

Tommi Oikarinen

Gemini-kielimallin hyödyntäminen ohjelmoinnissa

Insinööri

Tieto- ja viestintäteknikka

Kevät 2024



**KAMK • University
of Applied Sciences**

Tiivistelmä

Tekijä: Oikarinen Tommi

Työn nimi: Gemini-kielimallin hyödyntäminen ohjelmoinnissa

Tutkintonimike: Insinööri (AMK), tieto- ja viestintätekniikka

Asiasanat: tekoäly, ohjelmointi, koneoppiminen, neuroverkot, luonnollisen kielen käsittely

Opinnäytetyön tavoitteena oli tutkia Google DeepMindin kehittämää Gemini Pro 1.0 -kielimalliin perustuva Gemini-keskustelubottia ja sen kykyä toimia ohjelmoijan apuna ohjelmoinnissa. Tavoitteena oli selvittää, miten Gemini-keskustelubottia voidaan hyödyntää yleisissä ohjelmoinnin tehtävissä ja kuinka hyvin se suoriutuu niistä. Työssä perehdyttiin myös Gemini-keskustelubottiin keskeisesti liittyviin käsitteisiin, kuten tekoäly ja sen osa-alueet.

Gemini-keskustelubotin kykyä testattiin erilaisissa ohjelmoinnin yleisissä tehtävissä. Testit suoritettiin Visual Studio Code -ympäristössä käyttäen C-ohjelmointikieltä. Gemini-keskustelubotille annettiin tehtäväksi tuottaa ohjelmakoodia annettujen tekstikehotteiden perusteella, selittää ja kommentoida olemassa olevaa ohjelmakoodia, etsiä ja korjata virheitä ohjelmakoodista, optimoida ohjelmakoodia sekä kääntää ohjelmakoodia toiselle ohjelmointikielelle. Jokaisen testin onnistumista arvioitiin tarkastelemalla Gemini-keskustelubotin tuottamaa koodia ja sen selityksiä.

Testien perusteella havaittiin, että Gemini-keskustelubotti kykenee tuottamaan toimivaa C-koodia annettujen tekstikehotteiden perusteella. Se osasi myös selittää ja kommentoida monimutkaisinkin ohjelmakoodin toimintaa ymmärrettävästi. Gemini-keskustelubotti onnistui tunnistamaan ja korjaamaan syntaksi-, suoritus- ja loogisia virheitä testiohjelmakoodista. Lisäksi Gemini-keskustelubotti kykeni ehdottamaan pieniä, mutta hyödyllisiä optimointeja ohjelmakoodin luettavuuden ja tehokkuuden parantamiseksi. Lopuksi Gemini-keskustelubotti käänsi C-ohjelmakoodin JavaScriptiksi onnistuneesti.

Vaikka Gemini-keskustelubotti osoitti lupaavia tuloksia testeissä, on tärkeää huomata, että testit tehtiin melko yksinkertaisilla tehtävillä. Lisätutkimusta tarvitaan, jotta voidaan arvioida Gemini-keskustelubotin kykyä käsitellä monimutkaisempia tehtäviä. Tekoälytyökalujen rooli ohjelmoinnissa tulee kuitenkin kasvamaan tulevaisuudessa, ja Gemini-keskustelubotti on yksi esimerkki tästä kehityksestä. Opinnäytetyö osoitti, että Gemini-keskustelubotti voi toimia ohjelmoijalle hyödyllisenä apuvälineenä, joka voi tehostaa ohjelmointityötä ja parantaa ohjelmakoodin laatua.

Abstract

Author: Oikarinen Tommi

Title of the Publication: Gemini language model utilization in programming

Degree Title: Bachelor of Engineering, Information technology

Keywords: artificial Intelligence, programming, machine learning, neural networks, natural language processing

The goal of the thesis was to study the Gemini chatbot, based on the Gemini Pro 1.0 language model developed by Google DeepMind, and its capabilities as a programming assistant. The aim was to determine how the Gemini chatbot can be utilized in general programming tasks and how well it performs in them. This work also explored concepts central to the Gemini chatbot, such as artificial intelligence and its sub-fields.

The capabilities of the Gemini chatbot were tested in various general programming tasks. The tests were conducted in the Visual Studio Code environment using the C programming language. The Gemini chatbot was tasked with producing code based on provided instructions, explaining, and commenting on existing code, finding, and correcting errors in code, optimizing code, and translating code to another programming language. The success of each test was assessed by examining the code produced by the Gemini chatbot and its explanations.

Based on the tests, the Gemini chatbot can generate functional C code based on the given text prompts. It was also able to explain and comment on the functionality of even complex code in an understandable manner. The Gemini chatbot successfully identified and corrected syntax, execution, and logical errors in the test code. Furthermore, the Gemini chatbot was able to suggest small but useful optimizations to improve code readability and efficiency. Lastly, the Gemini chatbot successfully translated C code into JavaScript.

Although the Gemini chatbot showed promising results in the tests, it is important to note that the tests were conducted with relatively simple tasks. Further research is needed to assess the ability of the Gemini chatbot to handle more complex tasks. However, the role of AI tools in programming will continue to grow in the future, and the Gemini chatbot is one example of this development. This thesis demonstrated that the Gemini chatbot can be a useful tool for programmers, which can streamline the programming process and improve the quality of code.

Sisällys

1	Johdanto	1
2	Tekoäly ja suuret kielimallit ohjelmoinnissa.....	2
2.1	Tekoäly	3
2.2	Koneoppiminen	4
2.3	Luonnollisen kielenkäsittely	5
2.4	Neuroverkot ja syväoppiminen	5
2.5	Suuret kielimallit ja transformer-malli	6
2.6	Gemini-kielimallit ja keskustelubotit.....	8
3	Gemini-keskustelubotin testaus.....	10
3.1	Ohjelmakoodin tuottaminen.....	10
3.2	Ohjelmakoodin ymmärtäminen ja kommentointi	16
3.3	Ohjelmakoodista virheiden etsiminen ja korjaaminen	19
3.4	Ohjelmakoodin optimointi	27
3.5	Ohjelmakoodin kääntö toiselle ohjelmointikielelle	29
3.6	Tulokset	31
4	Yhteenveto	32
	Lähteet	33

1 Johdanto

Ohjelmoinnin ala kehittyi huimaa vauhtia ja tekoäly on alkanut saada yhä merkittävämmän roolin alan työkaluissa. Gemini-keskustelubotti, joka perustuu Google DeepMindin kehittämään Gemini Pro 1.0 -kielimalliin, on yksi esimerkki tästä kehityksestä. Tässä opinnäytetyössä selvitetään, miten Gemini-keskustelubotti voi toimia ohjelmoijan apuna ja missä yleisissä ohjelmoinnin eri tehtävissä sitä voidaan käyttää hyödyksi.

Gemini-keskustelubotti on vielä keväällä 2024 ilmainen ja kaikkien saatavilla, mutta keskustelubotista on olemassa myös kehittyneempi maksullinen versio, Gemini Advanced. Tässä työssä keskitytään kuitenkin ilmaiseen versioon ja sen ominaisuuksiin. Työssä arvioidaan testien perusteella Geminin kykyjä erilaisissa ohjelmoinnin yleisissä tehtävissä. Testit suoritetaan Visual Studio Code -ympäristössä C-ohjelmointikielellä.

Testeissä selvitetään, voiko Gemini-keskustelubotti tekstikehotteiden perusteella auttaa ohjelmoijaa tuottamaan ohjelmakoodia, selittämään ja kommentoimaan ohjelmakoodia, etsimään ja korjaamaan virheitä ohjelmakoodista, optimoimaan ohjelmakoodia ja kääntämään ohjelmakoodia toiselle ohjelmointikielelle.

Työ aloittaa tarkastelemalla suurten kielimallien teoreettisia mahdollisuuksia ohjelmoinnin apuvälineenä. Tämän jälkeen tutustutaan tekoälyn peruskäsitteisiin ja hieman sen kehityshistoriaan. Teoriaa seuraa käytäntö, jossa Geminin ominaisuuksia testataan ohjelmoinnin eri tehtävissä. Tulokset esitetään kuvakaappauksin ja ohjelmakoodiesimerkein. Lopuksi tehdään yhteenveto Geminin hyödyistä ohjelmoinnissa ja pohditaan vastaavien työkalujen tulevaisuudennäkymiä.

2 Tekoäly ja suuret kielimallit ohjelmoinnissa

Tekoälyn ja erityisesti suurten kielimallien yleistymisen on mullistamassa monia aloja, eikä ohjelmointi ole poikkeus. Viime vuosina on tapahtunut merkittävää edistystä tekoälyssä, erityisesti luonnollisen kielenkäsittelyn alalla. Läpimurrot syväoppimisessa ja neuroverkoissa ovat synnyttäneet suuret kielimallit, jotka on koulutettu valtavalla määrällä dataa. Ne pystyvät ymmärtämään ja tuottamaan ihmisen kaltaista tekstiä, mikä avaa uusia mahdollisuuksia ohjelmakoodin luomiseen, ymmärtämiseen ja dokumentointiin. [1.]

Yksi suurten kielimallien kiinnostavimmista sovelluksista on kyky tuottaa ja täydentää ohjelmakoodia eri ohjelmointikielillä. Mallit voivat luoda ohjelmakoodia pelkän tekstimuotoisen kuvauksen perusteella. Tämä voi säästää ohjelmoijien aikaa ja vaivaa rutiininomaisissa tehtävissä, kuten toistuvien ohjelmakoodirakenteiden luomisessa tai yleisten algoritmien toteuttamisessa. [1.]

Tutkimukset ovat osoittaneet, että näiden mallien avulla luotu ohjelmakoodi on laadukasta ja toimii odotetusti. Tulevaisuudessa voidaan odottaa tekoälyn ja suurten kielimallien roolin kasvavan entisestään ohjelmoinnissa. Ne voivat automatisoida rutiinitehtäviä, ehdottaa ratkaisuja ongelmiin ja jopa luoda kokonaisia ohjelmistoja itsenäisesti. Tämä vapauttaa ohjelmoijien aikaa keskittyä luovempiin ja strategisempiin tehtäviin. [1.]

