Automaattinen ohjelmakorjaus suurten kielimallien avulla

TURUN YLIOPISTO
Tietotekniikan laitos
LuK-tutkielma
Tietojenkäsittelytiede
Kesäkuu 2025
Aarni Kivelä

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

TURUN YLIOPISTO
Tietotekniikan laitos

Aarni Kivelä: Automaattinen ohjelmakorjaus suurten kielimallien avulla

LuK-tutkielma, 33 s.
Tietojenkäsittelytiede
Kesäkuu 2025

Suurten kielimallien (LLM) kehitys on avannut uusia mahdollisuuksia automatisoituun ohjelmakorjaukseen (APR), joka pyrkii tunnistamaan ja korjaamaan ohjelmistovirheitä ilman ihmisen väliintuloa. Erityisesti suurissa ohjelmistoprojekteissa APR voi vähentää huomattavasti virheiden korjaamiseen kuluvaa aikaa ja työtä.
Tämä tutkielma on kirjallisuuskatsaus, jossa analysoidaan viime vuosien merkittävimpiä julkaisuja aiheesta. Tutkielma tarkastelee LLM-pohjaisten APR-menetelmien etuja, rajoituksia ja toimintaperiaatteita verrattuina perinteisiin lähestymistapoihin, kuten symboliseen analyysiin ja geneettisiin algoritmeihin perustuviin APR-menetelmiin. Lisäksi tutkielma tarkastelee APR-prosessin käytettävyyttä reaalimaailman tilanteissa erityisesti LLM-pohjaisten APR-menetelmien näkökulmasta.
Tutkielmassa APR-prosessia käsitellään kahdessa toiminnallisessa päävaiheessa: virheen paikantaminen (FL) ja korjauksen generoiminen (PR). Kumpaankin päävaiheeseen liittyen tarkastellaan keinoja mukauttaa LLM:ien toimintaa APR-prosessiin erilaisten kehotestrategioiden ja LLM:ien hienosäädön avulla.
LLM-pohjaiset APR-menetelmät ovat osoittautuneet selvästi tehokkaammiksi, tarkemmiksi ja sovellettavammiksi kuin perinteiset lähestymistavat. Tutkimuskenttää hallitsevat dekooderiarkkitehtuuriin perustuvat mallit, joiden generatiiviset kyvykkyydet ja yleistämispotentiaali tekevät niistä erityisen soveltuvia ohjelmakorjaukseen. Lisäksi hybridimenetelmät, jotka yhdistävät perinteisiä tekniikoita LLM-pohjaisiin ratkaisuihin, tarjoavat lupaavan suunnan tulevalle tutkimukselle.

Asiasanat: kielimallit, ohjelmakorjaus, vian paikannus, kehote, hienosäätö, hybridimenetelmä

Sisällys

1 Johdanto **1**

2 Tausta **5**

2.1 Suuret kielimallit . . . 5

2.2 Automaattinen ohjelmakorjaus . . . 6

3 Kielimallien mukauttaminen **10**

3.1 Kehotteiden laatiminen . . . 10

3.2 Aihealuekohtainen hienosäätö . . . 11

4 APR-prosessin vaiheet **14**

4.1 Vian paikannus . . . 14

4.1.1 Perinteiset FL-menetelmät . . . 15

4.1.2 Testaukseen perustuva virheiden paikantaminen kielimalleilla . 16

4.1.3 Kielimallien suora hyödyntäminen virheiden paikantamiseen . 17

4.2 Ohjelmakorjaus . . . 20

5 Pohdinta **26**

6 Yhteenveto **31**

Lähdeluettelo **34**

1 Johdanto

Viime vuosina suurten kielimallien (engl. Large Language Models, LLM) kehitys on mullistanut monia tekoälyn sovellukohteita, kuten ohjelmistotekniikan eri osa-alueita. Yksi lupaava sovelluskohde on automatisoitu ohjelmakorjaus (engl. Automated Program Repair, APR), joka pyrkii automatisoimaan ohjelmakoodin virheiden tunnistamisen ja korjaamisen ilman ihmisen väliintuloa. Etenkin suuren kokoluokan ohjelmistokehitysprojekteissa, joissa koodia saattaa olla miljoonia rivejä, virheiden paikantaminen ja korjaaminen on ohjelmistokehittäjille erittäin aikaavievä prosessi. APR on siksi kerännyt merkittävän määrän akateemista huomiota, sillä sen avulla yksi ohjelmistokehitysprosessin vaivalloisimmista vaiheista voitaisiin trivialisoida.

APR mielletään yhdeksi haastavimmista ohjelmakoodin ylläpidollisista osa-alueista sen monimutkaisuuden vuoksi. Perinteiset APR-menetelmät hyödyntävät esimerkiksi symbolista analyysiä, geneettisiä algoritmeja sekä heuristiikkapohjaisia lähestymistapoja. Myös LLM:iä edeltäviä koneoppimiseen perustuvia lähestymistapoja, kuten syväoppimismalleja ja neuroverkkopohjaisia menetelmiä, on tutkittu ja sovellettu APR-menetelmiin. Juuri koneoppimiseen perustuvassa lähestymistavassa LLM-pohjaiset ratkaisut erottuvat muista menetelmistä valtavien esikoulutusaineistojensa vuoksi. LLM-pohjaisilla menetelmillä on mahdollista parantaa APR:n tehokkuutta, kattavuutta sekä käytettävyyttä, ja siten myös tehdä APR-prosessista oikeasti toimiva vaihtoehto ohjelmakehityksessä.

Koska kyseessä on alati muuttuva ja kehittyvä aihepiiri, tässä tutkielmassa tarkastellaan erilaisten viime vuosina kehitettyjen menetelmien hyötyjä ja toimintaperiaatteita mutta myös niitä haasteita ja haittapuolia, joita LLM:ien valjastamisessa APR-menetelmiin ilmenee. Tutkielma keskittyy nimenomaan APR:n hyödyntämiseen ohjelmakoodiin. Tutkielma ei ota kantaa APR-prosessin kehittämisen ulkoisiin tekijöihin, kuten APR-työkalujen integroimiseen työympäristöihin, vaan arvioi APR-prosessin elinkaaren sisäistä kehitystä LLM:ien yleistymisen yhteydessä. On hyvä huomata, että LLM:ien soveltaminen APR-menetelmiin on tullut mahdolliseksi vasta viime vuosina LLM:ien nopean kehityksen myötä. Sen vuoksi aihepiiriä tutkitaan paljon ja tutkimuksia julkaistaan nopealla tahdilla.

Tämä tutkielma toimii kirjallisuuskatsaksena aihepiirin lupaavimmille kehityskohteille ja näkökulmille. Tutkielma pyrkii myös vertailemaan ja arvioimaan ehdotettujen menetelmien tehokkuutta ja käytännöllisyyttä aiempiin perinteisiin ratkaisuihin verrattuna. Perinteisillä menetelmillä tai lähestymistavoilla tarkoitetaan niitä menetelmiä, jotka ovat olleet olemassa ennen LLM:ien yleistymistä, eivätkä ne siten hyödynnä LLM:iä ollenkaan. Tutkielma vastaa kolmeen tutkimuskysymykseen:

1. Mitä on automaattinen ohjelmakorjaus ja miksi sitä tutkitaan?

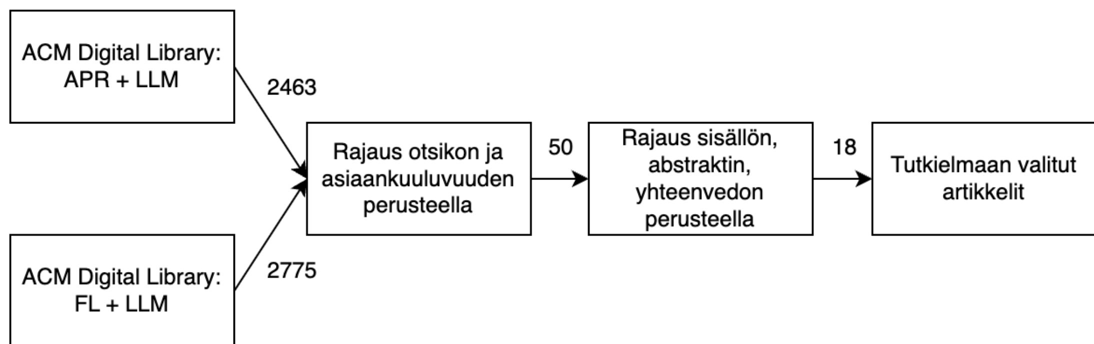
2. Miten suuria kielimalleja voidaan hyödyntää automaattisessa ohjelmakorjauksessa ja miten ne vertautuvat aiempiin lähestymistapoihin?

3. Onko automaattinen ohjelmakorjaus käytettävä ratkaisu ja miten LLM:ien integroiminen APR-prosessiin vaikuttaa sen käytettävyyteen?

Tutkielman tiedonhaku suoritettiin ACM Digital Libraryssä. Tärkeimmät hakulauseet olivat "Automated Program Repair" sekä "Fault Localization" yhdistettynä sanaan "LLM" tai "Large Language Model". Hakutuloksia rajattiin siten, että vain vuonna 2024 tai sen jälkeen hyväksytyt julkaisut otettiin mukaan tutkielmaan. Ensimmäinen hakulause tuotti 2463 hakutulosta ja toinen 2775 hakutulosta ACM

Digital Librarystä. Seuraavaksi hakuavaruutta rajasi huomattavasti lähteiden asiaankuuluvuus: tutkielmaan ei hyväksytty lähteitä, jotka käsittelevät automaattista ohjelmakorjausta tai vian paikannusta muissa konteksteissa, kuin ohjelmakoodin yhteydessä. Tuloksena oli noin 50 lupaavaa artikkelia, jotka valittiin tarkempaan arviointiin.

Julkaisut hyväksyttiin tutkielmaan niiden otsikon, tiivistelmän, johdannon ja yhteenvedon sisällön mukaan. Tiedonhaun tuloksena tutkielmaan valikoitui 18 vuonna 2024 tai sen jälkeen julkaistua ACM- tai IEEE-artikkelia, joista yhdeksän käsittelee huipputason saavuttaneita, toiminnallisesti ja strategisesti toisistaan poikkeavia LLM-pohjaisia ohjelmakorjaus- tai vianpaikannusmenetelmiä. Loput 9 käsittelevät LLM:ien hienosäätöä, kehotteiden laatimista ja APR-prosessin arviointia. Tiedonhaun jälkeen työhön on lisätty lähteitä pääosin jo kelpuutettujen lähteiden lähteistä ja olennaisten yksittäisten menetelmien julkaisuartikkeleista.



