Koodintarkastuksen automatisointi generatiivisen tekoälyn avulla

Turun yliopisto
Tietotekniikan laitos
TkK-tutkielma
Tietotekniikka
Huhtikuu 2025
Tami Nikander

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

TURUN YLIOPISTO
Tietotekniikan laitos

Tami Nikander: Koodintarkastuksen automatisointi generatiivisen tekoälyn avulla

TkK-tutkielma, 29 s.
Tietotekniikka
Huhtikuu 2025

Koodintarkastus on aikaa vievä mutta olennainen osa nykyaikaista ohjelmistokehitystä. Sen avulla vähennetään tuotantokoodiin päätyviä ohjelmointivirheitä sekä varmistetaan, että projektissa noudatetaan sovittuja ohjelmointikäytänteitä. Koodintarkastuksen automatisointi generatiivisen tekoälyn avulla voisi säästää kehittäjien aikaa tehostaen koko ohjelmistokehitysprosessia. Tässä tutkielmassa selvitettiin yksinkertaisen kirjallisuuskatsauksen avulla, miten koodintarkastusta on tähän mennessä pyritty automatisoimaan Transformer-arkkitehtuuriin perustuvien laajojen kielimallien avulla. Tutkielmassa tarkasteltiin Transformer-arkkitehtuurin vuonna 2017 julkaisemisen jälkeen julkaistua kirjallisuutta koodintarkastuksen automatisointiin pyrkivistä työkaluista ja kehittäjien käyttämistä tekoälyapuvälineistä. Tutkielmassa havaittiin, että generatiivista tekoälyä voidaan koodintarkastuksessa käyttää erilaisiin pienempiin tehtäviin, kuten kommenttien generointiin tai koodikorjausten ehdottamiseen. Lisäksi todettiin, että kehittäjät kokevat nykyiset käytössä olevat tekoälytyökalut hyödyllisiksi, mutta merkittävänä rajoitteena nykyaikaisissa automaattisissa koodintarkastustyökaluissa on kuitenkin kielimalleille esitettävä hyvin rajallinen projektin lähdekoodiin liittyvä tiedon määrä, joka haittaa mallien tarkkuutta. Mahdolliseksi ratkaisuksi tähän ongelmaan ehdotettiin RAG-menetelmien käytön tutkimista automaattisessa koodintarkastuksessa.

Asiasanat: koodintarkastus, laajat kielimallit, tekoäly

UNIVERSITY OF TURKU
Department of Computing

Tami Nikander: Koodintarkastuksen automatisointi generatiivisen tekoälyn avulla

Bachelor's Thesis, 29 p.
Information Technology
April 2025

Code review is a time-consuming but essential part of modern software development. It's used to reduce the number of bugs in production code and to enforce a project's code style and architecture guidelines. Code review automation with generative AI could save developer time, streamlining the software development process. Using a simple literature review, this paper explored currently proposed and used methods of code review automation based on large language models using the Transformer architecture. This paper examined literature published after the publication of the Transformer architecture in 2017 on code review automation tools and AI tools used by developers to assist in code review. In the paper it was observed that generative AI can be used for various smaller tasks in code review, like comment generation or suggesting code fixes. Moreover, it was noted that developers find existing AI-based tools useful, but a significant limitation of current methods is the limited information about a project's source code provided to language models, hindering model accuracy. Studying the possibility of utilizing RAG in automatic code review was suggested as a possible solution.

Keywords: code review, large language models, artificial intelligence

Sisällys

1 Johdanto **1**

2 Koodintarkastus **4**

2.1 Koodintarkastuksen alkuperä ja merkitys 4

2.2 Pull request -koodintarkastusprosessi 5

3 Koneoppiminen ja kielimallit **7**

3.1 Kone- ja syväoppiminen 7

3.2 Luonnollisen kielen käsittely 9

3.3 Kielimallit 10

3.3.1 Hienosäätö ja kehotteen suunnittelu 12

3.3.2 Hienosäätömenetelmät 13

4 Koodintarkastuksen automatisointi **15**

4.1 Koodintarkastuksessa automatisoitavat tehtävät 15

4.2 Koodintarkastuksen automatisointi käytännössä 21

4.3 Pohdinta 24

5 Yhteenveto **27**

Lähdeluettelo **30**

Kuvat

1.1 Kaavio hakutulosten käsittelystä ja valinnasta 3

2.1 Kaavio tyypillisestä pull request -koodintarkastusprosessista. 6

1 Johdanto

Ohjelmistokehitysprosessissa koodintarkastus on tärkeä vaihe, joka muun muassa varmistaa koodin huollettavuuden ja tiedon jakamisen tiimin kesken sekä tarjoaa mahdollisuuden ohjelmointivirheiden havaitsemiseen ja korjaamiseen. Manuaalinen koodintarkastus on kuitenkin aikaa vievä prosessi, joka vaatii kehittäjiltä useita tunteja viikossa. Esimerkiksi Microsoftin projekteissa kehittäjät käyttävät keskimäärin 4,7 tuntia ja avoimen lähdekoodin projekteissa noin 6,4 tuntia viikossa muiden kehittäjien koodin tarkastamiseen. [1] Koodintarkastuksessa kuluu merkittävästi aikaa pelkästään koodin ymmärtämiseen, minkä takia automaattisia menetelmiä on tutkittu koodintarkastuksen tehostamiseksi [2]. Tässä tutkielmassa selvitetään nykyisten automaattisten koodintarkastusmenetelmien tehokkuutta ja tarkastellaan, miten generatiivista tekoälyä voidaan hyödyntää koodintarkastuksessa. Tutkielmassa vastataan seuraaviin tutkimuskysymyksiin:

1. Kuinka generatiivista tekoälyä voidaan hyödyntää koodintarkastuksen automatisoinnissa?

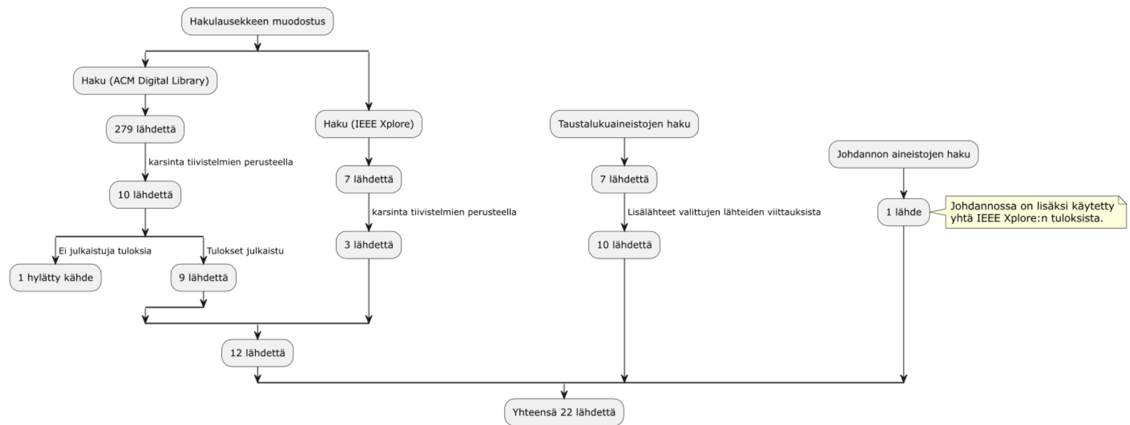
2. Kuinka tehokkaita ehdotetut automatisointimenetelmät ovat?

3. Mitkä ovat generatiivisen tekoälyn käytön haasteet koodintarkastuksessa?

Tutkielmassa käytetty tutkimusmenetelmä perustuu kirjallisuuskatsaukseen. Taustalukujen aineisto on kerätty Turun yliopiston Volter-tietokannasta hakemalla koodintarkastuksen kehitykseen liittyviä artikkeleita hakulausekkeella "code review evo-

lution" ja lisäksi hakemalla aiheen ja aineistotyypin perusteella syväoppimiseen ja kielimalleihin liittyviä kirjoja. Tutkimuskysymyksiin liittyvä aineisto on haettu ACM Digital Library ja IEEE Xplore -tietokannoista useassa vaiheessa muodostetulla hakulausekkeella. Hakutuloksia on rajattu Transformer-arkkitehtuurin esittelevän artikkelin julkaisuvuoden perusteella siten, että aikaisintaan vuonna 2017 julkaistut artikkelit otetaan huomioon. Muodostetulla hakulausekkeella ja rajauksella saaduista hakutuloksista on valittu tutkimuskysymyksiin liittyvät artikkelit otsikkojen ja tiivistelmien perusteella. Lisäaineistoa on kerätty myös valittujen artikkelien viitteistä.

Suurin osa tutkielmaan valitusta kirjallisuudesta on haettu samalla hakulausekkeella: `("code review" OR "automated code review") AND ("language model" OR "generative AI" OR "GPT-4" OR "generative pretrained transformer") AND (automation OR "AI-assisted code review")`. Hakulauseke on muodostettu keksimällä ensin aiheeseen sopiva alustava hakulauseke, joka annettiin Open AI:n GPT-4o-mallille ChatGPT-palvelun kautta parannettavaksi tutkimuskysymysten avulla. ChatGPT:llä paranneltua hakulauseketta korjattiin tämän jälkeen manuaalisesti poistamalla siitä tarpeettomia termejä ja yhdistelemällä samankaltaisia termejä kaarisulkeilla. Lopullisella hakulausekkeella saatiin ACM Digital Library ja IEEE Xplore -tietokannoista yhteensä 286 tulosta, joista suurin osa karsittiin pois tiivistelmien perusteella. Jäljelle jäi 13 artikkelia, joista vielä yksi jätettiin pois, koska kyseisen tutkimuksen tuloksia ei ollut julkaistu. Lopullisessa tutkielmassa lähteitä on yhteensä 22, joista 10 käytettiin taustalukuihin ja 12 vastaamaan tutkimuskysymyksiin.



