaluilla (Safi 2023; Shridhar & Bose 2021).

Toisaalta monet low-code-alustat tarjoavat kuitenkin mahdollisuuden perinteiseen ohjelmointiin jollain tasolla, mikä voi olla hyödyllistä monissa tilanteissa. Useimmissa alustoissa on tuki ohjelmointikielille, kuten Java tai JavaScript, joilla on mahdollista lisätä rajoitetusti lisää räätälöityjä toimintoja. (Bock & Frank 2021b.) Vaikka tämä ei vastaa aivan perinteisen ohjelmointikehityksen

tarjoamaa räätälöintimahdollisuutta, voi sen avulla kuitenkin hyödyntää molempien kehitystapojen parhaita puolia.

Low-code-alustojen käytössä on omat haasteensa erityisesti silloin, kun siirrytään yhdeltä alustalta toiselle. Siirtyminen vaatii usein koko toteutuksen uusimisen vastaamaan uuden alustan vaatimuksia, sillä low-code-alustat eivät ole keskenään täysin yhteensopivia. (Sufi 2023.) Tämä voi aiheuttaa merkittäviä hankaluuksia organisaatioille, jotka ovat investoineet aikaa ja resursseja tietyn alustan käyttöönottoon, sillä siirtyminen voi edellyttää kokonaan uuden kehitystyön aloittamista tai aiemmin kehitetyn sovelluksen sopeuttamista uuteen ympäristöön.

Khorram ym. (2020) nostavat esiin haasteen low-code-sovellusten testauksen suhteen. Liiketoimintakäyttäjien matala tekninen osaaminen voi koitua haasteeksi sovelluksen testausvaiheessa, jossa tekninen osaaminen olisi hyödyksi. Low-code-testauksen pitäisi tarjota korkeatasoista testiautomaatiota, ilman että se on riippuvainen käyttäjien teknisestä osaamisesta. Low-code-alustoilla ei ole lisäksi vakiintunutta testauskehystä, mikä vaikeuttaa eri low-code-alustojen testausta ja yhteensopivuutta. (Khorram ym. 2020.)

3.3 Käyttäjäystävällisyys ja kohderyhmät

Low-code-kehitys on tavallisesti käyttäjäystävällisempi vaihtoehto verrattuna perinteiseen ohjelmistokehitykseen, ja se mahdollistaa kehittämisen myös henkilöille, joilla ei ole paljon teknistä osaamista (Luo ym. 2021; Totterdale 2018). Low-code-ympäristöissä käyttäjiltä ei vaadita syvällistä ohjelmointiosaamista tai yksityiskohtaista ymmärrystä kehitysympäristöistä, vaan low-code-ympäristöjen vetämis- ja pudottamisominaisuudet ovat helppokäyttöisiä käyttäjille vähemmälläkin teknisellä osaamisella (Totterdale 2018).

Käyttäjätutkimukset ovat osoittaneet, että niin kokeneet kehittäjät kuin kehitystyössä kokemattomat henkilöt pystyvät omaksumaan low-code-työkalujen käytön nopeasti ja luomaan onnistuneesti sovelluksia jo alle tunnissa (Guthard ym. 2024). Tämä tekee alustoista houkuttelevan vaihtoehdon monille

organisaatioille, erityisesti jos käytössä olevat IT-resurssit ja budjetti ovat rajalliset. Low-code-kehityksellä voidaan paikata myös osaavien koodaajien puutetta, kun liiketoimintakäyttäjiä voidaan osallistaa mukaan kehitysprosesseihin ilman erityistä ohjelmointitaustaa (Sahay ym. 2020).

