Tampereen yliopisto

Teemu Loijas

SKAALAUTUVAN JA TUOTANTOKELPOISEN KOODIN SEKÄ DOKUMENTAATION TUOTTAMINEN METAGPT-KEHYKSELLÄ

Informaatioteknologian ja viestinnän tiedekunta
Diplomityö
Toukokuu 2024

TIIVISTELMÄ

Teemu Loijas: Skaalautuvan ja tuotantokelpoisen koodin sekä dokumentaation tuottaminen MetaGPT-kehyksellä
Diplomityö
Tampereen yliopisto
Tietotekniikan diplomi-insinöörin koulutusohjelma
Toukokuu 2024

MetaGPT on uudehko moniagenttinen viitekehys, jonka avulla voidaan luoda sekä lähdekoodia että dokumentaatiota. Tämä viitekehys yhdistää kokonaisen sovellusyrityksen hyödyntämällä erilaisia agentteja eri rooleissa. Näitä rooleja ovat pomo, tuotepäällikkö, arkkitehti, projektipäällikkö, insinööri sekä laadunvarmistaja. MetaGPT:llä voidaan luoda projekteja antamalla yhden rivin mittainen käsky.

Työssä tutkitaan MetaGPT:n tuottaman koodin ja dokumentaation tuotantokelpoisuutta sekä skaalautuvuutta. On tärkeää pystyä luomaan laadukasta ja skaalautuvaa koodia, jotta kehyksen hyödyistä pystytään oikeasti nauttimaan. Työssä luotiin Fastapi-pohjainen Python-sovellus, jonka lisäksi luotiin erillinen kyselyjä rajoittava moduuli. Moduuli jouduttiin lisäämään alkuperäiseen sovellukseen projektien lähdekooditiedostoja yhdistelemällä. Työssä käytetty tapa johtuu siitä, että moodulille tuotetussa dokumentaatiossa ei kerrota, miten moduulia tulisi käyttää sellaisenaan. Projektien dokumentaatioita ei yhdistelty, vaan todettiin, että kokonaan uuden dokumentaation kirjoittaminen olisi järkevin vaihtoehto.

Suurien koodimäärien tuottaminen tuntuu mahdottomalta eikä inkrementaalista koodin ja dokumentaation luontia tueta. Dokumentaatioiden yhdisteleminen on vaikeaa. Toimivan koodin luominen on vaikeaa ja samaan aikaan hieman eksperimentaalista. Halutun lopputuloksen saamiseksi voidaan joutua luomaan useita projekteja eikä siltikään välttämättä päästä haluttuun lopputulokseen. Todettiin, että MetaGPT ei pysty tuottamaan täysin tuotantokelpoista koodia eikä dokumentaatiota 0.3.0 versiolla. Myös skaalautuvuus on huonoa.

Avainsanat: MetaGPT, OpeanAI, skaalautuvuus, tuotantokelpoisuus

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck –ohjelmalla.

ABSTRACT

Teemu Loijas: Generating scalable and production-ready code and documentation with MetaGPT framework
Master of Science Thesis
Tampere University
Information Technology
May 2024

MetaGPT is a new multi-agent framework that can be used to create both source code and documentation. This framework connects an entire application enterprise by utilizing different agents in different roles. These roles include boss, product manager, architect, project manager, engineer and quality assurance. Projects can be created with MetaGPT by issuing a one-line command.

The thesis examines the production readiness and scalability of the code and documentation produced by MetaGPT. It is important to be able to create high quality and scalable code in order to really enjoy the benefits of the framework. In the thesis, a FastAPI based Python application was created, in addition to which a separate rate limiting module was created. The module had to be added to the original application by combining the source code files of the projects. The method used in the thesis is due to the fact that the documentation produced for the module does not tell how the module should be used as such. The documentation of the projects was not combined, but it was concluded that writing a completely new documentation would be the most reasonable option.

Producing large amounts of code seems impossible and incremental creation of code and documentation is not supported. Combining documentation is difficult. Creating working code is difficult and at the same time somewhat experimental. In order to get the desired end result, you may have to create several projects and still not necessarily reach the desired end result. It was found that MetaGPT is unable to produce fully production-ready code and documentation with version 0.3.0. Scalability is also poor.

Keywords: MetaGPT, OpenAI, scalability, production readiness

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

ALKUSANAT

MetaGPT on tuore moniagenttinen viitekehys, joka yhdistää lähdekoodin ja dokumentaation luomisen yhden rivin mittaisella komennolla. MetaGPT käyttää OpenAI:n tai muiden vastaavien palveluiden rajapintoja tekoälymallien hyödyntämiseksi. Nämä tekoälymallit luovat lähdekoodit sekä dokumentaatiot yhdessä MetaGPT:n kanssa.

Päätin tehdä diplomityöni tästä aiheesta, sillä etenkin lähdekoodin tuottaminen tekoälyn avustuksella on kasvanut huomattavasti viime vuosina. Työssä itseäni kiehtoo etenkin tuotetun koodin laatu ja määrä sekä tuotetun dokumentaation laatu. Useat sovelluskehittäjät olisivat iloisia, jos tekoäly pystyisi lähdekoodin lisäksi tuottamaan luotettavan ja laadukkaan dokumentaation samalla komennolla.

Haluaisin kiittää diplomityöohjaajiani Terhi Kilamoa ja Pekka Abrahamssonia arvokkaasta palautteesta, sparrauksesta ja oikeaan suuntaan ohjaamisesta työn aikana.

Tampereella, 16.5.2024

Teemu Loijas

SISÄLLYSLUETTELO

1. JOHDANTO 8
2. KIELIMALLIT, SKAALAUTUVUUS JA TUOTANTOKELPOISUUS 10
2.1 Kielimallit 10
2.2 Lähdekoodin ja dokumentaation skaalautuvuus 13
2.2.1 Skaalautuvuuden luokat 14
2.2.2 Skaalautuvuuden mallit 16
2.2.3 Skaalautuvuuden parantaminen 16
2.3 Lähdekoodin ja dokumentaation tuotantokelpoisuus 17
3. TUTKIMUSMENETELMÄ JA -KYSYMYS 19
3.1 Suunnittelututkimus yleisesti 19
3.2 Suunnittelututkimus tässä työssä 21
3.3 Tutkimuskysymys 21
4. TEKNOLOGIAT JA VIITEKEHYKSET 23
4.1 MetaGPT 23
4.2 Python 23
4.3 Pythonin virtuaaliympäristö 24
4.4 FastAPI 25
4.5 Docker 25
5. TOTEUTUS 28
5.1 MetaGPT:n asentaminen 28
5.2 MetaGPT:n konfigurointi 28
5.3 Testitapausten perusteella tehdyt huomiot 29
5.4 Fastapi-pohjaisen projektin luominen 29
5.5 Luodut tiedostot 29
5.5.1 Dokumentaatio 29
5.5.2 Lähdekoodi 31
5.6 Pythonin virtuaaliympäristön käyttäminen 31
5.7 Tarvittavat muutokset 32
5.7.1 Muutokset konfiguraatiotiedostoissa 32
5.7.2 Muutokset lähdekoodissa 33
5.8 Rate limit -moduulin luominen ja integroiminen olemassa olevaan projektiin 33
5.9 Syntyneen sovelluksen testaaminen 34
5.10 Inkrementaalinen koodin tuottaminen 36
6. TULOKSET JA POHDINTA 37
7. YHTEENVETO 40
LÄHTEET 41

LIITE A: SOVELLUKSEN MAIN.PY TIEDOSTO 42
LIITE B: RAJAPINNAN DOKUMENTAATIO 44
LIITE C: TUOTEVAATIMUSDOKUMENTTI 47
LIITE D: JÄRJESTELMÄSUUNNITTELUDOKUMENTTI 50

LYHENTEET JA MERKINNÄT

CI/CD	engl, Continuous Integration & Continuous Delivery. Jatkuva integrointi ja jatkuva toimitus
GPT	engl, Generative pre-trained transformer, generatiivinen esiopetettu muuntaja
HTTP	engl, Hypertext Transfer Protocol, hypertekstin siirtoprotokolla
Kubernetes	Avoimen lähdekoodin ohjelmisto konttien hallintaan suuremmissa mittakaavoissa
LLM	engl, Large Language Model, suuri kielimalli
On-premise	Paikan päällä. Käytetään usein esimerkiksi palvelimista, jotka eivät ole virtuaalisia tai pilvessä.
Overhead	Mikä tahansa ylimääräisen tai epäsuoran laskenta-ajan, muistin, kaistanleveyden tai muiden resurssien yhdistelmä, jota tarvitaan tietyn tehtävän suorittamiseen
PRD	engl, Product Requirements Document. Tuotevaatimusdokumentti
Stacktrace	Pinojälki

1. JOHDANTO

Nykymaailmassa tekoäly kasvattaa jatkuvasti rooliaan ohjelmistokehityksessä. Useimmiten tekoälyllä luodaan ainoastaan joko lähdekoodia tai dokumentaatiota.Tekoälyllä tuotetun materiaalin laatu on usein kyseenalaista. Vuonna 2023 on julkaistu ohjelmistokehys nimeltä MetaGPT, joka mahdollistaa sekä lähdekoodin että dokumentaation luomisen yhdellä komentorivikomennolla tekoälyä käyttäen.

Lähdekoodin ja dokumentaation luominen samanaikaisesti tekoälyllä voi parhaimmillaan tehostaa sovelluskehitystä huomattavasti. Tehokkuuden kasvattaminen toki vaatii hyvälaatuista koodia sekä dokumentaatiota. Jos tekoälyn avulla pystytään tuottamaan laadukasta materiaalia, voidaan saavuttaa myös huomattavia rahallisia sekä aikataulullisia säästöjä.

Tässä työssä tutkitaan MetaGPT:n luoman koodin ja dokumentaation skaalautuvuutta ja tuotantokelpoisuutta. Tutkimuskysymys: Voidaanko MetaGPT:n luomaa ohjelmistokoodia sekä dokumentaatiota sekä skaalata tehokkaasti että hyödyntää tuotantoympäristöissä?

MetaGPT:n kielimallina käytettin OpenAI:n gpt-4 kielimallia. Koodien ja dokumentaation luomiseksi MetaGPT:n koodiin tarvittiin muokkauksia. Liiallisen ja liian nopean koodimassan sekä dokumentaation luomisen takia koodin suorittamiseen tarvitsi laittaa taukoja, jotta OpenAI:n rajapintojen rajoitukset voitiin ohittaa.

Työssä rakennettiin Pythonilla FastAPI pohjainen rajapintasovellus, johon MetaGPT:llä voidaan luoda uusia Python moduuleita, joiden integroiminen olemassa olevaan koodiin tulisi olla suhteellisen helppoa. Työssä luotiin konfiguroitava rajapintojen rajoitusmoduuli, jonka luominen osoittautui yllättävän haastavaksi. Moduulia jouduttiin generoimaan useita kertoja, jotta saatiin toimivaa koodia, joka samalla pystyttiin integroimaan olemassa olevaan koodiin.

Työssä todetaan MetaGPT:n koodin ja dokumentaation skaalautuvan huonosti. MetaGPT osaa tuottaa Dockerfilejä, joiden avulla voidaan luoda Docker imageja, joita voidaan ajaa skaalautuvassa ympäristössä. Ongelmana on kuitenkin se, ettei MetaGPT:n luoma koodi ole suoraan skaalautuvaa varsinkaan monen Docker imagen ympäristössä, esimerkiksi Kubernetes klusterissa.

MetaGPT tarvitsee lisäkehitystä voidakseen olla hyödyllinen skalaautuvassa tuotantoympäristössä. Myös koodin laatu on hyvin vaihtelevaa ja MetaGPT:tä käyttävä

sovelluskehittäjä joutuu käymään tuotetun koodin lävitse ennen kuin se voidaan laittaa ajoon mihinkään ympäristöön. MetaGPT:n nykytilanteessa koodin kirjoittaminen itse on useasti nopeampi ja turvallisempi vaihtoehto kuin sen generointi. MetaGPT:n hyvänä puolena voidaan kuitenkin pitää sen tuottamaa dokumentaatiota, joka on usein sovelluskehittäjille pakkopullaa.