Suuret kielimallit tarjoavat jännittäviä mahdollisuuksia, mutta niitä käytettäessä on otettava huomioon useita rajoituksia. Nykyinen tutkimus usein perustuu yksinkertaisiin kehoitusstrategioihin, mikä jättää huomioonottamatta kehoitustekniikan potentiaalin suurten kielimallien suorituskyvyn parantamiseksi entisestään. Tärkeintä on keskittyä suurten kielimallien hyödyntämiseen ja integroimiseen erilaisiin sovelluksiin ja työnkulkuihin, jotta niiden potentiaali saadaan täysimääräisesti käyttöön. Lisäksi liiallinen luottamus Pythoniin ensisijaisena ohjelmointikielenä rajoittaa ymmärrystä suurten kielimallien kyvyistä muissa laajalti käytetyissä kielissä. Tutkimuksen laajentaminen kattamaan laajemman valikoiman kieliä on olennaista niiden täyden potentiaalin hyödyntämiseksi. [2.]

2.1 Tekoäly

Tekoälyn määritelmä ei ole yksiselitteinen, mutta yksinkertaistettuna se tarkoittaa tietokonejärjestelmien kykyä suorittaa tehtäviä, jotka normaalisti vaatisivat ihmisen älykkyyttä [3, s. 1–2]. Ihmisen älykkyyttä ei luonnehdita yhdellä piirteellä, vaan monien erilaisten kykyjen yhdistelmällä. Tekoälyn tutkimus on keskittynyt erityisesti seuraaviin älykkyyden osa-alueisiin: oppimiseen, päättelyyn, ongelmanratkaisuun, havaitsemiseen ja kielen käyttöön. [4.]

Tekoälyn tutkimus pyrkii saavuttamaan yhden kolmesta tavoitteesta: yleinen tekoäly, sovelletun tekoälyn tai kognitiivisen simulaation. Yleisen tekoälyn (jota kutsutaan myös vahvaksi tekoälyksi) tavoitteena on rakentaa koneita, jotka ajattelevat. Yleisen tekoälyn perimmäisenä tavoitteena on tuottaa kone, jonka yleinen älyllinen kyky on erottamaton ihmisen älyllisestä kyvystä. Sovelletun tekoälyn, joka tunnetaan myös nimellä edistynyt tietojenkäsittely, tavoitteena on tuottaa kaupallisesti kannattavia "älykkäitä" järjestelmiä - esimerkiksi "asiantuntija"-lääketieteellisen diagnoosin järjestelmiä ja osakekaupankäyntijärjestelmiä, jossa se on menestynyt huomattavasti. Kognitiivisessa simulaatiossa tietokoneita käytetään testaamaan teorioita siitä, miten ihmisen mieli toimii - esimerkiksi teorioita siitä, miten ihmiset tunnistavat kasvoja tai muistavat muistoja. Kognitiivinen simulaatio on jo nyt tehokas työkalu sekä neurotieteessä että kognitiivisessa psykologiassa. [4.]

Vaikka tekoälyn käsite on ollut olemassa 1950-luvulta lähtien, sen todellinen kukoistus on alkanut viime vuosikymmeninä. Varhainen tekoälyn tutkimus keskittyi ongelmanratkaisuun ja symbolisiin menetelmiin. 1960-luvulla Yhdysvaltain puolustusministeriö alkoi kouluttaa tietokoneita matkimaan ihmisen peruspäättelyä. Esimerkiksi DARPA:n (Defense Advanced Research Projects Agency) katukartoitushankkeet 1970-luvulla ja älykkäät henkilökohtaiset avustajat 2000-luvun alussa olivat tekoälyn edelläkävijöitä. 1950-luvulta 1970-luvulle hermoverkot herättivät innostusta "ajattelevista koneista". 1980-luvulta 2010-luvulle koneoppimisesta tuli suosittua, ja 2011-luvulta lähtien syväoppimisen läpimurto on ollut tekoälyn kasvun taustalla. Nykyään generatiivinen tekoäly on yksi nousevista teknologioista. [5.]

2.2 Koneoppiminen

Koneoppiminen on nopeasti kehittyvä tekoälyn osa-alue, joka keskittyy tietokoneohjelmien kykyyn oppia ja parantua ilman, että niitä on nimenomaisesti ohjelmoitu tehtävään. Se perustuu algoritmeihin ja tilastollisiin malleihin, jotka analysoivat dataa ja tunnistavat siitä piirteitä ja malleja. Tämän avulla ohjelmat voivat tehdä ennustuksia ja päätöksiä uudesta, ennennäkemättömästä datasta. [6, s. 1–2.]

Koneoppimisen sovellusalueet ovat laajat ja kasvavat jatkuvasti. Sitä käytetään jo nyt analysoimaan suuria datamääriä esimerkiksi biolääketieteessä, jossa sitä käytetään proteiinien rakenteen ja toiminnan ennustamiseen geneettisten sekvenssien perusteella. [7, s. 2.] Toisella saralla koneoppiminen mahdollistaa tehokkaan kasvojentunnistuksen ja on avainasemassa itseohjautuvien autojen kehityksessä. [8, s. 196; 9.]

Koneoppimisen algoritmit voidaan jakaa kolmeen pääluokkaan: valvottu oppiminen, valvomaton oppiminen ja vahvistusoppiminen. Valvotussa oppimisessa malli opetetaan merkittyjen esimerkkien avulla, joissa tulos on tiedossa. Tämän avulla malli voi oppia ennustamaan tulevia tuloksia. Esimerkki tästä on roskapostisuodatin, joka oppii tunnistamaan roskapostin merkittyjen sähköpostiviestien avulla. [3, s. 29.]

Valvomaton oppiminen puolestaan pyrkii löytämään piilotettuja malleja datasta ilman valmiiksi annettuja vastauksia. Yksi esimerkki tästä on klusterointi, jossa algoritmi jakaa datapisteet ryhmiin niiden ominaisuuksien perusteella. Tätä voidaan käyttää esimerkiksi kuvan segmentoinnissa. [10, s. 265–266.]

Vahvistusoppiminen eroaa edellisistä siinä, että malli oppii vuorovaikutuksessa ympäristönsä kanssa kokeilemalla ja erehtymällä. Se saa palautetta toimistaan, joiden avulla se oppii valitsemaan toimintoja, jotka maksimoivat pitkän aikavälin palkinnon. Tämä lähestymistapa on tärkeä esimerkiksi robottien ohjauksessa ja pelien pelaamisessa. [11, s. 2–3.]

Koneoppiminen tarjoaa valtavia mahdollisuuksia monilla eri aloilla. Se auttaa tekemään parempia päätöksiä, automatisoimaan tehtäviä ja ymmärtämään dataa syvemmällä tasolla. On kuitenkin tärkeää muistaa, että koneoppiminen ei ole maaginen ratkaisu kaikkiin ongelmiin. Se vaatii suuria datamääriä toimiakseen tehokkaasti ja mallit voivat olla alttiita vinoutumille datassa. [12, s. 13.]

2.3 Luonnollisen kielenkäsittely

Luonnollisen kielenkäsittely on kiehtova tieteenala, joka yhdistää kielitieteen, tietojenkäsittelytieteen ja tekoälyn pyrkimyksenä kehittää tietokoneohjelmia, jotka ymmärtävät, tulkitsevat ja tuottavat ihmiskieltä luonnollisella tavalla. Luonnollisen kielenkäsittely on avainasemassa monissa sovelluksissa, kuten konekäännöksessä, keskusteluboteissa, tekstianalyyseissä ja puheentunnistuksessa. [13, s. 4–5.]

Vaikka luonnollisen kielen käsittely ei olekaan uusi tiede, se on kokenut nopean kehityksen, jonka taustalla on useita tekijöitä. Ensinnäkin kiinnostus ihmisen ja koneen väliseen kommunikaatioon on kasvanut räjähdysmäisesti. Toisekseen eletään massadatan aikakautta, jossa valtavat määrät teksti- ja puhedataa on helposti saatavilla. Lisäksi laskentakapasiteetti on kasvanut huimasti, ja algoritmit ovat kehittyneet. Kaikki tämä on luonut otollisen ympäristön luonnollisen kielen käsittelyn kehittymiselle. [14.]

Luonnollisen kielen käsittelyn ytimessä on kyky purkaa ihmiskielen monimutkaisuutta. Kielen vivahteet, kuten homonyymit, sarkasmi ja kielikuvat, jotka ihmiset omaistavat vaivattomasti, ovat tietokoneille todellisia haasteita. Luonnollisen kielen käsittelyjärjestelmien on opittava tunnistamaan ja tulkitsemaan näitä vivahteita tarkasti voidakseen toimia tehokkaasti. Tätä varten ne hyödyntävät erilaisia tekniikoita, kuten puheentunnistusta, sanaluokitusta ja merkityksenmäärittystä. Nämä tekniikat auttavat konetta ymmärtämään tekstin rakennetta, yksittäisten sanojen merkitystä sekä niiden välisiä suhteita. Luonnollisen kielen käsittely kykenee myös tunnistamaan tekstissä mainittuja entiteettejä, kuten paikkoja ja henkilöitä, sekä tulkitsemaan tekstin yleistä sävyä ja tunteita. [15.]