Kuva 1.1: Vuokaavio tiedonhakuprosessista

Luku 2 pohjustaa tutkielman kannalta olennaisimmat aihealueet: suuret kielimallit sekä automaattisen ohjelmakorjauksen. Luvun tarkoituksena on esittää molempien aihealueiden keskeiset lähtökohdat ja siten luoda tutkielmalle vahva perusta.

Luku 3 tarkastelee merkittävimpiä lähestymistapoja LLM:ien mukauttamisessa APR-prosessiin. Tässä luvussa pohjustetaan LLM:ille annettujen kehotteiden mer-

kitystä ja niiden lupaavimia laatimis-strategioita, sekä LLM:ien aihealuekohtaista hienosäätöä APR-prosessiin sovellettuna.

Luku 4 esittelee ja analysoi uusimpien LLM-pohjaisten menetelmien lähestymistapoja sekä ohjelmakorjauksen että vianpaikannuksen vaiheissa. Luvun päätavoitteena on tarkastella uusimpien APR-menetelmien strategioita ja lähestymistapoja tehokkuuden, tarkkuuden, käytettävyyden ja käytännön soveltuvuuden näkökulmasta. Lisäksi tässä luvussa LLM-pohjaisia menetelmiä verrataan perinteisiin APR-menetelmiin niiden saavuttamien tulosten perusteella.

Luku 5 arvioi koko APR-prosessia sekä LLM-pohjaisia APR-menetelmiä. Tarkastelun kohteena ovat tutkielmassa tarkasteltujen LLM-pohjaisten menetelmien lähestymistavat, vahvuudet ja heikkoudet, erityisesti suhteessa nykyteknologian kehityksen huipputasoon. Lisäksi luvussa pohditaan lupaavia jatkotutkimuskohteita tutkielman sisällön perusteella, keskittyen erityisesti niihin osa-alueisiin, joissa esiintyy selkeitä puutteita tai kehitystarpeita.

Luku 6 on tutkielman päättävä yhteenveto, jossa käydään lyhyesti läpi tutkielman pääkohdat. Yhteenvedossa myös vastataan lyhyesti tutkimuskysymyksiin.

2 Tausta

2.1 Suuret kielimallit

Kielimallit (engl. Language Model, LM) ovat neuroverkkopohjaisia koneoppimismalleja, joiden tarkoitus on tuottaa tekstiä ennustamalla. Kielimallit ovat saavuttaneet merkittäviä tuloksia luonnollisen kielen käsittelyn (engl. Natural Language Processing, NLP) tutkimuksessa. [1] Kielimallin esikouluttaminen mahdollistaa sen, että malli voi hyödyntää laajaa empiiristä tietoa generatiivisissa ja arviointitehtävissä. Termille suuri kielimalli ei ole olemassa mitään tarkkaa kriteeriä, vaan sillä viitaataan yleisesti kielimallien huomattavaan tehokkuuden ja tarkkuuden paranemiseen, kun kielimallin parametrien määrä ylittää tietyn pisteen [1]. Suuret kielimallit jaetaan yleisesti kolmeen eri kategoriaan niiden sisäisten arkkitehtuurien perusteella [1] [2].

Dekooderi-arkkitehtuuriin (engl. Decoder-only) pohjautuvat LLM:t, kuten OpenAi:n GPT-sarja [3], generoivat tekstiä jatkuvan tokenien ennustamisen avulla ilman tarvetta syötteen koodikielelle muuntavalle enkooderille. Dekooderiarkkitehtuuriin perustuvat suuret kielimallit ovat siten tehokkaimpia juuri generoimiseen ja täydentämiseen liittyvissä tehtävissä. [1] Dekooderiarkkitehtuuriin pohjautuvat LLM:t kattavatkin tällä hetkellä suuren osan avoimen lähdekoodin viimeisintä kehitystä edustavista LLM:istä [4], ja siten myös merkittävän osan tähän tutkielmaan valikoituneista menetelmistä.

Enkooderi-arkkitehtuuriin (engl. Encoder-only) pohjautuvat LLM:t, kuten esimerkiksi BERT [5] hyödyntävät pelkästään enkooderikomponenttia syötteen prosessoimiseen ja koodikielelle muuntamiseen. Enkooderiarkkitehtuuri mahdollistaa syötteen rakenteellisen ja semanttisen ymmärryksen. Esimerkiksi BERTin kaksisuuntainen tokenisointimekanismi mahdollistaa mallille kyvyn huomioida kyseisen tokenin konteksti sekä vasemmalta että oikealta puolelta, johon dekooderipohjaiset LLM:t eivät pysty. [1]

Enkooderi-dekooderi-arkkitehtuuriin (engl. Encoder-Decoder) pohjautuvat LLM:t sisältävät sekä enkooderi- että dekooderikomponentit. Arkkitehtuurin ideana on, että LLM voi hyödyntää sekä enkooderin kyvyn tunnistaa syötteen rakenteelliset ja semanttiset ominaisuudet että dekooderin generatiiviset ominaisuudet konkreettisen ja kontekstuaalisesti oikean tuloksen tuottamiseen [1]. Enkooderi-dekooderi-arkkitehtuuri tarjoaa joustavammat mahdollisuudet käyttää kielimallia [1] ja mahdollistaa esimerkiksi koodin semantiikkaan liittyvien tehtävien toteuttamisen generatiivisesti [2]. LLM:t kuten T5 [6] ja siitä johdettu CodeT5 [7] käyttävät kyseistä arkkitehtuuria.

2.2 Automaattinen ohjelmakorjaus

Automaattisen ohjelmakorjauksen tarkoitus on korjata ohjelmakoodissa tapahtuvia virheitä ilman ihmisen väliintuloa [8] [9] [10]. Jotta toimiva APR-prosessi olisi mahdollinen saavuttaa, on tärkeä jakaa APR kahteen päävaiheeseen: vian paikannukseen (engl. Fault Localization, FL) ja korjauksen generoimiseen (engl. Program Repair, PR). Kumpikin päävaihe on välttämätön automaattisen prosessin saavuttamiseksi [11].

Perinteisesti vian paikannuksella tarkoitetaan sekä automaattisen ohjelmakorjauksen että manuaalisen debuggauksen vaihetta, jossa ongelman aiheuttava osa ohjelmakoodia pyritään paikantamaan mahdollisimman tarkasti [12]. Viat voivat

tämän tutkielman viitekehyksessä olla suoria koodin kääntämisen tai ajon aikaisten matalan tason virheitä aiheuttavien bugien (engl. Single-hunk bugs) lisäksi moniviitteisiä, hankalasti paikannettavia sekä korjattavia bugeja (engl. Multi-hunk bugs) [13]. Jotta APR olisi mahdollisimman käytettävä prosessi, vikoihin luetaan myös tietoturvaongelmia aiheuttavat heikkoudet (engl. vulnerabilities) ja ohjelman tehokkuutta rajoittavat pullonkaulat [13]. Siispä FL-vaiheessa on tärkeää paikantaa ongelmallisen koodin lisäksi myös sen konteksti mahdollisimman tarkasti ongelman oikeellisen kategorisoimisen saavuttamiseksi. Toimivan APR:n kannalta on tärkeää, että pelkän virheellisen kohdan paikantamisen lisäksi ymmärretään ongelmien syyt ja vaikutukset.

Toinen askel toimivan APR-prosessin saavuttamiseen on itse ohjelmakorjaus. PR-vaihe pyrkii generoimaan toimivan korjauksen (engl. patch) ongelman kontekstin mukaan mahdollisimman kattavasti ilman ihmisen väliintuloa. Yleisesti voidaan olettaa FL-vaiheen edeltävän PR-vaihetta, jotta generoidut korjaukset ovat mahdollisimman tarkkoja.

Sekä FL- että PR-vaiheen menetelmien suorituskykyä on tässä tutkielmassa pääosin testattu Defects4J Java-pohjaisella aineistolla. Defects4J koostuu virheistä, joista se tarjoaa korjaamattomat ja korjatut versiot, mikä mahdollistaa kontrolloidut testauskokeet Java-ohjelmille. Defects4J sisältää lisäksi täydelliset virheiden sijainnit, joita voidaan hyödyntää PR-vaiheen menetelmien testauksessa. [14]

Perinteiset APR-menetelmät ovat perustuneet symboliseen analyysiin, geneettisiin algoritmeihin ja heuristiikkapohjaisiin menetelmiin, joiden avulla ohjelmakoodia on pyritty seulomaan ja korjaamaan automaattisesti. Edellä mainitut menetelmät sisältävät kuitenkin huomattavia, perustavanlaatuisia puutteita tarkkuudessa, tehokkuudessa, joustavuudessa ja sovellettavuudessa. Tuoreimmat tutkimustulokset koneoppimiseen perustuvista ei-LLM-pohjaisista menetelmistä ovat osoittautuneet lupaaviksi, mutta näidenkin menetelmien tehokkuutta rajoittavat usein opetusaineis-

tojen niukkuus, yksipuolisuus ja mahdollinen aineistoille ylivirttyminen. [15] Vertailun helpottamiseksi tämän tutkielman yhteydessä perinteisiin APR-menetelmiin luetaan kaikki LLM:iä edeltävät lähestymistavat.

LLM:ien suurimpiin vahvuuksiin APR:ään aiemmin sovellettuihin menetelmiin verrattuna kuuluu niiden monipuolinen opetustietoaineisto, jonka myötä niille muodostuu erinoimainen kyky yleistää asioita. Laajan opetustietoaineistonsa ansiosta LLM:t ovat osoittautuneet eteviksi monissa eri ohjelmointikielissä ja niihin liittyvissä ohjelmointiparadigmoissa, kuten oliopohjainen tai funktionaalinen ohjelmointi. Tämä on jo itsessään suuri harppaus aiempiin APR-menetelmiin verrattuna. Opetustietoaineistonsa ansioista LLM:t ovat myös lähtökohtaisesti huomattavan joustavia ja usein sovellettavissa erilaisiin ohjelmistoihin. [15]

Hybridimenetelmillä tarkoitetaan sellaisia APR-lähestymistapoja, joissa perinteisempää ohjelma-analyysitekniikkaa yhdistetään LLM:n toimintaan. Siten hybridimenetelmiksi luokitellaan ne lähestymistavat, joissa FL- tai PR-vaiheen tuloksiin vaikuttavat pelkän LLM:n lisäksi jokin muukin ei-LLM-pohjainen prosessi.