Kuitenkaan kaikki low-code-alustat eivät ole yhtä käyttäjäystävällisiä, vaan oppimiskäyrä voi olla jyrkkä. Jotkin alustat tarjoavat monimutkaisia toimintoja, joiden oppiminen vaatii käyttäjältä aikaa. (Luo ym. 2021.) Lisäksi osalla alustoista graafinen käyttöliittymä ei ole kovin intuitiivinen, ja niiden tarjoamat mahdollisuudet ovat rajallisia. Jos opetusmateriaalit, kuten tutoriaalit ja ohjeet alustan käyttöön, ovat puutteellisia, voi käyttäjien olla haastavaa opetella uuden alustan toiminta. Osa järjestelmistä voi vaatia myös jonkin verran ohjelmointiosaamista, vaikka usein vaikka low-code-alustojen kohderyhmänä ovat nimenomaan käyttäjät, joilla tällaista teknistä osaamista ei ole paljon. (Sahay ym. 2020.)

Toisaalta myös perinteisen ohjelmoinnin opettelu voi olla aikaa vievää ja monivaiheista. Verrattuna perinteiseen ohjelmistokehitykseen, low-code-kehityksen oppiminen on kuitenkin helpompaa ja nopeampaa, mikä tekee siitä houkuttelevan vaihtoehdoin organisaatioille, jotka tarvitsevat nopeita ratkaisuja ja haluavat välttää perinteisen ohjelmistokehityksen kalliit kustannukset. (Luo ym. 2021.)

Low-code-alustojen etuna on myös niiden kyky yksinkertaistaa monia kehitystyön osa-alueita, kuten tietokannan hallintaa. Low-code-alustalla tietokannan hallinta ja käsittely on usein yksinkertaisempaa ja käyttäjäystävällisempää kuin perinteisessä ohjelmointikehityksessä (Totterdale 2018).

3.4 Tietoturva ja luotettavuus

Vaikka low-code-alustat tarjoavat yksinkertaista sekä helppoa ja nopeaa kehitystä, ovat nekin kuitenkin alttiita turvallisuusriskeille (Bargury 2021). Low-code-alustat tarjoavat automaattisesti kansainväliset tietoturvastandardit, kuten ISO/IEC 27001 ja PCIDSS (Sahinaslan ym. 2021), sekä erilaisia valmiiksi rakennettuja salaus- ja pääsynhallintatoimintoja, kuten kertakirjautumisen,

tunnistautumisen ja valtuutuksen (Tadi 2021; Sufi 2023). Tämä yksinkertaistaa tietoturvan hallintaa projektissa.

Perinteisessä kehityksessä taas yritys on itse kokonaan vastuussa oman ohjelmistonsa turvallisuudesta, mikä tuo mukanaan sekä haasteita että etuja. Toisaalta tämä mahdollistaa täyden kontrollin tietoturvauhkiin puuttumisessa ilman riippuvuutta ulkopuolisista toimijoista, mutta samalla se lisää vastuuta ja riskejä, mikä voi olla erityisen haastavaa pienemmille organisaatioille, joilla ei ole resursseja erikoistuneisiin tietoturvatiimeihin. (Shridhar & Bose 2021.)

Zenityn artikkelissa kuvaillaan keskeisimpiä turvallisuusriskejä low-code-kehityksessä (Bargury 2021). Niitä tunnistetaan seitsemän:

1. **Käyttöoikeuksien laajentuminen**: Low-code-sovellukset käyttävät usein henkilökohtaisia käyttäjätunnuksia järjestelmän roolien sijaan, mikä voi johtaa valtuuttamattomaan pääsyyn tai liiallisiin oikeuksiin, jos tunnuksia ei hallita asianmukaisesti.

2. **Tietovuoto**: Low-code-sovellukset liikuttavat usein dataa, mikä voi vahingossa päätyä organisaation rajojen ulkopuolelle, erityisesti silloin, jos käytetään valtuuttamattomia palveluita tai tallentamalla tietoja henkilökohtaisille levyille.

3. **Turvaton tunnistautuminen**: Monet low-code-sovellukset on tehnyt kehittäjät, joilla ei välttämättä ole asiantuntemusta turvallisessa tunnistautumisessa. Se voi johtaa turvattomiin yhteyksiin, joissa käytetään heikkoja salauksia.

4. **Väärät määritykset**: Jotkin low-code-alustat, kuten Power Platformin Portal Apps, mahdollistavat määritykset, jotka voivat vahingossa altistaa tietojen leviämisen valtuuttamattomille käyttäjille. Alustan tietoturvaohjeiden seuraaminen ja jatkuvan valvontaratkaisun käyttäminen voi pienentää tämän uhan riskiä.