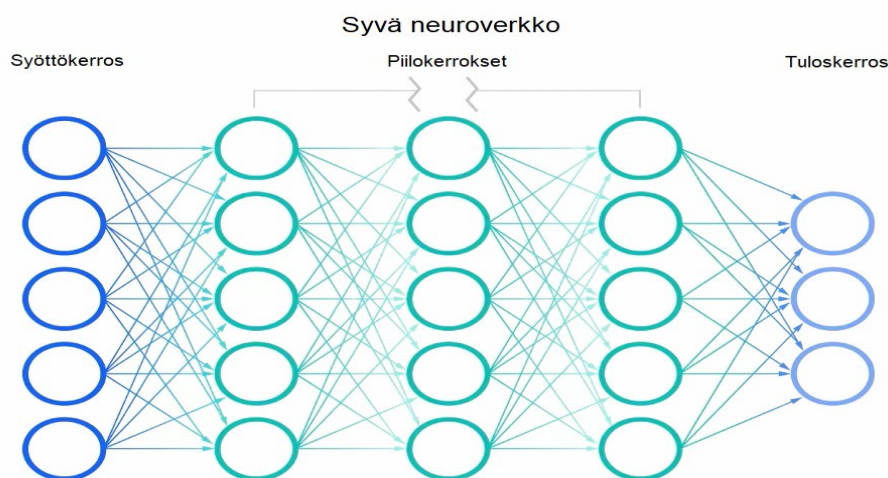
2.4 Neuroverkot ja syväoppiminen

Tekoälyn yksi merkittävimmistä osa-alueista on neuroverkot ja koneoppimiseen perustuva syväoppiminen. Neuroverkot ovat laskennallisia malleja, jotka on inspiroitu ihmisaivojen rakenteesta ja toiminnasta. Ne koostuvat toisiinsa kytkeytyneistä solmuista, eli neuroneista, jotka käsittelevät ja välittävät tietoa. Kuten ihmisaivot, neuroverkot oppivat kokemuksen kautta, vahvis-

taen yhteyksiään toistuvan harjoittelun myötä. Neuroverkkojen avulla voidaan mallintaa monimutkaisia ja epälineaarisia suhteita, mikä tekee niistä tehokkaan työkalun monenlaisiin tehtäviin. [3, s. 750.]

Neuroverkot eivät ole uusi keksintö, sillä ensimmäiset mallit kehitettiin jo 1940-luvulla. Teknologian kehitys ja laskentatehon kasvu ovat kuitenkin mahdollistaneet monimutkaisempien neuroverkkojen rakentamisen ja niiden hyödyntämisen yhä vaativammassa tehtävissä. [16, s. 9.]

Syvän oppimisen teho perustuu sen kykyyn oppia automaattisesti ilman ihmisen jatkuvaa ohjausta. Toisin kuin perinteisessä koneoppimisessa, jossa ihmiset määrittelevät datan ominaisuudet, syvät neuroverkot löytävät itse olennaiset piirteet ja hierarkiat datasta. Syvät neuroverkot koostuvat kolmesta tai useammasta kerroksesta, vaikka monet nykyaikaiset syvät neuroverkot sisältävät merkittävästi enemmän kerroksia. Syvässä neuroverkossa jokainen kerros käsittelee tietoa ja välittää sen eteenpäin, jolloin verkko oppii tunnistamaan monimutkaisia kuvioita valtavasta datamäärästä. [17.] Kuvassa 1 on syvän neuroverkon rakenne yksinkertaistettuna.



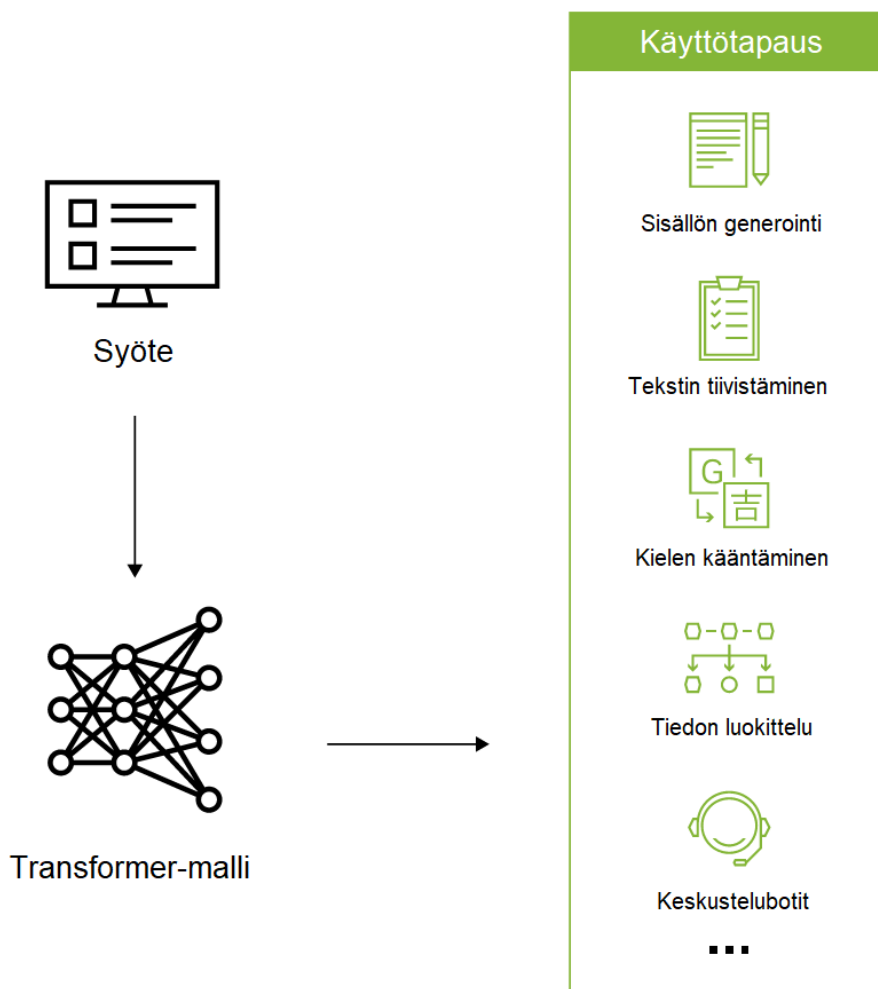
Kuva 1. Syvän neuroverkon rakenne [17].

2.5 Suuret kielimallit ja transformer-malli

Suurten kielimallien erityispiirteet erottavat ne muista tekoälymalleista. Ensinnäkin ne toimivat perustana erilaisille jatkokäyttöön tarkoitetuille tehtäville eliminoiden tarpeen kouluttaa yksittäisiä malleja jokaista toimintoa varten. Tämä säästää aikaa ja resursseja. Toiseksi useimmat suuret

kielimallit hyödyntävät transformer-malliarkehtuuria, joka on erinomainen peräkkäisen datan, kuten tekstin, käsittelyssä. Tämä mahdollistaa niiden kyvyn ymmärtää asiayhteyksiä ja suhteita tekstissä, mikä johtaa johdonmukaisempiin ja merkityksellisempiin tuloksiin. Lisäksi suuret kielimallit on koulutettu valtavilla tekstiaineistoilla, mikä antaa niille mahdollisuuden oppia monimutkaisia kielen rakenteita ja suhteita, mikä puolestaan johtaa niiden vaikuttaviin kykyihin. [18.]

Transformer-malli koostuu useista lohkoista tai kerroksista, jotka käyttävät yhdessä itsehuomio-kerroksia, eteenpäinsyöttökerroksia ja normalisointikerroksia syötteen prosessoimiseen ja ulostulon tuottamiseen. Nämä mallit ovat erityisen tehokkaita suurissa kielitehtävissä kahden keskeisen innovaation ansiosta: positionaaliset koodaukset ja itsehuomio. Positionaalinen koodaus upottaa syötteiden järjestyksen, mikä mahdollistaa sanojen ei-peräkkäisen syöttämisen neuroverkkoon. Itsehuomio antaa painoarvon kullekin syötetyn datan osalle sen merkityksen osoittamiseksi suhteessa muihin syötteisiin, mahdollistaen mallin keskittymisen olennaisiin osiin. Yhdessä nämä tekniikat mahdollistavat monimutkaisten suhteiden analysoinnin osien välillä pitkillä etäisyyksillä, ei-peräkkäisesti. Käsittelemällä tietoa ei-peräkkäin transformer-mallit voivat hajottaa monimutkaiset ongelmat pienempiin, samanaikaisiin laskelmiin, jotka sopivat hyvin rinnakkaiseen prosessointiin, helpottaen suurten merkitsemättömien tietojoukkojen ja massiivisten transformer-verkkojen laajamittaista analysointia. [19.] Kuvassa 2 on esitetty transformer-mallin toimintaa.



Kuva 2. Transformer-mallin toiminta [19].

2.6 Gemini-kielimallit ja keskustelubotit

Gemini on Google DeepMindin kehittämä ryhmä suuria kielimalleja sekä keskustelubotti. Gemini-kielimallit on alusta alkaen suunniteltu multimodaaliksi, eli ne voivat käsitellä ja tuottaa tekstiä, ohjelmakoodia, ääntä, kuvia ja videoita. Ne korvasivat joulukuussa 2023 Googlen aikaisemman suuren kielimallin PaLM 2:n, jota käytti Googlen keskustelubotti Bard. Google Bardin nimi vaihtui Geminiksi helmikuussa 2024. [20; 21.]

Huhtikuuhun 2024 mennessä, Gemini-kielimallista on olemassa versiot 1.0 ja 1.5. Versiosta 1.0 on mallit Nano, Pro ja Ultra ja versiosta 1.5 on vain Pro. Nano on Geminin kevyin malli ja se on

tehty suorittamaan mobiililaitteiden sisäisiä tehtäviä. Pro-malli on hyvä yleiskäyttöön, koska se skaalautuu erilaisiin tehtäviin. Ultra on tehokkain ja monipuolisin malli monimutkaisiin tehtäviin. Versiota 1.5 voisi käyttää Googlen Ai Studiossa, mutta se ei ole käytettävissä Suomessa alueellisen lukuksen takia. [22; 23.]

Gemini-kielimallin versio 1.0 perustuu aiemmin mainittuun transformer-malliin, jota on parannettu arkkitehtuuri- ja mallioptimoinnilla. Tämä mahdollistaa vakaan harjoittelun suurella mittakaavalla ja optimoidun päättelyn Googlen Tensor-suorittimella. Mallien koulutus tukee 32 768 merkin kontekstipituutta, joka on maksimimäärä merkkejä, joita kielimalli voi käsitellä kerralla. Mitä pidempi kontekstipituus on, sitä enemmän aiempaa keskustelua tai tekstiä malli voi huomioida vastatessaan. 32 768 merkkiä on noin 24 500 englannin kielistä sanaa. [24, s. 3; 25.]