2. KIELIMALLIT, SKAALAUTUVUUS JA TUOTANTOKELPOISUUS

2.1 Kielimallit

Kielimallit ovat monien teknologiakokemustemme perusta. Kielimalli on tekniikka tietyn sanasarjan esiintymisen todennäköisyyden laskemiseksi. Se yrittää jäljitellä tiettyjä ihmisen kielellisiä kykyjä oppien lukemattomia assosiaatioita sanojen välillä riippuen kontekstista, käyttäjän omista kielimalleista, muiden ihmisten kaavoista jne. Voidaan sanoa, että kielimalli pelaa aina peliä, jossa se yrittää ennustaa seuraavan sanan. Kielimallit käyttävät kolmea yleistä lähestymistapaa kieltä edustavan tiedon enkoodaamiseen: kielellistä, todennäköisyydellinen/tilastollinen sekä upotuksellista. (Eovito, Danilevsky 2022)

Kielellisessä lähestymistavassa malli enkoodaa kielioppirakenteet manuaalisesti, mikä on vaikeaa ja hidasta tehdä, mutta on samanaikaisesti selkeää ja täysin selitettävissä olevaa. Malli edustaa kieltä säännöstönä. Esimerkkinä voidaan käyttää englannin kielen sääntöä, jossa adjektiivi tulee yleensä ennen substantiivia, jota se muokkaa (”red book, eikä ”book red”), vaikka poikkeuksiakin on. (Eovito, Danilevsky 2022)

Todennäköisyydellisessä/tilastollisessa lähestymistavassa viitetekstiä käyttävä malli laskee sanan esiintymät ja luottaa niihin, kun malli koittaa ennustaa seuraavaa sanaa. Kielimalli voi oppia todennäköisyydet minkä tahansa kokoiselle sanasarjalle. Malli edustaa kieltä sanajonojen ja niihin liittyvien todennäköisyyksien joukkona. Nykyaikaisiin hermoverkkokielimalleihin (neural networks) verrattuna tilastolliset kielimallit ovat paljon paremmin ymmärrettyjä, mutta niiden suosio on hiipumassa. (Eovito, Danilevsky 2022)

Upotuksessa malli edustaa jokaista kielen sanaa vektorina suuressa ulottuvuudessa. Tätä mallia käytetään ensisijaisesti syvän neuroverkon kielimalleissa. Upotukset on lähestymistavoista vaikein ymmärtää. Sitä käytetään useimmissa huipputekniikan kielimalleissa. (Eovito, Danilevsky 2022)

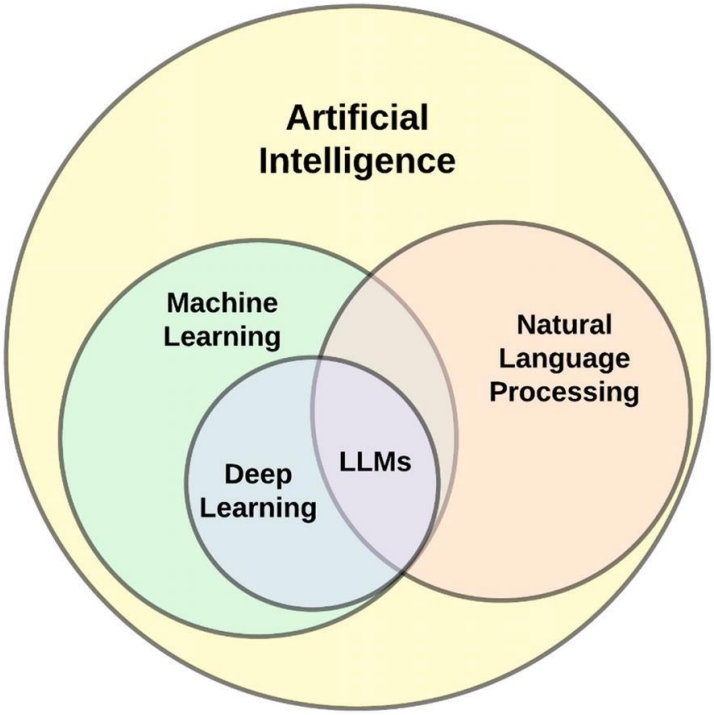
Kielimalli on luonnollisen kielen todennäköisyysmalli. Kielimallit ovat hyödyllisiä moniin erilaisiin tehtäviin, kuten puheentunnistukseen, konekäännöksiin, luonnollisen kielen luomiseen, optiseen merkin tunnistukseen, käsin kirjoituksen tunnistukseen, kielioppiin sekä tiedonhakuun. Suuret kielimallit, tällä hetkellä edistynein muotonsa, ovat yhdistelmä suurempia tietojoukkoja, eteenpäin vieviä neuroverkkoja (feedforward neural networks) ja muuntajia. Ne ovat syrjäyttäneet niin sanotut toistuvat

hermoverkkopohjaiset mallit (recurrent neural network-based models), jotka olivat aiemmin syrjäyttäneet puhtaat tilastolliset mallit.

Suuri kielimalli on kielimalli, joka on tunnettu kyvystään saavuttaa yleiskäyttöinen kielen ymmärtäminen ja luominen. Suuret kielimallit hankkivat kyvyt oppimalla tilastollisia suhteita tekstidokumenteista laskennallisesti intensiivisen itseohjatun ja puolivalvotun koulutusprosessin aikana. Suuret kielimallit ovat keinotekoisia hermoverkkoja, jotka seuraavat muuntaja-arkkitehtuuria. Niitä voidaan käyttää tekstin luomiseen ottamalla syöteteksti ja ennustamalla toistuvasti seuraava merkki tai sana. Vuoteen 2020 asti hienosäätö oli ainoa tapa mukauttaa mallia tiettyjen tehtävien suorittamiseen. Suuremmat mallit, kuten GPT-3, voidaan kuitenkin suunnitella nopeasti samanlaisten tulosten saavuttamiseksi. Niiden uskotaan hankkivan tietoa syntaksista, semantiikasta ja ontologiasta. (Thimira Amaratunga, 2023)

Merkittäviä esimerkkejä ovat OpenAI:n GPT-mallit (esim. GPT-3.5 ja GPT-4, joita käytetään ChatGPT:ssä), Googlen PaLM (käytetään Bardissa) ja Metan LLaMA sekä BLOOM, Ernie 3.0 Titan ja Anthropicin Claude 2. Vaikka edellä mainitut joskus vastaavat ihmisen suorituskykyä, eivät ne kuitenkaan ole täysin uskottavia kognitiivisia malleja. Ainakin toistuvien hermoverkkojen osalta on osoitettu, että ne joskus oppivat malleja, joita ihmiset eivät opi, mutta eivät opi malleja, joita ihmiset yleensä oppivat. (Thimira Amaratunga, 2023)

Suuret kielimallit ovat tulosta luonnollisen kielen käsittelyn, syvän oppimisen käsitteiden ja generatiivisten tekoälymallien yhdistelmästä (Kuva 1). (Thimira Amaratunga, 2023)



Kuva 1. Suuret kielimallit tekoälymaisemassa (Thimira Amaratunga, 2023)

Suurista kielimalleista on tullut tuttu nimi sen ansiosta, että ne ovat tuoneet generatiivisen tekoälyn yleisen edun eturintamaan, sekä pisteen, johon organisaatiot keskittyvät ottamaan käyttöön tekoälyä lukuisissa liiketoimintatoiminnoissa ja käyttötapauksissa.

Suuret kielimallit ovat perusmallien luokka, jotka on koulutettu valtaviin tietomääriin, jotta ne voivat tarjota perustavanlaatuiset ominaisuudet, joita tarvitaan useiden käyttötapausten ja sovellusten ohjaamiseen sekä monien tehtävien ratkaisemiseen. Tämä on jyrkässä ristiriidassa ajatuksen kanssa rakentaa ja kouluttaa toimialuekohtaisia malleja kullekin näistä käyttötapauksista erikseen, mikä on kohtuutonta useilla kriteereillä, tukahduttaa synergiaetuja ja voi jopa johtaa huonompaan suorituskykyyn.

Suuret kielimallit edustavat merkittävää läpimurtoa luonnollisen kielen käsittelyssä ja tekoälyssä, ja ne ovat helposti saatavilla käyttöliittymien, kuten Open AI:n Chat GPT-3 ja GPT-4, kautta, jotka ovat saaneet Microsoftin tuen.

Suuret kielimallit on suunniteltu ymmärtämään ja luomaan tekstiä ihmisen tavoin muun sisällön lisäksi niiden kouluttamiseen käytetyn valtavan datamäärän pohjalta. Ne pystyvät päättelemään asioita kontekstista riippuen, luomaan johdonmukaisia ja kontekstiin liittyviä vastauksia, kääntämään tekstiä muihinkin kieliin kuin englantiin, tiivistämään tekstiä, vastaamaan kysymyksiin ja jopa avustamaan luovassa kirjoittamisessa tai koodin luomisessa.

Ne pystyvät tekemään edellä mainittuja asioita miljoonien parametrien ansiosta, joiden avulla he voivat vangita kielen monimutkaisia malleja ja suorittaa laajan valikoiman kieleen liittyviä tehtäviä. Suuret kielimallit mullistavat sovelluksia eri aloilla chatboteista ja virtuaaliassistenteista sisällöntuotantoon, tutkimusapuun ja kielten kääntämiseen. Samalla kun suuret kielimallit kehittyvät ja paranevat, ne ovat valmiita muokkaamaan tapaamme olla vuorovaikutuksessa teknologian kanssa ja saada tietoa, mikä tekee niistä keskeisen osan nykyaikaista digitaalista maisemaa. (IBM, 2024)

Esimerkkinä suurista kielimalleista on OpenAI:n GPT-4. GPT-4 on suuri multimodaalinen malli, joka voi ratkaista vaikeita ongelmia tarkemmin kuin mikään aikaisemmista OpenAI:n malleista laajemman yleistietonsa ja kehittyneen päättelykykynsä ansiosta. GPT-4 malli on optimoitu keskustelulle, mutta toimii hyvin perinteisissä täydennystehtävissä kuten edeltäjänsä gpt-3.5-turbo. (OpenAI, 2024)

2.2 Lähdekoodin ja dokumentaation skaalautuvuus

Teknisestä ja taloudellisesta merkityksestään huolimatta lähdekoodin skaalautuvuudelle ei ole yleisesti hyväksyttyä määritelmää. Tässä tutkimuksessa määritelmänä käytetään seuraavaa (Williams and Smith 2004):

”Skaalautuvuus on sovellusjärjestelmän kyky – ilman muutoksia – kustannustehokkaasti tarjota suurempi suorituskyky, lyhyempi vasteaika ja/tai tukea useampia käyttäjiä laitteistoresurssien lisäyksen johdosta.”

Sovellus voidaan skaalata lisäämään suorituskykyä, käsittelemään enemmän käyttäjiä ja/tai lyhentämään vasteaikaa.

On mahdollista erottaa kaksi perusskaalausstrategiaa: vertikaalinen ja horisontaalinen. Vertikaalisessa skaalauksessa resursseja, joita ovat prosessori, muisti ja levy, lisätään yksittäiselle palvelimelle kapasiteetin kasvattamiseksi. Horisontaalisessa skaalauksessa sovelluksesta voidaan laittaa ajoon useita instansseja tai voidaan käyttää useita palvelimia kapasiteetin kasvattamiseksi. Horisontaalinen ja vertikaalinen skaalaus voivat tuottaa hyvin erilaisia tuloksia järjestelmästä riippuen. Tämän lisäksi niillä voi olla hyvin erilaiset kustannukset ja kustannusvaikutukset.

Yhden pullonkaulan poistaminen voi tuoda esiin toisen pullonkaulan. Esimerkiksi palvelimen kapasiteettia voidaan kasvattaa lisäämällä prosessoreita, mutta jossain kohtaa huomataan toisen resurssin, kuten levyn, olevan pullonkaula. Tällaisessa tilanteessa skaalausstrategiaa on muutettava tämän toissijaisen pullonkaulan ratkaisemiseksi. (Williams, Smith, 2004)

2.2.1 Skaalautuvuuden luokat

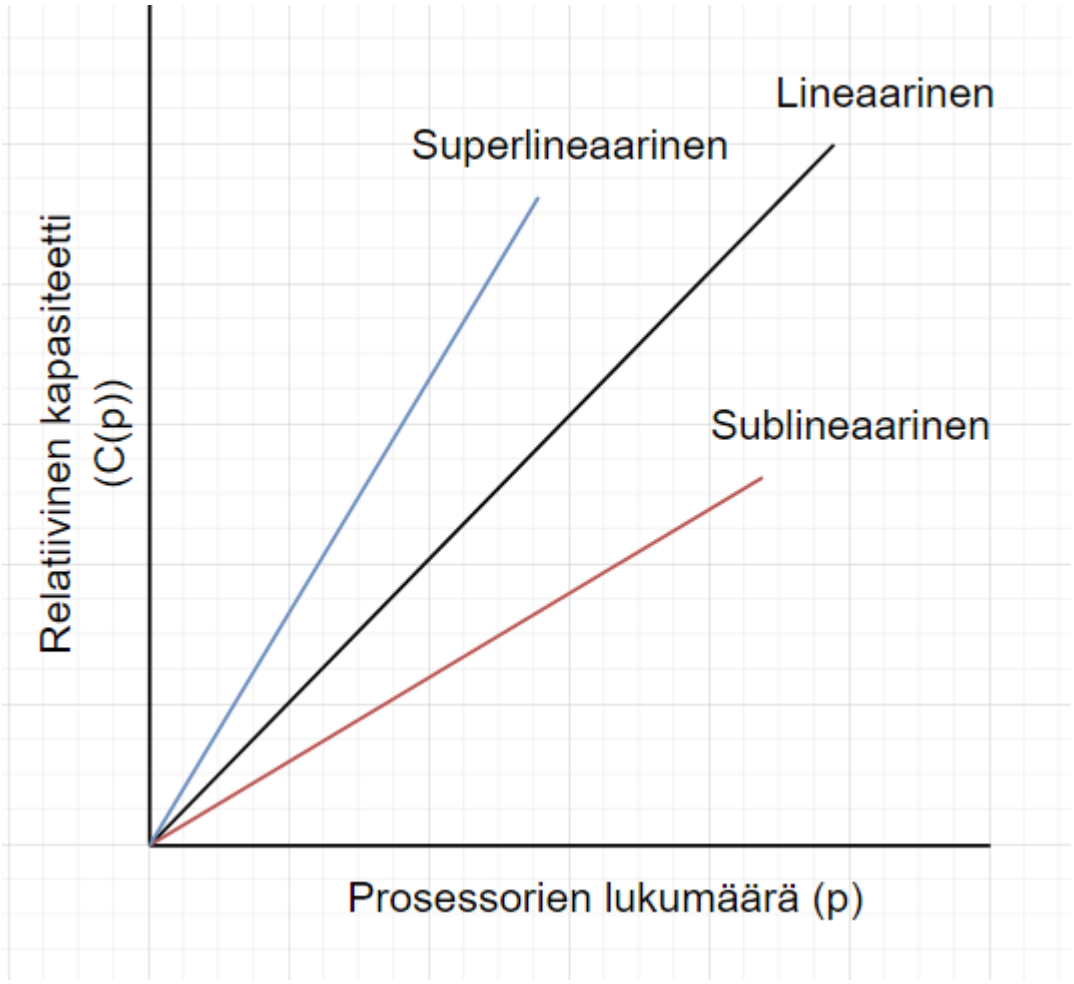
Skaalautuvuus voidaan jakaa kolmeen eri luokkaan. Tämä luokitusjärjestelmä on samanlainen kuin Alban ehdottama luokitusjärjestelmä (Alba 2002). Nämä luokat ovat lineaarinen skaalautuvuus, sublineaarinen skaalautuvuus sekä superlineaarinen skaalautuvuus.