Kuva 1.1: Kaavio hakutulosten käsittelystä ja valinnasta

Tutkielman toinen ja kolmas luku esittelevät tutkielmaan liittyviä taustatietoja. Toinen luku esittelee lyhyesti koodintarkastuksen historiaa ja kuvailee nykyaikaisen koodintarkastusprosessin kulkua, kun taas kolmas luku käsittelee tekoälyä, syväoppimista ja kielimallien toimintaperiaatteita. Neljännessä luvussa käsitellään koodintarkastuksen automatisointia generatiivisen tekoälyn avulla ja esitellään aiheeseen liittyviä tutkimuksia. Viidennessä luvussa vastataan tutkimuskysymyksiin edellisten lukujen perusteella ja esitetään yhteenveto tutkielman tuloksista. Lopuksi pohditaan mahdollisia aiheita jatkotutkimukselle.

2 Koodintarkastus

Tässä luvussa käsitellään lyhyesti koodintarkastuksen (engl. code review) merkitystä ohjelmistokehityksessä sekä esitellään Git-versionhallintajärjestelmän kanssa käytettävälle pull-based development -kehitysmallille tyypillinen pull request -koodintarkastusprosessi.

2.1 Koodintarkastuksen alkuperä ja merkitys

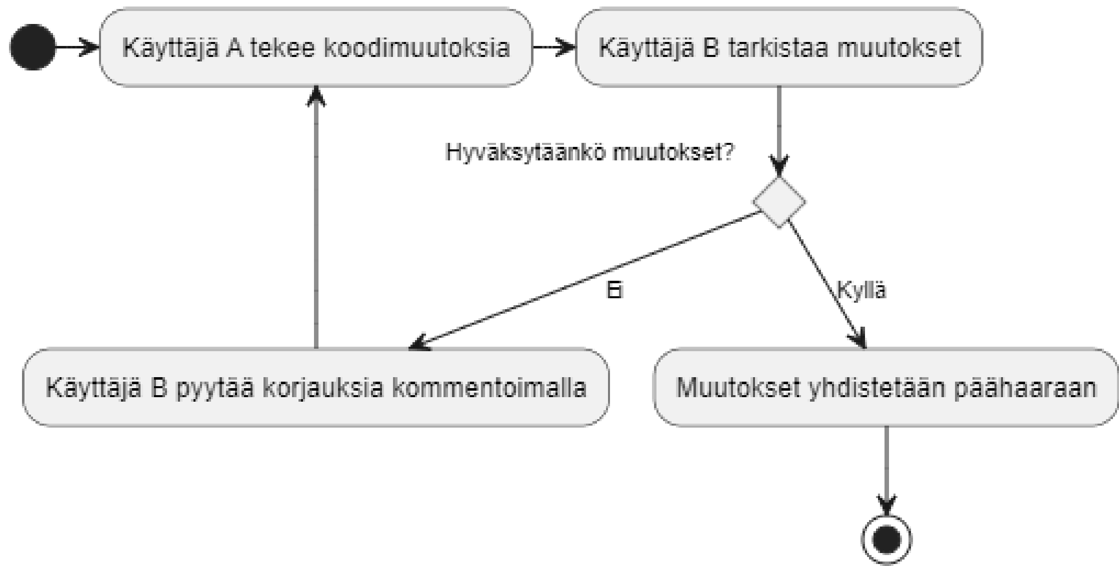
Koodintarkastus on ohjelmistokehitykseen liittyvä prosessi, jossa ohjelmakoodia tarkastetaan mahdollisten ohjelmointi-, tyyli- tai suunnitteluvirheiden löytämiseksi. Koodintarkastus on tärkeä osa ohjelmakoodin laadun varmistamista, ja sen avulla voidaan havaita virheitä ennen niiden päätymistä tuotantoympäristöön.

Vaikka koodintarkastuksen ydinajatus on pysynyt samana, sen sisältö on muuttunut merkittävästi ajan myötä. Ensimmäisenä muodollisena koodintarkastusmenetelmänä voidaan pitää Faganin tarkastusta (engl. Fagan inspection), jonka Michael Fagan esitteli vuonna 1976. Faganin tarkastuksessa ohjelmakoodi luettiin läpi ennen sen ajamista, ja jokainen ohjelman sisältämä haara tarkastettiin. Tämä menetelmä oli erittäin hyödyllinen aikana, jolloin tietokoneiden käyttöaika ja saatavuus olivat rajallisia ja korjausten testaaminen saattoi olla mahdollista vasta useiden viikkojen odottamisen jälkeen. [3] Faganin tarkastusten onkin todettu olevan tehokas tapa vähentää ohjelmistovirheitä ja jakaa tietoa tehdyistä muutoksista kehittäjien kesken [4]. Nykyaikaisille ohjelmistoille syvälliset tarkastukset ovat kuitenkin liian työläitä

kasvavan lähdekoodin määrän vuoksi, minkä takia nykyaikaisten koodintarkastusmenetelmien on täytynyt kehittyä nopeammiksi ja tehokkaammiksi.

2.2 Pull request -koodintarkastusprosessi

Nykyisin erityisesti avoimen lähdekoodin projekteissa suositaan pull-based development -kehitysmallia, jossa muutokset tehdään ensin muusta koodista erillisessä haarassa tai tietovarastossa ja yhdistetään päähaaraan pull request -muutospyynnöllä. Muutospyynnössä esitetään kootut lähdekoodiin tehdyt muutokset, jotka halutaan yhdistää päähaaraan. Myös muita mahdollisia tietoja, kuten muutosten tarkoitus, voidaan sisällyttää pyynnön kuvaukseen. Projektin ylläpitäjät tarkastavat pyynnössä esitetyt muutokset ja tarvittaessa kommentoivat niitä tai pyytävät niiden tekijää tekemään lisämuutoksia ennen koodin yhdistämistä päähaaraan. [5] Muutoksia voidaan pyytää iteratiivisesti useita kertoja, kunnes pyyntö lopulta joko hyväksytään tai hylätään. Ohjelmakoodin muutokset täytyy tällöin tarkastaa mahdollisesti useita kertoja yhden muutospyynnön kohdalla. [6]



Kuva 2.1: Kaavio tyypillisestä pull request -koodintarkastusprosessista.

3 Koneoppiminen ja kielimallit

Tässä luvussa esitellään kone- ja syväoppimisen perusteita sekä laajojen kielimallien (engl. Large Language Model, LLM) ja niihin liittyvien tekniikoiden toimintaperiaatteita.

3.1 Kone- ja syväoppiminen

Koneoppimista käytetään tyypillisesti tehtäviin, joihin on epäkäytännöllistä tai mahdotonta suunnitella kiinteää, valmiiksi kirjoitettua ohjelmaa. Goodfellow'n ym. kirjassa "Deep Learning" (2016) joitakin esitettyjä tehtävätyyppejä, joissa hyödynnetään usein koneoppimista, ovat esimerkiksi luokittelu sekä synteesi ja näytteenotto. Luokittelua käytetään, kun halutaan määrittää, mihin luokkaan jokin syöte kuuluu. Esimerkiksi esineiden tunnistaminen kuvista on yleinen luokittelutehtävä. Synteesiä ja näytteenottoa taas käytetään, jos halutaan luoda uutta koulutusaineiston kaltaista dataa. [7] Tällaisia tehtäviä ovat esimerkiksi kuvien luominen tekstin perusteella tai tekstin generointi.

Koneoppimisalgoritmien tehokkuuteen vaikuttaa merkittävästi datan esitystapa, eli algoritmille syötettävät tiedot ja niiden muoto. Datan esitystavan valinta voi kuitenkin olla hyvinkin vaikeaa erityisesti monimutkaisissa tehtävissä. Ratkaisuna tähän ongelmaan on esitystavan oppiminen (engl. representation learning), jossa koneoppimisalgoritmi löytää koulutuksen aikana tehtävän kannalta datan tärkeimmät piirteet. Algoritmi valitsee tällöin itse mahdollisimman tehokkaan esitystavan

datalle, jolloin ihmisen ei tarvitse määritellä sitä etukäteen. [7]

Esitystavan oppimisen hyödyntäminen koneoppimisalgoritmeissa vähentää algoritmin toteutukseen vaadittavaa manuaalista työtä, mutta se ei kuitenkaan välttämättä kykene esittämään raa'an datan perusteella monimutkaisia piirteitä, kuten esimerkiksi valaistusta tai kuvakulmia kuvantunnistuksessa. Esimerkiksi huonossa valaistuksessa otetussa kuvassa punainen esine voi näyttää lähes mustalta. Goodfellow ym. (2016) nimittävät näitä monimutkaisempia piirteitä variaatiotekijöiksi (engl. factors of variation). Erityisesti monimutkaisten variaatiotekijöiden – kuten kuvakulman yksittäisten pikseliarvojen perusteella – erottaminen raa'asta datasta on usein vaikeaa tai jopa mahdotonta, mikä tekee yksinkertaisten koneoppimisalgoritmien käytöstä haastavaa. [7]

Goodfellow ym. esittävät variaatiotekijöiden aiheuttamaan ongelmaan ratkaisuksi syväoppimista, jossa datan esitystavat perustuvat muihin saman datan yksinkertaisempiin esitystapoihin. Syväoppimisalgoritmien perusyksikköjä ovat monikerroksiset perseptronit (engl. multilayer perceptron, MLP), jotka muodostavat useita kerroksia, joissa on vaihteleva määrä perseptroneja. Monikerroksisten perseptronien arvot lasketaan edellisen kerroksen tuottamien arvojen perusteella, jolloin jokainen kerros saa syötteensä sitä edeltävän kerroksen perseptroneista. Optimaalinen esitystapa saadaan valitsemalla jokaisessa tasossa mallin seuraavaan kerrokseen vaikuttavat piirteet. Monikerroksinen algoritmi voi näin oppia tunnistamaan myös monimutkaisia yksinkertaisemmista piirteistä koostuvia piirteitä, kuten tiettyjen muotojen ja värien yhdistelmiä kuvissa. [7] Tämän ansiosta syväoppimisen avulla voidaan vähentää variaatiotekijöiden aiheuttamia ongelmia ja parantaa koneoppimisalgoritmien suorituskykyä vaikeammissa tehtävissä, joissa datan yksittäiset piirteet voivat muodostaa monimutkaisempia kokonaisuuksia.