5. **Riippuvuuden injektio**: Low-code-sovellukset luottavat voimakkaasti kolmannen osapuolen komponentteihin, jotka usein tulevat markkinapaikoista, jossa käyttäjät voivat jakaa komponentteja keskenään. Tämä tuo mukanaan tietoturvahyökkäyksen riskin. Kehittäjien tulisi tarkistaa kolmannen osapuolen komponenttien lähteet huolellisesti ennen niiden käyttämistä.

6. **Liiallinen jakaminen**: Low-code-alustat tekevät sovellusten, tietojen ja komponenttien jakamisesta helppoa, mikä voi johtaa liialliseen jakamiseen ja yrityksen käyttöoikeusmallin rikkomiseen.

7. **Luotettavaksi sovellukseksi tekeytyminen**: Käyttäjät usein luottavat low-code-sovelluksiin ilman tarkempaa tarkastelua, sillä ne ovat organisaation kehittämiä. Tämä antaa hyökkääjille mahdollisuuden esiintyä luotettavina sovelluksina ja käynnistää kalastelukampanjoita organisaatiossa. (Bargury 2021.)

Vaikka low-code-alustoilla on monia etuja kyberturvallisuudessa, kuten muun muassa nopea reagointi uusiin uhkiin, räätälöityjen ominaisuuksien luominen nopeasti sekä kustannusten säästö, kun vanhojen järjestelmien tilalle voidaan tehdä parannuksia ilman uusien työkalujen hankintaa, voivat ne aiheuttaa kuitenkin myös monia kyberturvallisuusriskejä. On tärkeää varmistaa, että kaikki luotu koodi käy läpi perusteellisen testauksen ja että koodin lähde on luotettava, erityisesti silloin, jos ratkaisu ostetaan kolmannen osapuolen toimittajalta. (Peruzzi 2023.) Koska low-code-alustat toimivat pääosin pilvessä, liittyy niihin myös paljon yleisiä pilvipalveluiden kyberturvallisuusuhkia, joita myös perinteinen ohjelmistokehitys kohtaa (Tissir ym. 2020).

Perinteisessä ohjelmistokehityksessä turvallisuudesta vastaavat usein kokeneet kehittäjät, kun taas low-code-kehittäjillä ei välttämättä ole yhtä paljon teknistä osaamista. Tämä luo haasteita turvallisuuden varmistamiseen, sillä kehittäjät voivat rikkoa alustan tarjoamia rajoja ja heikentää sovellusten turvallisuutta. (Schwartz 2021.)

3.5 Integraatiot ja laajennettavuus

Yksi low-code-alustojen keskeisistä eroista verrattuna perinteisiin kehitysalustoihin on niiden kokonaisvaltainen luonne: ne tarjoavat yhdessä ja samassa ympäristössä suurimman osan tarvittavista työkaluista ja komponenteista tietyn tyyppisten ohjelmistoprojektien toteutukseen. Perinteisessä ohjelmistokehityksessä joudutaan usein käyttämään erillisiä järjestelmiä ja sovittamaan yhteen eri ohjelmistoja, kuten kehitysympäristöjä, mallinnustyökaluja, tietokantajärjestelmiä ja käyttöliittymäkirjastoja. Low-code-alusta yksinkertaistaa työnkulkuja ja vähentää tarvetta hallita eri teknologioiden tuottamia irrallisia komponentteja tarjoamalla keskitetyn ympäristön, joka tukee tehokkaampaa ohjelmistokehitystä. (Bock & Frank 2021a.)

Kuitenkin Luo ym. (2021) tutkimuksen mukaan osa kehittäjistä kokee low-code-alustojen olevan puutteellisia mukautettavuuden suhteen. Esimerkiksi ulkoasut sekä muotoilut voivat olla vaikeasti muokattavissa. Perinteinen ohjelmistokehitys mahdollistaa rajattomat mahdollisuudet muokata ohjelmistoa tai sovellusta. (Luo ym. 2021.)