Lineaarinen skaalautuvuus tarkoittaa, että järjestelmän kapasiteetti p-prosessorilla on p kertaa yhden prosessorin kapasiteetti. Tämä tarkoittaa, että prosessorien määrän tuplaaminen tulee kasvattamaan kapasiteettia kaksinkertaiseksi.

Sublineaarinen skaalautuvuus tarkoittaa, että p-prosessorin järjestelmän kapasiteetti on pienempi kuin p kertaa kapasiteetti yhdellä prosessorilla. Tämä tarkoittaa, että prosessorien kaksinkertaistaminen ei kasvata järjestelmän kapasiteettia kaksinkertaiseksi.

Superlineaarinen skaalautuvuus tarkoittaa, että p-prosessorin järjestelmän kapasiteetti on yli p-kertainen yhdellä prosessorilla. Tämä tarkoittaa, että prosessorien kaksinkertaistaminen yli kaksinkertaistaa järjestelmän kapasiteetin.

Eri skaalautuvuusstrategiat on kuvattu kuviossa 1. Kuvion käyrät ovat ainoastaan suuntaa antavia.



Kuvio 1. Skaalautuvuuskategoriat

Lineaarista skaalautuvuutta voi esiintyä, jos rinnakkaisuuden aste sovelluksessa on sellainen, että se pystyy hyödyntämään tarjottuja lisäresursseja täysimääräisesti skaalaamalla. Esimerkkinä järjestelmä, joka vastaanottaa dataa useista eri lähteistä, käsittelee datan ja valmistelee datan jälkikäyttöä varten. Tällaisen järjestelmän voi olla mahdollista ajaa useita virtoja rinnakkain kapasiteetin lisäämiseksi. Jotta edellä mainittu sovellus skaalautuisi lineaarisesti, virrat eivät saa häiritä toisiaan (esimerkiksi kilpailemalla tietokannasta tai muista yhteisistä resursseista) tai ne eivät saa hyödyntää sovelluksen yhteistä tilaa. Jos virtoja häiritään edes toisella edellä mainitulla tavalla, skaalautuvuus vähenee alle lineaariseen. (Williams ja Smith 2004)

Sublineaarista skaalautuvuutta esiintyy, kun järjestelmä ei pysty käyttämään täysimääräisesti sille annettuja lisäresursseja. Sublineaarinen skaalautuvuus voi johtua esimerkiksi järjestelmän ominaisuuksista. Jos järjestelmä joutuu odottamaan resursseja, kuten tietokantaa ja sen lukitusta, järjestelmä ei pysty hyödyntämään kaikkia sille annettuja lisäprosessoreita. Sublineaarisuus voi järjestelmän ominaisuuksien lisäksi johtua suoritusympäristön ominaisuuksista, jotka vähentävät lisäprosessoreiden

prosessointitehoa: aikataulutuksen overhead, prosessorien välinen kiista jaetuista resursseista tai prosessorien välinen viestintä globaalin tilan ylläpitämiseksi. Nämä muuttujat voivat aiheuttaa suhteellisen kyvyn kasvaa hitaammin kuin lineaarisesti. (Williams ja Smith 2004)

Superlineaarinen skaalautuvuus vaikuttaa ensisilmäyksellä mahdottomalta. Superlineaarisuus on kuitenkin todellinen ilmiö. Se voidaan tunnistaa esimerkiksi tilanteessa, jossa järjestelmään lisätään prosessorin lisäksi muistia, levyä, verkkoliitäntöjä tai muita vastaavia resursseja. Nämä lisäresurssit kasvattavat muutakin kuin pelkkää prosessointitehoa. Tästä syystä prosessointikyky kasvaa yli lineaarisesti. Esimerkiksi, jos prosessorin lisäksi lisätään muistia, voi olla mahdollista tallentaa tietoja välimuistiin ja eliminoida tietokantakyselyt tietojen hakemiseksi. (Williams ja Smith 2004)

2.2.2 Skaalautuvuuden mallit

Sovellusten skaalautuvuuden analysointiin ja ennustamiseen käytettäviä malleja ovat (Williams ja Smith 2005):

- Lineaarinen skaalautuvuus
- Amdahlin laki
- Supersarjamalli
- Gustafsonin laki

Super-sarjamalli on Amdahlin lain jatke. Amdahlin laki ja Gustafsonin laki olivat kehitetty rinnakkaisten algoritmien ja arkkitehtuurien nopeuttamisen yhteydessä. Amdahlin laki ja supersarjamalli kuvaavat sublineaarista skaalutuvuutta. Gustafsonin laki kuvaa superlineaarista skaalautuvuutta. Voidaan todeta, ettei jokaisen järjestelmän tai sovelluksen skaalautuvuutta voida tutkia näiden mallien pohjalta, sillä kaikki järjestelmät eivät ole yhteensopivia edellä mainittujen mallien kanssa.

2.2.3 Skaalautuvuuden parantaminen

Skaalautuvuus ei ole ainoastaan laitteisto-ongelma. Prosessorien lisääminen on yleisin tapa skaalata järjestelmää, mutta ei kuitenkaan aina kustannustehokkain. Usein on mahdollista lisätä järjestelmän skaalautuvuutta yksinkertaisten ohjelmistoarkkitehtuuristen muutosten avulla. Ongelmajärjestelmissä tämä voi olla ainoa tapa saavuttaa skaalautuvuusvaatimukset.

Ohjelmistolähtöisiä menetelmiä skaalautuvuuden parantamiseksi (Williams ja Smith 2005):

- Kysynnän vähentäminen pullonkaularesurssissa
- Sovelluksen rinnakkaisuuden lisääminen
- Yhteisistä resursseista käytävän kilpailun vähentäminen
- Prosessorien välisen viestinnän vaikutusten vähentäminen

Valinta laitteisto- ja ohjelmistolähtöisistä lähestymistavoista skaalautuvuuden parantamiseksi riippuu useista tekijöistä. Järjestelmän skaalautuvuuden määrällinen arvio tarjoaa lähtökohdan näiden vaihtoehtojen punnitsemiselle.

2.3 Lähdekoodin ja dokumentaation tuotantokelpoisuus

Bigelowin (Bigelow, 2021) mukaan lähdekoodin tuotantokelpoisuutta tutkitaan koodin virheettömyydellä sekä sillä, miten sovellus selviää erilaisista mahdollisista virhetilanteista. Koodin virheettömyydellä tarkoitetaan koodin logiikan oikeellisuutta. Tuotantokelpoisen lähdekoodin tulee selvitä erilaisista virhetilanteista niin, että sovellus pysyy pystyssä virhetilanteista huolimatta.

Dokumentaation tuotantokelpoisuutta tutkitaan tarkastelemalla, että vastaako luotu dokumentaatio todellisuutta.

Tuotantokelpoisuuden tärkeimpiä ominaisuuksia ovat ominaisuudet ja toiminnallisuus, virheet ja niiden määrät, suorituskyky, käyttäjän hyväksyntä tai kokemus sekä turvallisuus.

Ominaisuudet ja toiminnallisuus tarkoittaa sitä, että sovelluksen tulee tehdä kaikki tuotevaatimusdokumentissa määritellyt asiat. Ohjelmistotestauksen periaate on validoida sovellusversio mitä tahansa dokumentoitua vaatimusta vastaan, joka on mainittuna esimerkiksi vaatimusmäärittelyssä.

Nykyaikaisessa ketterässä kehityksessä puuttuvat toiminnot ovat yleisiä ja täysin normaaleja, koska kehitystiimi julkaisee usein ohjelmiston uusia iteraatioita. Ohjelmiston on kuitenkin täytettävä tietyt perusvaatimukset ennen ensimmäistä käyttöönottoa. Käyttäjien on esimerkiksi voitava käyttää sovellusta sen tärkeimpiin tarkoituksiin.

Ei riitä, että julkaistava sovellusversio suorittaa tärkeimmät ominaisuudet ja toiminnot. Nämä on tehtävä tarkasti ja tuotettava odotetut vastaukset tiedettyihin syötteisiin. Testaajien tulee etsiä vakaata versiota, jonka virheprosentit ovat hyväksyttävissä tai vahvistetuissa rajoissa. Virheet ja niiden liian suuri määrä voivat viivästyttää

käyttöönottoa. Korkeat virheprosentit voivat jopa kyseenalaistaa ohjelmiston laadun ja käynnistää laajemman tuotteen arkkitehtuurin ja koodauksen tarkastelun.

Ohjelmistoihin sovelletaan usein suorituskykyvaatimuksia tai ohjelmistomäärittelyn määrittelemiä mittareita. Mittareita voivat muun muassa olla:

- Laskentanopeudet, kuten tapahtumat sekunnissa tai vasteaika
- Infrastruktuurin tai resurssien käyttö, kuten sovelluksen prosessorin käyttö
- Verkkometriikka, kuten viive tai samanaikaiset käyttäjäyhteydet ja muut tietopisteet
- Muut asiat, joita kehitystiimi pitää tärkeänä

Laadunvarmistusammattilaiset suorittavat usein suorituskykytestauksen myöhemmässä sovelluskehityksen vaiheessa. Suorituskykytestit ovat kuitenkin tärkeitä julkaistavan sovellusversion validoimiseksi vallitseviin mittareihin nähden. Tämän lisäksi ennen julkaisua on varmistettava, että kaikki tuotannon suorituskykyä seuraavat ohjelmistot ovat valmiina tukemaan tuotantoon vietävää sovellusta, kun se vihdoin saapuu tuotantoympäristöön.

Käyttäjien hyväksyntätestaus tai kokemus mittaa, kuinka paljon todelliset käyttäjät pitävät ohjelmistosta ja kuinka helposti he voivat sitä käyttää. Sovellustiimit käyttävät yleensä käyttäjäkokemusmittaria arvioidakseen käyttäjien tyytyväisyyttä ottamalla huomioon sovellusten vasteajat, epäonnistuneet pyynnöt ja muut mahdolliset sovellustiimin päättämät asiat. Jos käyttäjät eivät voi helposti käyttää hyväksyntätestaukseen vietyä versiota, sovelluksen käyttöönotto voi viivästyä tai pahimmillaan peruuntua kokonaan.

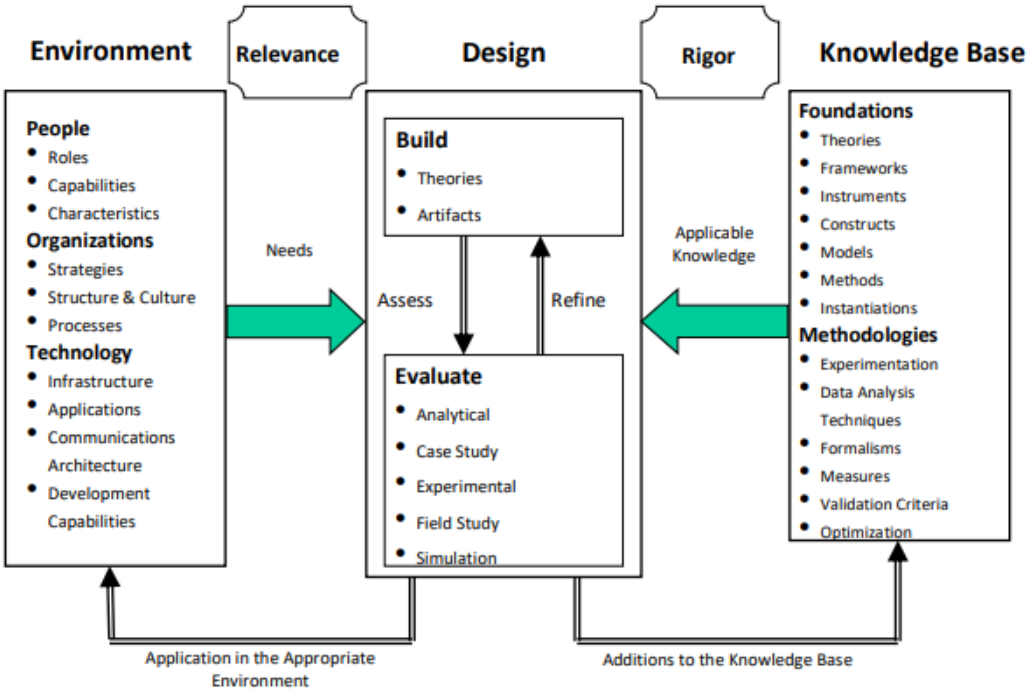
Sovelluksen turvallisuus tai tietoturva on nykypäivänä todella tärkeässä roolissa tuotantoympäristöissä. Tietoturvatestaukseen kuuluu usein mahdollisten haavoittuvuuksien, valtuutus- ja todennustekniikoiden, tietojen salauksen sekä haitallisen toiminnan vastustamisen testaus ja mahdollinen korjaaminen.