3.2 Luonnollisen kielen käsittely

Luonnollisen kielen käsittely (engl. Natural Language Processing, NLP) on tekoälyn osa-alue, jonka tarkoituksena on tulkita ja tuottaa ihmisille merkityksellistä tekstiä. NLP-algoritmit muodostavat perustan laajoille kielimalleille, joissa käytetään useita samoja tekniikoita, kuin yksinkertaisemmissakin NLP-algoritmeissa. [8]

Amaratunga kertoo kirjassaan "Understanding Large Language Models: Learning Their Underlying Concepts and Technologies" (2023) NLP-algoritmien aloittavan tavallisesti tekstin käsittelyn jakamalla sen ensin pienempiin yksiköihin eli tokeneihin. Tokenisointi (engl. tokenization) voi tarkoittaa esimerkiksi lauseiden jakamista sanoihin tai sanojen jakamista merkkeihin tai sanan osiin. Tokenisointimenetelmiä on useita, ja niiden valinta riippuu käytettävästä aineistosta ja tehtävästä. [8]

NLP-algoritmin koulutus vaatii koulutusaineistoa, joka voi sisältää esimerkiksi tekstiä, äänitteitä, kuvia tai useiden aineistotyyppien yhdistelmiä. Koulutusaineisto muodostaa mallin korpuksen, johon perustuu mallin kyky ymmärtää ja tuottaa luonnollista kieltä. Korpuksen sisältämät tokenit muodostavat sanaston, joka on kokoelma kaikista tokeneista, joita malli voi käsitellä. [8]

Luonnollisen kielen käsittelyssä mallin sanastoon kuuluvat tokenit esitetään yleensä vektoreina, joita kutsutaan sanojen upotuksiksi (engl. word embeddings). Sanojen upotus voidaan toteuttaa usealla eri tavalla, mutta yleisimpiä upotusmenetelmiä ovat muun muassa bag-of-words (BoW) ja word2vec -upotukset. Bag-of-words-upotus on yksinkertaisesti toteutettavissa oleva menetelmä, jossa teksti muutetaan vektoriksi, jonka pituus on yhtä suuri kuin tokenien määrä mallin sanastossa. Vektorin jokainen alkio kertoo, kuinka monta kertaa alkion indeksiä vastaava tokeni esiintyy tekstissä. Tämä menetelmä ei kuitenkaan säilytä sanojen tai tokenien järjestystä tekstissä eikä ota huomioon sanojen välisiä merkityssuhteita. Word2vec-upotus taas esittää tokenit vektoreina jatkuvassa moniulotteisessa avaruudessa, jossa sa-

mankaltaiset sanat ovat lähellä toisiaan ja niiden sijainti suhteessa muihin vektoreihin vastaa tokenin tai sanan suhdetta muihin tokeneihin tai sanoihin. Tämä menetelmä vaatii vähemmän muistia kuin BoW-upotus erityisesti suurilla sanastoilla, sillä word2vec-upotuksessa vektorien pituus ei riipu suoraan sanaston koosta, kun taas suurilla sanastoilla BoW-upotuksen vektorit kasvavat hyvin pitkiksi. Word2vec-upotuksen vektorit täytyy kuitenkin oppia koulutusaineiston perusteella, joten upotusmallin koulutus voi vaatia paljon enemmän aikaa ja laskentatehoa kuin BoW-upotusmenetelmän käyttöönotto. Word2vec-upotusmenetelmä on toteutuksen suhteellisesta monimutkaisuudesta huolimatta ominaisuuksiensa vuoksi hyödyllisempi käyttötarkoituksiin, joissa sanojen väliset suhteet ovat tärkeitä. [8]

3.3 Kielimallit

Kielimallit ovat malleja, joiden tarkoitus on ennustaa seuraava sana tai tokeni mallille annettavan syötteen perusteella. Amaratungan mukaan kielimallit voidaan jakaa toteutustapansa perusteella kahteen luokkaan: N-grammimalleihin ja neuroverkkoihin perustuviin kielimalleihin. N-grammimallit ovat yksinkertaisempia ja ennustavat seuraavan sanan todennäköisyyden $N-1$:n edellisen sanan perusteella. N-grammimallit tarvitsevat syötteekseen aina $N-1$ sanaa, minkä takia mallit eivät sovellu hyvin vaihtelevan pituisten syötteiden käsittelyyn. [8]

Eri pituisten syötteiden käsittelyyn soveltuvat paremmin neuroverkkoihin perustuvat kielimallit, jotka voivat käsitellä vaihtelevan pituisia syötteitä. Esimerkiksi takaisinkytketyt neuroverkot (engl. Recurrent Neural Network, RNN) soveltuvat NLP-tehtäviin, koska ne voivat säilyttää sisäisen tilan, joka voi sisältää tietoa edellisten sanojen asiayhteydestä. RNN-mallien koulutuksessa käytettävä vastavirta-algoritmi aiheuttaa kuitenkin katoavan gradientin ongelman, jonka takia mallin ensimmäisten tasojen väliset painot voivat kehittyä erittäin hitaasti myöhempien tasojen välisiin painoihin verrattuna. Tämän takia NLP-tehtävissä käytetään usein RNN-

arkkitehtuuriin perustuvaa pitkäkestoista lyhytkestomuisti -arkkitehtuuria (engl. Long Short-Term Memory, LSTM), joka ratkaisee katoavan gradientin ongelman ja säilyttää edelleen RNN-mallien kyvyn säilyttää tietoa aiempien sanojen välisistä suhteista. RNN-pohjaiset kielimallit ovat kuitenkin melko hitaita, sillä niiden suorittaminen vaatii paljon peräkkäistä laskentaa. [8]

Peräkkäistä laskentaa vaativien mallien vaihtoehdoksi on kehitetty Transformer-arkkitehtuuri, joka mahdollistaa tehokkaamman rinnakkaisen laskennan [8]. Vaswanin ym. esittelemä Transformer-arkkitehtuuri käyttää huomiomekanismia, jonka avulla malli voi asettaa eri painoja syötteen eri osille, jotka syötetään edelleen mallin seuraaville kerroksille. Huomioon perustuva Transformer-malli joutuu suorittamaan vain $O(1)$ peräkkäistä operaatiota n-pituisella syötteellä, kun taas RNN-arkkitehtuuriin perustuva neuroverkko vaatii $O(n)$ peräkkäistä operaatiota. [9] Transformer-mallissa tarvittavien peräkkäisten operaatioiden määrä ei siis kasva merkittävästi syötteen pituuden kasvaessa, mikä mahdollistaa erityisesti pitkien syötteiden käsittelyn tehokkaasti.

Rinnakkaisen laskennan mahdollistamisen ansiosta Transformer-arkkitehtuuria käytetään useimmiten laajoissa kielimalleissa, eli kielimalleissa, joilla on hyvin suuri määrä kouluttamisen aikana säädettäviä parametreja ja suuri määrä koulutusaineistoa. Laajojen kielimallien koko mitataan yleensä parametrien määrällä, joka vaikuttaa mallin oppimiskykyyn ja mallin muistiin tallennettavan tiedon määrään. Esimerkiksi OpenAI:n GPT-3–malli käyttää 175:a miljardia parametria. Suuri parametrien määrä vaatii kuitenkin myös paljon laskentatehoa ja muistia mallin kouluttamiseen ja käyttämiseen. [8] Tämän takia laajojen kielimallien suorittaminen kuluttajille tarkoitetuilla laitteilla ei usein ole käytännöllistä.