APR-prosessin tavoitetila on täysi automaattisuus. Jotta prosessin käyttäminen olisi ylipäätään kannattavaa, sen on tunnistettava ongelmakohdat luotettavasti ja generoitava korjaukset tarkasti. Samaan aikaan prosessin tulee olla käytännöllisesti hyödyllinen työkalu. Prosessin pitää olla joustava, kattava ja yleistettävä, jotta sitä voitaisiin hyödyntää toisistaan ratkaisevasti eriäviin tilanteisiin kuten eri ohjelmistotyyppeihin, ohjelmointikieliin ja paradigmoihin. [15].

Liiallinen luottamus LLM:ien yleispätevyyteen johtaa nopeasti aihealuekohtaisen tarkentavan aineiston puutteen tai vaikeiden ja monitahoisten ongelmien vuoksi virheisiin, kuten hallusinaatioihin. Toimiva APR-prosessi vaatiikin usein erityistä kehotteiden suunnittelustrategiaa tai aihealuekohtaista opetusaineistoa virheiden ja hallusinaatioiden välttämiseksi. APR-prosessin tarkkuuden kehittämistä on viime vuosina tutkittu huomattavan paljon. [4] [12] [16]. Seuraavan luvun tarkoituksena

on tarkastella erilaisia tapoja mukauttaa LLM:ien käyttäytymistä ja siten optimoida niiden tarkkuutta, kun niitä sovelletaan APR-prosessin kumpaankin päävaiheeseen.

3 Kielimallien mukauttaminen

3.1 Kehotteiden laatiminen

Välttämätön ensimmäinen askel LLM:n hyödyntämisessä mihin tahansa tiettyyn tehtävään on kehotteiden suunnitteleminen (engl. prompt engineering). Huolellisen kehotteiden laatimisen on osoitettu olevan kriittistä LLM:n toimivuuden kannalta [17] [18]. Kehotteet voidaan laatia monella eri tavalla, mutta yleensä niiden tulee sisältää vähintäänkin seuraavat asiat: LLM tarvitsee ohjeen tai kysymyksen, josta se saa tiedon suoritettavasta toimenpiteestä kuten korjauksen laatimisesta. FL-vaiheessa LLM tarvitsee virheen sisältävän palan ohjelmakoodia, joka voi olla esimerkiksi virheen sisältävä funktio tai luokka. PR-vaiheessa LLM voi hyödyntää samaa ohjelmakoodin osaa kontekstina FL-vaiheesta saadun virheen tarkan sijainnin lisäksi. [17]

Tavanomaisiin kehotestrategioihin lukeutuvat nollan ja harvan esimerkin kehotteet. Nollan esimerkin kehotteella (engl. Zero-Shot Learning, ZSL) tarkoitetaan LLM:n ohjeistamista tehtävään, johon sitä ei ole erityisesti koulutettu. Harvan esimerkin kehotteella (engl. Few-Shot Learning, FSL) taas tarkoitetaan vastaavasti sitä, että kehote sisältää jotain aihepiiriin liittyvää dataa, jota LLM voi hyödyntää tehtävässä, kuten esimerkkejä tehtävän suorittamisesta. [1] [2]

Etenkin dekooderiarkkitehtuuriin pohjautuvat LLM:t ovat osoittautuneet tehokkaiksi kehotteiden ohjaamina, sillä ne hyötyvät merkittävästi kehotteen sisällön laa-

dusta ja kattavuudesta. Niiden erinoimainen kyky analysoida ja tuottaa luonnollista kieltä tarjoaa hyvät valmiudet LLM:n soveltamiselle APR-prosessiin pelkän kehotteen varassa, mikä mahdollistaa ZSL- ja FSL-menetelmien hyödyntämisen erityisesti matalan tason virheiden korjaamiseen. [1] [2]

Edellä mainittuja pidemmälle viedyt kehotteiden laatimismenetelmät ovat myös saaneet paljon huomiota. Kaksi hyviä tuloksia tuottanutta menetelmää ovat ajatusketjujen (engl. Chain of Thought, CoT) ja suunnittelun perustelujen (engl. Design Rationale, DR) lisääminen kehotteeseen. CoT-kehottaminen perustuu vaiheittaiseen päättelyyn, missä LLM:n kehotteeseen yksinkertaisesti lisätään ohje: "Let's think step by step"[19] [18]. Jo pelkästään kyseisen fraasin on todettu parantavan esimerkiksi ChatGPT:n tekemiä valintoja, ja siten sen kykyä paikantaa virheitä. [19] CoT-lähestymistapa voidaan viedä pidemmälle iteratiivisen elinkaaren ja esimerkiksi harvojen esimerkkien eli FSL:n avulla. LLM:ää kehotetaan iteraation aikana toistuvasti CoT-kehotteilla, joiden tuloksista LLM kerää lupaavimmat generoidut korjaukset ja hyödyntää niitä uudelleen FSL-periaatteella CoT-iteraation aikana, kunnes konteksutaalisesti lähin korjaus alkuperäiseen vihreelliseen koodiin löydetään. [18]. On myös todettu, että suunnittelun perustelujen lisääminen kehotteeseen korostaa kehotettavan LLM:n koodin semanttista ymmärrystä huomattavasti. Koska dekooderipohjaiset LLM:t ovat eteviä luonnollisen kielen käsittelyssä, luonnollisella kielellä kirjoitetut suunnitteluperustelut toimivat ohjelmakoodin lailla arvokkaana kontekstina. [20].

3.2 Aihealuekohtainen hienosäätö

Hallusinaatiot ovat suurimpia haasteita LLM:ien hyödyntämisessä APR-prosessiin. Etenkin dekooderipohjaisilla LLM:illä on generatiivisten kykyjensä vuoksi taipumus tuottaa joko semanttisesti tai kontekstuaalisesti virheellistä tekstiä tai koodia. Ohjelmakoodin yhteydessä, missä faktuaalisuus on perimmäinen vaatimus toimivalle

koodille, semanttisesti tai kontekstuaalisesti virheelliset generoidut korjaukset voivat olla äärimmäisen haitallisia ja vaikeasti korjattavia. [21]

Hallusinaatioita esiintyy useimmiten tilanteissa, joissa kielimallilla ei ole riittävästi aihealuekohtaista tietoa käyttökontekstista. Tutkimuksissa on osoitettu, että LLM:t ovat alttiimpia hallusinaatioille tehtävissä, joihin liittyvää dataa esiintyy harvoin niissä laajoissa aineistoissa, joilla mallit on koulutettu. Aihealuekohtaisen hienosäädön (engl. Domain-specific fine-tuning) avulla malleja voidaan kouluttaa halutuille aihealueille, mikä vähentää hallusinaatioiden riskiä. [21] Lisäksi LLM:ien tarkkuutta tiettyyn tehtävään voidaan parantaa huomattavasti kouluttamalla niitä aihealue- tai tehtäväkohtaisesti. [4] [16] On hyvä huomata, että hienosäätö ei ole APR:lle ominainen prosessi, vaan sitä hyödynnetään kaikissa LLM:ien sovelluskohteissa, kun tarvitaan aihealuekohtaista tarkennusta.

Koko mallin hienosäätö (engl. Full Model Fine-Tuning, FMFT) on suoraviivainen tapa mukauttaa malleja tiettyyn tarkoitusperään. FMFT:ssä LLM koulutetaan jonkin aihepiirin tietoaineistoilla. Siten varmistetaan LLM:ien tehokkuus sille osoitetussa tehtävässä. APR:ään liittyvissä tehtävissä FMFT:n valjastaminen LLM:ien hienosäätämiseen on tutkimusten mukaan mahdollistanut 46–164 % suuremman määrän vikojen korjaamista kuin LLM:illä ilman hienosäätöä [4].

FMFT takaa mallin suorituskyvyn mukaisen tehokkuuden tehtävässä, mutta siihen liittyy myös merkittäviä ongelmia. LLM:t ovat viime vuosina kasvaneet parametrimääriltään valtaviksi: ne sisältävät jopa satoja miljardeja parametreja [16]. Koska FMFT vaatii kaikkien LLM:n valtavan neuroverkon painokertoimien mukauttamisen, prosessista tulee sitä hitaampaa ja kalliimpaa, mitä suurempi LLM on kyseessä. Kokonaisen LLM:n hienosäätäminen FMFT-menetelmällä on siis laskennallisesti erittäin raskasta ja kuluttaa paljon energiaa [16] [4]. Toinen merkittävä ongelma FMFT-menetelmän hyödyntämisessä on sopivien tietoaineistojen puute. LLM:ien laajuuden vuoksi tarpeeksi suuria aihealuekohtaisia tietoaineistoja toimi-

van FMFT:n saavuttamiseksi onkin hyvin vaikea laatia. Jotta lopputulos olisi mahdollisimman toimiva, tietoaineistojen tulisi olla tarpeeksi kattavia, laadukkaita ja ajankohtaisia. [16]

FMFT:stä koituvien ongelmien ratkaisemiseksi on kehitelty monia erilaisia tapoja hienosäätää LLM:iä aihealuekohtaisiin tehtäviin. Parametrien kannalta tehokas hienosäätö tai kevyt hienosäätö (engl. Parameter Efficient Fine-Tuning, PEFT) mahdollistaa LLM:n mukauttamisen tehokkaammin, edullisemmin ja nopeammin kuin FMFT [4]. PEFT itsessään on sateenvarjotermi ja sisältää monia lupaavia toisistaan eroavia lähestymistapoja hienosäätöön. PEFT-menetelmät perustuvat siihen, että runkona toimivan esikoulutetun LLM:n olemassaolevia parametreja ei muokata, vaan mukauttaminen kohdistetaan ainoastaan PEFT-menetelmien lisäparametreille, joiden määrän ei tarvitse ylittää 1–5 % koko LLM:n alkuperäisten parametrien määrästä [4].

4 APR-prosessin vaiheet

4.1 Vian paikannus

Vian paikantaminen koodista on avainasemassa ensimmäisenä askeleena kaikissa toimivissa automaattisissa ohjelmakorjausprosesseissa [22]. Tässä luvussa esitellään toimivan APR-prosessin mahdollistavan FL-prosessin tärkeimmät vaatimukset, menetelmät ja haasteet.