Gemini-keskustelubotista on ilmainen ja maksullinen vaihtoehto. Ilmainen vaihtoehto on nimeltään pelkkä "Gemini" ja maksullinen "Gemini Advanced". Molemmat keskustelubotit käyttävät Gemini-kielimallin versiota 1.0, mutta ilmaisversio käyttää mallia Pro ja maksullinen vaihtoehto mallia Ultra. [26.]

3 Gemini-keskustelubotin testaus

Ohjelmoijat etsivät jatkuvasti tapoja tehostaa työtään ja parantaa ohjelmakoodinsa laatua. Tekoälytyökalut tarjoavat tähän mielenkiintoisia mahdollisuuksia, ja tässä osiossa testattiin, miten Gemini-keskustelubottia voidaan hyödyntää päivittäisissä ohjelmointitehtävissä. Gemini-keskustelubotista olisi saatavilla parempi ja suorituskykyisempi versio ”Gemini Advanced”, mutta tämä opinnäytetyö on toteutettu ilman budjettia.

Testit valittiin kattamaan erilaisia tilanteita, joissa ohjelmoija voi hyötyä näiden tekoälytyökalujen käytöstä. Testien avulla selvitettiin keskustelubotin kyvykkyyttä ohjelmakoodin tuottamisessa, ohjelmakoodin ymmärtämisessä ja kommentoinnissa, ohjelmakoodista virheiden etsimisessä ja korjaamisessa, ohjelmakoodin optimoinnissa sekä ohjelmakoodin kääntämisessä toiselle ohjelmointikielelle. Vaikka keskustelubotti ymmärtää suomea, sille annetut tekstikehotteet annettiin sille natiivilla kielellä, eli englanniksi, ettei suomen kielen hankala kielioppi aiheuta sille ongelmia ymmärryksessä ja siten vaikuta testien tuloksiin. Sille annetut tekstikehotteet, sen antamat vastaukset ja Visual Studio Codessa suoritettavat ohjelmakoodit on havainnoitu kuvissa.

3.1 Ohjelmakoodin tuottaminen

Ohjelmakoodia harvoin kannattaa kirjoittaa alusta alkaen itse. On hyvin todennäköistä, että joku on jo tehnyt saman työn ja tehnyt sen paremmin. Kaikki valmiit ratkaisut eivät kuitenkaan sovellu kaikkiin tarpeisiin, jolloin ohjelmoijan on tietyissä tilanteissa tarpeen kehittää oma ohjelmakoodi, jotta saavutetaan haluttu toiminnallisuus tai suorituskyky. Tällaisissa tapauksissa kannattaa oman ajan ja vaivan säästämiseksi käyttää valmiita ratkaisuja, jos sellaisia löytyy.

Ensimmäisessä testissä testattiin keskustelubotin kykyä tuottaa ohjelmakoodia annetun tekstikehotteen perusteella. Sitä käskettiin kirjoittamaan C-ohjelma, joka ottaa kaksi komentoriviparametria: syötetiedoston nimen ja tulostiedoston nimen. Ohjelman pitäisi lukea syötetiedoston sisältö ja salata kyseinen sisältö käyttäen yksinkertaista salausalgoritmia ja sitten kirjoittaa salattu sisältö tulostiedostoon. Keskustelubotille annettu tekstikehote ohjelmakoodin tuottamiseen näkyy Kuvassa 3.

```
Write a C program that takes two command-line arguments: the name of an input file and the name of an output file. The program should read the contents of the input file, encrypt the contents using a simple encryption algorithm of your choice, and then write the encrypted contents to the output file.
```

Kuva 3. Keskustelubotille annettu tekstikehote ohjelmakoodin tuottamiseen.

Keskustelubotti tuotti ohjelmakoodin, joka käytti Caesarin salakirjoitus -salausta. Siinä kirjaimet siirretään eteenpäin aakkosissa tietyn määrän verran. Se on yksi yksinkertaisimmista salausmenetelmistä, jonka takia se on helppo murtaa, mutta se soveltuu hyvin opetustarkoituksiin. Kuvassa 4 näkyy keskustelubotin tuottama ohjelmakoodi kommentteineen vastauksena edellä mainittuun tekstikehoteeseen. Sen lisäksi keskustelubotti on selittänyt ohjelmakoodin osien toimintaa Kuvassa 5. Selityksessä se ehdottaakin käyttämään vahvempaa salausalgoritmia OpenSSL-kirjastosta.

```

#include <stdio.h>

int main(int argc, char* argv[]) {
    // Check for correct number of arguments
    if (argc != 3) {
        printf("Usage: %s <input_file> <output_file>\n", argv[0]);
        return 1;
    }

    // Open input and output files
    FILE* input_file = fopen(argv[1], "r");
    if (input_file == NULL) {
        printf("Error opening input file: %s\n", argv[1]);
        return 1;
    }
    FILE* output_file = fopen(argv[2], "w");
    if (output_file == NULL) {
        printf("Error opening output file: %s\n", argv[2]);
        fclose(input_file);
        return 1;
    }

    // Caesar cipher shift amount (adjust as needed)
    int shift = 3; // Shift each character by 3 positions

    // Read and encrypt characters
    char ch;
    while ((ch = fgetc(input_file)) != EOF) {
        if (isalpha(ch)) {
            // Handle uppercase and lowercase separately
            if (isupper(ch)) {
                ch = 'A' + ((ch - 'A' + shift) % 26);
            } else {
                ch = 'a' + ((ch - 'a' + shift) % 26);
            }
        }
        fputc(ch, output_file);
    }

    // Close files
    fclose(input_file);
    fclose(output_file);

    printf("File encrypted successfully!\n");
    return 0;
}

```

Kuva 4. Keskustelubotin tuottama ohjelmakoodi annetusta tekstikehotteesta.

Explanation:

1. **Includes:** `stdio.h` is included for file operations.
2. **Arguments:** The program checks if two arguments are provided. The first is the input file name, and the second is the output file name.
3. **File Handling:** The program opens the input and output files. Error handling is included to check for opening failures.
4. **Shift Amount:** An integer variable `shift` defines the Caesar cipher shift amount. You can adjust this value to change the encryption strength (higher value for stronger encryption).
5. **Reading and Encryption:**
 - A `while` loop reads each character (`ch`) from the input file using `fgetc` .
 - The `isalpha` function checks if the character is alphabetic.
 - If the character is alphabetic:
 - For uppercase characters, the modulo operation (`%`) ensures the shift stays within the alphabet range (`A` to `Z`).
 - Similar logic applies to lowercase characters (`a` to `z`).
 - The encrypted character is written to the output file using `fputc` .
6. **Closing Files:** Both files are closed using `fclose` .
7. **Success Message:** A confirmation message is printed on successful encryption.

Note: This is a simple encryption example using a Caesar cipher. For robust security, consider using stronger encryption algorithms from libraries like OpenSSL.

Kuva 5. Keskustelubotin selitys tuottamastaan ohjelmakoodista.

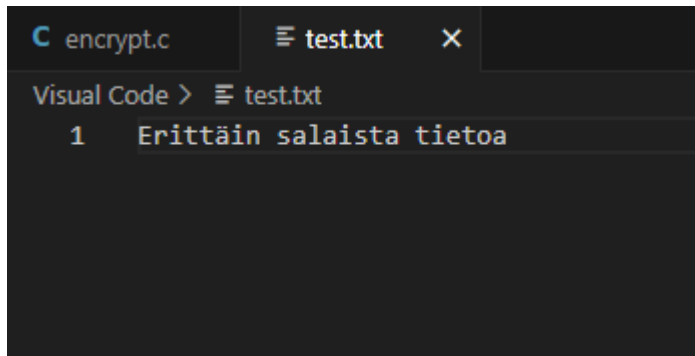
Seuraavaksi keskustelubotin tuottama ohjelmakoodi kopioitiin koodieditoriin ja luotiin salattava tekstitiedosto, jolla ohjelmakoodin toiminta voitiin testata. Kuvassa 6 tuo kyseinen ohjelmakoodi näkyy koodieditorissa ja Kuvassa 7 näkyy salattavan tekstitiedoston sisältö.

```

C encrypt.c x test.txt
Visual Code > C encrypt.c > main(int, char * [])
1  #include <stdio.h>
2
3  int main(int argc, char* argv[]) {
4      // Check for correct number of arguments
5      if (argc != 3) {
6          printf("Usage: %s <input_file> <output_file>\n", argv[0]);
7          return 1;
8      }
9
10     // Open input and output files
11     FILE* input_file = fopen(argv[1], "r");
12     if (input_file == NULL) {
13         printf("Error opening input file: %s\n", argv[1]);
14         return 1;
15     }
16     FILE* output_file = fopen(argv[2], "w");
17     if (output_file == NULL) {
18         printf("Error opening output file: %s\n", argv[2]);
19         fclose(input_file);
20         return 1;
21     }
22
23     // Caesar cipher shift amount (adjust as needed)
24     int shift = 3; // Shift each character by 3 positions
25
26     // Read and encrypt characters
27     char ch;
28     while ((ch = fgetc(input_file)) != EOF) {
29         if (isalpha(ch)) {
30             // Handle uppercase and lowercase separately
31             if (isupper(ch)) {
32                 ch = 'A' + ((ch - 'A' + shift) % 26);
33             } else {
34                 ch = 'a' + ((ch - 'a' + shift) % 26);
35             }
36         }
37         fputc(ch, output_file);
38     }
39
40     // Close files
41     fclose(input_file);
42     fclose(output_file);
43
44     printf("File encrypted successfully!\n");
45     return 0;
46 }
47

```

Kuva 6. Keskustelubotin tuottama ohjelmakoodi koodieditorissa.