Transformer-mallit voidaan jakaa niiden toimintaperiaatteen perusteella kolmeen eri luokkaan. Autoregressiiviset mallit, kuten OpenAI:n GPT-mallit, ennustavat yhden seuraavan tokenin syötteen perusteella. Pidempien tekstien generoimiseksi au-

toregressiiviselle mallille annetaan syötteeksi mallin tuottama teksti ja alkuperäinen teksti useita kertoja. Autoenkoodaavat mallit – kuten BERT (Bidirectional Encoder Representations from Transformers) – käsittelevät koko syötteen kerralla ja ennustavat syötteestä peitetyt sanat. Seq2seq (Sequence-to-Sequence) -mallit – kuten OpenNMT (Open Neural Machine Translation) – soveltuvat tehtäviin, joissa syötteen ja tulosteen pituus voivat poiketa toisistaan merkittävästi. Kielimalleja, jotka käyttävät näiden eri luokkien yhdistelmiä kutsutaan hybridimalleiksi. [8]

3.3.1 Hienosäätö ja kehotteen suunnittelu

Monia kielimalleja – kuten Open AI:n GPT-3-mallia – ei yleensä ole suunniteltu mihinkään tiettyyn tehtävään. Tämän takia kielimallin hyödyntäminen tarkasti määriteltyyn tehtävään vaatii yleensä mallin kouluttamisen jälkeen lisätoimia, joilla malli saadaan tuottamaan haluttuja vastauksia. Tavallisimmat menetelmät laajojen kielimallien käyttäytymisen muokkaamisen ovat kehotteen suunnittelu (engl. prompt engineering) ja hienosäätö (engl. fine-tuning). [8]

Kehotteen suunnittelussa pyritään saavuttamaan laajan kielimallin haluttu käyttäytyminen tai vastaustapa antamalla mallille alkukehote, jossa mallille annetaan tehtävään liittyvät toimintaohjeet. Kehotteen suunnittelun avulla mallin käyttäytymistä voidaan muokata hyvin nopeasti, ja yhtä valmiiksi koulutettua mallia voidaan käyttää monipuolisiin tehtäviin. Kielimallien vastauksissa on kuitenkin aina niille ominaista satunnaisuutta, minkä takia malli voi joskus tuottaa odottamattomia tuloksia myös tarkasti suunnitellulla kehotteella. Kehotteen suunnittelussa on myös ylisovittamisen riski; kielimalli voi oppia toistamaan kehotteessa annettuja esimerkkejä, johtaen liian toistuviin vastauksiin. Kehotteen optimointi riittävän tarkkojen vastauksien saavuttamiseksi ja ylisovittamisen välttämiseksi voi vaatia paljon aikaa ja useita kehotteen iteraatioita erityisesti monimutkaisissa tehtävissä. [8]

Kehotteen suunnittelun sijaan tai sen lisäksi laaja kielimalli voidaan paremmin

sovittaa sille annettuun tehtävään hienosäädön avulla. Kielimallin koulutuksen aikana malli saa korpuksesta yleistietoa, joka ei välttämättä ole riittävää hyvin erikoistuneisiin tehtäviin. Hienosäätö sisältää mallin lisäkouluttamisen tehtävään liittyvällä datalla. Hienosäädetty malli voi edelleen hyödyntää esikouluttamisen aikana opittua tietoa, kuten kielirakenteita, sanastoa ja yleistietoa, minkä takia hienosäätö voidaan toteuttaa esikouluttamiseen verrattuna nopeasti ja pienellä määrällä koulutusaineistoa. Onnistuneen hienosäädön tuloksena on hyvin erikoistunut malli, joka pystyy suorittamaan tehtävän aiempaa paremmin. Myös hienosäädön riskinä on kuitenkin ylisovittaminen, jonka seurauksena malli voi keskittyä liikaa suppeaan hienosäätöaineistoon ja "unohtaa" joitakin esikoulutuksen aikana opittuja tietoja. [8]

3.3.2 Hienosäätömenetelmät

Laajojen kielimallien hienosäätö voidaan toteuttaa usealla eri menetelmällä, joiden soveltuvuus riippuu käyttötarkoituksesta. Hienosäädöstä puhuttaessa ilman tarkempaa määritelmää tarkoitetaan mallin kaikkien parametrien muuttamista hienosäädön aikana. Tämä kuitenkin vaatii paljon tilaa, sillä jokaista hienosäädettyä tehtävää varten tarvitaan erillinen kopio koko mallista. [10] Ratkaisuksi tähän ongelmaan on kehitetty kevyempiä hienosäätömenetelmiä, jotka eivät vaadi esikoulutetun mallin alkuperäisten parametrien muuttamista. [10] [11]

Li ja Liang esittelivät vuonna 2021 etuliitehienosäädön (engl. prefix tuning), jossa opitaan etuliitevektori, joka liitetään mallin jokaiseen kerrokseen kaikilla syötteillä. Autoregressiivisissä malleissa (esim. GPT) etuliitevektori sijoitetaan syötteen alkuun ennen varsinaista syötettä, kun taas autoenkoodavissa malleissa (esim. BERT) syötteelle ja tulosteelle opitaan erilliset etuliitevektorit. Etuliitevektorit voivat koostua mistä tahansa halutun tulosteen tuottavista tokeneista, mutta niiden ei tarvitse olla sanoja tai muutenkaan kuulua mallin korpukseen. [10]

Hu ym. esittelivät vuonna 2021 Low-Rank Adaptation (LoRA) -menetelmän, jossa hienosäädön aikana opitaan uudet mallin esikoulutettuihin painoihin lisättävät painot. LoRA-hienosäädössä mallille, jonka esikoulutettu painomatriisi on $W_0 \in \mathbb{R}^{d \times k}$, opitaan uudet painot $A \in \mathbb{R}^{r \times k}$ ja $B \in \mathbb{R}^{d \times r}$. Tekijä r (rank) valitaan siten, että $r \ll \min(d, k)$. Tällöin hienosäädetyn mallin painot saadaan laskemalla $W = W_0 + \Delta W = W_0 + BA$, missä W_0 on mallin alkuperäiset painot sisältävä matriisi ja A ja B ovat hienosäädetyt painomatriisit. Koska hienosäädettävien parametrien määrä on paljon pienempi, kuin täydellisesti hienosäädetyissä malleissa, hienosäätö on laskennallisesti tehokkaampaa ja malli vaatii vähemmän muistia. LoRA-menetelmällä hienosäädetyt painot voidaan myös säilyttää erillään mallin alkuperäisistä painoista, mikä mahdollistaa yhdistämisen muihin hienosäätömenetelmiin. [11]

4 Koodintarkastuksen automatisointi

Tässä luvussa käsitellään koodintarkastuksen automatisointia generatiivisen tekoälyn avulla ja vertaillaan aiemmissa tutkimuksissa tuotettuja menetelmiä keskenään. Tutkimuskysymyksiin vastataan seuraavassa luvussa tässä luvussa käsiteltyjen tutkimusten perusteella.

4.1 Koodintarkastuksessa automatisoitavat tehtävät

Yhden ensimmäisistä tutkimuksista, joka käsitteli koodintarkastuksen automatisointia Transformer-arkkitehtuuriin perustuvalla laajalla kielimallilla toteuttivat Tufano ym. vuonna 2021. Tufano ym. käyttivät tutkimuksessaan NMT-mallia OpenNmt-tf-Python-kirjaston avulla. Malli esikoulutettiin noin 17 tuhannella ryhmällä $\langle C_s, R_{nl} \rangle \to C_r$, joissa C_s on kehittäjän kirjoittama koodikatkelma ennen sen tarkastamista, R_{nl} koodiin liittyvä muutosehdotus ja C_r muutosehdotuksen perusteella korjattu koodikatkelma. Tutkimuksen tarkoituksena oli kehittää malli, joka voi toteuttaa koodintarkastuksen aikana pyydetyt korjaukset automaattisesti suorittamalla koulutusvaihetta vastaavan muunnoksen $\langle C_s, R_{nl} \rangle \to C_r$ hyödyntäen kahta enkooderia, sekä ehdottaa koodimuutoksia ennen koodintarkastusta jättäen luonnollisella kielellä

kirjoitetun palautteen pois mallin syötteestä, jolloin suoritetaan muunnos $C_s \rightarrow C_r$ vain yhtä enkooderia käyttäen. Yhden enkooderin malli tuotti parhaimmillaan 15,76 % täydellisiä vastauksia ja saavutti 0.9145:en BLEU-4-mediaanin. Kahden enkooderin malli taas tuotti 30,72 % täydellisiä vastauksia ja saavutti 0.9543:en BLEU-4-mediaanin. On kuitenkin huomioitava, että Tufano ym. arvioivat kahden enkooderin mallia lähinnä tapauksissa, joissa yhden enkooderin malli ei kyennyt tuottamaan tyydyttävää ratkaisua. Yhden enkooderin mallin onnistuneista ratkaisuista 67,93 % sisälsivät ohjelman toiminnallisuuden muutoksia, kun taas kahden enkooderin mallin ratkaisuista toiminnallisuutta muutettiin 74,19 prosentissa kaikista onnistuneista ratkaisuista. Tufano ym. päättelivät, että kahden enkooderin mallille annettavat kommentit auttavat hieman muutosehdotusten tuottamisessa, mutta eivät kuitenkaan merkittävästi auta mallia keksimään uudentyyppisiä muutoksia. [12] Yhden enkooderin malli, joka ei saanut kommentteja, suoriutui siis melko hyvin monipuolisten muutosten tuottamisesta, mutta kahden enkooderin malli oli tarpeen monimutkaisempien muutosten tapauksissa, joissa ihmisen täytyi osoittaa muutoksen tarve.

Li ym. julkaisivat vuonna 2022 tutkimuksen, jossa he kehittivät T5:een (Text-to-Text Transfer Transformer) perustuvan CodeReviewer-mallin [13]. Mallilla on yhteensä 12 enkooderi- ja dekooderikerrosta ja 223 miljoonaa parametria. CodeReviewer koulutettiin GitHubista kerätyillä pull request -muutospyynnöillä[1] ja Gerrit Code Review -alustalta[2] kerätyillä kommenteilla. Kuten Tufano ym., Li ym. pyrkivät myös automatisoimaan muutosten toteuttamisen, mutta tämän lisäksi korjausten tarpeellisuuden arvioinnin yksinkertaisena binäärisenä luokitteluna sekä muutoksia ehdottavien kommenttien generoinnin koodimuutosten perusteella. Näiden kolmen vaiheen avulla pyrittiin automatisoimaan koodintarkastusprosessi kokonaisuudessaan: Jos malli päätti, että muutoksia tarvitaan ennen hyväksymistä, malli generoi tarvittavia muutoksia kuvaavat kommentit. Tämän jälkeen malli toteutti

[1] `https://docs.github.com/en/pull-requests`

[2] `https://www.gerritcodereview.com/`

näissä kommenteissa pyydetyt muutokset. CodeReviewerin F1-tulos oli 72,53 ja oikeellisuus 73,89, jotka olivat merkittävästi parempia, kuin tutkimuksessa vertaillut CodeT5-, T5- ja Transformer-mallien tulokset. Näistä seuraavaksi parhaiten suoriutui CodeT5, jonka F1-tulos oli 64,16 ja oikeellisuus 67,07. [13] Vaikka sekä CodeT5 että CodeReviewer perustuvat 12-kerroksiseen arkkitehtuuriin, CodeReviewer saavutti huomattavasti korkeamman tuloksen. CodeT5-mallia ei kuitenkaan koulutettu CodeReviewerin tapaan koodintarkastukseen [13], mikä todennäköisesti selittää tulosten eron. Tulos osoittaa oikein valitun koulutusdatan parantavan merkittävästi kielimallin suorituskykyä koodintarkastuksessa.