FL-vaiheen päätarkoituksena on kaventaa hakuavaruutta PR-vaihetta varten: jotta virhe voidaan korjata, sen syy ja sijainti täytyy tietää tarkasti. Tarkempi virheen sijaintitieto mahdollistaa täsmällisemmän korjauksen. Nykyaikaiset ohjelmistokehitysprosessit ovat usein erittäin laajoja ja sisältävät monimutkaista koodia. Kun kehitysprosessien laajuus kasvaa, hakuavaruuden rajaamisesta tulee keskeinen tekijä APR-prosessin tehostamisessa. [23]

Testit tarjoavat FL-prosessille arvokasta informaatiota virheen laadusta ja sijainnista. Testit tarkoittavat ohjelman osia, jotka tarkistavat, toimiiko koodi odotetulla tavalla. Niiden avulla pyritään estämään virheiden päätyminen tuotantoon [24]. LLM-pohjainen vianpaikannus voidaan jakaa kahteen kategoriaan sen perusteella, hyödyntävätkö menetelmät korjattavan ohjelman testikattavuutta toiminnassaan vai eivät.

4.1.1 Perinteiset FL-menetelmät

Ennen LLM-pohjaisten menetelmien käsittelemistä on tarpeen nostaa esiin merkittävimmät perinteiseksi luokitellut FL-prosessin lähestymistavat, sillä niillä on monia hyödyllisiä ominaisuuksia ja niitä käytetään usein pohjana hybridimenetelmille.

Spektrianalyysiin pohjautuva vian paikannus (engl. Spectrum-based fault-localization, SBFL) on osoittautunut tehokkaimmaksi perinteiseksi testeihin perustuvaksi FL-menetelmäksi. SBFL arvioi koodirivien virhealttiutta sen perusteella, kuinka usein ne esiintyvät epäonnistuneissa testitapauksissa suhteessa onnistuneisiin tapauksiin. SBFL vaatii kattavaa testidataa sekä läpäisevistä että epäonnistuvista testeistä, jotta se kykenee paikallistamaan vian todellisen sijainnin. [22] [24] Tutkijoiden keskuudessa suosittuja SBFL-menetelmiä ovat Ochiai [25] ja Tarantula [26]. Perinteisistä FL-menetelmistä SBFL on tämän tutkielman kannalta olennaisin, sillä sen todetun suorituskyvyn lisäksi se on osoittanut lupaavia tuloksia LLM:ien kanssa hybridimenetelmiin sovellettuna [22] [24] [27] [19].

Mutaatioihin perustuva vian paikannus (engl. Mutation-based fault-localization, MBFL) on toinen perinteinen lähestymistapa FL:ään. Se perustuu koodirivien mutaatioiden eli pienten tahallisten muutosten ja alkuperäisten koodirivien vertailuun virheen tarkan sijainnin löytämiseksi. MBFL on intensiivisistä laskennallisista vaatimuksistaan johtuen SBFL:ää huomattavasti monimutkaisempi prosessi ja sen vuoksi tehoton useissa tapauksissa. [12] Suosittuja MBFL-menetelmiä ovat FIFL [28] ja Metallaxis [29].

LLM-pohjaiset FL-tekniikat voidaan lukea osaksi koneoppimista hyödyntävää vian paikannusta (engl. Machine Learning Fault Localization, MLFL). Perinteiset MLFL-menetelmät ovat testipohjaisia, koska ne soveltavat koneoppimista ja syväoppimista jonkin perinteisen FL-menetelmän, kuten SBFL tai MBFL, tuloksiin. Kaikista lupaavimpia MLFL-ratkaisuja nykypäivänä ovat kuitenkin LLM-pohjaiset ratkaisut: niiden on osoitettu kykenevän suoriutumaan tehtävistään perinteisiä mene-

telmiä paremmin [12] [22]. Huomionarvoisia LLM:iä edeltäviä huippuluokan oppimiseen perustuvia menetelmiä ovat esimerkiksi DeepFL [30] ja MTL-Transfer-FL [23].

4.1.2 Testaukseen perustuva virheiden paikantaminen kielimalleilla

Viimeisimmissä tutkimuksissa, jotka kohdistuvat LLM:ien avulla automatisoituun FL-prosessiin, testaukseen perustuvat FL-menetelmät ovat osoittaneet lupaavia tuloksia. Huomattavan suuri osa tähän tutkielmaan seulotuista tutkimuksista käsittelee hybridimenetelmiä, joissa perinteisiä APR- tai FL-menetelmiä pyritään tehostamaan LLM:ien avulla. Perinteisillä FL-menetelmillä, kuten SBFL, on helppo luoda toimiva pohja automaattiselle FL-prosessille, jota on luonteva jatkaa LLM-pohjaisella komponentilla. Hybridimenetelmää käyttämällä voidaan valita kummankin järjestelmän vahvat puolet [24].

Tässä tutkielmassa käsitellään kolmea vuonna 2024 esitettyä, hyviä tuloksia saavuttanutta, LLM-pohjaista testaukseen perustuvaa MLFL-tekniikkaa: AutoFL [24], FuseFL [19] ja SemiAutoFL [22]. Kaikki menetelmät hyödyntävät OpenAI:n dekooderipohjaista GPT-sarjaa.

AutoFL on täysin automaattinen FL-menetelmä, joka kykenee suorittamaan FL-prosessin vain yhdellä epäonnistuneella testillä, sillä epäonnistuneilla testeillä on mahdollista saada paljon informaatiota virheen laadusta ja sijainnista. Tämä vähentää tarvetta laajalle testikattavuudelle. Ensin menetelmä ohjaa LLM:n hyödyntämään ohjelman omia funktiokutsuja, joiden avulla se kerää testifunktioista FL-prosessiin tarvitsemansa tietoaineiston. [24]

Vastaavasti täysin automaattisen FuseFL:n tavoitteena on virheiden paikantaminen hyödyntämällä useita tietolähteitä, kuten SBFL-prosessin tuloksia, testitapausten suoritustuloksia sekä ohjelmakoodin suunnitteluperusteluita. FuseFL laatii

kehotteet ajatusketjuina, jonka jälkeen sen LLM-komponentti pyrkii paikantamaan virheen tarkan sijainnin ohjelmakoodista. [19]

SemiAutoFL on AutoFL:n pohjalta syntynyt puoliautomaattinen lähestymistapa. Puoliautomaattisuus tarkoittaa, että sen toiminnan elinkaareen kuuluu myös manuaalinen vaihe. SemiAutoFL:n LLM:lle annetaan kehotteena osa ohjelmakoodia ja siihen liittyvät testit. Sen jälkeen LLM saattaa pyytää kehittäjältä lisätietoja oleellisista ohjelmakoodin osista, mikä tekee menetelmästä puoliautomaattisen. [22]

Merkittävä haaste perinteisten FL-menetelmien käytettävyydelle on se, etteivät ne tarjoa perusteluja sille, miksi tietyt koodirivit on merkitty epäilyttäviksi [19] [24]. Tämä johtaa myös siihen, ettei ratkaisevaa kontekstuaalista informaatiota virheen laadusta välity kehittäjälle, tai APR:n tapauksessa korjausta tekevän LLM:n kehotteeseen, mikä heikentää menetelmän tarkkuutta ja siten sen käyttökelpoisuutta merkittävästi.

AutoFL:ää ja FuseFL:ää käsittelevissä julkaisuissa nousee esiin perinteisten FL-menetelmien päätöksenteon heikko selitettävyys. Sen sijaan AutoFL ja FuseFL pyrkivät virheen paikantamistulostensa lisäksi generoimaan dekooderipohjaisella LLM:illään perustelut tekemilleen päätöksille. [24] [19].

4.1.3 Kielimallien suora hyödyntäminen virheiden paikantamiseen

Perinteiset FL-menetelmät, kuten SBFL ja MBFL, perustuvat testeistä saatuun tietoon, kuten informaatioon virheen laadusta ja sijainnista. Sen vuoksi myös niiden päälle rakennetut LLM:iä hyödyntävät hybridimentelmät vaativat testeistä saatua dataa. Testidatan hyödyntäminen on luontevaa ja tehokasta, sillä sen kerääminen on helppoa, ja se sisältää runsaasti olennaista tietoa koodin ajonaikaisesta käyttäytymisestä. Testidatan hyödyntämisessä on kuitenkin tiettyjä ratkaisevia heikkouksia.

Ensinnäkin testidataa hyödyntävät FL-menetelmät edellyttävät, että saatavilla on kattavia ja luotettavia testitapauksia. Testeihin perustuva FL voi helposti epäonnistua ulkoisten tekijöiden, kuten huonosti kirjoitettujen testien vuoksi, vaikka prosessi itsessään olisi toimiva. Esimerkiksi SBFL luottaa aina testikattavuuteen ja on siten herkkä testisarjan ominaisuuksille. Virhe voi helposti päästä tuotantoon, jos testitapaukset eivät sisällä sitä lainkaan. [12]

Edellämainituista syistä LLM:ien yleistymisen myötä virheiden paikantamiseen on syntynyt uusi, lupaava lähestymistapa, jossa LLM:iä hyödynnetään suoraan koodin analysoimiseen. Yang ym. [12] esittelevät LLMAO:n, automaattisen LLM-pohjaisen vikojen paikannusmenetelmän, joka ei perustu testaukseen. LLMAO paikantaa virheet rivin tarkkuudella ja toteuttaa vikojen paikannusprosessin täysin ilman testidataa. Se hyödyntää pelkästään suurten kielimallien kykyä analysoida koodia ja tunnistaa mahdollisia virhekohtia. [12]

Toisin kuin enkooderiarkkitehtuuriin pohjautuvilla LLM:illä, dekooderiarkkitehtuuriin perustuvilla LLM:illä on merkittävä heikkous tekstin semanttisten yhteyksien tunnistamisessa. Koska ne on suunniteltu ensisijaisesti tekstin generointiin, ne hyödyntävät kontekstia vain vasemmalla puolellaan olevista tokeneista, eli aiemmasta tekstistä. Tämä rajoitus heikentää niiden kykyä hahmottaa laajempia semanttisia yhteyksiä erityisesti tilanteissa, joissa ymmärrys edellyttäisi myös tulevan tekstin huomioimista. Tämä on merkittävä heikkous, kun FL-prosessissa luotetaan täysin LLM:n kykyyn analysoida koodia virheiden varalta. [12]

Vaikka enkooderipohjaiset LLM:t omaavat ennestään kaksisuuntaisen koodin ymmärryksen, LLMAO pohjautuu dekooderiarkkitehtuuriin perustuvaan LLM:ään ja pyrkii mukauttamaan sen FL-tehtäviin hyödyntämällä runkomallin päälle hienosäädettyjä kaksisuuntaisia sovitintasoja (engl. Adapter Layer) [12]. Sovitintasojen avulla mukauttaminen kuuluu kevyeeseen hienosäätöön, ja niiden avulla voidaan te-

hokkaasti muokata runkona toimivaa LLM:ää ilman tarvetta päivittää sen painoja [12].