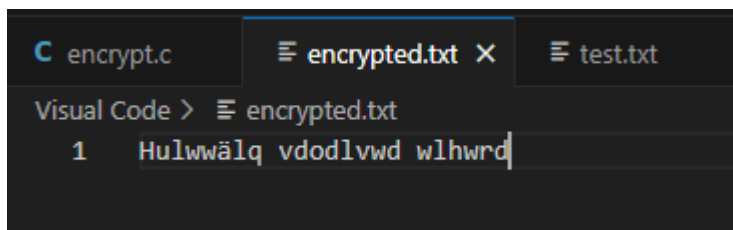
```

C encrypt.c test.txt X
Visual Code > test.txt
1 Erittäin salaista tietoa

```

Kuva 7. Salattavan tekstitiedoston sisältö.

Seuraavaksi ohjelmakoodi käännettiin ja ajettiin, jolloin se salasi tuon luodun tekstitiedoston Caesarin salakirjoitus -salauksella. Kuvassa 8 nähdään salatun tekstitiedoston sisältö.



```

C encrypt.c encrypted.txt X test.txt
Visual Code > encrypted.txt
1 Hulwwälq vdodlvwd wlhwrđ

```

Kuva 8. Salatun tekstitiedoston sisältö.

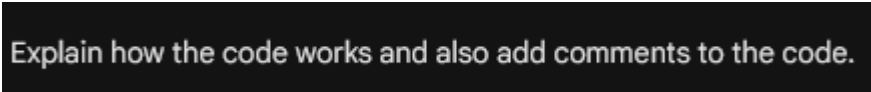
Keskustelubotin tuottama ohjelmakoodi toimi, niin kuin pitikin. Se oli C-ohjelma, joka otti kaksi komentoriviparametria: syötetiedoston nimen ja tulostiedoston nimen. Ohjelman piti myös lukea syötetiedoston sisältö, salata sisältö käyttäen jotain yksinkertaista salausalgoritmia ja sitten kirjoittaa salattu sisältö tulostiedostoon. Tuon ohjelmakoodin tuottamiseen meni keskustelubotilta alle 10 sekuntia ja se vaatisi noin keskitason C-ohjelmointitaitoja, jos sen joku ihminen kirjoittaisi.

Jos tuo ohjelmakoodi ei olisi toiminut halutulla tavalla, voi keskustelubotille antaa lisätietoja tai tarkentaa annettua tekstikehotetta. Tekstikehotteeseen olisi voinut myös lisätä joitain vaatimuksia, joita ohjelmakoodissa pitää olla. Tässä tapauksessa se olisi voinut olla vaikka jokin tietty salausalgoritmi, jolla tekstitiedoston sisältö salataan.

3.2 Ohjelmakoodin ymmärtäminen ja kommentointi

Ohjelmakoodin toiminnan ymmärtäminen on ohjelmoijalle yhtä olennainen taito, kuin itse toimivan ohjelmakoodin kirjoittaminen. Jos kirjoittamaansa ohjelmakoodia ei ymmärrä, ei sitä myöskään pysty asianmukaisesti kommentoimaan, jolloin muutoksien tekeminen ja virheiden etsiminen siitä hankaloituu. Hyvä kommentointi auttaa paitsi ohjelmakoodin kirjoittajaa itseään ymmärtämään sen toimintaa myöhemmin, mutta myös muita ohjelmoijia, jotka työskentelevät ohjelmakoodin parissa tulevaisuudessa.

Toisessa testissä testattiin keskustelubotin kykyä ymmärtää ja kommentoida sille annettua ohjelmakoodia. Sille annettiin vaikeasti luettava ohjelmakoodi, joka ei sisältänyt vihjeitä ohjelmakoodin toiminnasta kommenttien tai muuttujien nimien muodossa. Annettu ohjelmakoodi pyrkii löytämään alkuluvut käyttäjän määrittelemältä alueelta. Se aloittaa pyytämällä käyttäjältä alueen ylärajaa, jonka sisältä halutaan löytää alkuluvut. Kun käyttäjä antaa ylärajan, ohjelma käyttää `generate_primes` -funktioita löytääkseen alkuluvut määrättyltä alueelta. Tämä funktio kutsuu rekursiivisesti itseään aloittaen laskennan numerosta 2, koska 2 on pienin alkuluku. Funktio tarkistaa jokaisen luvun annetun ylärajan ja luvun 2 väliltä, onko se alkuluku vai ei. Jos luku on alkuluku, se tulostetaan näytölle. Alkulukujen löytämiseksi ohjelma käyttää `is_prime` -funktioita. Tämä funktio hyödyntää optimoituja tarkistusmenetelmiä ja rekursiota, jotta vältetään turhia tarkistuksia ja parannetaan suorituskykyä. Kun kaikki alkuluvut on löydetty, ne tulostetaan näytölle käyttäjän nähtäväksi. Kuvassa 9 näkyy sille annettu tekstikehote ohjelmakoodin ymmärtämiseen ja kommentointiin ja Kuvassa 10 sille annettu ohjelmakoodi.



Explain how the code works and also add comments to the code.

Kuva 9. Keskustelubotille annettu tekstikehote ohjelmakoodin ymmärtämiseen ja kommentointiin.

```

C abc.c x
Visual Code > C abc.c > ...
1  #include <stdio.h>
2
3  int a(int n) {
4      if (n <= 1) return 0;
5      if (n <= 3) return 1;
6      if (n % 2 == 0 || n % 3 == 0) return 0;
7
8      int i = 5;
9      while (i * i <= n) {
10         if (n % i == 0 || n % (i + 2) == 0)
11             return 0;
12         i += 6;
13     }
14     return 1;
15 }
16
17 void b(int n, int c) {
18     if (c > n) return;
19
20     if (a(c))
21         printf("%d ", c);
22
23     b(n, c + 1);
24 }
25
26 int main() {
27     int n;
28     scanf("%d", &n);
29
30     b(n, 2);
31
32     return 0;
33 }
34

```

Kuva 10. Keskustelubotille annettu ohjelmakoodi, sen ymmärtämiseen ja kommentointiin.

Keskustelubotti antoi kommentoidun ohjelmakoodin ja selityksen sen toiminnasta vaihe vaiheelta. Kommenteissa ja selityksissä se kuvasi ohjelman toimintaa helposti ymmärrettävästi. Kuvassa 11 näkyy keskustelubotin kommentoima ohjelmakoodi, ja Kuvassa 12 on selitys ohjelmakoodin toiminnasta vaihe vaiheelta.

```

C abcc x
Visual Code > C abcc > ...
1  #include <stdio.h>
2
3  // Function to check if a number is prime
4  int a(int n) {
5      // Base cases: 0 and 1 are not prime
6      if (n <= 1) return 0;
7
8      // 2 and 3 are prime
9      if (n <= 3) return 1;
10
11     // Even numbers and numbers divisible by 3 are not prime (except 2 and 3)
12     if (n % 2 == 0 || n % 3 == 0) return 0;
13
14     // Start checking for divisibility from 5
15     int i = 5;
16     while (i * i <= n) {
17         // Check if n is divisible by i or i + 2 (optimized check for 6k +/- 1)
18         if (n % i == 0 || n % (i + 2) == 0) return 0;
19
20         // Increment i by 6 to check next potential divisors (6k +/- 1)
21         i += 6;
22     }
23
24     // If no divisors found in the loop, n is prime
25     return 1;
26 }
27
28 // Function to print prime numbers up to a given limit
29 void b(int n, int c) {
30     // Base case: stop recursion when c exceeds limit
31     if (c > n) return;
32
33     // Check if c is prime using function a
34     if (a(c)) {
35         // Print the prime number
36         printf("%d ", c);
37     }
38
39     // Recursively call b to check and print next number (c + 1)
40     b(n, c + 1);
41 }
42
43 int main() {
44     int n;
45
46     // Get the upper limit for prime numbers from user
47     printf("Enter the limit: ");
48     scanf("%d", &n);
49
50     // Start printing primes from 2 (first prime number)
51     printf("Prime numbers less than or equal to %d are: ", n);
52     b(n, 2);
53
54     printf("\n");
55     return 0;
56 }
57

```

Kuva 11. Keskustelubotin kommentoima ohjelmakoodi.

Explanation:

1. `a(int n)` : This function checks if a number `n` is prime.
 - It handles base cases for 0, 1, 2, and 3.
 - Even numbers and multiples of 3 (except 2 and 3) are not prime.
 - It uses a loop to check for divisibility by potential prime factors starting from 5.
 - The loop optimization checks for divisibility by `i` and `i + 2` to cover all possible divisors in the form $6k \pm 1$ (where k is an integer).
 - If no divisors are found, the number is prime.
2. `b(int n, int c)` : This function prints prime numbers up to a limit `n`.
 - It uses recursion to iterate through numbers from `c` to `n`.
 - It calls function `a` to check if the current number `c` is prime.
 - If `c` is prime, it prints the number.
 - The function then recursively calls itself with `c + 1` to check the next number.
3. `main()` : This is the main function where the program starts.
 - It prompts the user to enter an upper limit for prime numbers.
 - It calls function `b` to start printing prime numbers from 2 (the first prime) up to the given limit `n`.

This code effectively finds and prints prime numbers less than or equal to a user-specified limit.

Kuva 12. Keskustelubotin selitys annetusta ohjelmakoodista.

Keskustelubotti antoi vastauksena hyvin kommentoidun ohjelmakoodin ja sekä syvemmän selityksen sille, miten tuo ohjelmakoodi toimii. Se lisäsi myös muutaman printf-funktion helpottamaan ohjelmakoodin ymmärrystä käyttäjälle, vaikka sitä siltä ei pyydetty.

3.3 Ohjelmakoodista virheiden etsiminen ja korjaaminen

Ohjelmointivirheet ovat osa ohjelmointia ja niitä tulee vastaan kaikille ohjelmoijille. Virheitä voi olla monenlaisia ja niistä yleisiä ovat esimerkiksi syntaksi-, suoritus- ja loogiset virheet. Syntaksi- virheet johtuvat ohjelmakoodin rakenteen virheistä, jotka estävät kääntäjää ymmärtämästä sitä.