Vuonna 2023 Lu ym. esittelivät LLaMA-Reviewer-mallin, joka perustuu Meta AI:n kehittämään LLaMA-kielimalliin[3] [14]. Mallilla pyrittiin automatisoimaan koodintarkastusprosessi Lin ym. [13] käyttämällä kolmivaiheisella menetelmällä, mutta toisin kuin CodeReviewer, LLaMA-Revieweriä ei esikoulutettu koodintarkastukseen. Mallin pohjana toimiva 32-kerroksinen ja 6,7 miljardia parametria sisältävä LLaMA on esikoulutettu laajalla luonnollisen kielen korpuksella, joten tarkempi koulutus LLaMA:n erikoistamiseksi LLaMA-Reviewer-malliksi toteutettiin parametritehokkailla hienosäätömenetelmillä (engl. Parameter-Efficient Fine-Tuning, PEFT). Tutkimuksessa käytettiin kahta eri PEFT-menetelmää: LoRA ja etuliitehienosäätö. Lu ym. havaitsivat, että LoRA-hienosäädöllä saavutettiin etuliitehienosäätöä paremmat tulokset, ja LLaMA-Reviewer suoriutui hyvin erityisesti kommenttien generoinnissa ehdotettujen muutosten perusteella. Muutosten tarpeellisuuden arvioinnissa taas LLaMA-Reviewerin tulokset olivat vaihtelevia, ja malli saavutti matalimman tarkkuuden verrattuna muihin vertailtuihin malleihin. [14] Tutkimuksessa on kuitenkin huomioitava, että toisin kuin esimerkiksi vertailuun käytetty CodeReviewer, LLaMA-Reviewerä ei ole esikoulutettu koodintarkastukseen, mikä saattaa selittää tulosten vaihtelun. LLaMA:n esikoulutuksen suuremman luonnollista kieltä sisäl-

[3] `https://www.llama.com/`

tävän korpuksen ansiosta LLaMA-Reviewer kuitenkin suoriutuu hyvin luonnollista kieltä vaativista tehtävistä, kuten kommenttien generoinnista. [14]

Tufano ym. arvioivat vuonna 2024 julkaistussa tutkimuksessaan Lin ym. kehittämän CodeReviewer-mallin sekä kahden muun kielimallin (T5CR ja CommentFinder) suorituskykyä koodintarkastuksessa [15]. Tufano ym. totesivat vertailluista malleista T5CR:n olevan tehokkain koodityyliin liittyvissä muutoksista, mutta ei niinkään koodin toiminnallisuuteen liittyvissä muutoksissa. Tämän arvioidaan johtuvan malleille annetun tiedon määrästä ja esitystavasta: T5CR saa syötteenään vain muutetun koodikatkelman, kuten yhden metodin, kun taas CodeReviewer saa kaikkia muutospyynnössä tehtyjä koodimuutoksia kuvailevan diff-tiedoston, mikä auttaa erityisesti useissa tiedostoissa tehtyjen muutosten vuorovaikutusten tunnistamisessa. Tässäkin menetelmässä on kuitenkin ongelmana, että kielimalli ei saa tietoa muutosten ulkopuolisista tiedostoista, kuten käytettävissä olevista muuttumattomista luokista. Tutkimuksessa vertailtiin koodintarkastukseen suunniteltuja malleja myös ChatGPT:hen, joka osoittautui hieman paremmaksi pyydettyjen muutosten toteuttamisessa kommenttien perusteella. ChatGPT oli kuitenkin huonompi muutoksia ehdottavien kommenttien generoinnissa. [15] Tutkimuksessa ei kuitenkaan mainittu käytettyä GPT:n versiota tai ChatGPT:lle annettua kehotetta, joten ChatGPT:n todellista suorituskykyä on tämän tutkimuksen perusteella vaikea arvioida luotettavasti.

Vuonna 2024 Guo ym. julkaisivat tutkimuksen, jossa he vertailivat ChatGPT:n ja CodeReviewer-mallin suorituskykyä koodin parantamisessa kommenttien perusteella, dokumentoiden Tufanon ym. vertailua [15] tarkemmin tutkimuksessa käytetyt menetelmät ChatGPT:n arviointiin [16]. Tutkimuksessa CodeReviewer hienosäädettiin uudelle, Lin ym. käyttämästä samasta repositoriosta kerätylle datasetille, jotta CodeReviewer voisi käyttää uudempaa koulutusaineistoa hyväkseen. ChatGPT:n mallina Guo ym. käyttivät GPT-3.5-Turboa ilman hienosäätöä, sekä GPT-4:ää ta-

pauksiin, joissa 3.5-Turbo antoi vääriä vastauksia. Tutkimuksessa kokeiltiin viittä eri kehotetta eri lämpötiloilla OpenAI:n rajapinnan[4] kautta. Parhaimmalla kehotteen ja lämpötilan yhdistelmällä ChatGPT saavutti CodeRevieweria paremmat metriikat verrattuna CodeRevieweriin kaikilla mallien arviointiin käytettävillä dataseteillä, paitsi alkuperäisellä Lin ym. tutkimuksessa muodostetulla CodeReview-datasetillä. Guo ym. arvioivat, että ChatGPT oli tästä päätellen tehokkaampi yleistämään koulutusdatasta opittuja tietoja uusille tapauksille ja totesivat, että CodeReviewerin parempi suorituskyky alkuperäisellä datasetillä johtuu luultavasti siitä, että malli oli koulutettu samaan datasettiin kuuluvilla kommenteilla. Tutkimuksessa huomattiin kuitenkin, että ChatGPT:n suorituskyky kärsi rajallisesta tiedosta projektin rakenteesta, epäselvistä muutoksista ja epäselvistä muutosten sijainneista. Tämä sai GPT-3.5-Turbon hallusinoimaan virheellisiä ratkaisuja tiedon puutteen takia. Guo ym. ehdottivat mahdolliseksi ratkaisuksi tähän kehittyneemmän kielimallin, kuten GPT-4:n käyttöä. [16] Päätelmä rajallisen tiedon ja tiedon esitystavan vaikutuksista mallin suorituskykyyn vastaa Tufanon ym. havaintoja. [15]

Esikoulutettuja kielimalleja koodintarkastuksen automatisoinnissa tutkivat myös Yu ym. vuonna 2024 [17]. Tutkimuksen tavoitteena oli parantaa automaattisen koodintarkastuksen ymmärrettävyyttä, jotta ihmistarkastajan olisi helpompi puuttua automaattisen tarkastajan tekemiin virheisiin. Yu ym. esittelivät Carllm (Comprehensibility of Automated Code Review using Large Language Models) -järjestelmän. Carllm:n tarkoituksena oli luoda pull request -pyyntöihin kommentteja viisivaiheisen prosessin avulla: ensin Carllm teki binäärisen päätöksen siitä, tarvitseeko koodi lainkaan korjauksia. Jos koodissa oli jotain korjattavaa, muutosta tarvitseva koodi paikannettiin. Tämän jälkeen ongelmasta luotiin kuvaus, ehdotettiin ratkaisua ongelmaan ja lopuksi luotiin kommentti ongelman kuvauksen ja ehdotetun ratkaisun perusteella. Toisin kuin aiemmat koodintarkastuksen automatisointiin tähtäävät

[4] `https://platform.openai.com/docs/api-reference/chat`

järjestelmät, Carllm pyysi myös ihmisiltä vahvistusta tarkastusprosessin jokaisessa vaiheessa. Mikäli ihminen hyväksyi Carllm:n ehdotuksen, siirryttiin tarkastuksessa seuraavaan vaiheeseen. Vertaillessaan Carllm-järjestelmää muihin automaattisiin koodintarkastustyökaluihin Yu ym. havaitsivat, että Carllm kykeni arvioimaan muutosten tarpeellisuuden huomattavasti paremmin, kuin muut työkalut, kuten CodeReviewer, LLaMA-Reviewer, GPT-3.5-Turbo ja GPT-4. Carllm oli myös Yun ym. vertailussa aikaisempia menetelmiä tehokkaampi selkeiden kommenttien tuottamisessa: parhaimman 13 miljardin parametrin CodeLLaMA-mallin pohjalta hienosäädetyn Carllm-mallin tuottamista kommenteista 36,3 % luokiteltiin selkeiksi ja 16,4 % epäselviksi, kun taas CodeReviewerin kommenteista vain 20,7 % olivat selkeitä ja 53,2 % epäselviä ja LLaMA-Reviewerin kommenteista 15,3 % olivat selkeitä ja 40,7 % epäselviä. [17] Tulokset osoittavat, että LLaMA:n ja GPT:n kaltaiset yleiskäyttöiset kielimallit eivät välttämättä sovellu yksinään koodintarkastukseen luultavasti koodintarkastukseen liittymättömän koulutusdatan takia, mutta myös, että ainoastaan koodilla koulutetut kielimallit suoriutuvat heikosti koodintarkastuksessa tarvittavassa loogisessa päättelyssä ja luonnollisen kielen tehtävissä, kuten selkeiden kommenttien generoinnissa [17]. Koodiin erikoistuneiden mallien heikompi suorituskyky NLP-tehtävissä johtuu todennäköisesti siitä, että mallien koulutusdata ei juurikaan sisällä merkityksellistä määrää luonnollista kieltä.