Käss ym. (2022) mukaan integraatiot ovat yksi keskeisimmistä haasteista low-code-kehityksen käyttöönotolle. Vaikka integraatiot ovat usein yksinkertaisia alustalla itsessään, ne voivat kuitenkin vaatia huomattavaa panostusta toisiin järjestelmiin, kuten useita organisaation hyväksyntöjä järjestelmään pääsemiseksi. Tämä voi hidastaa kehitysprosessia tai lisätä vaatimuksia käyttäjien tekniselle osaamiselle. (Käss ym. 20222.)

Low-code-alustat tarjoavat myös usein mahdollisuuden käyttää ulkoisia tietolähteitä API:en kautta, jolloin dataa voidaan tallentaa joko alustan sisäisiin tai ulkoisiin tietokantoihin. Alustoilla on usein myös kirjastot käytössä yleisille vakiotoiminnoille, esimerkiksi matemaattisille funktioille. Low-code-alustoilla erilaiset kehityskomponentit on yhdistetty yhteen ja samaan ympäristöön, mikä vähentää tarvetta vaihtaa eri järjestelmien välillä ja helpottaa ohjelmointiprosessia. Tämä on suuri etu verrattuna perinteiseen ohjelmointikehitykseen, jossa on usein

käytössä monia eri työkaluja ja järjestelmiä, mikä voi tehdä kehityksestä moni-vaiheista ja aikaa vievää. (Bock & Frank 2021b.)

4 Käyttötapaukset ja esimerkit

4.1 Low-code kehityksen käyttö

Low-code-kehitys on saanut laajaa suosiota esimerkiksi mobiilisovellusten kehittämisessä, joissa toimitusaika on usein lyhyt ja kehitysprosessin nopeuttaminen luo huomattavia etuja. Luo ym. (2021) tutkimuksen mukaan low-code-alustat mahdollistavat sovellusten nopean kehittämisen, mikä tekee niistä houkuttelevan vaihtoehdon organisaatioille. Lisäksi aloilla, joilla pyritään parantamaan toiminnan tehokkuutta prosessi- ja työnkulkuautomaatioiden avulla, low-code-menetelmät ovat nousseet suosituiksi kehitysvaihtoehdoiksi (Luo ym. 2021).

Low-code-kehitys sopii myös organisaatioille, joilla on rajalliset IT-resurssit ja budjetti. Niiden avulla voidaan toteuttaa valmiita sovelluksia kaikkine ominaisuuksineen nopeassa tahdissa, joka säästää kustannuksia ja helpottaa kehitysprosessia. Organisaatio voi kuitenkin joutua muokkaamaan vaatimuksiaan projektille sen mukaan, mitä kehitykseen valitulla low-code-alustalla pystytään toteuttamaan, sillä kaikki työkalut ja ominaisuudet eivät ole samanlaisia eri low-code-palveluntarjoajilla. (Sahay ym. 2020.)

Nopean kehityksen lisäksi low-code-alustat tarjoavat myös huoletonta ylläpitoa. Alustaorganisaatio vastaa sen parannuksista ja ohjelmistopäivityksistä, joka yksinkertaistaa järjestelmän ylläpitoa käyttäjälle, ja on siten vähemmän vaativa ratkaisu yritykselle, jolla on vähemmän IT-resursseja käytettävissä (Shridhar & Bose 2021). Low-code alustat mahdollistavat myös paremman yhteensopivuuden liiketoiminnan ja IT-osaston välillä, sillä ne tarjoavat visuaalisia työkaluja, joiden avulla liiketoimintakäyttäjät voivat osallistua sovellusten kehittämiseen ilman syvällistä teknistä osaamista. Näin saadaan ratkaistua ongelma, joka voi syntyä, jos liiketoiminnan visiot digitaalisesta kehityksestä ovat liian haastavia kääntää toteutuksiksi esimerkiksi heikon viestinnän tai IT-asiantuntijoiden puutteellisen toimialaosaamisen vuoksi. (Sufi 2023.)