LLMAO on osoittautunut erittäin kilpailukykyiseksi muihin FL-menetelmiin verrattuna, ja se on saavuttanut merkittäviä tuloksia virheiden paikantamisessa ilman testejä. Mallia on testattu Defects4J-datasetillä ja se on osoittautunut huomattavasti tehokkaammaksi kuin SBFL-menetelmät, kuten Ochiai, ja MLFL-menetelmät, kuten MTL-Transferia edeltävä malli TransferFL. [12]

Toggle on tämän tutkielman ainoa LLM-pohjainen APR-menetelmä, joka hyödyntää eri arkkitehtuurin omaavia LLM:iä FL- ja PR-vaiheessaan. Togglen FL-prosessi hyödyntää enkooderi-dekooderipohjaisen CodeT5-mallin [7] enkooderikomponenttia ja mahdollistaa virheiden paikantamisen tokenien tarkkuudella saavuttaen siten hienojakoisemman FL-prosessin kuin dekooderipohjainen LLMAO. PR-vaiheessa Toggle hyödyntää dekooderipohjaista LLM:ää. [17]

Toisin kuin LLMAO, Togglen FL-prosessi ottaa syötteekseen jo valmiiksi paikannetun viallisen funktion, joka on löydetty esimerkiksi Ochiai SBFL-menetelmällä. Toggle voidaan siten luokitella hybridimenetelmäksi. Toggle hyödyntää LLMAO:n tavoin suoraan LLM:ien kykyä analysoida koodia virheiden paikantamiseksi, eikä siten myöskään vaadi testejä viallisen kohdefunktion paikantamisen jälkeen. Togglen LLM on hienosäädetty tunnistamaan virheettömän etuliitteen (engl. prefix) ja jälkiliitteen (engl. suffix) rajat, joiden väliin jäävät tokenit tunnistetaan virheellisiksi. Hienojakoinen virheiden paikannus on Togglen tärkein keino sen virheiden korjausprosessin tarkentamiseen, jota käsitellään luvussa 4.2 [17].

Togglen pääasiallisen vian paikannusmallin ja korjausmallin lisäksi sillä on valinnainen CodeT5-malliin perustuva säätömoduuli, jolla virheen paikannuksen tuloksia pyritään tarkentamaan ennen korjausvaihetta. Kokeelliset tulokset osoittavat, että säätömalli parantaa johdonmukaisesti LLM:ien virheiden korjaustarkkuutta. [17]

4.2 Ohjelmakorjaus

Perinteiset APR-tekniikat voidaan luokitella heuristiikkapohjaisiin, rajoitepohjaisiin, mallipohjaisiin ja oppimispohjaisiin lähestymistapoihin. [18] [31] Heuristiikkapohjaiset lähestymistavat, kuten ARJA-e, [32] hyödyntävät esimerkiksi evoluutiopohjaista geneettistä ohjelmointia ja satunnaishakua virheiden korjaamiseksi. Mallipohjaiset APR-tekniikat kuten Tbar [33] voivat korjata suuren määrän virheitä tehokkaasti ennalta määriteltyjä malleja noudattaen. Mallipohjaiset APR-menetelmät ovat heikosti yleistettäviä ja epäkäytännöllisiä, sillä niiden toiminta rajoittuu näihin ennalta määriteltyihin malleihin. [18].

Viime vuosina on tutkittu laajasti myös syväoppimiseen ja neuroverkkoihin perustuvia oppimispohjaisia lähestymistapoja. Esimerkiksi hybridimenetelmä MTL-Transfer-PR hyödyntää neuroverkkopohjaista konekäännöskomponenttia (engl. Neural Machine Transfer, NMT) paikkakandidaattien ennustamiseen ja sillä on saavutettu huomattavan kilpailukykyisiä tuloksia verrattuna muihin oppimiseen perustuviin menetelmiin. [23] NMT-lähestymistavat ovat myös huomattavan paljon alttiimpia opetusaineistosta johtuvalle ylivirittymiselle, kuin LLM-pohjaiset lähestymistavat, sillä niillä ei ole käytettävissä LLM:ien valtavia tietoaineistoja. [31]. Tämän vuoksi NMT-menetelmät ovat vahvasti riippuvaisia virhetietoaineistoista, ja siten melko epäkäytännöllisiä. [18] LLM-pohjaiset ratkaisut ovat myös osoittautuneet merkittävän paljon tehokkaammiksi kuin syväoppimiseen perustuvat ei-LLM-pohjaiset lähestymistavat [17].

Tässä tutkielmassa esitellään luvussa 4.1 esitellyn Toggle APR -menetelmän lisäksi kolme muuta vuosina 2024-2025 ehdotettua LLM:iä hyödyntävää APR-menetelmää: DRCodePilot [20], GiantRepair [10] ja ThinkRepair [18]. Kaikilla edellämainituilla menetelmillä on oma lähestymistapansa APR-prosessiin. Lisäksi ne ovat kaikki saavuttaneet huipputason tuloksia omina julkaisuajankohtinaan.

Tämän luvun tarkoitus on käsitellä nimenomaan APR-menetelmien PR-vaihetta, jossa ohjelmakoodiin on tarkoitus generoida mahdollisimman oikeellisia korjauksia (patch) ja tarkastella PR-vaiheen hyötypotentiaalia. Huomionarvoista on, että kaikki mainitut mallit hyödyntävät PR-vaiheessaan jotakin dekooderiarkkitehtuuriin pohjautuvaa LLM:ää niiden tarjoamien generatiivisten etujen vuoksi. Kaikkien neljän menetelmän suorituskykyä on myös pääasiassa arvioitu niiden kyvyllä korjata virheitä onnistuneesti.

Kyseiset neljä APR-menetelmää eivät suinkaan ole ainoat LLM-pohjaiset menetelmät, vaan ne on valittu pitkälti niiden toisistaan eriävien lähestymistapojen, julkaisuajankohtien ja niiden saavuttamien huomattavien tulosten vuoksi. Muita LLM:iin pohjautuvia APR-menetelmiä ovat esimerkiksi SelfAPR [34], RING [35], Mentat [36], AlphaRepair [37], ChatRepair [38]. Myös OpenAI:n GPT-sarjan [3] dekooderipohjaisia malleja on testattu paljon APR-prosessissa.

Toggle on APR-menetelmä, joka hyödyntää virheiden paikantamiseen enkooderipohjaista LLM:ää ja niiden korjaamiseen dekooderipohjaista LLM:ää. PR-vaiheessaan Toggle hyödyntää erikokoisia dekooderipohjaisia LLM:iä, kuten GPT-sarjan eri malleja. Togglen PR-vaiheessa generoitujen korjausten tarkkuus perustuu moneen eri tekijään. Kehotteena korjausmalli saa valmiiksi paikannetun viallisen funktion, ja FL-vaiheessa paikannetun, tarvittaessa säätömallilla säädetyn virheen sijainnin funktion sisältä. Täten virheen sijainti on hyvin tarkasti tiedossa, mikä tarkentaa korjausta huomattavasti. Korjausvaiheessa hyödynnetyt LLM:t on myös erikseen hienosäädetty tehtävää varten. Tutkimus osoittaa, että suuremmilla LLM:illä on johdonmukaisesti parempi korjaustarkkuus hienosäädön jälkeen. Hossain ym. toteavat artikkelissaan, että Defects4J:llä testattuna Toggle saavutti johdonmukaisesti muita menetelmiä parempia tuloksia ohjelmakorjauksessa. [17]

DRCodePilot on APR-menetelmä joka on kehitetty tehokkaan LLM:n, GPT-4 turbo -kielimallin [39] ympärille. DRCodePilotin merkittävin kehitysaskel on suun-

nittelun perustelujen sisällyttäminen GPT-4:lle annettuun kehotteeseen. DRCodePilotin FL-prosessi hyödyntää runkonaan toimivaa tehokasta GPT-4-mallia ja ohjelmakoodin suunnittelun perusteluja vikojen paikantamiseen suoraan ohjelmakoodista. Tämän jälkeen DRCodePilot generoi korjausluonnoksen ja käyttää palauteperusteista, itseohjautuvaa kehikkoa (engl. framework), jossa GPT-4 matkii ihmiskehittäjien suunnnitteluperusteluja ohjelmistokehitysprosessin eri vaiheissa ja tarkentaa tuotoksiaan saamansa palautteen perusteella. [20]

DRCodePilotin saavuttamat kokeelliset tulokset ovat kuitenkin hyvin vaikuttavia. Sitä on testattu vertailuaineistossa, joka koostuu 938 virhe-korjausparista kahdesta avoimen lähdekoodin tietovarastosta GitHubissa ja Jirassa. Tällä aineistolla DRCodePilot saavuttaa johdonmukaisesti paremmat tulokset kuin pelkkä GPT-4. Lisäksi Zhao ym. mukaan DRCodePilot on uraa uurtava APR-menetelmä suunnitteluperustelujen yhdistämisessä LLM:ien kehotteisiin APR-prosessissa [20].

GiantRepair on tässä tutkielmassa käsitellyistä APR-menetelmistä uusin. Se on hybridimenetelmä, eikä se siten nojaudu kokonaan LLM:ien generatiivisiin ominaisuuksiin, vaan hyödyntää niiden ohella myös ohjelma-analyysia LLM:ien tulosten tarkentamiseksi. GiantRepair hyödyntää varsinaisessa korjausvaiheessaan virheitä sisältävää koodia ja siihen kohdistettuja dekooderipohjaisten LLM:ien laatimia korjausyrityksiä. [10]

Fengjie ym. [10] nostavat esille merkittävän heikkouden viimeaikaisessa APR-tutkimuksessa, jota GiantRepair pyrkii korjaamaan. Olemassa olevat APR-menetelmät hyödyntävät usein LLM:iä suoraan korjauksen generoimiseen ilman erillistä optimointia ja nojautuvat täysin ennen korjauksen varsinaista generointia suoritettuihin toimenpiteisiin, kuten laadittuihin kehotteisiin ja hienosäätöön. Tämä johtaa alhaisempaan suorituskykyyn ja tarkkuuteen korjauksissa. Fengjie ym. mukaan LLM:ien suoraan tuottamat korjausehdotukset tarjoavat kuitenkin arvokasta

informaatiota korjausprosessille, vaikka korjausehdotukset eivät välttämättä olisikaan täysin oikeita. [10]

GiantRepairin PR-prosessi toimii kolmessa päävaiheessa: ensimmäiseksi GiantRepair konstruoi LLM:ien ehdottamista korjauksista korjausrungon (engl. skeleton). Tämä tapahtuu eristämällä LLM:ien ehdottamat koodimuutokset ja viallisen koodin puumalliin. [10] On hyvä huomata, että tämä vertailu tapahtuu lauseen tasolla (engl. statement-level), eli tunnistus on epätarkempi kuin esimerkiksi Togglen hyödyntämä tokenien tasolla tapahtuva vihreellisen koodin tunnistus.