Edellä mainittujen lisäksi tuotantokelpoisuutta voisi varmasti tutkia hyvin monelta eri kannalta, mutta työssä ei paneuduta asiaan syvällisemmin.

3. TUTKIMUSMENETELMÄ JA -KYSYMYS

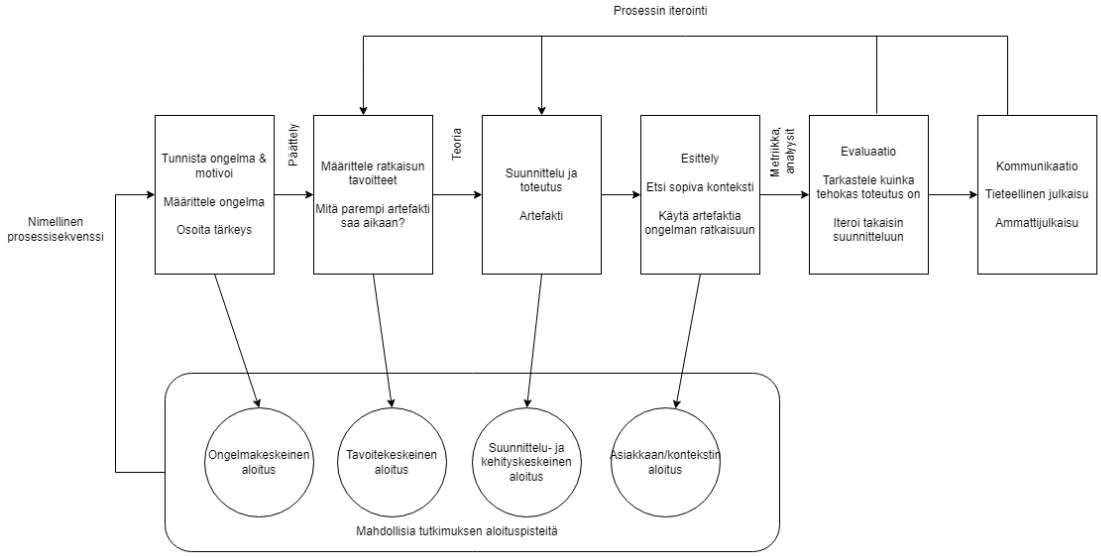
3.1 Suunnittelututkimus yleisesti

Suunnittelututkimus on ongelmanratkaisuparadigma, joka yrittää parantaa ihmisen ymmärrystä innovatiivisten artefaktien luomisen kautta. Yksinkertaisimmillaan kehitystutkimuksen tähtäimenä on kehittää teknologiaa ja tiedettä luomalla innovatiivisia artefakteja, jotka ratkaisevat ongelmia ja parantavat ympäristöä, jossa ne ovat instantioitu (kuva 2). (Brocke, Hevner, Maedche, 2020)



Kuva 2. Kehittämistutkimuksen viitekehys (Brocke, Hevner, Maedche, 2020)

Suunnittelututkimuksen prosessi kuvataan kuvassa 3. Prosessi sisältää kuusi eri vaihetta: ongelman tunnistus ja motivaatio, ratkaisun tavoitteiden määritelmät, suunnittelu ja toteutus, esittely sekä evaluaatio ja kommunikointi. Näiden lisäksi prosessissa on neljä mahdollista sisääntulopistettä: ongelmakeskeinen aloitus, tavoitekeskeinen ratkaisu, kehityskeskeinen aloitus ja asiakas/konteksti aloitus.



Kuva 3. Suunnittelututkimuksen prosessikuvaus

Kuvan 3 ensimmäisessä prosessin kohdassa, ongelman tunnistaminen ja motivointi, määritellään tutkimusongelma ja perustellaan ratkaisun arvo. Ratkaisun arvon perusteleminen saa aikaan kaksi asiaa: se motivoi tutkijaa ja tutkimuksen yleisöä etsimään ratkaisua ja se auttaa yleisöä arvostamaan tutkijan käsitystä ongelmasta. Tämän toiminnan edellyttämiä resursseja ovat tieto ongelman tilasta ja sen ratkaisemisen tärkeydestä.

Ongelman tunnistamisen ja motivoinnin jälkeen on vuorossa ratkaisun tavoitteiden määrittely. Ratkaisun tavoitteet voidaan päätellä ongelman määrittelystä ja tiedosta, mikä on mahdollista ja toteutettavissa. Tavoitteet voivat olla kvantitatiivisia, esimerkiksi termejä, joissa toivottu ratkaisu olisi parempi kuin nykyiset, tai laadullisia, esimerkiksi kuvaus siitä, kuinka uuden artefaktin odotetaan tukevan ratkaisuja ongelmiin, joita ei ole tähän mennessä käsitelty. Tavoitteet tulee päätellä järkevästi ongelman määrittelystä.

Kohta kolme on suunnittelu ja kehitys. Tässä kohdassa luodaan artefakti. Käsitteellisesti artefakti voi olla mikä tahansa suunniteltu esine tai asia, jonka suunnitteluun on käytetty tutkimusaikaa. Suunnitteluun ja kehitykseen kuuluu artefaktin halutun toiminnallisuuden ja sen arkkitehtuurin määrittäminen ja todellisen artefaktin luominen.

Tehtävässä neljä on nimeltään esittely. Esittelyssä tarkoituksena on osoittaa artefaktin käyttö yhden tai useamman ongelman ratkaisemiseen. Tämä voi sisältää artefaktin käytön kokeilussa, simuloinnissa, tapaustutkimuksessa, todistuksessa tai muussa tarkoituksenmukaisessa toiminnassa.

Kohtana viisi on arviointi. Arvioinnissa mitataan, kuinka hyvin artefakti tukee ongelman ratkaisua. Tämä kohta sisältää ratkaisun tavoitteiden vertaamisen todellisiin havaittuihin

tuloksiin artefaktin käytöstä kyseisessä kontekstissa. Riippuen ongelman ja artefaktin luonteesta, arvioinnilla voi olla monia erilaisia muotoja. Arvioinnin lopussa tutkijat voivat päättää, palataanko takaisin vaiheeseen kolme parantaakseen artefaktin tehokkuutta vai jatketaanko viestintään ja jätetäänkö parannukset seuraaviin projekteihin tai tutkimuksiin.

Kohdassa kuusi on kyseessä viestintä. Viestinnässä nimensä mukaisesti viestitään ongelman eri näkökohdat ja suunniteltu artefakti viestitään asianomaisille sidosryhmille. Asianmukaisia viestintämuotoja käytetään tutkimuksen tavoitteista ja yleisöstä riippuen.

3.2 Suunnittelututkimus tässä työssä

Työn ongelmana ja motivaationa on luoda sovellus MetaGPT:llä, joka sisältäisi tuotantokelpoista lähdekoodia ja dokumentaatiota. Pienien koodimäärien generoiminen ei säästä huomattavasti sovelluskehittäjien työaikaa. Jos koodi ja dokumentaatio ovat huonolaatuista, niillä ei tee mitään.

Ratkaisun tavoitteena on luoda tuotantokelpoista sekä skaalautuvaa lähdekoodia että dokumentaatiota MetaGPT:llä. Edellisen ohessa koitetaan, että pystytäänkö MetaGPT:n avulla luomaan yli 1000 koodiriviä.

Suunnittelu ja kehitys kohdassa luodaan työn artefakti, joka on tietyllä komentorivikäskyllä luotu koodi ja dokumentaatio.

Esittelyssä esitellään tuotetut koodit ja dokumentit, jotka MetaGPT on tuottanut. Esitellään myös mahdollisesti luodun sovelluksen eri käyttötapauksia, jos tuotetut lähdekoodit tämän mahdollistavat. Vaihtoehtoisesti voidaan todeta, ettei tuotetut lähdekoodit ole käyttökelpoisia.

Arviointi tarkoittaa tässä työssä tuotettujen koodien ja dokumenttien laadun, skaalautuvuuden ja koon tarkastelua. Tutkimus on tässä työssä hyvinkin tutkivaa, sillä MetaGPT ei ole tuttu entuudestaan. Työn lomassa arvioinnin kriteeristöä voidaan joko löyhentää tai kiristää saatujen tulosten perusteella.

Viestinnässä esitellään tuotetuiden koodien ja dokumenttien koko ja laatu. Käytännössä nämä kirjataan tämän dokumentin loppupuolelle.

3.3 Tutkimuskysymys

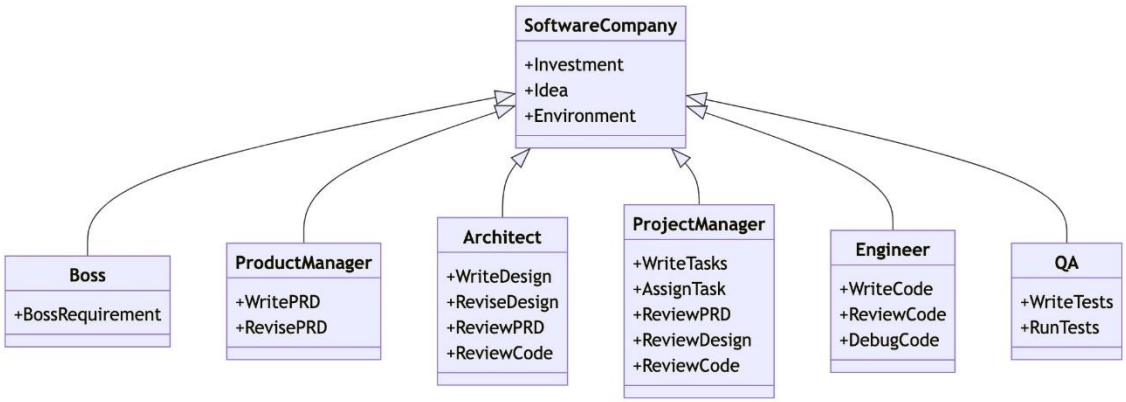
Tässä suunnittelutieteellisessä tutkimuksessa tutkitaan MetaGPT:n luoman koodin skaalautuvuutta. Tutkimuskysymys: Voidaanko MetaGPT:n luomaa ohjelmistokoodia

sekä dokumentaatiota sekä skaalata tehokkaasti että hyödyntää tuotantoympäristöissä? Tutkimuskysymys on tärkeä, sillä skaalautumattoman ohjelmakoodin kirjoittaminen ei ole arvokasta taloudellisesti. Koodin vieminen tuotantoympäristöön vaatii koodilta tietynlaisia laadullisia ominaisuuksia. On tärkeää, että koodi täyttää nämä ominaisuudet.

4. TEKNOLOGIAT JA VIITEKEHYKSET

4.1 MetaGPT

MetaGPT (MetaGPT, 2024) on monen agentin kehys, joka määrittää GPT:ille erilaisia rooleja muodostaakseen yhteistyökykyisen ohjelmistokokonaisuuden monimutkaisia tehtäviä varten. MetaGPT:lle annetaan yhden rivin määrityksen, jonka pohjalta se tuottaa käyttäjätarinoita, vaatimuksia, tietorakenteita, rajapintoja, asiakirjoja sekä analyyseja. Sisäisesti MetaGPT sisältää tuotepäälliköitä, arkkitehtejä, projektipäälliköitä sekä insinöörejä, jotka ikään kuin yhdessä rakentavat edellä mainittuja asioita. MetaGPT tarjoaa ohjelmistoyrityksen koko prosessin sekä huolellisesti organisoidut käyttöprosessit (kuva 4). Tämän tutkimuksen tarkoituksena on selvittää MetaGPT:n skaalautuvuutta.



Kuva 4. MetaGPT:n arkkitehtuuri (MetaGPT, 2024)

MetaGPT:tä voidaan ajaa myös Sparkin, Anthropicin ja Azuren kielimalleilla. Sitä voidaan myös hyödyntää hakuprosesseissa käyttämällä serpapia, tai Googlea. MetaGPT mahdollistaa myös TTS:n eli text to speechin. Käytännössä syötteenä voidaan antaa tekstiä ja MetaGPT luo puhetta.

4.2 Python

Python on tulkattu, oliosuuntautunut, korkean tason ohjelmointikieli, jossa on dynaaminen semantiikka. Sen korkeatasoiset sisäänrakennetut tietorakenteet yhdistettynä dynaamiseen tyypitykseen ja dynaamiseen sidontaan tekevät siitä erittäin houkuttelevan nopeassa sovelluskehityksessä sekä käytettäväksi skriptaus- tai liimauskielenä olemassa olevien komponenttien yhdistämiseen. Pythonin

yksinkertainen, helposti opittava syntaksi korostaa luettavuutta ja vähentää siten ohjelman ylläpitokustannuksia. Python tukee moduuleja ja paketteja, mikä kannustaa ohjelmien modulaarisuuteen ja koodin uudelleenkäyttöön. Python-tulkki ja laaja standardikirjasto ovat saatavilla lähde- tai binäärimuodossa maksutta kaikille tärkeimmille alustoille, ja niitä voidaan jakaa vapaasti.