Koska lähes kaikki low-code-alustat toimivat pilvessä, on se helppo ja nopea tapa ottaa organisaatiossa käyttöön pilvipohjaiset teknologiat, joihin siirtyminen on nykyaikana suosittua (Sufi 2023; McKendrick 2022). Myös tietokantojen ylläpitäminen sujuu helposti low-code-alustoilla (Totterdale 2018).

Kuitenkin, vaikka low-code-alustat tarjoavat monia etuja perinteiseen ohjelmistokehitykseen verrattuna, niillä on myös omat rajoitteensa. Esimerkiksi kustomointi on usein rajallisempaa low-code-alustoilla. Jos organisaatio kaipaa laajaa räätälöintiä ohjelmistossaan, on usein parempi valita perinteinen ohjelmistokehitys, jossa kustomointiin on rajattomat mahdollisuudet. (Safi 2023; Shridhar & Bose 2021; Tadi 2021.) Lisäksi vaikka low-code-kehitys yksinkertaistaakin tietokantojen hallintaa, low-code-alustojen automaattiset tietokantakyselyt voivat olla tehottomia suurten tietomäärien tai monimutkaisten liitoskyselyiden kanssa. Perinteinen ohjelmistokehitys on usein parempi vaihtoehto silloin, kun on tarve käsitellä raskaita tietokantakyselyitä, sillä se tarjoaa paremman tietokannan optimoinnin ja kyselyjen suorituskyvyn. (Tadi 2021.)

Kehitysprojekteille voi olla myös teknisiä vaatimuksia, jotka rajoittavat kehitystyökalujen valintaa. Jos esimerkiksi vaaditaan integraatiota perinteisiin järjestelmiin, esimerkiksi ERP- tai CRM-järjestelmiin, voivat ne vaatia hyvin räätälöityä API-käyttöä, johon low-code-alustat eivät välttämättä sovi. Perinteinen ohjelmistokehitys antaa yritykselle paremman mahdoolisuuden valita itse joustavasti infrastruktuurin, jotta operatiivinen tehokkuus saadaan taattua. Myös tilanteissa, joissa on kyse sovelluksista joissa turvallisuusvaatimukset ovat korkeat, on perinteinen kehitys varmempi valinta, sillä se tarjoaa paremman hallinnan turvallisuusmäärityksiin. (Tadi 2021.)

Low-code-alustoja käyttäessä on hyvä ottaa huomioon myös uhka alustan vanhentumisesta. Ajan myötä voi käydä niin, ettei alustaa enää päivitetä, sen yksittäiset ominaisuudet poistuvat käytöstä tai sen käyttö mahdollisesti loppuu kokonaan. Myös alustan lisensointimalli ja hinta voi muuttua projektin aikana, joka vaikuttaa kehitysprojektin budjettiin. Perinteinen ohjelmistokehitys tarjoaa enemmän joustavuutta, eikä siinä olla yhden palveluntarjoajan varassa. (Tadi 2021.)

Käss ym. (2023) mainitsevat ohjelmistokehityksen tehostamisen tärkeimpänä low-code-teknologian käyttöönoton ajurina. Sen sijaan haasteita sen käyttöönotossa aiheuttavat low-code-kulttuurin puute sekä muutosvastarinta. Monet yritykset ovat tottuneet perinteisiin kehitysmenetelmiin, joten low-code-ympäristöön siirtyminen voi olla vaikeaa. (Käss ym. 2023.)

4.2 Esimerkkejä low-code-kehityksen käyttötapauksista

Totterdalen (2018) tutkimuksessa esiteltiin käyttötapaus, jossa low-code-kehitystä hyödynnettiin tutkimusdatan keräyksen tukena. Tutkimuksessa kehitettiin työkalu tutkimusdatan keräämiseen terveydenhoitoalalle Mendix-alustan avulla. Low-code-kehitystä voidaan hyödyntää siis nopeuttamaan tiedon keruuta, sekä se tarjoaa paikan tutkimustietojen tallentamiseen. Tämä voi auttaa vähentämään riskejä tiedon suojaamiseen liittyen. (Totterdale 2018.)