OpenAI:n GPT-malleihin perustuvia ratkaisuja tutkivat myös Wang ym. vuonna 2024 julkaistussa tutkimuksessaan, jossa esiteltiin ChatGPT:hen perustuva CoRe (Collaborative LLM-based agents for Code Reviewer Recommendation) -järjestelmä [18]. Toisin kuin monissa aiemmissa koodintarkastuksen kielimalleilla automatisointia käsittelevissä tutkimuksissa, Wang ym. eivät pyrkineet automatisoimaan koodin kommentointia tai muutosten toteutusta, vaan kehittivät automaattisen menetelmän sopivan ihmistarkastajan löytämiseen pull request -pyynnön sisältämien muutosten perusteella. CoRe-järjestelmä koostuu yhteensä kuudesta GPT-3.5-Turbo-16k-

mallia käyttävästä agentista, joista jokaisella on oma tehtävänsä. Agentit tekevät yhteistyötä kommunikoimalla keskenään pull request -pyynnön sisällöstä ja rakenteesta ja tuottavat lopuksi luettelon pull request -pyynnölle sopivimmista tarkastajista perusteluineen. Wang ym. vertasivat CoRe-järjestelmän tekemiä päätöksiä aiemmin kehitettyyn RevFinder-työkaluun, joka ehdottaa pull request -pyynnöille tarkastajia muutettujen tiedostojen tiedostopolkujen perusteella antamalla tarkastajille tarkastettavaksi pyyntöjä, jotka liittyvät heidän asiantuntemukseensa. Tutkimuksen tulokset osoittivat, että CoRe suoriutui RevFinder-työkalua paremmin, sekä paremmin kuin yksi samaa GPT-mallia käyttävä ChatGPT-instanssi ilman yhteistyötä muiden agenttien kanssa. [18] Wang'n ym. tutkimuksen tuloksista voidaan päätellä, että laajat kielimallit kykenevät suorittamaan tehtävänsä tehokkaammin, mikäli niille annetaan hyvin tarkasti rajattu tehtävä. Erikoistuneet kielimallit voivat suorittaa monimutkaisempia tehtäviä ja käsitellä monimutkaisempaa tietoa kommunikoimalla keskenään.

4.2 Koodintarkastuksen automatisointi käytännössä

Useissa automaattisia koodintarkastusmenetelmiä käsittelevissä tutkimuksissa [12]–[18] työkaluja on arvioitu ainoastaan laboratorio-olosuhteissa vertailemalla työkalujen tuottamia kommentteja tai korjauksia ennalta määriteltyihin ratkaisuihin tai luokittelemalla työkalujen tuottamat tulokset manuaalisesti sen perusteella, kuinka selkeitä tai oikeita ratkaisut ovat olleet tutkijoiden mielestä. Kuten Yu ym. kuitenkin tutkimuksessaan totesivat, merkkijonojen vertailumenetelmät kuten BLEU eivät sovellu täysin koodintarkastuksen oikeellisuuden arviointiin: oikeina ratkaisuina käytetyssä datasetissä voi esiintyä virheitä ja kielimalleille ominainen tulosteiden satunnaisuus voi tuottaa yhtä toimivia mutta toisistaan eroavia ratkaisuja samoilla

syötteillä. [17] Tulosteiden manuaalinen arviointi yksitellen puolestaan on erittäin työlästä ja hidasta, jonka takia vain pieni määrä mallin tuottamista ratkaisuista on käytännöllistä arvioida manuaalisesti tutkimuksen yhteydessä. Tämän takia koodintarkastustyökaluja tulisi myös arvioida käytännössä, jotta niiden todellista hyödyllisyyttä ohjelmistokehityksessä voitaisiin arvioida.

Xiaon ym. vuonna 2024 julkaisemassa tutkimuksessa [19] arvioitiin GitHubin kehittämää Copilot for PRs -työkalua[5], jonka tarkoitus on auttaa kehittäjiä muutospyyntöjen laatimisessa generoimalla niille kuvauksia automaattisesti. Xiao ym. vertailivat Copilotilla generoituja muutospyyntöjä käyttäjien manuaalisesti tekemiin muutospyyntöihin. Tutkimus osoitti, että Copilot for PRs -työkalulla tehdyt muutospyynnöt hyväksytään keskimäärin nopeammin ja 1.57 kertaa todennäköisemmin kuin käyttäjien tekemät muutospyynnöt. Copilot for PRs -työkalun käyttöönoton myös todettiin yleistyvän tasaisesti. Monet kehittäjät eivät kuitenkaan käyttäneet Copilotin generoimia kuvauksia sellaisenaan: Generoiduista kuvauksista poistettiin usein tarpeettomia tietoja, kuten yhteenvetoja muutosten vaikutuksista ja linkkejä muutettuihin koodiriveihin. Kehittäjät tekivät myös projektin terminologiaan liittyviä korjauksia, ja joissakin tapauksissa luopuivat Copilotin käytöstä kokonaan vääränlaisen generoidun kuvauksen takia. Tulokset osoittavat, että Copilot for PRs -työkalu on kehittäjille hyödyllinen, mutta kehittäjien tekemistä muutoksista tulisi vielä oppia paremmin, millaista sisältöä kehittäjät odottavat Copilotilta. [19] Tämä olisi mahdollista saavuttaa esimerkiksi hienosäätämällä malli kehittäjien tekemillä korjauksilla generoituun sisältöön.

Vuonna 2024 Vijayvergiya ym. kehittivät T5-malliin perustuvan AutoCommenter-työkalun, jonka tehtävänä oli tunnistaa kohdat, joissa ohjelmakoodi poikkeaa suositelluista ohjelmointikäytännöistä. Käytännöistä poikkeaviin ongelmakohtiin malli tuotti kommentin ja varmuusarvion (confidence score) tuottamansa kommentin oi-

[5] `https://docs.github.com/en/copilot/using-github-copilot/using-github-copilot-for-pull-requests`

keellisuudesta. Varmuusarvion perusteella mallin tuottama kommentti joko hylättiin tai julkaistiin kehittäjien korjattavaksi. Tutkimuksen tavoitteena oli määrittää, voiko AutoCommenteria käyttää teollisessa ympäristössä Googlen sisäisesti joissakin projekteissa. AutoCommenterin tehokkuutta arvioitiin keräämällä palautetta kehittäjiltä, joista lähes 90 % suhtautuivat työkaluun myönteisesti, ja AutoCommenterin tuottamista kommenteista noin 60 % koettiin hyödyllisiksi. Jotkut mallin tuottamista huomautuksista perustuivat kuitenkin vanhentuneeseen tietoon ja sen toteutustavan takia tietojen päivitys olisi vaatinut mallin kouluttamista tai hienosäätöä uudella datasetillä, joten Vijayvergiya ym. päättivät suodattaa jotkin vanhentuneeseen tietoon perustuvat kommentit pois. [20]

Koodintarkastuskommenttien automaattista korjausta käytännössä tutkittiin Froemmgenin ym. vuonna 2024 julkaistussa tutkimuksessa, jossa kehitettiin T5:een perustuvan ihmisten jättämiä kommentteja ratkaiseva malli, joka otettiin koekäyttöön joissakin Googlen projekteissa [21]. Mallin tuottamat ratkaisut annettiin kehittäjille hyväksyttäväksi, jos malli oli riittävän varma ratkaisustaan. Tämä ylimääräinen hyväksymisvaihe mahdollisti matalamman kynnysvarmuuden käytön, jolloin useampi mallin tuottamista ratkaisuista esitettiin kehittäjille. Malli koulutettiin yli 3 miljardilla koodiesimerkillä, joista noin 60 miljoonaa liittyi koodintarkastukseen. Mallin suorituskykyä saatiin parannettua entisestään hienosäätämällä se ainoastaan ratkaisuilla koodintarkastuskommentteihin. Tutkimuksen tulokset osoittivat, että kehittäjät hyväksyivät hienosäädetyn mallin antamista ratkaisuista keskimäärin 7,5 % ja suhtautuivat positiivisesti työkaluun. [21] Hyväksyttyjen ratkaisujen melko matala osuus on mahdollisesti selitettävissä Yun ym. tutkimuksessa [17] havaitun rajoitteen avulla: mikäli mallin koulutusaineisto sisältää vain pieniä määriä luonnollista kieltä, malli ei välttämättä kykene tuottamaan selkeitä tai oikeita ratkaisuja tehokkaasti kommenttien perusteella.

Koodintarkastusympäristöön integroitujen työkalujen lisäksi kehittäjät ovat myös

käyttäneet Open AI:n ChatGPT-palvelua apunaan koodintarkastuksessa. Watanabe ym. tutkivat vuonna 2024 kehittäjien ChatGPT:n käyttöä [22]. Tutkimuksessa tutkittiin kehittäjien reaktioita ChatGPT:llä generoituihin ehdotettuihin ratkaisuihin muutospyyntöjen yhteydessä käydyissä keskusteluissa. Tutkituissa tapauksissa 64,0 % kehittäjistä suhtautuivat positiivisesti ChatGPT:llä generoituihin ratkaisuihin ja 30,7 % suhtautuivat negatiivisesti. Negatiivisista vastauksista yhteensä noin 34 % johtuivat siitä, että ChatGPT:n generoima koodi ei sopinut käyttötarkoitukseen tai ei toiminut odotetulla tavalla. Watanabe ym. myös huomauttivat, että 65,7 % negatiivisista vastauksista liittyivät tapauksiin, joissa ratkaisua ehdottanut kehittäjä ei ollut tarkastanut ChatGPT:n tuottamaa koodia, vaan tarjosi generoitua koodia suoraan ratkaisuna. Tämä osoittaa, että ChatGPT:llä luotu koodi on syytä tarkastaa ennen sen käyttöönottoa. [22]

4.3 Pohdinta

Tämän tutkielman tavoitteena oli tutkia nykykirjallisuudessa esiteltyjä automaattisia koodintarkastusmenetelmiä kirjallisuuskatsauksen keinoin. Tässä alaluvussa käydään läpi tutkielmaa tehdessä opittuja asioita koodintarkastuksen automatisoinnista ja vastataan tutkielman alussa esitettyihin tutkimuskysymyksiin.