Usein ohjelmistokehittäjät rakastuvat Pythoniin sen tarjoaman lisääntyneen tuottavuuden vuoksi. Koska käännösvaihetta ei ole, edit-test-debug-sykli on uskomattoman nopea. Python-ohjelmien virheenkorjaus on helppoa: bugi tai vääränlainen syöte ei koskaan aiheuta segmentointivirhettä. Sen sijaan, kun tulkki havaitsee virheen, se aiheuttaa poikkeuksen. Kun ohjelma ei havaitse poikkeusta, tulkki tulostaa stacktracen. Lähdetason debuggeri mahdollistaa paikallisten ja globaalien muuttujien tarkastelun, mielivaltaisten lausekkeiden arvioinnin, keskeytyspisteiden asettamisen, koodissa siirtymisen rivi kerrallaan ja niin edelleen. Pythonin debuggeri on kirjoitettu itse Pythonilla, mikä todistaa Pythonin voiman. Toisaalta usein nopein tapa ohjelman virheenkorjaukseen on lisätä muutama tulostettava lauseke lähteeseen: nopea edit-test-debug-sykli tekee tästä yksinkertaisesta lähestymistavasta erittäin tehokkaan. (Python, 2024)

4.3 Pythonin virtuaaliympäristö

Pythonin venv-moduuli tukee kevyiden virtuaaliympäristöjen luomista, joista jokaisella on omat Python-pakettinsa asennettuna hakemistoihinsa. Virtuaalinen ympäristö luodaan olemassa olevan Python-asennuksen päälle. Virtuaaliympäristö voidaan eristää pohjalla olevasta Python-asennuksesta, jolloin vain virtuaaliympäristöön eksplisiittisesti asennetut paketit ovat saatavilla (Python, 2024).

Virtuaaliympäristöä käytettäessä yleiset asennustyökalut, kuten pip, asentavat Python-paketit ympäristöön ilman, että sitä tarvitsee erikseen pyytää.

Virtuaaliympäristön ominaisuuksia ovat muun muassa:

- Sisältää tietyn Python-tulkin, ohjelmistokirjastot ja binäärit, joita tarvitaan projektin (kirjaston tai sovelluksen) tukemiseen. Edellä mainitut ovat eristettynä muiden virtuaaliympäristöjen ohjelmistoista ja käyttöjärjestelmään asennetuista Python-tulkeista ja kirjastoista.
- Perinteisesti virtuaaliympäristö sisältyy hakemistoon, joka on “venv” tai “.venv”.
- Virtuaaliympäristöä ei laiteta versionhallintaan.

- Ympäristöä pidetään kertakäyttöisenä. Sen pitäisi olla helppoa poistaa ja luoda uudelleen tyhjästä. Virtuaaliympäristöön ei sijoiteta mitään projektikoodia.
- Virtuaaliympäristöä ei pidetä siirrettävänä tai kopioitavana.

4.4 FastAPI

FastAPI (FastAPI, 2024) on moderni, nopea, suorituskykyinen verkkokehys API-rajapintojen rakentamiseen Python 3.8+ -versiolla. Sen tärkeimmäit ominaisuudet ovat:

- Nopeus: erittäin korkea suorituskyky. Sama kuin NodeJS:llä ja Go:lla. Yksi nopeimmista Python-kehyksistä.
- Nopea koodaaminen: Lisää ominaisuuksien kehittämisnopeutta noin 200-300%.
- Vähemmän virheitä: Vähentää noin 40% sovelluskehittäjän aiheuttamista virheistä.
- Intuitiivinen: Erinomainen editorin tuki. Täydennys kaikkialla. Vähemmän aikaa virheiden debuggaukseen.
- Helppo: Suunniteltu helppokäyttöiseksi ja helposti opittavaksi. Vähemmän aikaa dokumentaation lukemiseen.
- Lyhyt koodi: Minimoi koodin duplikoinnin. Vähentää Bugeja. Useita ominaisuuklsia jokaisesta parametrikuvauksesta.
- Vankka: Saa tuotantokoodia automaattisella interaktiivisella dokumentaatiolla.
- Standardeihin perustuva: Perustuu ja on täysin yhteensopiva API:n avoimiin standardeihin, OpenAPI ja JSON Schema.

4.5 Docker

Docker (Docker, 2024) on avoin alusta sovellusten kehittämiseen, toimittamiseen ja suorittamiseen. Dockerin avulla voidaan erottaa sovellukset infrastruktuurista, joka nopeuttaa ohjelmistojen toimittamista. Dockerin avulla infrastruktuuria voidaan hallita samalla tavalla kuin sovelluksia. Käyttämällä Dockerin menetelmiä koodin toimituksessa, testauksessa ja käyttöönotossa, voidaan viivettä koodin kirjoituksen ja tuotantoon viemisen välillä pienentää huomattavasti.

Docker tarjoaa mahdollisuuden pakata ja suorittaa sovelluksia eristetyssä ympäristössä, jota kutsutaan kontiksi. Eristyksen ja suojauksen ansiosta useita kontteja voidaan ajaa samanaikaisesti yhdellä ”isännällä”. Kontit ovat kevyitä ja sisältävät kaiken sovelluksen

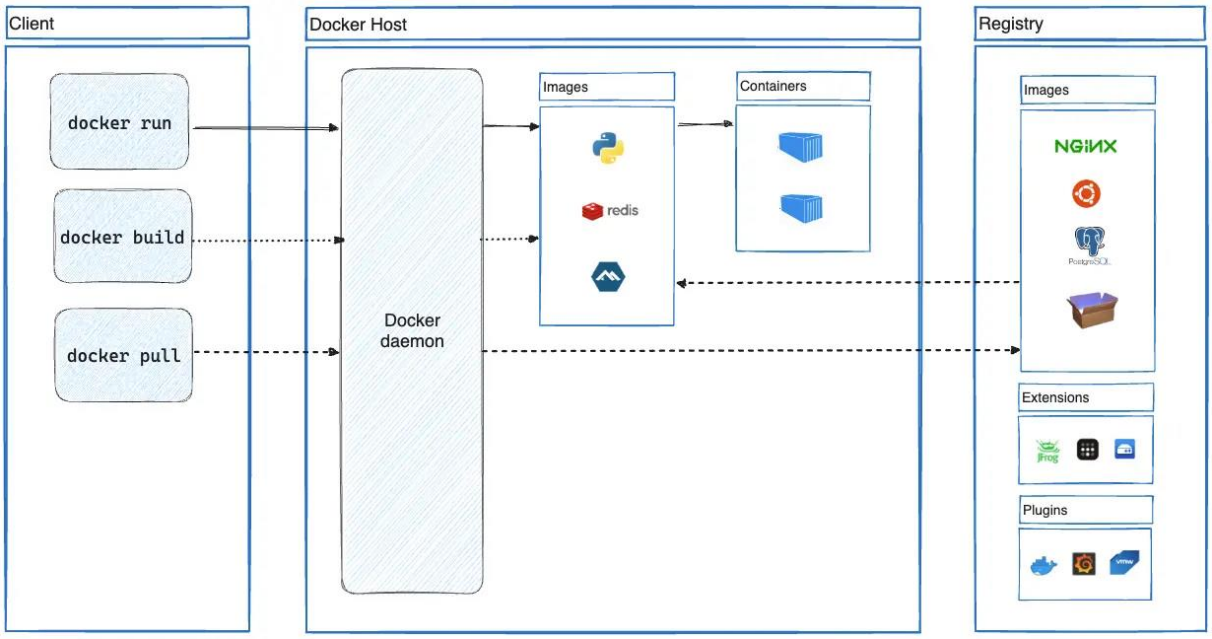
suorittamiseen tarvittavan, jolloin isännälle asennetuista paketeista ja/tai sovelluksista ei tarvitse välittää. Kontteja voidaan jakaa muiden käyttäjien kanssa, jolloin voidaan varmistua siitä, että jokaisella konttia tarvitsevalla on sama versio kontista.

Docker tarjoaa työkalut ja alustan konttien elinkaaren hallintaan. Kontti voidaan luoda kehitysvaiheessa, jolloin samaa konttia voidaan käyttää sekä jakeluun että testaukseen. Testauksen jälkeen sama kyseinen kontti voidaan viedä tuotantoon sellaisenaan tai ajaa orkestroituna palveluna riippumatta tuotantoympäristön tyypistä (on-premise vai pilvi).

Docker yksinkertaistaa kehitystyön elinkaarta sallimalla kehittäjien työskennellä standardoiduissa ympäristöissä käyttämällä paikallisia kontteja, jotka tarjoavat sovelluksia ja palveluita. Kontit sopivat erinomaisesti jatkuvaan integrointiin ja toimitukseen (CI/CD).

Dockerin siirrettävyys ja kevyt ”luonne” helpottavat työkuormien dynaamista hallintaa, sovellusten ja palveluiden laajentamista tai purkamista liiketoiminnan tarpeiden mukaan lähes reaaliajassa. Dockerin keveys ja nopeus tarjoavat kustannustehokkaan vaihtoehdon virtuaalikoneille, joka mahdollistaa suuremman palvelinkapasiteetin käyttämisen.

Docker käyttää asiakas-palvelin-arkkitehtuuria. Asiakas keskustelee Docker daemonin kanssa, joka huolehtii konttien rakentamisesta, ajamisesta ja jakelusta. Asiakas ja daemon voivat toimia samassa järjestelmässä tai asiakas voidaan yhdistää etädaemoniin. Asiakas ja daemon kommunikoivat REST API:n, UNIX sockettien tai verkkoliitännän kautta. Toinen Dockerin asiakasohjelma on Docker Compose, jonka avulla voidaan työskennellä sovellusten kanssa, jotka koostuvat konteista (kuva 5).



Kuva 5. Dockerin arkkitehtuuri (Docker, 2024)

5. TOTEUTUS

5.1 MetaGPT:n asentaminen

MetaGPT:n asentamiseen tarvitaan toimiva Python 3 asennus. Tutkimuksessa Python on jo valmiiksi asennettuna työasemalla. MetaGPT:n käytössä voidaan hyödyntää myös npm paketteja. Oletetaan myös npm:n olevan asennettuna työasemalla. Tarkastetaan Python-asennuksen toimivuus ”python –version” komennolla. Kloonataan git repositorio ”git clone https://github.com/geekan/MetaGPT.git”. Siirrytään MetaGPT:n hakemistoon, jonka edellinen komento on luonut. Asennetaan tarvittavat Pythonin paketit ”python -m pip install -r requirements.txt” komennolla. Asennetaan erilaisten dokumenttien luontia varten tarvittavat npm paketit komennolla ”npm install -g @mermaid-js/mermaid-cli”.

MetaGPT:n toimivuus voidaan testata esimerkiksi komennolla ”python startup.py ”Write a cli snake game””. Jos asennus on onnistunut, MetaGPT lähtee luomaan uutta sovellusta.

5.2 MetaGPT:n konfigurointi

Konfiguroidaan MetaGPT käyttämään OpenAI:n gpt-4 kielimallia. Asetetaan OpenAI:n rajapinta-avain OPENAI_API_KEY kenttään. MAX_TOKENS ja RPM ovat vaihdettavissa olevia arvoja, joita voi koittaa muokata erilaisten tulosten saamiseksi. MAX_TOKENS kuvaa yhden sovelluksen luontiin käytettävien tokenien maksimiarvon. OpenAI laskuttaa rajapinnan käytöstä tokenien perusteella. RPM tarkoittaa on englanniksi ”Requests Per Minute”, joka voidaan suomentaa ”kutsua minuutissa”.

```
11  OPENAI_API_KEY: "xx-xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
12  OPENAI_API_MODEL: "gpt-4"
13  MAX_TOKENS: 3500
14  RPM: 10
```

Kuva 6. MetaGPT:n konfiguraatiotiedosto

OpenAI:n rajapinnan rajoitteiden vuoksi write_code.py tiedoston run-metodiin on lisätty time.sleep(60) rivi, joka hidastaa koodin luomista. Tämä hidastus mahdollistaa rajapinnan jatkuvan käytön MetaGPT:lle ilman, että rajapinta palauttaa rate limiting virhettä.

5.3 Testitapausten perusteella tehdyt huomiot

Ennen varsinaisen projektin luomista tehtiin useita testiprojekteja. Testiprojektien perusteella voidaan todeta MetaGPT:n tuottaman koodin olevan hyvin vaihtelevan laatuista. Lähdekoodille syntyvä dokumentaatio vastaa todellisuutta ja on näin ollen laadullisesti huomattavasti parempaa kuin lähdekoodi.

5.4 Fastapi-pohjaisen projektin luominen

Alkuperäinen fastapi_articles -projekti on luotu syöttämällä MetaGPT:n käynnistysskriptille parametrina ”Write a FastAPI based application with no authentication. The application should allow users to handle articles.”. Tämän pohjalta MetaGPT on luonut tarvittavat dokumentit, konfiguraatiotiedostot ja itse kooditiedostot, joihin sisältyvät sekä testitiedostot että lähdekooditiedostot.