Bhattacharyya & Kumar (2021) tutkivat low-code-kehityksen käyttöä toimitusketjujen hallinnassa ja niiden digitalisoinnin parantamiseksi. Tutkimuksen mukaan low-code-alustojen käyttö voi parantaa toimitusketjun läpinäkyvyyttä ja tukea datavetoista päätöksentekoa. Lisäksi se mahdollistaa yrityksille paremman reaaliaikaisen prosessinhallinnan ja nopeuttaa markkinoille pääsyä. Erityisesti pienet ja keskikokoiset yritykset voivat hyötyä tästä. (Bhattacharyya & Kumar 2021.)

5 Johtopäätökset

Low-code-alustat tarjoavat merkittäviä etuja erityisesti organisaatioille, joilla on rajoitetut IT-resurssit ja jotka tarvitsevat nopeita ja kustannustehokkaita ratkaisuja. Alustojen visuaaliset työkalut ja ”vedä ja pudota” -ominaisuudet tekevät niistä helppokäyttöisiä, ja ne mahdollistavat liiketoimintakäyttäjien osallistumisen kehitysprosesseihin ilman syvällistä teknistä osaamista. Tämä nopeuttaa kehitystä ja vähentää IT-osaston kuormitusta.

Toisaalta low-code-kehityksen rajoitteet saattavat olla esteenä sen käyttöönotossa. Rajoitteet liittyvät erityisesti skaalautuvuuteen, integraatioihin ja kustomointiin, joiden suhteen mahdollisuudet ovat edelleen paremmat perinteisen ohjelmistokehityksen parissa. Tästä syystä onkin oleellista, että ennen kehitystavan valintaa perehdytään huolellisesti projektin luonteeseen sekä siihen, minkälaisia resursseja kehitykseen on. Erityisesti suuremmissa ja monimutkaisemmissa projekteissa kallistutaan edelleen perinteisen ohjelmistokehityksen puoleen.

Low-code-alustojen käyttöön liittyy myös riskejä, jotka korostuvat erityisesti silloin, jos kehittäjillä ei ole vahvaa teknistä osaamista. Mahdolliset tietoturva-aukot sekä kolmannen osapuolen lisäosien hyödyntäminen voi aiheuttaa haittaa, jos ohjelmistoon pääsee haittaohjelmia. Lisäksi haasteita voi olla siinä, jos ohjelmisto halutaankin siirtää alustalta toiselle, sillä low-code-alustat eivät usein ole toistensa kanssa yhteensopivia.

Low-code-kehityksen suosio kasvaa jatkuvasti, mutta sen käyttöönotossa on tärkeää huomioida organisaation tarpeet ja varmistaa, että tietoturva on hallinnassa. Erityisesti pienille ja keskikokoisille yrityksille low-code-alustat tarjoavat mahdollisuuden tehostaa liiketoimintaprosesseja ja nopeuttaa markkinoille pääsyä, mutta on tärkeää ymmärtää myös alustojen rajoitukset ja varmistaa, että ne soveltuvat yrityksen ja kehitysprojektin tarpeisiin pitkällä aikavälillä.

Lähteet

Aggarwal, K.K., Singh, Yogesh. 2005. Software engineering. New Age International.

Awad, M. A. 2005. A comparison between agile and traditional software development methodologies. University of Western Australia, 30, s. 1–69.

Bargury, Michael. 2021. The 7 Deadly Sins of Low-Code Security and How to Avoid Them. Verkkoaineisto. Zenity. <https://zenity.io/blog/security/low-code-security-risks-7-sins-and-how-to-overcome-every-single-one> 17.11.2021. Luettu 8.5.2025.

Bhattacharyya, Som Sekhar & Kumar, Saurabh. 2023. Study of deployment of “low code no code” applications toward improving digitization of supply chain management. Journal of Science and Technology Policy Management, 14(2), s. 271–287.

Beck, Kent, Beedle, Mike, Van Bennekum, Arie, Cockburn, Alistair, Cunningham, Ward, Fowler, Martin, Grenning, James, Highsmith, Jim, Hunt, Andrew, Jeffries, Ron, Kern, Jon, Marick, Brian, Martin, Rober C., Mellor, Steve, Schwaber, Ken, Sutherland, Jeff & Thomas, Dave. 2001. Manifesto for Agile Software Development.