Esimerkiksi puuttuvat sulut tai puolipisteet aiheuttavat syntaksivirheen. Syntaksivirheet havaitaan yleensä kääntämisvaiheessa ja ne estävät ohjelmaa kääntymästä. Suoritusvirheet ilmenevät, kun ohjelma suoritetaan ja se kohtaa odottamattoman tilan tai yrittää suorittaa jotain mahdotonta. Esimerkiksi yritys jakaa muuttuja numerolla 0 voi aiheuttaa suoritusvirheen. Näistä virheistä yleensä vaikein löytää on looginen virhe, jossa ohjelma ei kaadu, mutta toimii ei tarkoitetulla tavalla. Niitä kääntäjä ei tulkitse virheiksi, joten niistä ei tule virheilmoituksia, toisin kuin syntaksi- ja suoritusvirheistä. Niiden etsiminen ja korjaaminen on aikaa vievää puuhaa, varsinkin jos ohjelmoija käy manuaalisesti ohjelmakoodia läpi katselmoimalla.

```

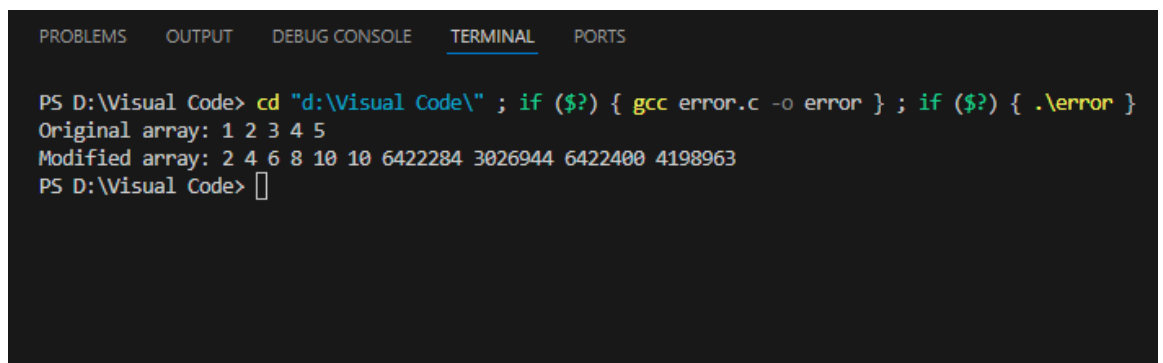
C error.c x
Visual Code > C error.c > ...
1  #include <stdio.h>
2
3  void modifyArray(int *arr, int size) {
4      for (int i = 0; i < size; i++) {
5          arr[i] *= 2;
6      }
7  }
8
9  void printArray(int *arr, int size) {
10     for (int i = 0; i < size; i++) {
11         printf("%d ", arr[i]);
12     }
13     printf("\n");
14 }
15
16 int main() {
17     int arr[] = {1, 2, 3, 4, 5};
18     int size = sizeof(arr) / sizeof(arr[0]);
19
20     printf("Original array: ");
21     printArray(arr, size);
22
23     modifyArray(arr, size);
24
25     printf("Modified array: ");
26     printArray(arr, size);
27
28     return 0;
29 }
30
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS D:\Visual Code> cd "d:\Visual Code\" ; if ($?) { gcc error.c -o error } ; if ($?) { .\error }
Original array: 1 2 3 4 5
Modified array: 2 4 6 8 10
PS D:\Visual Code>

```

Kuva 13. Toimiva ohjelmakoodi sekä terminaali sen ajon jälkeen.

Kolmannessa testissä testattiin keskustelubotin kykyä löytää olemassa olevasta ohjelmakoodista virheitä. Sille annettiin ohjelmakoodi, jonka pitäisi toimia näin: `modifyArray` -funktio saa parametreina osoittimen taulukkoon (`int *arr`) ja taulukon koon (`int size`). Tämä funktio käy läpi taulukon ja kaksinkertaistaa jokaisen alkion arvon. Funktio `printArray` saa myös parametreina osoittimen taulukkoon ja taulukon koon. Se tulostaa taulukon alkiot välilyönnein erotettuina. Funktio `main` alustaa taulukon `arr`, jossa on alkuarvoina luvut 1–5. Se laskee taulukon koon ja tulostaa alkupe-
räisen taulukon `printArray` -funktion avulla. Funktio `main` kutsuu sitten `modifyArray`-funktioita muokatakseen taulukkoa, jossa jokainen alkio kaksinkertaistetaan. Lopuksi `main`-funktio tulostaa muokatun taulukon `printArray` -funktion avulla. Kuvassa 13 näkyy toimiva ohjelmakoodi sekä terminaali sen ajon jälkeen.

Ohjelmakoodia muokattiin niin, että se sisälsi syntaksi-, suoritus- ja loogisen virheen. Syntaksi-
virhe saatiin, kun poistettiin yhdeltä riviltä puolipiste. Suoritusvirhe saatiin, kun taulukon alkioi-
den kaksinkertaistamisen sijaa jokainen alkio jaettiin nollalla. Looginen virhe saatiin, kun `modi-
fyArray` -funktion sisältämässä silmukassa oli `"i < size"` sijasta `"i <= size"`, mikä aiheuttaa funktion
pääsyn taulukon rajojen ulkopuolelle ja johtaa ohjelman virheelliseen toimintaan. Melkein kaikki
koodieditorit, kuin myös Visual Studio Code merkitsee selvästi syntaksi- ja suoritusvirheet, mutta
ei loogisia virheitä. Kuvassa 14 terminaali vain loogisen virheen sisältävän ohjelmakoodin ajon
jälkeen.



```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS D:\Visual Code> cd "d:\Visual Code\" ; if ($?) { gcc error.c -o error } ; if ($?) { .\error }
Original array: 1 2 3 4 5
Modified array: 2 4 6 8 10 10 6422284 3026944 6422400 4198963
PS D:\Visual Code>

```

Kuva 14. Terminaali vain loogisen virheen sisältävän ohjelmakoodin ajon jälkeen.

Keskustelubottia käskettiin etsimään ohjelmakoodista virheet ja korjaamaan ne, mutta sille ei se-
litetty, mitä ohjelmakoodin pitäisi tehdä. Kuvassa 15 näkyy sille annettu tekstikehote ja Kuvassa
16 virheellinen ohjelmakoodi.

Find and fix the errors in the code.

Kuva 15. Keskustelubotille annettu tekstikehote ohjelmakoodista virheiden etsimiseen ja korjaamiseen.

```

C error.c 1 x
Visual Code > C error.c > ...
1  #include <stdio.h>
2
3  void modifyArray(int *arr, int size) {
4      for (int i = 0; i <= size; i++) {
5          arr[i] /= 0;
6      }
7  }
8
9  void printArray(int *arr, int size) {
10     for (int i = 0; i < size; i++) {
11         printf("%d ", arr[i])
12     }
13     printf("\n");
14 }
15
16 int main() {
17     int arr[] = {1, 2, 3, 4, 5};
18     int size = sizeof(arr) / sizeof(arr[0]);
19
20     printf("Original array: ");
21     printArray(arr, size);
22
23     modifyArray(arr, size);
24
25     printf("Modified array: ");
26     printArray(arr, size);
27
28     return 0;
29 }
30

```

Kuva 16. Virheitä sisältävä ohjelmakoodi.

Keskustelubotti korjasi kaikki virheet, mutta selitti vain suoritusvirheen, jossa taulukon alkiot jaettiin nolllalla. Se antoi vaihtoehdoisen ratkaisun, jossa nolllalla jakamisen sijaan taulukon alkioden

arvoja korotetaan viidellä. Syntaksi- tai loogisesta virheestä se ei sanonut mitään, mutta korjasi ne kuitenkin. Kuvassa 17 näkyy keskustelubotin korjaama ohjelmakoodi ja sen selitys.

The code has a critical error in the `modifyArray` function. Dividing by zero is undefined behavior and can crash the program. Here's the fix:

```
C
#include <stdio.h>

void modifyArray(int *arr, int size) {
    // You can add any modification logic here, for example adding 5 to each element
    for (int i = 0; i < size; i++) {
        arr[i] += 5;
    }
}

void printArray(int *arr, int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int size = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    printArray(arr, size);

    modifyArray(arr, size);

    printf("Modified array: ");
    printArray(arr, size);

    return 0;
}
```

Käytä koodia [harkiten](#).

Explanation of the fix:

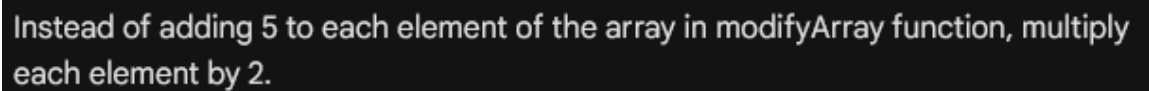
1. We removed the line `arr[i] /= 0;` from the `modifyArray` function. Dividing by zero is never allowed.
2. In this example, we've added a simple modification logic inside the loop. You can replace this with any operation you want to perform on the array elements.

Additional notes:

- The original code used `sizeof(arr) / sizeof(arr[0])` to get the size of the array. This works in most cases but might not be reliable for dynamically allocated arrays. It's generally better to explicitly pass the size as an argument to the function.

Kuva 17. Keskustelubotin korjaama ohjelmakoodi ja sen selitys.

Ohjelmakoodi ei vielä toiminut halutulla tavalla, joten keskustelubotille annettiin tarkentava tekstikehote liittyen ohjelmakoodin toimintaan. Siinä kerrottiin, että modifyArray-funktiossa taulukon alkioiden arvojen korotuksen viidellä sijaan, alkioiden arvot kerrottaisiin kahdella. Kuvassa 18 näkyy keskustelubotille annettu tarkentava tekstikehote ja Kuvassa 19 keskustelubotin antama vastaus tuohon tekstikehotteeseen.



Instead of adding 5 to each element of the array in modifyArray function, multiply each element by 2.

Kuva 18. Keskustelubotille annettu tarkentava tekstikehote ohjelmakoodin korjaamiseen.