Toinen vaihe on korjausten luominen kontekstin mukaan. Luoduista korjausrungoista pyritään luomaan korjauksille ehdokaspaikkoja staattisen analyysin avulla. Paikan ohjelmakohtainen konteksti pyritään huomioimaan mahdollisimman kattavasti vertailemalla ehdokaspaikan ja virheellisen koodin sekä ehdokaspaikan ja LLM:n generoiman korjauksen samankaltaisuutta. Yhteiset koodielementit pyritään säilyttämään ja samankaltaisia ehdokaspaikkoja pyritään suosimaan. [10]

Lopuksi GiantRepair arvioi ehdokaspaikkojen oikeellisuuden ajamalla niille erikseen laaditut testit. Paikat, jotka läpäisevät kaikki testit, luokitellaan järkeenkäyviksi (engl. plausible) korjauksiksi. Järkeenkäyviä paikkoja vertaillaan vielä keskenään, painottaen samankaltaisuutta alkuperäiseen koodiin ja LLM:ien generoimiin pohjustaviin korjauksiin. [10]

Fengjie ym. [10] nostavat esille myös toisen tärkeän huomion APR-prosessin käytännöllisestä hyötypotentiaalista. Useimmiten automaattisia PR-menetelmiä on testattu täydelliseen FL:ään perustuen, eli virheen sijainti tiedetään PR-vaiheessa varmasti. Täydellisen FL:än hyödyntäminen testauksessa on luontevaa, sillä se eliminoi FL-vaiheesta koituvia epätarkkuuksia ja tarkentaa siten itse testattavan PR-menetelmän arviointia. Täydellinen FL-prosessi ei kuitenkaan ole realistinen kokonaisen APR-prosessin yhteydessä, joka rajoittaa sillä testattujen PR-menetelmien reaalimaailman potentiaalia. [10]

GiantRepairia on testattu sekä täydelliseen että automatisoituun epätäydelliseen FL-prosessiin perustuen. Kummassakin tapauksessa GiantRepair on osoittautunut paremmaksi ratkaisuksi kuin suoraan LLM:ien generoimiin korjauksiin nojautuvat menetelmät. Defects4J v1.2 -viitekehyksessä se korjasi keskimäärin 27,78 % enemmän bugeja ja Defects4J v2.0 -viitekehyksessä 23,40 % enemmän bugeja verrattuna siihen, että LLM:ien tuottamia korjauksia olisi käytetty suoraan. Olettaen että FL-prosessi on täydellinen, GiantRepair korjasi vähintään 42 virhettä enemmän kuin muut huipputasoiset APR-menetelmät. Realistisemmassa tapauksessa, automatisoidun eli epätäydellisen FL:n yhteydessä, GiantRepair korjasi silti vähintään 7 virhettä enemmän kuin parhaiten suoriutunut verrokkimenetelmä. [10]

ThinkRepair on viimeinen tässä tutkielmassa käsiteltävä APR-lähestymistapa. Se perustuu kehotteiden laatimiseen CoT-menetelmällä ja iteratiiviseen elinkaareen, missä sen kahta päävaihetta voidaan toistaa kunnes toimiva korjaus saavutetaan. [18]

Ensimmäisessä eli tiedonkeruuvaihessa ThinkRepair kerää ajatusketjuja. Ensin LLM:lle laaditaan kehote, jossa sille kerrotaan konkreettisesti korjaustehtävän kuvaus ja CoT-fraasi: "Let's think step by step". ThinkRepair on lisäksi suunniteltu kokonaisten yksittäisten funktioiden korjaamiseen, eikä sillä ole erillistä FL-vaihetta. Kehote sisältää myös viallisen, korjattavan funktion. Kehotteen avulla LLM tuottaa mahdollisia korjauksia koko annetusta funktiosta ja niihin liittyviä ajatusketjuja, joiden tarkoitus on kuvata päättelyprosessia virheen tunnistamisessa ja korjaamisessa. Tuloksena on kokoelma esimerkkejä jotka testataan esimerkiksi alkuperäisen tietokannan kuten Defects4J tukemilla testisarjoilla. Ainoastaan ne korjatut funktiot jotka läpäisevät kaikki testit säilytetään vaiheen lopussa. [18]

Korjausvaiheessa virhe on tarkoitus korjata aiemmasta vaiheesta saadun tietopohjan avulla. Tietopohjasta valitaan muutama esimerkki harvan otoksen oppimiseen: viallinen funktio, ehdotettu korjaus ja sen ajatusketju. Valinta perustuu vialli-

sen funktion semanttiseen samankaltaisuuteen tietopohjan esimerkkien kanssa. Valinnan tulos ja viallinen kohdefunktio yhdistetään sitten uuteen kehotteeseen, jotta LLM saa kontekstia iteraation aiemmissa sykleissä epäonnistuneista ja onnistuneista korjauksista ja niiden ajatusketjuista. [18]

ThinkRepairin iteratiivinen lähestymistapa korjaukseen on saavuttaa huippuluokan tuloksia Defects4J V1.2 -datasetissä täsmällisellä vian sijainnilla, saavuttaen 27 %:sta jopa 344 %:iin parempia tuloksia kuin aiemmat menetelmät. ThinkRepairin on osoitettu myös voivan korjata tasaisesti enemmän yksilöllisiä virheitä Defects4J V1.2 ja V2.0 -dataseteistä, joita muut aiemmin julkaistut LLM-pohjaiset APR-menetelmät, kuten AlphaRepair [37] ja BaseChatGPT [38], eivät kyenneet korjaamaan. [18]

5 Pohdinta

Kuten tutkielmassa käy ilmi, LLM-pohjaiset APR-menetelmät ovat osoittautuneet tehokkaammiksi kuin perinteiset menetelmät. Paremman tehokkuuden ja tarkkuuden lisäksi merkittävimpiä etuja LLM-pohjaisilla menetelmillä ovat niiden joustavuus ja yleistämiskyky. LLM:ien ansiosta APR-menetelmiä voidaan nykyään soveltaa useisiin eri ohjelmointikieliin. LLM:ien laaja peruskoulutus mahdollistaa monikielisyyden. Se on huomattava edistysaskel, joka ei ole ollut saavutettavissa perinteisillä menetelmillä.

Dekooderipohjaiset LLM:t edustavat tällä hetkellä kooltaan suurimpia LLM:iä. Dekooderipohjaisten LLM:ien menestys on selvästi vaikuttanut voimakkaasti APR-menetelmien kehitykseen. On osoitettu, että ohjelmistokehitystehtävissä LLM:n suorityskyky paranee johdonmukaisesti sen koon kasvaessa [12]. FL-vaiheessa, jossa pyritään rajaamaan hakuavaruutta ja paikantamaan virheitä semanttisen koodianalyysin avulla, dekooderimallien yksisuuntainen (vasemmalta oikealle) rakenne rajoittaa kykyä hyödyntää täyttä kontekstia virhepaikkojen paikantamisessa. [12]

Vaikka enkooderipohjaiset LLM:t tarjoavat tarkempia valmiuksia semanttiseen koodin ymmärtämiseen, neljä viidestä tässä tutkielmassa käsitellystä FL-vaiheen menetelmästä hyödyntää dekooderipohjaista arkkitehtuuria (ks. Taulukko 5.1). Näillä menetelmillä on kyky vähentää riippuvuutta ulkoisista lähteistä, kuten testikattavuudesta, ja tuottaa perusteltuja selityksiä paikannuspäätöksille.

Taulukko 5.1: FL-menetelmien ryhmittely ominaisuuksien mukaan

FL	Dekooderipohjainen	Hybridimenetelmä	Hyödyntää testejä
AutoFL	X	X	X
FuseFL	X	X	X
SemiAutoFL	X	X	X
LLMAO	X		
ToggleFL		X	X

PR-vaiheessa dekooderipohjaisten mallien generointikyky korostuu erityisesti korjauksen tuottamisessa, mikä selittää niiden laajan käytön (ks. Taulukko 5.2). Dekooderimallien on osoitettu olevan myös hyvin alttiita kehotteille ja hyötyvän merkittävästi erilaisista kehotelähestymistavoista, kuten ajatusketjuista ja suunnitteluperusteluista. [18] [20] Tutkielmassa tarkastelluista menetelmistä kaikki neljä hyödyntävät dekooderipohjaisia LLM:iä ohjelmakorjauksessa.

Taulukko 5.2: PR-menetelmien ryhmittely ominaisuuksien mukaan

PR	Dekooderipohjainen	Hybridimenetelmä	Hyödyntää testejä
TogglePR	X	X	
GiantRepair	X	X	
ThinkRepair	X		X
DRCodePilot	X		X

Hybridimenetelmät vaikuttavat kevyeltä ja tehokkaalta tavalta parantaa APR-prosessin kumpaakin päävaihetta, sillä jo olemassa olevien menetelmien pohjalta on helppo rakentaa LLM:iä hyödyntäviä komponentteja, jotka tehostavat prosessia ja korjaavat sen perinteisten komponenttien puutteita. AutoFL [24] ja FuseFL [19] ovat kumpikin hybridimenetelmiä, jotka keskittyvät tulosten selitettävyyteen. Menetelmän selitettävyys parantaa tuloksen laatua ja kontekstuaalista informaatiota. On osoitettu, että kehotteen laatu parantaa johdonmukaisesti PR-vaiheen tarkkuutta [18].

Hybridimenetelmät pohjautuvat vahvasti perinteisiin menetelmiin, joiden toiminta perustuu testeistä saatuun dataan. [19] Testipohjaisuus aiheuttaa kuitenkin APR-prosessille selkeitä ulkoisia haasteita, jotka eivät riipu prosessin sisäisistä vai-

heista. Testipohjaista APR:ää ei voida soveltaa edes 90 % ohjelmistokehitysprosessissa ilmenevistä virheistä, koska tarvittavia virheen osoittavia testejä ei ole olemassa ennen kuin ihminen on korjannut virheen. [9]. Tässä tutkielmassa käsitellyistä LLM-pohjaisista ratkaisuista huomattavan suuri osa hyödyntää testejä joko FL-vaiheessa virheiden paikantamiseen tai PR-vaiheessa korjausten validoimiseen, joten ongelma on huomattava. Luonteva jatkotutkimuskohde voisi siis olla testiautomaation yhdistäminen testipohjaiseen APR:ään. Automaattisilla testeillä niihin perustuvan APR:n kattavuutta voitaisiin laajentaa ohjelman osiin, joita ei ole vielä korjattu tai testattu.