Bock, Alexander C. & Frank, Ulrich. 2021a. In Search of the Essence of Low-Code: An Exploratory Study of Seven Development Platforms. Teoksessa: ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C). Fukuoka, Japani, s. 57–66.

Bock, Alexander C. & Frank, Ulrich, 2021b. Low-code platform. Business & Information Systems Engineering, 63, s. 733–740.

Boehm, Barry W. 1986. A Spiral Model of Software Development and Enhancement. ACM SIGSOFT Software engineering notes, 11(4), s. 14–24.

Cass, Stephen. 2024. Top Programming Languages 2024. Verkkoaineisto. IEE Spectrum. <https://spectrum.ieee.org/top-programming-languages-2024> 22.8.2024. Luettu 2.5.2025.

Cockburn, Alistair & Highsmith, Jim. 2001a. Agile software development: The business of innovation. Computer, 34(9), s. 120–127.

Cockburn, Alistair & Highsmith, Jim. 2001b. Agile software development: The people factor. Computer, 34(11), s. 131–133.

Gunnell, Marshall. 2024. Software Development Life Cycle (SDLC). Verkkoaineisto. Techopedia. <https://www.techopedia.com/definition/22193/software-development-life-cycle-sdlc> Päivitetty 26.7.2024. Luettu 19.4.2025.

Guthard, Till, Kosiol, Jens & Hohlfeld, Oliver. 2024. Low-code vs. the developer: An empirical study on the developer experience and efficiency of a no-code platform. Teoksessa: ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS Companion '24), 22.–27. syyskuuta 2024, Linz, Itävalta. ACM, s. 856–865.

Khorram, Faezeh, Mottu, Jean-Marie & Sunyé, Gerson. 2020. Challenges & opportunities in low-code testing. Teoksessa: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, s. 1–10.

Kurbatova, Zarina, Golubev, Yaroslav, Kovalenko, Vladimir & Bryksin, Timofey. 2021. The IntelliJ platform: a framework for building plugins and mining software data. Teoksessa: 2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW), s. 14–17.

Käss, Sebastian, Strahringer, Susanne & Westner, Markus. 2022. Drivers and inhibitors of low code development platform adoption. Teoksessa: 2022 IEEE 24th Conference on Business Informatics (CBI), s. 196–205.

Käss, Sebastian, Strahringer, Susanne & Westner, Markus. 2023. Practitioners' perceptions on the adoption of low code development platforms. IEEE access, 11, s. 29009–29034.

Luo, Yajing, Liang, Peng, Wang, Chong, Shahin, Mojtaba & Zhan, Jing. 2021. Characteristics and Challenges of Low-Code Development: The Practitioners' Perspective. Teoksessa: ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). 11.–15. lokakuuta 2021, Bari, Italia. New York: ACM, s. 1–11.

Martin, Robert C., 2021. Clean Craftsmanship: Disciplines, Standards, and Ethics. Addison-Wesley Professional.

Matthes, Eric. 2023. Python crash course: A hands-on, project-based introduction to programming. no starch press.

McKendrick, Joe. 2022. Low-code no-code market keeps growing, and that means shifts in technology roles. Verkkoaineisto. ZDNET. <https://www.zdnet.com/article/low-code-no-code-market-keeps-growing-portending-shifts- in-technology-roles/> 3.11.2022. Luettu 5.5.2025.

Mendix Platform. Verkkoaineisto. Mendix. <https://www.mendix.com/platform/> Luettu 2.5.2025.

Pereira, Rui, Couto, Marco, Ribeiro, Francisco, Rua, Rui, Cunha, Jácome, Fernandes, João Paulo & Saraiva, João. 2021. Ranking programming languages by energy efficiency. Science of Computer Programming, 205, s. 1–30.