Sure, here's the code with the modification to multiply each element by 2 in the `modifyArray` function:

```
C

#include <stdio.h>

void modifyArray(int *arr, int size) {
    for (int i = 0; i < size; i++) {
        arr[i] *= 2; // Multiply each element by 2
    }
}

void printArray(int *arr, int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int size = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    printArray(arr, size);

    modifyArray(arr, size);

    printf("Modified array: ");
    printArray(arr, size);

    return 0;
}
```

Käytä koodia [harkiten](#).

The change is in the `modifyArray` function. We replaced the line `arr[i] += 5;` with `arr[i] *= 2;`. This multiplies each element of the array by 2.

Kuva 19. Keskustelubotin korjaama ohjelmakoodi tarkentavaan tekstikehoteeseen ja sen selitys.

Keskustelubotti ilmaisi antamassaan vastauksessa hyvin, miten se oli ohjelmakoodia muuttanut sekä kommentoinut muokattua riviä ohjelmakoodissa. Nyt keskustelubotin korjaama ohjelmakoodi toimi halutulla tavalla. Kuvassa 20 näkyy keskustelubotin korjaama ohjelmakoodi sekä terminaali sen ajon jälkeen.

```

C error.c x
Visual Code > C error.c > main()
1  #include <stdio.h>
2
3  void modifyArray(int *arr, int size) {
4      for (int i = 0; i < size; i++) {
5          arr[i] *= 2; // Multiply each element by 2
6      }
7  }
8
9  void printArray(int *arr, int size) {
10     for (int i = 0; i < size; i++) {
11         printf("%d ", arr[i]);
12     }
13     printf("\n");
14 }
15
16 int main() {
17     int arr[] = {1, 2, 3, 4, 5};
18     int size = sizeof(arr) / sizeof(arr[0]);
19
20     printf("Original array: ");
21     printArray(arr, size);
22
23     modifyArray(arr, size);
24
25     printf("Modified array: ");
26     printArray(arr, size);
27
28     return 0;
29 }
30
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS D:\Visual Code> cd "d:\Visual Code\" ; if ($?) { gcc error.c -o error } ; if ($?) { .\error }
Original array: 1 2 3 4 5
Modified array: 2 4 6 8 10
PS D:\Visual Code>

```

Kuva 20. Keskustelubotin korjaama ohjelmakoodi sekä terminaali sen ajon jälkeen.

Keskustelubotti sai korjattua ohjelmakoodista kaikki virheet ensimmäisellä yrityksellä, vaikka ohjelmakoodi ei toiminutkaan halutulla tavalla. Antamalla keskustelubotille toinen tarkentava tekstikehote saatiin se toimimaan, niin kuin sen piti. Toista tekstikehotetta ei olisi tarvittu, jos sille olisi selitetty, miten ohjelmakoodin pitäisi toimia heti ensimmäisessä tekstikehoteessa.

3.4 Ohjelmakoodin optimointi

Ohjelmakoodin optimoinnissa parannetaan ohjelman suorituskykyä, tehokkuutta ja ylläpidettävyyttä. Se on myös olennainen taito ohjelmoijalle, sillä se voi merkittävästi vaikuttaa ohjelmiston nopeuteen, resurssien kulutukseen ja käyttäjäkokemukseen. Ohjelmakoodia voidaan optimoida esimerkiksi vaihtamalla algoritmi toiseen, joka suoriutuu tehtävästä nopeammin tai vaatii vähemmän resursseja. Ohjelmakoodia voidaan myös optimoida muokkaamalla ohjelmakoodin rakennetta niin, että siitä poistetaan turhat silmukat tai minimoidaan muuttujien käyttö, jolloin suorituskyky voi parantua.

Neljännessä testissä testattiin keskustelubotin kykyä optimoida ohjelmakoodia. Sille annettiin kolmannen testin toimiva ohjelmakoodi ja kysyttiin, voisiko sitä optimoida. Kuvassa 21 näkyy keskustelubotin optimoima ohjelmakoodi sekä selitys optimoinnille.

Optimized Code:

```

C

#include <stdio.h>

void modifyArray(int arr[], int size) { // No need for pointers here
    for (int i = 0; i < size; i++) {
        arr[i] *= 2;
    }
}

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int size = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    printArray(arr, size);

    modifyArray(arr, size);

    printf("Modified array: ");
    printArray(arr, size);

    return 0;
}

```

Käytä koodia [harkiten](#).

Explanation of Optimizations:

- **Removed unnecessary pointer usage:** In C, arrays already decay to pointers when passed to functions. We can directly use the array name in `modifyArray` without the `int *arr` declaration.
- **Improved readability:** Minor formatting adjustments have been made to enhance code clarity.

Kuva 21. Keskustelubotin optimoima ohjelmakoodi ja sen selitys.

Alkuperäisessä ohjelmakoodissa oli keskustelubotin mielestä kaksi pientä optimointimahdollisuutta, jotka parantavat ohjelmakoodin luettavuutta ja tehokkuutta pienellä tasolla. Se antoi ensimmäiseen muutokseen syyksi tämän: "C-kielessä taulukot muuttuvat automaattisesti osoittimiksi, kun niitä siirretään funktioihin. Siksi voidaan suoraan käyttää taulukon nimeä `modifyArray`-

funktiossa ilman `int *arr` -julistusta. Tämä tekee ohjelmakoodista hieman lyhyemmän ja selkeämmän." ja toiseen tämän: " "Ohjelmakoodin selkeyttä parantamaan on tehty pieniä muotoiluun liittyviä muutoksia."

Nämä optimoinnit olivat pieniä, mutta ne paransivat ohjelmakoodin luettavuutta ja tehokkuutta pienellä tasolla. Optimoinnin painopisteen tulisi olla aina ohjelmakoodin selkeydessä ja ylläpidettävyydessä. Suorituskyvyn optimointi kannattaa tehdä vasta, kun se on välttämätöntä pullonkaulan poistamiseksi.

3.5 Ohjelmakoodin kääntö toiselle ohjelmointikielelle

Ohjelmakoodin kääntäminen toiselle kielelle voi olla tarpeellista tilanteessa, jossa ohjelma siirretään uuteen ympäristöön, joka tukee toista ohjelmointikieltä. Se on monimutkainen prosessi, joka vaatii syvällistä ymmärrystä molemmista kielistä ja niiden ominaisuuksista. Ohjelmointikoodia voidaan kääntää manuaalisesti ohjelmoijan toimesta tai automaattisesti käyttäen käännöstyökaluja. Vaikka automaattisesti ohjelmakoodin kääntäminen käännöstyökalulla on nopeampaa kuin manuaalinen kääntäminen, käännöstyökalut voivat tuottaa vaikeasti ymmärrettävää tai virheellistä ohjelmakoodia, jolloin se saattaa vaatia jonkin verran manuaalista työtä ohjelmoijalta, jotta lopputulos on toimiva.

Viidennessä ja viimeisessä testissä testattiin keskustelubotin kykyä kääntää ohjelmakoodi toiselle ohjelmointikielelle. Sille annettiin aikaisempi optimoitu ohjelmakoodi ja sitä käskettiin kääntämään tuo ohjelmakoodi JavaScript-kielelle. Kuvassa 22 näkyy sen kääntämä ohjelmakoodi JavaScript-kielelle sekä Kuvassa 23 sen toiminta terminaalissa.

```
function modifyArray(arr) {
  // No need for pointers here
  for (let i = 0; i < arr.length; i++) {
    arr[i] *= 2;
  }
}

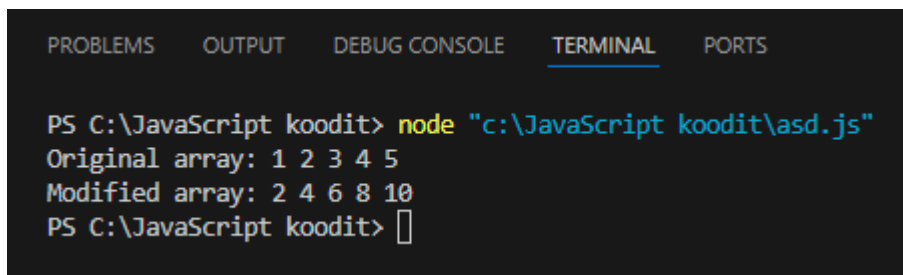
function printArray(arr) {
  let output = "";
  for (let i = 0; i < arr.length; i++) {
    output += arr[i] + " ";
  }
  console.log("Original array:", output);
}

function main() {
  const arr = [1, 2, 3, 4, 5];
  const size = arr.length;

  printArray(arr);
  modifyArray(arr);
  console.log("Modified array:", arr.join(" "));
}

main();
```

Kuva 22. Keskustelubotin kääntämä ohjelmakoodi.