5.5 Luodut tiedostot

5.5.1 Dokumentaatio

Dokumentaatiopuolelle MetaGPT luo aina seuraavat tiedostot:

- api_spec_and_tasks.md
- prd.md
- system_design.md

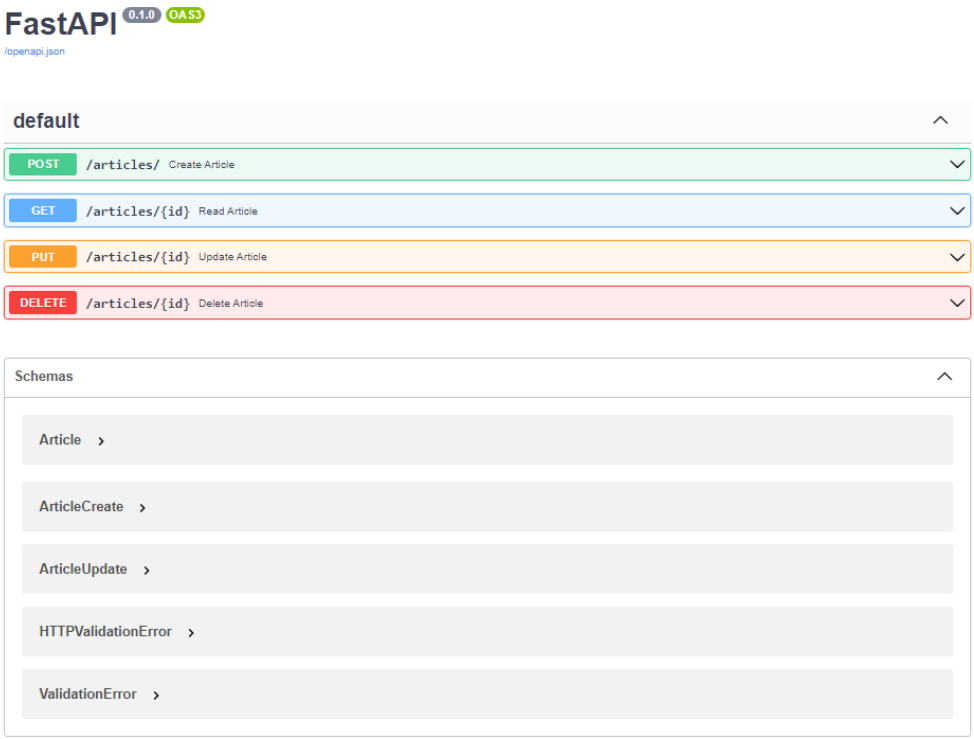
Yllä olevan listan ensimmäinen tiedosto sisältää tarvittavat Python-kirjastot ja niiden versiot, muut mahdolliset kirjastot ja paketit, koko rajapinnan kuvauksen, loogisen analyysin, tehtävälistan, jaettua tietoa ja kaiken MetaGPT:lle epäselvän.

PRD-dokumentti (tuotevaatimusasiakirja) sisältää alkuperäiset vaatimukset, tuotetavoitteet, käyttäjätarinat, kilpailija-analyysin, kilpailukykyisen kvadranttikaavion, vaatimusanalyysin, vaatimukset, käyttöliittymäsuunnittelun luonnoksen ja kaiken epäselvän tuotevaatimuksiin liittyvän.

Viimeinen dokumentti sisältää järjestelmän kuvauksen. Siihen sisältyvät toteutuksen lähestymistapa, projektin nimi, tiedostolista, tietorakenteet sekä rajapintojen kuvaukset, sekvenssikaavion sekä epäselvät asiat järjestelmän kuvaukseen liittyen.

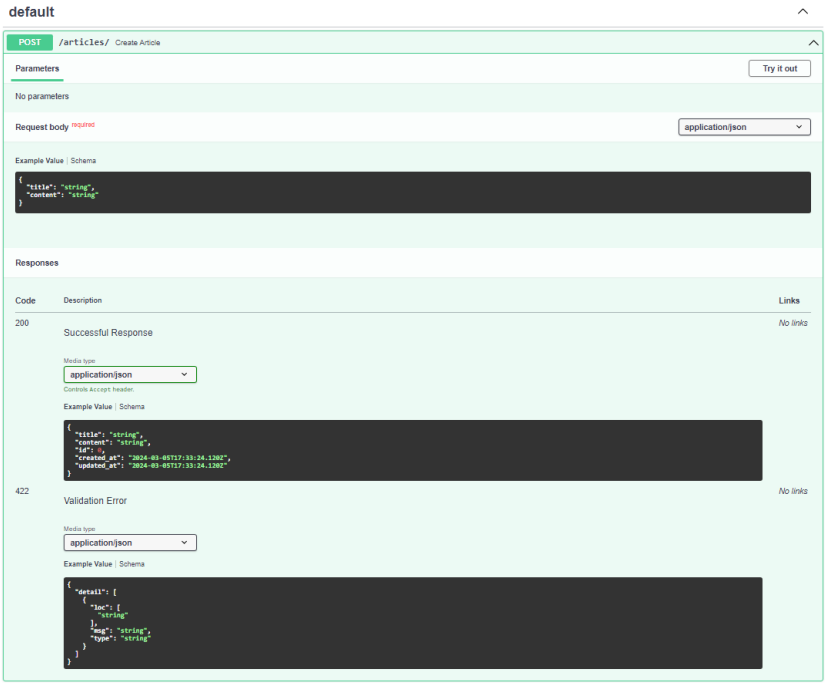
Tässä työssä nämä dokumentit on luotu sekä pohjalle rakennetulle ”perusversiolle” sekä ratelimiting-moduulille.

MetaGPT:n itse generoiman dokumentaation lisäksi FastAPI luo OpenAPI-pohjaisen rajapintadokumentaation http://localhost:8000/docs osoitteeseen sovelluksen ollessa käynnissä (Kuva 7).



Kuva 7. FastAPI:n OpenAPI-pohjainen rajapintadokumentaatio

Dokumentaatio kuvaa jokaisen rajapinnan päätepisteen kuvan 8 mukaisesti.



Kuva 8. Rajapinnan päätepisteen dokumentaatio

5.5.2 Lähdekoodi

Järjestelmäkuvausdokumentin mukaisesti MetaGPT on luonut niin sanotulle perusversiolle tiedostot:

- main.py
- models.py
- schemas.py
- database.py
- alembic.ini
- Dockerfile
- requirements.txt
- tests/test_main.py

Voidaan siis todeta, että MetaGPT on luonut sovelluksen ajamiseen tarvittavat Pythontiedostot sekä ohjelmiston ajamisympäristöön liittyvät tiedostot.

Ratelimiting-moduulille on luotu seuraavat lähdekooditiedostot:

- main.py
- config.py
- rate_limiter.py

Vaikka työssä puhutaan erillisestä moduulista, asioiden yksinkertaistamiseksi moduulin koodit lisättiin niin sanottuun perusversioon. Näin ollen molemmissa projekteissa syntyneet main.py ja models.py tiedostot vaativat yhdistelyä, jotta ohjelma saatiin ajoon.

5.6 Pythonin virtuaaliympäristön käyttäminen

Virtuaaliympäristöt luodaan suorittamalla komento ”venv” (kuva 9).

```
python -m venv .
```

Kuva 9. Virtuaaliympäristön luominen komennolla ”venv”.

Kuvan 9 mukaisen komennon suorittaminen luo kohdehakemiston ja sijoittaa sinne tarvittavat kirjastot.

Virtuaaliympäristö saadaan käyttöön Windows-ympäristössä ajamalla ”activate.bat” skriptin (Kuva 10).

```
venv\Scripts\activate.bat
```

Kuva 10. Virtuaaliympäristön aktivointi

5.7 Tarvittavat muutokset

Sovelluksen käynnistymistä ja järkevää ajamista varten tarvittiin muutamia muutoksia sekä konfiguraatio- että lähdekooditiedostoihin.

5.7.1 Muutokset konfiguraatiotiedostoissa

MetaGPT:n luomassa requirements.txt tiedostossa tarvittiin seuraavat muutokset:

- Pydantic versiosta 1.8.2 versioon 1.10.1
- Lisättiin uvicornin versio 0.23.2

Uvicorn-kirjasto lisättiin, jotta sovellusta voitaisiin ajaa myös ilman Dockeria. Kirjaston lisäämisen lisäksi lähdekoodiin tarvittiin muutama koodirivi. Näistä lisää kappaleessa 5.7.2. Jos sovellusta ajetaan Dockerilla, toimii se myös ilman uvicornia ja edellä mainittuja lähdekoodirivejä.

Pydanticin versiopäivitys vaati suurempaa tutkimista, sillä virhe, jonka sovellus antoi ennen päivitystä, ei ollut kovin selkeä (kuva 11).

```
File "C:\Python311\Lib\inspect.py", line 2725, in __init__
raise ValueError('{!r} is not a valid parameter name'.format(name))
ValueError: 'not' is not a valid parameter name
```

Kuva 12. Sovelluksen käynnistyessä antama virhe Pydanticin 1.8.2 versiolla.

Alkuperäisessä Dockerfilessä kontin ulospäin avattu portti oli 80. Portti vaihdettiin porttiin 3001, jotta vältytään erilaisilta käyttöoikeusongelmilta, joita alle 1000 portit aiheuttavat. Docker image rakennettiin kuvan 12 mukaisella komennolla, jonka jälkeen se laitettiin ajoon Docker Desktopin kautta. Docker Desktopissa kontille tehtiin porttiohjaus portista 3001 porttiin 80, jotta sovellus osaa vastaanottaa sisään tulevat kyselyt.

```
docker build -t fastapiapplication:0.0.1 .
```

Kuva 12. Docker imagen rakentaminen.

5.7.2 Muutokset lähdekoodissa

Luotua sovellusta haluttiin ajaa ilman Dockeria. Tätä varten tarvittiin uvicorn-kirjaston lisääminen projektiin sekä main-metodin lisääminen (kuva 13). Lisätty metodi luo tilanteen, jossa sovellus jää kuuntelemaan konfiguroitua porttia. Ilman tätä metodia sovellus sammuu heti sen käynnistyttyä. Soveiluksen ajaminen lokaalisti ilman Dockeria nopeuttaa ja yksinkertaistaa testausiteraatioita huomattavasit.

```
if __name__ == "__main__":
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

Kuva 13. Main-metodin lisääminen projektiin

5.8 Rate limit -moduulin luominen ja integroiminen olemassa olevaan projektiin

Työssä luotiin erillinen konfiguroitava rate limit -moduuli, jonka ajatuksena oli mahdollisimman helppo integrointi alkuperäiseen sovellukseen. Täydelliseen moduuliajatteluun ei päästy, vaan työssä jouduttiin tyytymään leikkaa ja liimaa ratkaisuun. Ajatuksena oli kuitenkin se, että leikkaa ja liimaa ratkaisun pitää olla niin yksinkertainen, että sovelluskehittäjän ei tarvitse tehdä itse mitään koodimuutoksia.

Moduuli luotiin antamalla MetaGPT:lle komento "Write a configurable rate limiting module for fastapi based application".

Myös rate limit -moduulissa tarvittiin Pydanticin päivitys versiosta 1.8.2 versioon 1.10.1. Tämän lisäksi requirements.txt tiedostoon lisättiin slowapin versio 0.1.4.

Moduulissa syntyneet Python-tiedostot ovat:

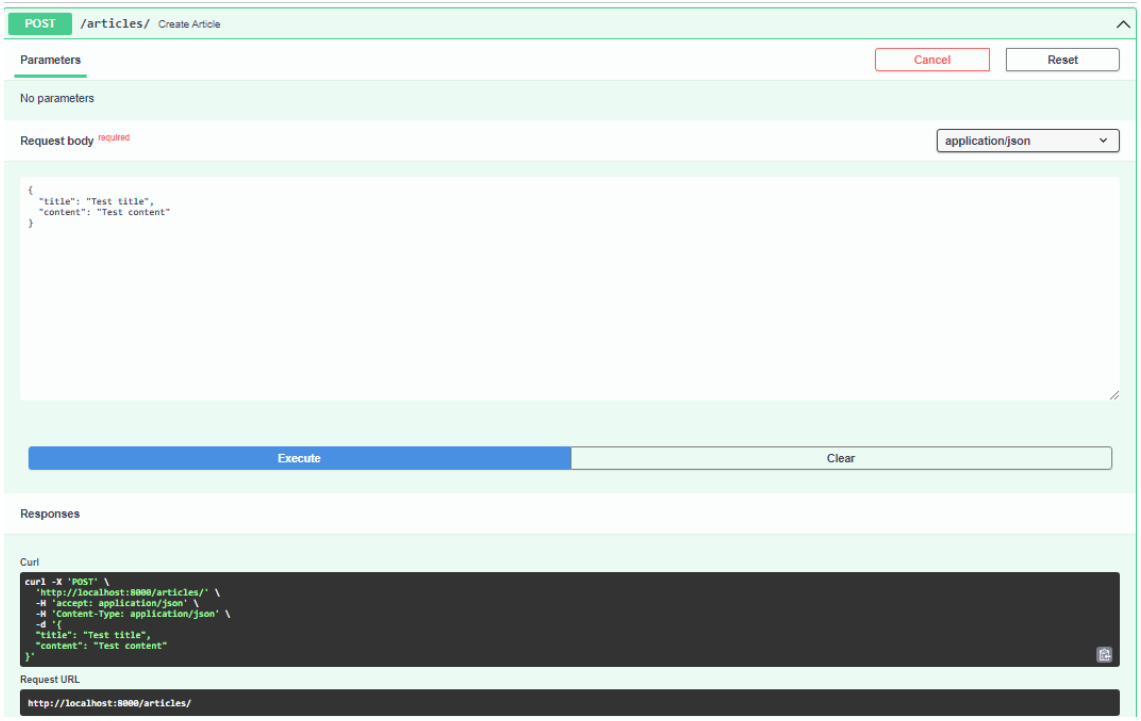
- main.py
- config.py
- rate_limiter.py

Listauksesta kaksi jälkimmäistä tiedostoa siirrettiin suoraan alkuperäisen projektin lähdekooditiedostojen hakemistoon. Listauksesta kaksi jälkimmäistä tiedostoa yhdisteltiin alkuperäisen projektin samannimisten tiedostojen kanssa sopivalla tavalla, jotta sovellus saatiin käyntiin. Yhdistely vaati vähintään keskinkertaista Python-osaamista.

5.9 Syntyneen sovelluksen testaaminen

Sovellus voidaan konfiguroida muuttamalla config.py tiedostossa olevia muuttujia. Tiedosto sisältää MAX_REQUESTS ja TIMEFRAME muuttujat. MAX_REQUESTS kuvaa kutsujen määrää ja TIMEFRAME aikamäärettä sekunteina, jossa kyselyjen määrää tarkastellaan. Testitapauksessa käytetään MAX_REQUESTS-muuttujassa arvoa 2 ja TIMEFRAME-muuttujassa arvoa 60.

Luodaan artikkeli lähettämällä POST-kutsu (kuva 14). Artikkelin otsikoksi asetetaan ”Test title” ja sisällöksi ”Test content”.

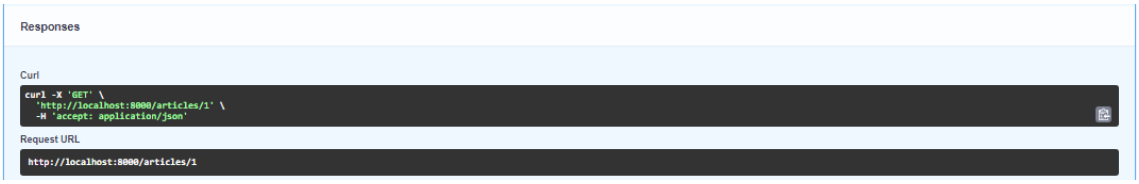


Kuva 14. Kutsu artikkelin lisäämiseksi.

Sovellus (kuva 15) palauttaa vastauksen HTTP-tilakoodin 200. Vastauksessa on myös annettu otsikko ja sisältö. Näiden lisäksi on automaattisesti kasvava id, luontiaika sekä päivitysaika.

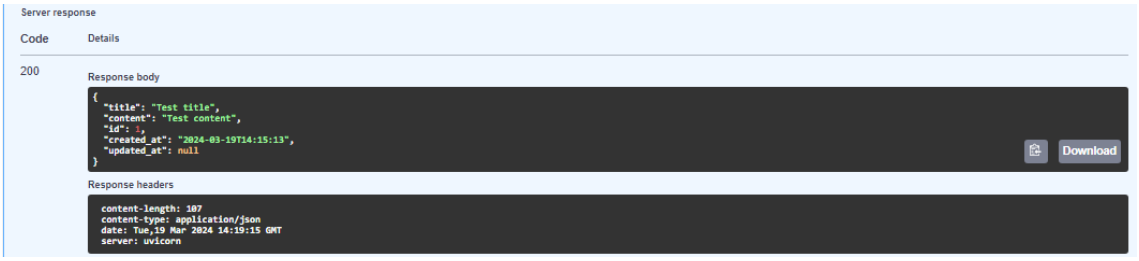
Kuva 15. Sovelluksen vastaus artikkelin lisäämiseen.

Tämän jälkeen voidaan hakea luotu artikkeli GET-kutsulla (kuva 16).



Kuva 16. Artikkelin hakeminen GET-kutsulla

Sovellus vastaa POST-kutsua vastaavilla tiedoilla (kuva 17).

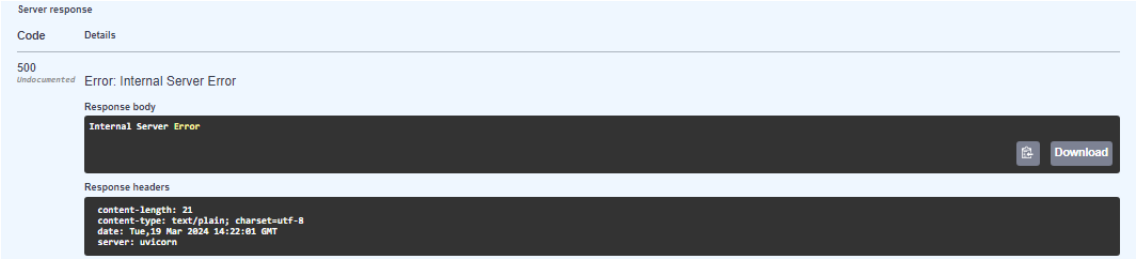


Kuva 17. Sovelluksen vastaus artikkelin hakemiseen.

Näiden kahden kutsun jälkeisiä kutsuja palvelin tulee rajoittamaan, jos ne on tehty seuraavan minuutin aikana. Pyydetään uudelleen palvelimelta artikkelia id:llä 1. Palvelin rajoittaa kyselyitä, mutta ei osaa hallita siitä aiheutuvaa virhettä. Virhe aiheuttaa vastaukseen HTTP-tilakoodin 500 (kuva 19), vaikka sovelluksen lokille tulee virhe, jossa mainitaan HTTP-tilakoodi 429 (kuva 18). Sovellus kuitenkin jatkaa toimintaansa ja TIMEFRAME-muuttujassa määritellyn aikamääreen jälkeen alkaa taas vastailemaan kyselyihin normaalisti. Sovellus siis osaa rajoittaa kyselyiden määrää, mutta ei ilmoita siitä kyselyn tekijälle asianmukaisella tavalla ja HTTP-tilakoodilla.