Peruzzi, Joe. 2023. The Pros and Cons of Low-code in Cybersecurity Environments. Verkkoaineisto. Carahsoft. <https://www.carahsoft.com/blog/tychon-pros-and-cons-of-low-code-in-cybersecurity-blog-2023> 10.3.2023. Luettu 8.5.2025.

Royce, Winston W. 1970. Managing the development of large software systems: concepts and techniques. Teoksessa: IEEE WESCON, Elokuu 1970. Proceedings of the 9th International Conference on Software Engineering, s. 328–338.

Sahay, Apurvanand, Indamutsa, Arsene, Di Ruscio, Davide & Pierantonio, Alfonso. 2020. Supporting the understanding and comparison of low-code development platforms. Teoksessa: 2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), s. 171–178.

Sahinaslan, Ender, Sahinaslan, Onder & Sabancioglu, Mehmet. 2021. Low-code application platform in meeting increasing software demands quickly: SetXRM. AIP Conference Proceedings, 2334(1). AIP Publishing.

Schwaber, Ken, 1997. Scrum development process. Business object design and implementation: OOPSLA'95 workshop proceedings 16 October 1995, Austin, Texas, s. 117–134. Springer London.

Sommerville, Ian, 1998. Systems engineering for software engineers. Annals of Software Engineering, 6(1), s. 111–129.

Shridhar, Shreyas & Bose, Siddharth. 2021. Analysis of low code-no code development platforms in comparison with traditional development methodologies. International Journal for Research in Applied Science and Engineering Technology, 9(12), s. 508–513.

Stoica, Marian, Mircea, Marinela & Ghilic-Micu, Bogdan. 2013. Software Development: Agile vs. Traditional. Informatica Economică, 17(4), s. 64–76.

Sufi, Fahim. 2023. Algorithms in low-code-no-code for research applications: A practical review. Algorithms, 16(2), s. 108.

Schwartz, Karen D. 2021. App Development: Staying Secure Using Low-Code Platforms. Verkkoaineisto. ITPro Today. <https://www.itprotoday.com/it-

security/app-development-staying-secure-using-low-code-platforms> 12.11.2021. Luettu 8.5.2025.

Tadi, Sri Rama Chandra Charan Teja. 2021. Scalability and Performance Benchmarking of Low-Code Platforms vs. Traditional Development in Large-Scale Enterprise Applications. Journal of Scientific and Engineering Research, 8(8), s. 262–273.

The AI-powered low-code platform. Verkkoaineisto. OutSystems. <https://www.outsystems.com/low-code-platform/> Luettu 5.5.2025.

The Definitive Guide to Low-Code Development. Verkkoaineisto. Mendix. <https://www.mendix.com/low-code-guide/> Luettu 2.5.2025.

The IntelliJ Platform. 2025. Verkkoaineisto. JetBrains. <https://plugins.jetbrains.com/docs/intellij/intellij-platform.html> Päivitetty 29.4.2025. Luettu 1.5.2025.

The state of low-code/no-code. 2021. Verkkoaineisto. Creatio. <https://www.creatio.com/page/sites/landings/files/2021-05/Report-May.pdf> Luettu 6.5.2025.

Tissir, Najat, El Kafhali, Said & Aboutabit, Noureddine. 2020. Cybersecurity management in cloud computing: semantic literature review and conceptual framework proposal. Journal of Reliable Intelligent Environments, 7(2), s. 69–84.

Totterdale, Robert L. 2018. Case study: The utilization of low-code development technology to support research data collection. Issues in information systems, 19(2), s. 132–139.

SDLC - Overview. Verkkoaineisto. TutorialsPoint. <https://www.tutorialspoint.com/sdlc/sdlc_overview.htm> Luettu 19.4.2025.

Visual Studio Code documentation. Verkkoaineisto. Microsoft. <https://code.visualstudio.com/docs> Luettu 1.5.2025.

Waszkowski, Robert. 2019. Low-code platform for automating business processes in manufacturing. IFAC-PapersOnLine, 52(10), s. 376–381.

What is Power Apps? 2024. Verkkoaineisto. Microsoft. <https://learn.microsoft.com/en-gb/power-apps/powerapps-overview> 7.8.2024. Luettu 2.5.2025.