Koodintarkastuksen automatisointiin generatiivisen tekoälyn avulla on kehitetty useita menetelmiä, joilla pyritään helpottamaan koodintarkastuksen kehittäjille aiheuttamaa työkuormaa [12]–[14], [17], [18]. Tufano ym. kehittivät vuonna 2021 ensimmäisen tässä tutkielmassa tutkitusta kirjallisuudesta löytyvän laajaan kielimalliin perustuvan automaattisen koodintarkastustyökalun, joka pystyi parantelemaan joitakin koodikatkelmia odotetuilla tavoilla [12]. Seuraavana vuonna Li ym. kehittivät CodeReviewer-työkalun, joka oli suorituskyvyltään parempi kuin Tufanon ym. esittämä menetelmä [13]. Vuonna 2023 Lu ym. kehittivät LLaMA-kielimalliin perustuvan LLaMA-Reviewer-työkalun, jonka he sovittivat koodintarkastukseen hie-

nosäätämällä normaalilla luonnollisella kielellä koulutetun mallin koodintarkastusesimerkeillä. LLaMA-Reviewer hyötyi tästä koulutusmenetelmästä ja suoriutui siten aiempia menetelmiä paremmin tehtävistä, jotka vaativat luonnollisen kielen käsittelyä, kuten tarkastuskommenttien lukemisesta ja muutosten toteuttamisesta niiden perusteella. [14] Hienosäätöä kielimallin suorituskyvyn parantamiseen käyttivät myös Yu ym. vuonna 2024 Carllm:n kehityksessä. Yun ym. kehittämä Carllm oli tehokkaampi automaattisessa koodintarkastuksessa ehdotettuja muutoksia perustelevan ajatusketjumallin ansiosta. [17] Ohjelmakoodiin tehtävien korjausten lisäksi generatiivista tekoälyä voidaan käyttää koodintarkastuksessa muihinkin tehtäviin, kuten sopivimman tarkastajan valintaan muutosten sisällön perusteella. Tähän tarkoitukseen kehittivät Wang ym. vuonna 2024 CoRe-järjestelmän, jossa useat eri rooleissa toimivat tekoälyagentit tekevät yhteistyötä sopivimman ihmistarkastajan valitsemiseksi. [18]

Suurta osaa tutkituista automaattisista koodintarkastustyökaluista on arvioitu ainoastaan laboratorio-olosuhteissa joko vertailemalla niiden tuottamia vastauksia ihmisten määrittelemiin oikeisiin vastauksiin tekstin samankaltaisuusmetriikoiden avulla tai manuaalisesti luokittelemalla tuotetut vastaukset joko oikeiksi tai vääriksi [12]–[14], [17], [18]. Työkalujen soveltuvuuden tutkiminen ainoastaan teoreettisin menetelmin on kuitenkin työlästä, eikä myöskään tarjoa täydellistä katsausta siihen, miten kehittäjät käyttäisivät työkaluja työssään. Vijayvergiyan ym. ja Froemmgenin ym. tutkimuksissa arvioitiin kommentteja generoivia ja ratkaisevia työkaluja Googlen sisäisissä projekteissa kysymällä kehittäjiltä palautetta työkalujen käytöstä. Molemmissa tutkimuksissa kehittäjät suhtautuivat yleisesti positiivisesti työkaluihin, mutta huomauttivat myös generoitujen ratkaisujen epätarkkuuksista. [20], [21] Vastaavia tuloksia havaittiin myös mm. Xiaon ym. tutkimuksessa, jossa tarkasteltiin kehittäjien kokemuksia GitHub Copilot for PRs -työkalusta [19]. Myös Watanaben ym. kehittäjien kokemuksia ChatGPT:n käytöstä koodintarkastukses-

sa arvioinut tutkimus osoitti, että suurin osa kehittäjistä suhtautui positiivisesti generatiivisen tekoälyn käyttöön, mutta tekoäly kuitenkin generoi melko usein virheellisiä ratkaisuja, joita ei voi käyttää sellaisenaan. [22] Yleisesti kehittäjien asenne koodintarkastuksen automatisointia kohtaan generatiivisella tekoälyllä on siis positiivinen, mutta tekoälypohjaiset työkalut tekevät virheitä niin usein, että niiden käyttö vaatii valvontaa ja palautetta ihmiseltä.

Tutkielman perusteella voidaan todeta, että generatiivisen tekoälyn käyttö koodintarkastuksessa on jo nyt mahdollista ihmistarkastajan tukena, mutta ei voi vielä korvata ihmisen kykyä arvioida koodimuutoksia laajemmassa kontekstissa. Automaattiset koodintarkastustyökalut voivat kuitenkin säästää aikaa koodintarkastusprosessissa auttamalla kehittäjiä löytämään virheitä ja parannusehdotuksia koodista nopeammin. Kehittäjät suhtautuvat tarkasteltuihin työkaluihin yleisesti ottaen positiivisesti, mutta tekoälyn ehdottamat ratkaisut tarvitsevat usein korjauksia ennen niiden käyttöönottoa. Koodintarkastustyökalujen kehittämistä tulisi siis jatkaa riittävän tarkkuuden ja luotettavuuden saavuttamiseksi. On myös syytä huomioida, että tekoälyn nopean kehittymisen takia tämän tutkielman kirjoitushetkellä julkaistut työkalut saattavat olla jo vanhentuneita tai niiden suorituskyky saattaa olla parantunut uusien mallien myötä. Tutkielman tulokset eivät siis välttämättä vastaa nykytilannetta täysin, mutta antavat kuitenkin katsauksen siihen, miten generatiivista tekoälyä on mahdollista käyttää koodintarkastuksessa.

5 Yhteenveto

Transformer-arkkitehtuuriin perustuvan generatiivisen tekoälyn käyttöä koodintarkastuksen automatisoinnissa on tutkittu toistaiseksi melko vähän. Tässä tutkielmassa generatiivisen tekoälyn nykyistä käyttöä ja tutkittuja toteutuksia koodintarkastuksessa tutkittiin kirjallisuuskatsauksen avulla. Koodintarkastuksen automatisoinnin havaittiin voivan säästää aikaa ja parantaa koodin laatua [19]. Automaattiset koodintarkastustyökalut nostivat myös muutospyyntöjen hyväksymisprosenttia. Vaikka suuri osa nykyisestä tutkimuksesta on toteutettu laboratorio-olosuhteissa, on Transformer-mallien tehokkuutta myös käytännössä arvioitu rajallisesti. Tutkielmassa käsitellyn aineiston perusteella tutkimuskysymyksiin saatiin seuraavat vastaukset:

TK 1. **Kuinka generatiivista tekoälyä voidaan hyödyntää koodintarkastuksen automatisoinnissa?**

Generatiivista tekoälyä voidaan hyödyntää koodintarkastuksen automatisoinnissa monin eri tavoin. Tähän mennessä julkaistuissa tutkimuksissa generatiivista tekoälyä on käytetty muutosten tarpeellisuuden arviointiin ennen varsinaista koodintarkastusta, muutoksia ehdottavien kommenttien generointiin, koodin automaattiseen korjaamiseen kommenttien perusteella, muutosten kuvausten kirjoittamiseen sekä ihmisten suorittamissa tarkastuksissa sopivan tarkastajan valintaan.

TK 2. **Kuinka tehokkaita ehdotetut automatisointimenetelmät ovat?**

Nykyisissä aihetta käsittelevissä tutkimuksissa tekoälymallien tekemät ratkaisut ovat olleet ajoittain kaukana ihmisten määrittelemistä oikeista ratkaisuista. Vaikka monet koodintarkastukseen suunnitellut kielimallit saavuttavat vaikuttavia tuloksia tarkkuudellaan, on virheellisten tai puutteellisten ratkaisujen generointi edelleen merkittävä riski. Tämän takia nykyiset mallit eivät sovellu koodintarkastuksen automatisointiin täysin ilman ihmisten valvontaa. Tästä huolimatta käytännössä testatut mallit, kuten GitHub Copilot for PRs, on kehittäjien keskuudessa koettu koodintarkastuksessa hyödyllisiksi ja aikaa säästäviksi työkaluiksi.

TK 3. **Mitkä ovat generatiivisen tekoälyn käytön haasteet koodintarkastuksessa?**

Merkittävä haaste koodintarkastuksen automatisoinnissa on tällä hetkellä mallien epätarkkuus ja siitä johtuva tarve generoitujen ratkaisujen manuaaliselle tarkastamiselle. Koodintarkastuksen täydellinen automatisointi ei siis vielä nykyisillä malleilla ole mahdollista. Osittain tämä johtuu todennäköisesti kielimallien rajallisesta koodin kontekstin ymmärryksestä; kielimallit eivät saa nykyisillä toteutuksilla tietoa saman projektin muista osista koodimuutosten ulkopuolella, joten malleilla ei ole käytettävissä hyödyllistä tietoa esimerkiksi saatavilla olevista apufunktioista tai -luokista, projektin rakenteesta tai koodityylistä. Tässä hyödyksi voisi olla RAG-menetelmien (Retrieval Augmented Generation) käyttäminen, joka antaisi kielimallin hakea lisää tietoa muualta projektista tarvittaessa.