```
Traceback (most recent call last):
  File "/usr/local/lib/python3.9/site-packages/uvicorn/protocols/http/h11_impl.py", line 408, in run_asgi
    result = await app(  # type: ignore[func-returns-value]
  File "/usr/local/lib/python3.9/site-packages/uvicorn/middleware/proxy_headers.py", line 84, in __call__
    return await self.app(scope, receive, send)
  File "/usr/local/lib/python3.9/site-packages/fastapi/applications.py", line 199, in __call__
    await super().__call__(scope, receive, send)
  File "/usr/local/lib/python3.9/site-packages/starlette/applications.py", line 112, in __call__
    await self.middleware_stack(scope, receive, send)
  File "/usr/local/lib/python3.9/site-packages/starlette/middleware/errors.py", line 181, in __call__
    raise exc from None
  File "/usr/local/lib/python3.9/site-packages/starlette/middleware/errors.py", line 159, in __call__
    await self.app(scope, receive, _send)
  File "/usr/local/lib/python3.9/site-packages/starlette/middleware/base.py", line 25, in __call__
    response = await self.dispatch_func(request, self.call_next)
  File "/app/main.py", line 20, in rate_limit_middleware
    raise HTTPException(status_code=429, detail="Too Many Requests")
fastapi.exceptions.HTTPException
```

Kuva 18. Sovelluslokille tuleva virhe HTTP-tilakoodilla 429.



Kuva 19. Palvelimen vastaus, kun kyselyitä on rajoitettu.

5.10 Inkrementaalinen koodin tuottaminen

Työssä käytetty MetaGPT:n 0.3 versio ei mahdollista inkrementaalisen koodin tuottamista. Inkrementaalisella koodin tuottamisella tarkoitetaan sitä, että MetaGPT osaisi hyödyntää jo valmiita kooditiedostoja. Uudemmissa versioissa tämä on jo mahdollista.

6. TULOKSET JA POHDINTA

Tuotantokelpoisen koodin tuottaminen on haastavaa työssä käytetyllä MetaGPT:n versiolla. Kehyksen tuottamassa koodissa on usein loogisia virheitä, joka aiheuttaa tilanteita, joissa koodi toimii väärin. Joissain tapauksissa koodissa koitetaan jopa käyttää metodeita, joita ei olla implementoitu ollenkaan.

Työssä tuotettu lopullinen sovellus on lineaarisesti skaalautuva. Rajapintojen läpi menevä liikenne voidaan tuplata prosessorien määrän tuplaamisella. Dokumentaation skaalautuvuus työssä on sublineaarista, jos skaalautuvuutta tutkitaan siitä näkökulmasta, että erillisiä sovelluksia yhdistetään toisiinsa. Näissä tapauksissa dokumentaatio joudutaan käsin yhdistelemään, joka on työlästä. MetaGPT:llä pystytään tuottamaan ainoastaan noin 200-300 lähdekoodiriviä. 1000 koodirivin tuottaminen vaikuttaa tällä hetkellä lähes mahdottomalta.

Todetaan, että MetaGPT:tä ei ole kannattavaa käyttää tuotantokäyttöön tarkoitetun koodin tuottamiseen. Sovelluskehittäjät pystyvät nopeammin ja luotettavammin tuottamaan omaa koodia. Sovelluskehittäjän on aina käytävä läpi kehyksen tuottamat koodit turvallisuuden vuoksi. Usein koodin lukeminen ja ymmärtäminen vie yhtä paljon aikaa kuin itse koodin tuottaminen.

Työssä tuotettu koodi on arkkitehtuurillisesti skaalautuvaa. Modulaarisuus on yksi tapa mahdollistaa skaalautuvuus. Näin ei saada aikaan suuria monoliitteja, jotka pitkällä tähtäimellä voivat käydä liian suuriksi, hitaiksi ja vaikeiksi ylläpidettäviksi. Voidaan todeta, että MetaGPT:llä voidaan tuottaa skaalautuvaa koodia, jos luodaan erillisiä pienehköjä moduuleja. Jos koitetaan luoda koodimäärällisesti suurempaa sovellusta, skaalautuvan koodin luominen on haastavampaa. Tässä tapauksessa pohjalle luotu Fastapi-pohjainen sovellus on siinä mielessä skaalattava, että siitä voitaisiin teoriassa ajaa monia instansseja eri konteissa. Skaalautuvaan lopputulokseen on päästy osittain sen takia, että tutkimuksen alussa tehtyjen testitapausten pohjalta todettiin suurten koodimäärien tuottaminen vaikeaksi, ellei jopa mahdottomaksi. Tämän pohjalta ruvettiin miettimään tietynlaisia sovelluksia, jotka ovat lähes automaattisesti skaalautuvia. Toki arkkitehtuurisessa mielessä skaalautuvuuteen ei riitä pelkkä koodi. Koodin ympärille tarvitaan toimiva infrastruktuuri, joka hoitaa esimerkiksi konttien automaattisen lisäämisen ja poistamisen tarpeen mukaan. Näitä infrastruktuurillisia koodeja MetaGPT ei pystynyt tuottamaan tässä tutkimuksessa.

Dockerfilellä, joka tässäkin tutkimuksessa generoitui, voidaan luoda ja ajaa kontteja ympäristöstä riippumatta. Dockerfile ja Docker mahdollistavat sovelluksen ja koodin helpon siirrettävyyden, rakentamisen ja ajamisen. Dockerkontteja voidaan ajaa esimerkiksi Kubernetesympäristössä. Kubernetes mahdollistaa konttien ajamisen ja hallitsemisen suuren mittakaavan ympäristöissä.

Dokumentaatiopuolelle MetaGPT tuottaa useita kolme eri dokumentteja. Ensimmäinen näistä sisältää rajapinnan kuvauksen ja sovelluksen tehtävät. Toinen sisältää erilaiset määrittelyyn liittyvät dokumentaatiot. Alkuperäisen fast api sovelluksen ja siihen liitetyn rate limiting moduulin dokumentaatiot ovat laadultaan hyviä. MetaGPT:n tuottamat dokumentit vastaavat hyvin sen tuottamaa koodia. Dokumentaation laatu on hyvä ja se varmasti hyödyttää sovelluskehittäjän tekemistä. Voidaan kuitenkin todeta, että yksinkertaiselle sovellukselle on helppoa ja nopeaa kirjoittaa dokumentaatio. Olisi mielenkiintoista nähdä, miten MetaGPT tulevaisuudessa suoriutuu laajemman lähdekoodimassan dokumentoinnista.

Eri MetaGPT:llä luotujen projektien dokumentaatioiden yhdistäminen voi olla haastavaa. Jos projektien lähdekoodit joudutaan yhdistelemään niin, että alkuperäinen koodi muuttuu, dokumenttien yhdistäminenkin voi olla hidasta ja vaikeaa. Tässä tilanteessa koko dokumentaation kirjoittaminen alusta loppuun uudelleen voi olla järkevämpi vaihtoehto. Jos taas eri projektit ovat esimerkiksi kokonaan irrallisia moduuleita, dokumentaation yhdistäminen järkeväksi kokonaisuudeksi onnistunee helposti ja tehokkaasti. Dokumentaatiopuolen skaalautuvuus riippuu laajalti koodin tai projektin toteutustavasta. Modulaarisen projektin dokumentointi MetaGPT:llä on tehokkaampaa kuin monoliitin dokumentointi.

Tällä MetaGPT:n versiolla ei voida luoda skaalautuvaa koodia ja dokumentaatiota tehokkaasti, ellei tiedetä jo etukäteen, että tuotettava sovellus on lähes automaattisesti skaalautuva. Skaalautuvuus toteutuu osittain Dockerfilen ansiosta, mutta todellisuudessa koodi itsessään ei ole skaalautuvaa. Myös koodin ympärillä pyörivä infrastruktuuri tai sen koodit puuttuvat täysin.

Työssä luotu sovellus täyttää vaatimuksensa testiympäristöissä. MetaGPT:lle annettuna vaatimuksena oli luoda konfiguroitava kyselyitä rajoittava moduuli, joka ei käytä Redistä tai muita kolmannen osapuolen kirjastoja. Sovellus osaa rajoittaa kyselyiden määrää konfiguroinnissa annettujen muuttujien perusteella, mutta ei osaa vastata oikealla HTTP-tilakoodilla rajoituksen tapahtuessa. Nämä väärät tilakoodit eivät vastaa standardeja, jonka takia tuotantoympäristöissä voitaisiin törmätä ongelmiin.

MetaGPT:llä ei voida tehokkaasti luoda tuotantokelpoista sekä skaalautuvaa lähdekoodia ja dokumentaatiota. Uskon tämän olevan mahdollista tulevaisuudessa, kun MetaGPT ja sen hyödyntämät rajapinnat kehittyvät. Tällä hetkellä sovelluskehittäjän on tehokkaampaa ja turvallisempaa kirjoittaa itse nämä molemmat.

Tulevaisuudessa MetaGPT:n kehittyessä uskon sen pystyvän tuottamaan laadukkaampaa koodia niin, että sovelluskehittäjien työt tehostuvat kehystä käyttämällä. Yhtenä suurista harppauksista voidaan pitää inkrementaalisen koodin tuottamisen tukemista, joka käytännössä mahdollistaa jo olemassa olevien lähdekooditiedostojen lukemisen ja koodin tuottamisen niiden pohjalta. Tämän ominaisuuden käyttäminen voi olla kiellettyä joissain organisaatioissa tietoturvasyistä.

7. YHTEENVETO

Jo nyt lähdekoodin ja dokumentaation luominen tekoälyn avulla on suuressa roolissa ohjelmistokehityksessä. Tulevaisuudessa tekoälyn käytön määrä tulee varmasti kasvamaan huomattavasti. Kielimallien kehittyessä entisestään voidaan tuottaa huomattavasti laadukkaampaa materiaalia.

Myös MetaGPT:n ojhelmistopäivitykset tulevat vaikuttamaan positiivisesti materiaalin tuottamiseen. Työssä käytetyn 0.3 version ominaisuuksiin ei vielä kuulu inkrementaalinen lähdekoodin ja dokumentaation tuottaminen. Jo julkaistuissa versioissa tämä ominaisuus on käytössä. Tämä on ehdottomasti tärkeimpiä ominaisuuksia oikeissa ohjelmistoprojekteissa.

Työssä päädyttiin melko oletettuun lopputulokseen. Alusta asti oli selvää, että suurten ja laadukaidden materiaalien tuottaminen tulee olemaan hankalaa jopa GPT-4 kielimallilla. Tähän ajatukseen saatiin heti vahvistusta, kun todettiin matopelinkin luomisen olevan haastavaa. Yksittäisten metodien tai todella yksinkertaisten rajapintojen tuottaminen on mahdollista MetaGPT:llä. Ajallinen säästö on hyvin minimaalinen ellei jopa negatiivinen. Myös laatu on todennäköisesti huonompaa kuin ohjelmistokehittäjällä. MetaGPT luo myös dokumentaation, mutta voidaan todeta, että pienet ja yksinkertaiset metodit ja rajapinnat eivät tarvitse erillistä dokumentaatiota, jos ne on kirjoitettu järkevästi. Rajapinnoilla on myös nykyisin useasti automaattinen dokumentaatio Swaggerin tai muun vastaavan kirjaston avulla.

Tekoälyä voidaan hyödyntää pienissä määrin ohjelmistoprojekteissa. Tekoälyllä tuotetun materiaalin kanssa pitää kuitenkin olla hyvin tarkkana, sillä tekoäly luulee välillä tietävänsä ja ymmärtävänsä asioita, joita se ei oikeasti ymmärrä. Tekoälyä hyödyntävällä ohjelmistokehittäjäl tulee olla hyvä ymmärrys tuotetusta materiaalista, jotta sitä voidaan käyttää projektissa.

LÄHTEET

[1] [Williams and Smith 2004] L. G. Williams and C. U. Smith, “Web Application Scalability: A ModelBased Approach,” Proceedings of the Computer Measurement Group, Las Vegas, December, 2004.

[2] Austin Eovito, Marina Danilevsky – Language Models in Plain English, 2022. Saatavissa: https://learning.oreilly.com/library/view/language-models-in/9781098109073/ch01.html#why_do_you_need_to_read_this_report

[3] [Alba 2002] E. Alba, “Parallel Evolutionary Algorithms Can Achieve Super-Linear Performance,” Information Processing Letters, vol. 82, pp. 7-13, 2002.

[4] Thimira Amaratunga – Understanding Large Language Models, 2023. Saatavissa: https://learning.oreilly.com/library/view/understanding-large-language/9798868800177/

[5] FastAPI, 2023. Saatavissa: https://fastapi.tiangolo.com

[6] Lloyd Williams, Connie Smith – QSEM^{SM} Quantitative Scalability Evaluation Method, 2005. Saatavissa: https://spe-ed.com/classic-site/papers/qsem.pdf

[7] Stephen Bigelow – A production readiness checklist for software development, 2001. Saatavissa: https://www.techtarget.com/searchsoftwarequality/tip/A-production-readiness-checklist-for-software-development

[8] MetaGPT, 2024. Saatavissa: https://github.com/geekan/MetaGPT

[9] Python – What is Python? Executive Summary, 2024. Saatavissa: https://www.python.org/doc/essays/blurb/

[10] Python – venv – Creation of virtual environments, 2024. Saatavissa: https://docs.python.org/3/library/venv.html

[11] Jan vom Brocke, Alan Hevner, Alexander Maedche – Introduction to Design Science Research, 2020. Saatavissa: https://www.researchgate.net/publication/345430098_Introduction_to_Design_Science_Research

[12] OpenAI – Models. Luettu 11.5.2024. Saatavissa: https://platform.openai.com/docs/models/gpt-4-and-gpt-4-turbo

[13] IBM – What are large language models (LLMs)?. Luettu 11.5.2024. Saatavissa: https://www.ibm.com/topics/large-language-models

[14] Docker – Docker overview. Luettu 15.5.2024. Saatavissa: https://docs.docker.com/get-started/overview/

LIITE A: SOVELLUKSEN MAIN.PY TIEDOSTO