Hallusinaatiot ovat suuri ongelma etenkin dekooderipohjaisia LLM:iä hyödynnettäessä. Niitä ilmenee usein, kun LLM:llä ei ole aihealuekohtaista informaatiota ja kontekstia aiheesta, johon sitä pyritään hyödyntämään. Hienosäädön on osoitettu olevan hallusinaatioita ehkäisevä toimenpide. [20] Hienosäätö on kuitenkin usein kallis prosessi eikä kovinkaan joustava menetelmä [4]. Lisäksi hienosäädön on osoitettu lisäävän riskiä ylivirittymiselle siihen aineistoon, jolla mallia hienosäädetään. [15] On siis aiheellista tutkia myös muita tapoja vähentää hallusinaatioita.

Mitä enemmän APR-prosessi nojautuu LLM:iin, sitä enemmän hallusinaatiot tulevat tuottamaan ongelmia. Truth-O-Meter [21] on vuonna 2024 julkaistu LLM:ien generoimaa luonnollista kieltä tarkastava järjestelmä, joka hyödyntää ulkoisia tietolähteitä fatkapohjaisuuden tarkistamiseen (engl. Evidence Retrieval, ER). Yksi jatkotutkimuskohde voisi siis olla ER-pohjaisen faktantarkistusjärjestelmän yhdistäminen APR-prosessiin hallusinaatioiden vähentämiseksi ilman APR-menetelmän LLM:än parametreja mukauttavaa hienosäätöä. ER-pohjaisen menetelmän avulla hienosäädön haasteita, kuten dataseteistä johtuvia ongelmia, ylioppimista ja alkuperäisen mallin tehokkuuden heikkenemistä, voitaisiin välttää tarkistamalla korjauksia ulkoisiin tietolähteisiin, kuten verkosta haettuihin datasetteihin, verraten. ER-pohjainen prosessi ei muuttaisi LLM:n sisäistä toimintaa mitenkään, vaan toimisi

kehotteen tavoin ulkoisena ohjeena LLM:ille niiden ominaisen joustavuuden säilyttäen.

Kehotteiden laatiminen on avainasemassa kaikissa tässä tutkielmassa käsitellyissä tutkimuksissa sekä FL- että PR-vaiheissa. Yin ym. [18] tuovat esiin, että APR-mentelmät voivat toimia ilman erityistä hienosäätöä ja tuottaa erinomaisia tuloksia pelkkien kehotteiden ja LLM:n peruskoulutuksen avulla.

Kolme neljästä PR-vaiheen mallista hyödyntää LLM:ien ohjeistamiseen pelkästään kehotteiden laatimista eri menetelmillä. Ainoastaan Toggle hienosäätää korjausmalliaan ja pyrkii mahdollisimman tarkkaan kehotteeseen hienojakoisen FL-vaiheensa ja säätömallinsa avulla [17]. Kehotteen tarkkuus vaikuttaisi olevan avainasemassa korjausten oikeellisuuden kannalta.

Kehotteiden manuaalinen laatiminen on kallista [18], eikä se myöskään tue täysin automaattisen ohjelmakorjauksen vaatimuksia, sillä APR-prosessin tulisi toimia itsenäisesti. Potentiaalinen jatkotutkimuskohde voisi siis olla automaattinen kehotteiden laatiminen LLM-pohjaisella FL-prosessilla. PR-vaiheessa manuaalisen kehotteiden laatimisen voisi korvata FL-vaiheesta automaattisesti johdetuilla kehotteilla, jotka sisältävät ainakin tehtävänannon, vian sijainnin ja muun olennaisen kontekstin.

Kehotteet jotka sisältävät ohjeita luonnollisella kielellä, kuten ajatusketjut (CoT) ja suunnitteluperustelut (DR), parantavat LLM:ien suorituskykyä merkittävästi [18] [20]. Myös näitä kehotteita olisi mahdollista laatia automaattisesti LLM-pohjaisilla menetelmillä. Esimerkiksi AutoFL ja FuseFL ovat kumpikin FL-hybridimenetelmiä, jotka keskittyvät generoimaan perusteluja ohjelmakoodista paikantamilleen virheille. Testikäytössä FuseFL:n ja AutoFL:n generoimat perustelut ovat osoittautuneet kehittäjille hyödyllisiksi. [24] [19] Näin ollen samoilla perusteluilla voitaisiin laatia kehotteita LLM-pohjaisille PR-vaiheen menetelmille. Tämä vähentäisi tarvetta kalliille ja aikaavievälle manuaalisten kehotteiden laatimiselle. PR-vaiheen menetelmien

testaus kehotteilla, jotka sisältävät täydellisen FL-tuloksen, ei ole realistista [10], joten automaattisesti laadittuihin kehotteisiin perustuva lähestymistapa parantaisi PR-vaiheen testauksen todenmukaisuutta.

6 Yhteenveto

Tämän tutkielma on uusimpien LLM-pohjaisten APR-ratkaisujen kirjallisuuskatsaus. Sen lisäksi tavoitteena on arvioida LLM-pohjaisten APR-ratkaisujen suorituskykyä ja käytettävyyttä perinteisiin menetelmiin verrattuna.

1. Mitä on automaattinen ohjelmakorjaus ja miksi sitä tutkitaan?
2. Miten suuria kielimalleja voidaan hyödyntää automaattisessa ohjelmakorjauksessa ja miten ne vertautuvat aiempiin lähestymistapoihin?
3. Onko automaattinen ohjelmakorjaus käytettävä ratkaisu ja miten LLM:ien integroiminen APR-prosessiin vaikuttaa sen käytettävyyteen?

Automaattinen ohjelmakorjaus pyrkii korjaamaan ohjelmakoodin virheitä ilman ihmisen väliintuloa. APR-prosessi jakautuu kahteen toiminnalliseen päävaiheeseen. Vian paikannusvaiheen eli FL-vaiheen tarkoitus on rajoittaa prosessin hakuavaruutta mahdollisimman tarkasti, ja kerätä olennaista tietoa virheen semantiikasta ja kontekstista. Korjauksen generoimisvaiheen, eli PR-vaiheen, tarkoituksena on generoida mahdollisimman tarkka ja toimiva korjaus ohjelmakoodiin FL-vaiheen tuloksia hyödyntäen.

Toimivan APR-prosessin tarve on merkittävä. On osoitettu, että debuggaus, testaus ja ohjelman toimivuuden varmistaminen voivat viedä jopa kolme neljäsosaa kehitysbudjetista ja kuluttaa jopa puolet kehittäjien työajasta. [9]

Jotta LLM:ien täysi potentiaali APR-sovelluksissa voidaan hyödyntää, niiden toimintaa on mukautettava. Välttämätön ensimmäinen askel on suunnitella kehote, joka ohjeistaa LLM:n mahdollisimman kattavasti ja tarkasti suorittamaan sille annettu tehtävä. Tutkimukset osoittavat, että kehotteiden laatu ja sisältö vaikuttavat olennaisesti LLM:n suorituskykyyn APR-prosessissa. Kehotteiden lisäksi LLM:iä voidaan hienosäätää aihealuekohtaisella datalla, mikä tehostaa sen suorituskykyä kyseiseen aihepiiriin liittyvissä tehtävissä.

Hybridimenetelmät, joissa yhdistetään perinteisiä ohjelma-analyysitekniikoita LLM-pohjaisiin menetelmiin, ovat tutkijoiden keskuudessa suosittu lähestymistapa APR-menetelmiin, sillä hybridimenetelmien avulla voidaan valikoida luontevasti sekä LLM:n että kunkin perinteisen komponentin vahvuudet.

Kumpikin APR:n päävaihe on perinteisesti pohjautunut testikattavuuteen, joka on siten heijastunut myös useisiin hybridimenetelmiin. FL-vaiheessa testit ovat tehokas tapa kerätä olennaista dataa ohjelman ja virheen laadusta, vaikka menetelmää rajoittavat ulkoiset syyt, kuten testikattavuuden puute. PR-vaiheessa testit ovat tehokas tapa validoida generoituja korjauksia, vaikka myös PR-prosessissa voi kehittyä sama testikattavuuden puutteesta koituva ongelma. LLM:ien yleistymisen myötä on siten kehittynyt mahdollisuus hyödyntää LLM:ien luontaisia tekstinymmärryskykyjä koodin analysoimiseen suoraan ilman erillistä tarvetta testeille.

LLM-pohjaiset APR-menetelmät ovat selvästi ylittäneet perinteisten menetelmien suorituskyvyn. Lisäksi LLM:ien laajat koulutusaineistot tarjoavat APR-menetelmille merkittävän edun, kun niitä sovelletaan eri käyttötarkoituksissa. Laajat koulutusaineistot auttavat vähentämään riskiä LLM:ien ylioppimiselle.

Nykyisellä kehitystasolla APR-tekniikoiden luontevimmat käyttökohteet ovat puoliautomaattisia ratkaisuja, joissa ne tukevat kehittäjän työtä apuvälineinä. Eladawy ym. [9] osoittavat tutkimuksessaan, että oikean ehdotuksen saaminen APR:ltä auttaa kehittäjiä johdonmukaisesti ymmärtämään ohjelmakoodin virheitä. [9]

APR:n tuleva kehitys vaikuttaa lupaavalta. Tässä tutkielmassa käsiteltyjen LLM-pohjaisten APR-menetelmien parissa on havaittavissa selvää kehitystä. Vuonna 2025 julkaistu GiantRepair [10] on saavuttanut uuden huipputason Defects4J-datasetillä testattuna ja samalla se on ylittänyt vuonna 2024 julkaistun Togglen [17] saavuttamat tulokset. LLM-pohjaisten APR-menetelmien kehitys siis jatkuu ja täysin automaattisen, toimivan ja käytettävän APR-prosessin saavuttaminen on epäilemättä paljon todennäköisempää, kuin aikaisemmilla lähestymistavoilla.

Lähdeluettelo

[1] X. Hou et al., ”Large Language Models for Software Engineering: A Systematic Literature Review”, *ACM Transactions on Software Engineering and Methodology*, vol. 33, s. 1–79, 2024. DOI: 10.1145/3695988.