Transformer-arkkitehtuuri oli suuri askel eteenpäin generatiivisen tekoälyn tehokkuudessa, jota on onnistuttu hyödyntämään jo joissakin määrin koodintarkastuksen automatisoinnissa. Transformer-arkkitehtuuriin perustuvat kielimallit kehittyvät myös edelleen nopeasti, minkä takia tässä tutkielmassa käsitellyt ratkaisut

eivät käytä esimerkiksi OpenAI:n uusimpia GPT-malleja, kuten GPT-4o:ta. Jatkotutkimuksena voisi siis tutkia uudempien kielimallien suorituskykyä koodintarkastuksessa. Koska malleille välitettävän tiedon todettiin tässä tutkielmassa olevan rajoittava tekijä mallien suorituskyvyssä, olisi syytä tutkia myös mahdollisia keinoja tämän ongelman ratkaisemiseen RAG-menetelmien avulla. Tämä voisi kehittyneempien kielimallien lisäksi edistää merkittävästi koodintarkastuksen automatisointia vähentäen ihmisten antaman palautteen tarvetta.

Lähdeluettelo

[1] A. Bosu, J. C. Carver, C. Bird, J. Orbeck ja C. Chockley, ”Process Aspects and Social Dynamics of Contemporary Code Review: Insights from Open Source Development and Industrial Practice at Microsoft”, *IEEE Transactions on Software Engineering*, vol. 43, nro 1, s. 56–75, tammikuu 2017, ISSN: 1939-3520. DOI: 10.1109/TSE.2016.2576451.

[2] X. Zhou, K. Kim, B. Xu, D. Han, J. He ja D. Lo, ”Generation-based Code Review Automation: How Far Are We?”, teoksessa *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*, toukokuu 2023, s. 215–226. DOI: 10.1109/ICPC58990.2023.00036. url: https://ieeexplore.ieee.org/document/10174115/?arnumber=10174115.

[3] S. Eldh, ”Code Review Evolution”, *IEEE Software*, vol. 41, nro 5, s. 4–8, syyskuu 2024, ISSN: 1937-4194. DOI: 10.1109/MS.2024.3416648.

[4] A. Aurum, H. Petersson ja C. Wohlin, ”State-of-the-art: software inspections after 25 years”, en, *Software Testing, Verification and Reliability*, vol. 12, nro 3, s. 133–154, 2002, ISSN: 1099-1689. DOI: 10.1002/stvr.243.

[5] Z.-X. Li, Y. Yu, G. Yin, T. Wang ja H.-M. Wang, ”What Are They Talking About? Analyzing Code Reviews in Pull-Based Development Model”, en, *Journal of Computer Science and Technology*, vol. 32, nro 6, s. 1060–1075, marraskuu 2017, ISSN: 1000-9000, 1860-4749. DOI: 10.1007/s11390-017-1783-2.

[6] J. Jiang, J. Lv, J. Zheng ja L. Zhang, "How Developers Modify Pull Requests in Code Review", *IEEE Transactions on Reliability*, vol. 71, nro 3, s. 1325–1339, syyskuu 2022, ISSN: 1558-1721. DOI: 10.1109/TR.2021.3093159.

[7] I. Goodfellow, Y. Bengio ja A. Courville, *Deep learning* (Adaptive computation and machine learning). Cambridge, Massachusetts: The MIT Press, 2016, ISBN: 978-0-262-03561-3. url: https://www.deeplearningbook.org/.

[8] T. Amaratunga, *Understanding Large Language Models: Learning Their Underlying Concepts and Technologies*, en. Berkeley, CA: Apress, 2023, ISBN: 9798868800160. DOI: 10.1007/979-8-8688-0017-7. url: https://link.springer.com/10.1007/979-8-8688-0017-7.

[9] A. Vaswani, N. Shazeer, N. Parmar et al., "Attention Is All You Need", elokuu 2023, arXiv:1706.03762. DOI: 10.48550/arxiv.1706.03762. url: http://arxiv.org/abs/1706.03762.

[10] X. L. Li ja P. Liang, "Prefix-Tuning: Optimizing Continuous Prompts for Generation", teoksessa *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, C. Zong, F. Xia, W. Li ja R. Navigli, toim., Online: Association for Computational Linguistics, elokuu 2021, s. 4582–4597. DOI: 10.18653/v1/2021.acl-long.353. url: https://aclanthology.org/2021.acl-long.353.

[11] E. J. Hu, Y. Shen, P. Wallis et al., "LoRA: Low-Rank Adaptation of Large Language Models", nro arXiv:2106.09685, lokakuu 2021, arXiv:2106.09685. DOI: 10.48550/arXiv.2106.09685. url: http://arxiv.org/abs/2106.09685.

[12] R. Tufano, L. Pascarella, M. Tufano, D. Poshyvanyk ja G. Bavota, "Towards Automating Code Review Activities", teoksessa *Proceedings of the 43rd International Conference on Software Engineering*, sarja ICSE '21, Madrid, Spain:

IEEE Press, maaliskuu 2021, s. 163–174, ISBN: 978-1-4503-9085-9. DOI: 10.1109/ICSE43902.2021.00027. url: https://dl.acm.org/doi/10.1109/ICSE43902.2021.00027.

[13] Z. Li, S. Lu, D. Guo et al., ”Automating code review activities by large-scale pre-training”, teoksessa *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, sarja ESEC/FSE 2022, New York, NY, USA: Association for Computing Machinery, maaliskuu 2022, s. 1035–1047, ISBN: 978-1-4503-9413-0. DOI: 10.1145/3540250.3549081. url: https://dl.acm.org/doi/10.1145/3540250.3549081.

[14] J. Lu, L. Yu, X. Li, L. Yang ja C. Zuo, ”LLaMA-Reviewer: Advancing Code Review Automation with Large Language Models through Parameter-Efficient Fine-Tuning”, teoksessa *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, lokakuu 2023, s. 647–658. DOI: 10.1109/ISSRE59848.2023.00026. url: https://ieeexplore.ieee.org/document/10299938/?arnumber=10299938.

[15] R. Tufano, O. Dabić, A. Mastropaolo, M. Ciniselli ja G. Bavota, ”Code Review Automation: Strengths and Weaknesses of the State of the Art”, *IEEE Transactions on Software Engineering*, vol. 50, nro 2, s. 338–353, helmikuu 2024, ISSN: 1939-3520. DOI: 10.1109/TSE.2023.3348172.

[16] Q. Guo, J. Cao, X. Xie et al., ”Exploring the Potential of ChatGPT in Automated Code Refinement: An Empirical Study”, teoksessa *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, sarja ICSE ’24, New York, NY, USA: Association for Computing Machinery, 2024, s. 1–13, ISBN: 9798400702174. DOI: 10.1145/3597503.3623306. url: https://dl.acm.org/doi/10.1145/3597503.3623306.

[17] Y. Yu, G. Rong, H. Shen et al., "Fine-tuning Large Language Models to Improve Accuracy and Comprehensibility of Automated Code Review", *ACM Trans. Softw. Eng. Methodol.*, 2024, Just Accepted, ISSN: 1049-331X. DOI: 10.1145/3695993. url: https://dl.acm.org/doi/10.1145/3695993.

[18] L. Wang, Y. Zhou, H. Zhuang et al., "Unity Is Strength: Collaborative LLM-Based Agents for Code Reviewer Recommendation", teoksessa *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, sarja ASE '24, New York, NY, USA: Association for Computing Machinery, lokakuu 2024, s. 2235–2239, ISBN: 9798400712487. DOI: 10.1145/3691620.3695291. url: https://dl.acm.org/doi/10.1145/3691620.3695291.

[19] T. Xiao, H. Hata, C. Treude ja K. Matsumoto, "Generative AI for Pull Request Descriptions: Adoption, Impact, and Developer Interventions", *Research Artifact - Generative AI for Pull Request Descriptions: Adoption, Impact, and Developer Interventions*, vol. 1, nro FSE, 47:1043–47:1065, 2024. DOI: 10.1145/3643773.

[20] M. Vijayvergiya, M. Salawa, I. Budiselić et al., "AI-Assisted Assessment of Coding Practices in Modern Code Review", en, teoksessa *Proceedings of the 1st ACM International Conference on AI-Powered Software*, Porto de Galinhas Brazil: ACM, heinäkuu 2024, s. 85–93, ISBN: 979-8-4007-0685-1. DOI: 10.1145/3664646.3665664. url: https://dl.acm.org/doi/10.1145/3664646.3665664.

[21] A. Froemmgen, J. Austin, P. Choy et al., "Resolving Code Review Comments with Machine Learning", teoksessa *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, sarja ICSE-SEIP '24, New York, NY, USA: Association for Computing Machinery,

2024, s. 204–215, isbn: 979-8-4007-0501-4. doi: 10.1145/3639477.3639746. url: https://dl.acm.org/doi/10.1145/3639477.3639746.

[22] M. Watanabe, Y. Kashiwa, B. Lin, T. Hirao, K. Yamaguchi ja H. Iida, ”On the Use of ChatGPT for Code Review: Do Developers Like Reviews By ChatGPT?”, en, teoksessa *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, Salerno Italy: ACM, kesäkuu 2024, s. 375–380, isbn: 9798400717017. doi: 10.1145/3661167.3661183. url: https://dl.acm.org/doi/10.1145/3661167.3661183.