```
from typing import List
from fastapi import FastAPI, HTTPException, Depends
from sqlalchemy.orm import Session
from sqlalchemy.exc import SQLAlchemyError
from database import SessionLocal, engine
import models, schemas
import uvicorn
from rate_limiter import RateLimiter
from fastapi import FastAPI, Request, HTTPException

models.Base.metadata.create_all(bind=engine)

app = FastAPI()
rate_limiter = RateLimiter()

@app.middleware("http")
async def rate_limit_middleware(request: Request, call_next):
    user_id = str(request.client.host)
    if not rate_limiter.check_rate_limit(user_id):
        raise HTTPException(status_code=429, detail="Too Many Requests")
    response = await call_next(request)
    return response

# Dependency
def get_db():
    db = SessionLocal()
    try:
        yield db
    except SQLAlchemyError as e:
        db.rollback()
        raise HTTPException(status_code=500, detail=str(e))
    finally:
        db.close()

@app.post("/articles/", response_model=schemas.Article)
def create_article(article: schemas.ArticleCreate, db: Session = Depends(get_db)):
    try:
        db_article = models.Article(title=article.title, content=article.content)
        db.add(db_article)
        db.commit()
        db.refresh(db_article)
        return db_article
    except SQLAlchemyError as e:
        db.rollback()
        raise HTTPException(status_code=500, detail=str(e))

@app.get("/articles/{id}", response_model=schemas.Article)
def read_article(id: int, db: Session = Depends(get_db)):
    db_article = db.query(models.Article).filter(models.Article.id == id).first()
    if db_article is None:
        raise HTTPException(status_code=404, detail="Article not found")
    return db_article

@app.put("/articles/{id}", response_model=schemas.Article)
def update_article(id: int, article: schemas.ArticleUpdate, db: Session = Depends(get_db)):
    try:
        db_article = db.query(models.Article).filter(models.Article.id == id).first()
```

```
        if db_article is None:
            raise HTTPException(status_code=404, detail="Article not found")
        db_article.title = article.title
        db_article.content = article.content
        db.commit()
        db.refresh(db_article)
        return db_article
    except SQLAlchemyError as e:
        db.rollback()
        raise HTTPException(status_code=500, detail=str(e))

@app.delete("/articles/{id}")
def delete_article(id: int, db: Session = Depends(get_db)):
    try:
        db_article = db.query(models.Article).filter(models.Article.id == id).first()
        if db_article is None:
            raise HTTPException(status_code=404, detail="Article not found")
        db.delete(db_article)
        db.commit()
        return {"detail": "Article deleted"}
    except SQLAlchemyError as e:
        db.rollback()
        raise HTTPException(status_code=500, detail=str(e))

if __name__ == "__main__":
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

LIITE B: RAJAPINNAN DOKUMENTAATIO

Required Python third-party packages

- fastapi==0.65.2
- sqlalchemy==1.4.15
- alembic==1.4.3
- pydantic==1.8.2
- pytest==6.2.4
- docker==5.0.0

Required Other language third-party packages

- No third-party packages required in other languages

Full API spec

```
openapi: 3.0.0
info:
  title: FastAPI Articles API
  version: 1.0.0
paths:
  /articles:
    post:
      summary: Create a new article
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Article'
      responses:
        '201':
          description: Article created
  /articles/{id}:
    get:
      summary: Get an article by ID
      parameters:
        - name: id
          in: path
          required: true
          schema:
            type: integer
    put:
      summary: Update an article by ID
      parameters:
        - name: id
          in: path
          required: true
          schema:
            type: integer
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Article'
      responses:
```

```
            '200':
              description: Article updated
        delete:
          summary: Delete an article by ID
          parameters:
            - name: id
              in: path
              required: true
              schema:
                type: integer
          responses:
            '204':
              description: Article deleted
    components:
      schemas:
        Article:
          type: object
          properties:
            id:
              type: integer
            title:
              type: string
            content:
              type: string
            created_at:
              type: string
              format: date-time
            updated_at:
              type: string
              format: date-time
```

Logic Analysis

- ['main.py', 'Contains the FastAPI application instance and the API routes']
- ['models.py', 'Contains the SQLAlchemy models']
- ['schemas.py', 'Contains the Pydantic models for data validation']
- ['database.py', 'Contains the SQLAlchemy session and engine configuration']
- ['alembic.ini', 'Contains the Alembic configuration']
- ['Dockerfile', 'Contains the Docker configuration']
- ['requirements.txt', 'Contains the Python dependencies']
- ['tests/test_main.py', 'Contains the Pytest tests']

Task list

- requirements.txt
- database.py
- models.py
- schemas.py
- main.py
- tests/test_main.py
- Dockerfile
- alembic.ini

Shared Knowledge

'database.py' contains the SQLAlchemy session and engine configuration, which is used by 'models.py' to define the SQLAlchemy models. 'schemas.py' uses Pydantic to define models for data validation, which are used by the API routes in 'main.py'. 'tests/test_main.py' contains the

Pytest tests for the API routes. The 'Dockerfile' contains the Docker configuration to containerize the application. 'alembic.ini' contains the Alembic configuration for database migrations.

Anything UNCLEAR

The requirement is clear to me.

LIITE C: TUOTEVAATIMUSDOKUMENTTI

Original Requirements

Write a FastAPI based application with no authentication. The application should allow users to handle articles.

Product Goals

- Create a FastAPI application
- Enable users to handle articles
- Ensure the application is efficient and user-friendly

User Stories

- As a user, I want to be able to create an article
- As a user, I want to be able to edit an article
- As a user, I want to be able to delete an article
- As a user, I want to be able to view all articles
- As a user, I want to be able to view a single article

Competitive Analysis

- Competitor A: Offers similar features but lacks in performance
- Competitor B: Has a user-friendly interface but lacks some features
- Competitor C: Offers more features but is not as efficient
- Competitor D: Has a robust system but lacks in user-friendliness
- Competitor E: Offers a similar product but lacks in customer support

Competitive Quadrant Chart

quadrantChart

title Reach and engagement of campaigns

x-axis Low Reach --> High Reach

y-axis Low Engagement --> High Engagement

quadrant-1 We should expand

quadrant-2 Need to promote

quadrant-3 Re-evaluate

quadrant-4 May be improved

Competitor A: [0.3, 0.6]

Competitor B: [0.45, 0.23]

Competitor C: [0.57, 0.69]

Competitor D: [0.78, 0.34]

Competitor E: [0.40, 0.34]

Our Target Product: [0.5, 0.6]

Requirement Analysis

The application needs to be built using FastAPI. It should allow users to create, view, edit, and delete articles. No authentication is required.

Requirement Pool

- ['P0', 'Build the application using FastAPI']
- ['P0', 'Implement functionality to create articles']
- ['P0', 'Implement functionality to view articles']

- ['P1', 'Implement functionality to edit articles']
- ['P1', 'Implement functionality to delete articles']

UI Design draft

The application should have a clean and minimalistic design. It should have a navigation bar at the top with options to create, view, edit, and delete articles. The main area of the application should display the articles.

Anything UNCLEAR

No unclear points at this moment.

LIITE D: JÄRJESTELMÄSUUNNITTELUDOKUMENTTI

Implementation approach

We will use the FastAPI framework to build the application. FastAPI is a modern, fast (high-performance), web framework for building APIs with Python 3.6+ based on standard Python type hints. For the database, we will use SQLAlchemy which is a SQL toolkit and Object-Relational Mapping (ORM) system for Python, along with Alembic for database migrations. Pydantic will be used for data validation and settings management using Python type annotations. For testing, we will use Pytest, a framework that makes it easy to write simple and scalable test cases. We will also use Docker for creating a containerized version of the application, which ensures that it runs the same regardless of the environment.

Python package name

fastapi_articles

File list

- main.py
- models.py
- schemas.py
- database.py
- alembic.ini
- Dockerfile
- requirements.txt
- tests/test_main.py

Data structures and interface definitions

```
classDiagram
    class Article{
        +int id
        +str title
        +str content
        +datetime created_at
        +datetime updated_at
        +__init__(id: int, title: str, content: str, created_at: datetime, updated_at: datetime)
        +create_article(title: str, content: str)
        +get_article(id: int)
        +update_article(id: int, title: str, content: str)
        +delete_article(id: int)
    }
```

Program call flow

```
sequenceDiagram
    participant M as Main
    participant A as Article
    M->>A: create_article("Title", "Content")
    A-->>M: Article
    M->>A: get_article(1)
    A-->>M: Article
    M->>A: update_article(1, "New Title", "New Content")
```

```
A-->>M: Article
M->>A: delete_article(1)
A-->>M: None
```

Anything UNCLEAR

The requirement is clear to me.