[2] Z. Sha et al., ”llasm: Naming Functions in Binaries by Fusing Encoder-only and Decoder-only LLMs”, *ACM Transactions on Software Engineering and Methodology*, s. 3 702 988, 2024. DOI: 10.1145/3702988.

[3] T. B. Brown et al., ”Language Models are Few-Shot Learners”, *arXiv preprint arXiv:2005.14165v4*, 2020. DOI: 10.48550/arXiv.2005.14165.

[4] G. Li, C. Zhi, J. Chen, J. Han ja S. Deng, ”Exploring Parameter-Efficient Fine-Tuning of Large Language Model on Automated Program Repair”, teoksessa *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, Sacramento, CA, USA: ACM, 2024, s. 719–731. DOI: 10.1145/3691620.3695066.

[5] J. Devlin, M.-W. Chang, K. Lee ja K. Toutanova, ”BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”, *arXiv preprint arXiv:1810.04805v2*, 2019. DOI: 10.48550/arXiv.1810.04805.

[6] C. Raffel et al., ”Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer”, *arXiv preprint arXiv:1910.10683v4*, 2023. DOI: 10.48550/arXiv.1910.10683.

[7] Y. Wang, W. Wang, S. Joty ja S. C. H. Hoi, "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation", *arXiv preprint arXiv:2109.00859v2*, 2021. DOI: 10.48550/arXiv.2109.00859.

[8] Q. Zhang et al., *A Systematic Literature Review on Large Language Models for Automated Program Repair*, arXiv preprint arXiv:2405.01466v2, 2024. DOI: 10.48550/arXiv.2405.01466.

[9] H. Eladawy, C. Le Goues ja Y. Brun, "Automated Program Repair, What Is It Good For? Not Absolutely Nothing!", teoksessa *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, Lisbon, Portugal: ACM, 2024, s. 1–13. DOI: 10.1145/3597503.3639095.

[10] F. Li, J. Jiang, J. Sun ja H. Zhang, "Hybrid Automated Program Repair by Combining Large Language Models and Program Analysis", *ACM Transactions on Software Engineering and Methodology*, s. 3 715 004, 2025. DOI: 10.1145/3715004.

[11] S. Jiang, J. Zhang, W. Chen, B. Wang, J. Zhou ja J. Zhang, "Evaluating Fault Localization and Program Repair Capabilities of Existing Closed-Source General-Purpose LLMs", teoksessa *Proceedings of the 1st International Workshop on Large Language Models for Code*, Lisbon, Portugal: ACM, 2024, s. 75–78. DOI: 10.1145/3643795.3648390.

[12] A. Z. H. Yang, C. Le Goues, R. Martins ja V. Hellendoorn, "Large Language Models for Test-Free Fault Localization", teoksessa *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, Lisbon Portugal: ACM, 2024, s. 1–12. DOI: 10.1145/3597503.3623342.

[13] K. Huang et al., "An Empirical Study on Fine-Tuning Large Language Models of Code for Automated Program Repair", teoksessa *2023 38th IEEE/ACM*

International Conference on Automated Software Engineering (ASE), Luxembourg, Luxembourg: IEEE, 2023, s. 1162–1174. DOI: 10.1109/ASE56229.2023.00181.

[14] R. Just, D. Jalali ja M. D. Ernst, ”Defects4J: a database of existing faults to enable controlled testing studies for Java programs”, teoksessa *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, San Jose, CA, USA: ACM, 2014, s. 437–440. DOI: 10.1145/2610384.2628055.

[15] A. Zirak ja H. Hemmati, ”Improving Automated Program Repair with Domain Adaptation”, *ACM Transactions on Software Engineering and Methodology*, vol. 33, s. 1–43, 2024. DOI: 10.1145/3631972.

[16] J. Liu, C. Sha ja X. Peng, ”An Empirical Study of Parameter-Efficient Fine-Tuning Methods for Pre-Trained Code Models”, teoksessa *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Luxembourg, Luxembourg: IEEE, 2023, s. 397–408. DOI: 10.1109/ASE56229.2023.00125.

[17] S. B. Hossain et al., ”A Deep Dive into Large Language Models for Automated Bug Localization and Repair”, *Proceedings of the ACM on Software Engineering*, vol. 1, s. 1471–1493, 2024. DOI: 10.1145/3660773.

[18] X. Yin, C. Ni, S. Wang, Z. Li, L. Zeng ja X. Yang, ”ThinkRepair: Self-Directed Automated Program Repair”, teoksessa *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, Vienna, Austria: ACM, 2024, s. 1274–1286. DOI: 10.1145/3650212.3680359.

[19] R. Widyasari, J. W. Ang, T. G. Nguyen, N. Sharma ja D. Lo, ”Demystifying Faulty Code: Step-by-Step Reasoning for Explainable Fault Localization”, teoksessa *2024 IEEE International Conference on Software Analysis, Evolu-*

tion and Reengineering (SANER), Rovaniemi, Finland: IEEE, 2024, s. 568–579. DOI: 10.1109/SANER60148.2024.00064.

[20] J. Zhao, D. Yang, L. Zhang, X. Lian, Z. Yang ja F. Liu, ”Enhancing Automated Program Repair with Solution Design”, teoksessa *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, Sacramento, CA, USA: ACM, 2024, s. 1706–1718. DOI: 10.1145/3691620.3695537.

[21] B. Galitsky, A. Chernyavskiy ja D. Ilvovsky, ”Truth-O-Meter: Handling Multiple Inconsistent Sources Repairing LLM Hallucinations”, teoksessa *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, Washington, DC, USA: ACM, 2024, s. 2817–2821. DOI: 10.1145/3626772.3657679.

[22] S. Bin Murtaza, A. Mccoy, Z. Ren, A. Murphy ja W. Banzhaf, ”LLM Fault Localisation within Evolutionary Computation Based Automated Program Repair”, teoksessa *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, Melbourne, VIC, Australia: ACM, 2024, s. 1824–1829. DOI: 10.1145/3638530.3664174.

[23] X. Wang et al., ”MTL-TRANSFER: Leveraging Multi-task Learning and Transferred Knowledge for Improving Fault Localization and Program Repair”, *ACM Transactions on Software Engineering and Methodology*, vol. 33, s. 1–31, 2024. DOI: 10.1145/3654441.

[24] S. Kang, G. An ja S. Yoo, ”A Quantitative and Qualitative Evaluation of LLM-Based Explainable Fault Localization”, *Proceedings of the ACM on Software Engineering*, vol. 1, s. 1424–1446, 2024. DOI: 10.1145/3660771.

[25] R. Abreu, P. Zoeteweij ja A. J. Van Gemund, ”An Evaluation of Similarity Coefficients for Software Fault Localization”, teoksessa *2006 12th Pacific Rim*

International Symposium on Dependable Computing (PRDC'06), 2006, s. 39–46. DOI: 10.1109/PRDC.2006.18.

[26] J. A. Jones ja M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique", teoksessa *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, Long Beach, CA, USA: ACM, 2005, s. 273–282. DOI: 10.1145/1101908.1101949.

[27] S. Shan, Y. Huo, Y. Su, Y. Li, D. Li ja Z. Zheng, "Face It Yourselves: An LLM-Based Two-Stage Strategy to Localize Configuration Errors via Logs", teoksessa *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, Vienna, Austria: ACM, 2024, s. 13–25. DOI: 10.1145/3650212.3652106.

[28] L. Zhang, L. Zhang ja S. Khurshid, "Injecting mechanical faults to localize developer faults for evolving software", teoksessa *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, Indianapolis, Indiana, USA: ACM, 2013, s. 765–784. DOI: 10.1145/2509136.2509551.

[29] S. Moon, Y. Kim, M. Kim ja S. Yoo, "Ask the Mutants: Mutating Faulty Programs for Fault Localization", teoksessa *Verification and Validation 2014 IEEE Seventh International Conference on Software Testing*, 2014, s. 153–162. DOI: 10.1109/ICST.2014.28.

[30] X. Li, W. Li, Y. Zhang ja L. Zhang, "DeepFL: integrating multiple fault diagnosis dimensions for deep fault localization", teoksessa *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Beijing, China: ACM, 2019, s. 169–180. DOI: 10.1145/3293882.3330574.

[31] Y. Ouyang, J. Yang ja L. Zhang, "Benchmarking Automated Program Repair: An Extensive Study on Both Real-World and Artificial Bugs", teok-

sessa *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, Vienna, Austria: ACM, 2024, s. 440–452. DOI: 10.1145/3650212.3652140.

[32] Y. Yuan, ”ARJA-e for the First International Competition on Automated Program Repair”, teoksessa *Proceedings of the 5th ACM/IEEE International Workshop on Automated Program Repair*, Lisbon, Portugal: ACM, 2024, s. 50–52. DOI: 10.1145/3643788.3648019.

[33] K. Liu, A. Koyuncu, D. Kim ja T. F. Bissyandé, ”TBar: revisiting template-based automated program repair”, New York, NY, USA: ACM, 2019, s. 31–42. DOI: 10.1145/3293882.3330577.

[34] H. Ye, M. Martinez, X. Luo, T. Zhang ja M. Monperrus, ”SelfAPR: Self-supervised Program Repair with Test Execution Diagnostics”, teoksessa *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, Rochester, MI, USA: ACM, 2023. DOI: 10.1145/3551349.3556926.

[35] H. Joshi, J. C. Sanchez, S. Gulwani, V. Le, I. Radiček ja G. Verbruggen, ”Repair is nearly generation: multilingual program repair with LLMs”, *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 37, nro 4, s. 5131–5140, 2023. DOI: 10.1609/aaai.v37i4.25642.

[36] F. Ribeiro, ”Large Language Models for Automated Program Repair”, teoksessa *Companion Proceedings of the 2023 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, Cascais, Portugal: ACM, 2023, s. 7–9. DOI: 10.1145/3618305.3623587.

[37] C. S. Xia ja L. Zhang, ”Less training, more repairing please: revisiting automated program repair via zero-shot learning”, teoksessa *Proceedings of the 30th*

ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Singapore, Singapore: ACM, 2022, s. 959–971. DOI: 10.1145/3540250.3549101.

[38] C. S. Xia ja L. Zhang, "Automated Program Repair via Conversation: Fixing 162 out of 337 Bugs for 0.42 Each using ChatGPT", teoksessa *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, Vienna, Austria: ACM, 2024, s. 819–831. DOI: 10.1145/3650212.3680323.

[39] OpenAI, *OpenAI API Models*, 2024. viitattu 26. toukokuuta 2025. url: https://platform.openai.com/docs/models.