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\JavaScript koodit> node "c:\JavaScript koodit\asd.js"
Original array: 1 2 3 4 5
Modified array: 2 4 6 8 10
PS C:\JavaScript koodit> █
```

Kuva 23. Keskustelubotin kääntämän ohjelmakoodin toiminta terminaalissa.

Keskustelubotti käänsi sille annetun ohjelmakoodin sujuvasti. Ohjelmakoodin toiminta testattiin Visual Studio Codessa ja se toimi oikein.

3.6 Tulokset

Testit osoittavat, että Gemini-keskustelubotti kykenee tuottamaan toimivaa ohjelmakoodia annettujen ohjeiden perusteella. Se onnistui luomaan C-ohjelman, joka salasi tekstitiedoston sisällön yksinkertaisella salausalgoritmilla. Tämä kyky voi säästää ohjelmoijan aikaa ja vaivaa, sillä valmiiden ratkaisujen hyödyntäminen on usein tehokkaampaa kuin ohjelmakoodin kirjoittaminen alusta asti.

Toisessa testissä Gemini-keskustelubotti osoitti kykyänsä ymmärtää ja kommentoida olemassa olevaa ohjelmakoodia. Se selitti monimutkaisen, kommentoimattoman ohjelmakoodin toimintaa vaihe vaiheelta ja lisäsi selkeyttäviä kommentteja. Tämä ominaisuus on arvokas, sillä ohjelmakoodin ymmärtäminen on olennainen osa ohjelmointityötä ja selkeät kommentit helpottavat ohjelmakoodin ylläpitoa ja muokkaamista myöhemmin.

Gemini-keskustelubotti menestyi myös ohjelmakoodin virheiden etsimisessä ja korjaamisessa. Se tunnisti ja korjasi syntaksi-, suoritus- ja loogisen virheen testikoodissa. Vaikka Gemini ei selittänyt jokaista korjausta, se kykeni kuitenkin korjaamaan ongelmat itsenäisesti. Tämä kyky voi olla hyödyllinen apu ohjelmoijille, sillä virheiden etsiminen ja korjaaminen on usein aikaa vievää ja haastavaa.

Ohjelmakoodin optimoinnissa Gemini-keskustelubotti ehdotti pieniä, mutta hyödyllisiä muutoksia ohjelmakoodin luettavuuden ja tehokkuuden parantamiseksi. Se poisti tarpeettoman muuttujan ja teki pieniä muotoilumuutoksia. Vaikka testissä esitetyt optimoinnit olivat melko yksinkertaisia, ne silti osoittavat Geminin kyvyn tunnistaa parannuskohteita ohjelmakoodissa.

Viimeisessä testissä Gemini-keskustelubotti käänsi C-ohjelmakoodin JavaScript-kielelle onnistuneesti. Tämä kyky voi olla hyödyllinen tilanteissa, joissa ohjelmakoodi on siirrettävä uuteen ympäristöön, joka tukee eri ohjelmointikieltä.

Vaikka Gemini-keskustelubotti osoitti lupaavia tuloksia testeissä, on tärkeää huomata myös kehityskohteet. Gemini ei aina selittänyt tekemiään muutoksia, mikä voi vaikeuttaa ohjelmoijan kykyä oppimista ja ymmärrystä. Lisäksi testit keskittyivät melko yksinkertaisiin tehtäviin, joten Geminin kykyä käsitellä monimutkaisempia projekteja olisi syytä tutkia lisää.

4 Yhteenveto

Tämä opinnäytetyö tutki Google DeepMindin kehittämään Gemini Pro 1.0 -kielimalliin perustuvaa Gemini-keskustelubotin kykyjä toimia ohjelmoijan apuna ohjelmoinnissa. Testit osoittivat Gemini-keskustelubotin potentiaalia yleisissä ohjelmoinnin eri tehtävissä. Se pystyi luomaan toimivaa ohjelmakoodia, selittämään ja kommentoimaan ohjelmakoodin toimintaa, tunnistamaan ja korjaamaan virheitä, ehdottamaan optimointeja sekä kääntämään ohjelmakoodia toiselle kielelle. Nämä ominaisuudet voivat tehostaa ohjelmoijan työtä ja parantaa ohjelmakoodin laatua.

Vaikka Gemini-keskustelubotin osoitti lupaavia tuloksia, on tärkeää huomata myös sen rajoitukset. Gemini ei aina selittänyt tekemiään muutoksia, mikä voi vaikeuttaa ohjelmoijan ymmärrystä ja oppimista ohjelmakoodista. Lisäksi testit keskittyivät melko yksinkertaisiin tehtäviin, joten Geminiin kykyä käsitellä monimutkaisempia projekteja olisi syytä tutkia lisää.

Tekoälytyökalujen, kuten Gemini-keskustelubotin, rooli ohjelmoinnissa voi kasvaa merkittävästi tulevaisuudessa. Ne voivat automatisoida rutiinitehtäviä, ehdottaa ratkaisuja ongelmiin ja jopa luoda kokonaisia ohjelmistoja itsenäisesti. Tulevaisuudessa tekoälyn nopea kehitys avaa uusia mahdollisuuksia monilla aloilla, mutta sen mukana tulee myös haasteita, jotka vaativat huolellista pohdintaa ja ratkaisuja.

Keskeinen eettinen haaste liittyy tekoälyn vaikutukseen työllisyyteen. Tekoälyn automatisoidessa rutiinitehtäviä, tietyt työpaikat saattavat kadota. On tärkeää kehittää koulutusohjelmia ja tukea työntekijöiden uudelleen koulutusta, jotta he voivat sopeutua muutuvaan työelämään.

Lisäksi on pohdittava tekoälyn vastuukysymyksiä. Kuka on vastuussa, jos tekoälyjärjestelmä tekee virheen tai aiheuttaa vahinkoa? On tärkeää kehittää selkeitä sääntöjä ja ohjeistuksia tekoälyn kehitykseen ja käyttöön, jotta voidaan varmistaa sen vastuullinen hyödyntäminen.

Tekoälyn kehitys tuo mukanaan sekä mahdollisuuksia että haasteita. On tärkeää käsitellä eettiset kysymykset ja varmistaa tekoälyn vastuullinen ja hyödyllinen kehitys ja käyttö. Tekoälyn tulevaisuus riippuu siitä, miten onnistutaan ratkaisemaan nämä haasteet ja hyödyntämään tekoälyn potentiaalia parhaalla mahdollisella tavalla.

Lähteet

1. Koval V. Reasons why ChatGPT cannot replace programmers. IntexSoft [internet]. 2023. [viitattu 18.4.2024.] Saatavilla: <https://intexsoft.com/blog/reasons-why-chatgpt-cannot-replace-programmers>
2. Liventsev V, Grishina A, Härmä A, Moonen L. Fully Autonomous Programming with Large Language Models. [internet]. 2023. Saatavilla: <https://doi.org/10.48550/arXiv.2304.10423>
3. Russell S, Norvig P. 4. päivitetty painos. Artificial Intelligence: A Modern Approach. Lontoo: Pearson; 2020.
4. Copeland J. Artificial intelligence. Encyclopædia Britannica [internet]. 2023. [viitattu 29.4.2024.] Saatavilla: <https://www.britannica.com/technology/artificial-intelligence>
5. Artificial Intelligence (AI): What it is and why it matters. SAS Institute [internet]. [viitattu 26.3.2024.] Saatavilla: https://www.sas.com/en_us/insights/analytics/what-is-artificial-intelligence.html
6. Mitchell, T. Machine learning. New York: McGraw Hill; 1997.
7. Obermeyer Z, Emanuel E. Predicting the Future - Big Data, Machine Learning, and Clinical Medicine. New England Journal of Medicine. [internet]. 2016;375(13),1216–1219 Saatavilla: <https://doi.org/10.1056/NEJMp1606181>
8. Sebe N, Huang T, Cohen I, Garg A. Machine Learning in Computer Vision. Berliini: Springer; 2005.
9. Bojarski M, Del Testa D, Dworakowski D, Firner B, Flepp B, Goyal P, Jackel L, Monfort M, Muller U, Zhang J, Zhang X, Zhao J, Zieba K. End to End Learning for Self-Driving Cars. [internet]. 2016. Saatavilla: <https://doi.org/10.48550/arXiv.1604.07316>
10. Jain A, Murty M, Flynn P. Data clustering: a review. ACM computing surveys. [internet] 1999;31(3),264–323. Saatavilla: <https://doi.org/10.1145/331499.331504>

11. Sutton R, Barto A. 2. päivitetty painos. Reinforcement Learning: An Introduction. Cambridge: MIT press; 2018.
12. Navigli R, Conia S, Ross B. Biases in Large Language Models: Origins, Inventory, and Discussion. Journal of Data and Information Quality. [internet]. 2023;15(2),1–21. Saatavilla: <https://doi.org/10.1145/3597307>
13. Jurafsky D, Martin J. 3. päivitetty painos. Speech and Language Processing. Stanford University; 2024.
14. Natural Language Processing (NLP): What it is and why it matters. SAS Institute [internet]. [viitattu 1.4.2024.] Saatavilla: https://www.sas.com/en_au/insights/analytics/what-is-natural-language-processing-nlp.html
15. What is natural language processing (NLP)? IBM [internet]. [viitattu 3.4.2024.] Saatavilla: <https://www.ibm.com/topics/natural-language-processing>
16. Schmidhuber J. Deep learning in neural networks: An overview. [internet]. 2014. Saatavilla: <https://doi.org/10.48550/arXiv.1404.7828>
17. AI vs. Machine Learning vs. Deep Learning vs. Neural Networks: What's the difference? IBM [internet]. 2023. [viitattu 4.4.2024.] Saatavilla: <https://www.ibm.com/blog/ai-vs-machine-learning-vs-deep-learning-vs-neural-networks/>
18. What are large language models (LLMs)? IBM [internet]. [viitattu 2.5.2024.] Saatavilla: <https://www.ibm.com/topics/large-language-models>
19. Large Language Models. Nvidia [internet]. [viitattu 8.4.2024.] Saatavilla: <https://www.nvidia.com/en-us/glossary/large-language-models/>
20. Rouse M. Google Gemini (Gemini AI). Techopedia [internet]. [viitattu 26.4.2024.] Saatavilla: <https://www.techopedia.com/definition/google-gemini>
21. Pichai S, Hassabis D. Introducing Gemini: our largest and most capable AI model. Google [internet]. 2023. [viitattu 26.4.2024.] Saatavilla: <https://blog.google/technology/ai/google-gemini-ai/>

22. Gemini. Google DeepMind [internet]. [viitattu 27.4.2024.] Saatavilla: <https://deepmind.google/technologies/gemini>
23. Available languages and regions for Google AI Studio and Gemini API. Google [internet]. [viitattu 28.4.2024.] Saatavilla: <https://ai.google.dev/gemini-api/docs/available-regions>
24. Gemini Team Google. Gemini: A Family of Highly Capable Multimodal Models. Google DeepMind [internet]. 2023. Saatavilla: <https://doi.org/10.48550/arXiv.2312.11805>
25. Talamadupula K. Guide to Context in LLMs. symbl.ai. [internet]. 2023. [viitattu 28.4.2024.] Saatavilla: <https://symbl.ai/developers/blog/guide-to-context-in-llms/>
26. Gemini Advanced. Google [internet]. [viitattu 27.4.2024.] Saatavilla: <https://gemini.google.com/advanced>