Jarno Laaksonen

PAIKALLISESTI AJETUN LAAJAN KIELIMALLIN (LLM) INTEGROINTI ASIAKASPALVELUSOVELLUKSEEN

Opinnäytetyö

Liiketalouden ammattikorkeakoulututkinto

Tietojenkäsittelyn koulutus

2025

Kaakkois-Suomen ammattikorkeakoulu

Kaakkois-Suomen ammattikorkeakoulu

Tutkintonimike	Tradenomi (AMK)
Tekijä/Tekijät	Jarno Laaksonen
Työn nimi	Paikallisesti ajetun laajan kielimallin (LLM) integrointi asiakaspalvelusovellukseen
Vuosi	2025
Sivut	36 sivua
Työn ohjaaja	Jukka Selin

TIIVISTELMÄ

Opinnäytetyössä tutkitaan paikallisesti ajettavan laajan kielimallin (Large Language Model, LLM) integrointia puheohjattuun asiakaspalvelusovellukseen. Tavoitteena on selvittää, miten LLM voidaan toteuttaa ilman pilvipalveluja ja millaisia hyötyjä tai haasteita tällainen lähestymistapa tuo verrattuna perinteisiin pilvipohjaisiin ratkaisuihin. Työssä tarkastellaan teoreettisesti LLM-mallien toimintaa sekä vertaillaan paikallisen ja pilvipohjaisen käytön ominaisuuksia muun muassa kustannusten, tietoturvan ja skaalautuvuuden näkökulmista.

Kehittämisosuudessa rakennetaan toimiva prototyyppi, jossa asiakaspalveluprosessi toteutetaan lähes täysin paikallisesti ajettavilla komponenteilla. Sovelluksen rakenne perustuu Python-ohjelmointikieleen ja FastAPI-kehykseen, ja siinä hyödynnetään muun muassa Whisper-mallia puheentunnistukseen, Ollamaa kielimallin suorittamiseen sekä LangChainia älykkään toiminnallisuuden laajentamiseen. Puhelinrajapinta toteutetaan Twilion avulla, ja vastaukset palautetaan käyttäjälle puhesynteesinä.

Työn tuloksena kehitettiin kokonaisuus, jossa asiakas voi soittaa puhelun, esittää kysymyksen ja saada vastauksen täysin paikallisesti tuotettuna – ilman verkkoyhteyttä tai ulkopuolista datankäsittelyä. Paikallisesti ajettava ratkaisu osoittautui teknisesti mahdolliseksi, ja se tarjoaa merkittäviä etuja erityisesti tietoturvan ja kustannusten hallinnan näkökulmasta. Haasteita liittyi erityisesti suorituskykyyn ja komponenttien yhteensovittamiseen, mutta ne ratkaistiin kehitystyön aikana onnistuneesti.

Asiasanat: tekoäly, kielimallit, puheentunnistus, asiakaspalvelu, ohjelmointi

South-Eastern Finland
University of Applied Sciences

Degree title	Bachelor of Business Administration
Author (authors)	Jarno Laaksonen
Thesis title	Integration of a Locally Run Large Language Model (LLM) into a Customer Service Application
Time	2025
Pages	36 pages
Supervisor	Jukka Selin

ABSTRACT

The thesis explores the integration of a locally run Large Language Model (LLM) into a voice-controlled customer service application. The goal is to determine how an LLM can be implemented without cloud services and to identify the benefits and challenges of this approach compared to traditional cloud-based solutions. The study examines the theoretical operation of LLMs and compares local and cloud-based usage in terms of cost, data security, and scalability.

In the development section, a working prototype is built, in which the customer service process is executed almost entirely using locally run components. The application is built with the Python programming language and the FastAPI framework, utilizing the Whisper model for speech recognition, Ollama for running the language model, and LangChain for extending intelligent functionality. The phone interface is implemented with Twilio, and responses are returned to the user as speech synthesis.

As a result of the project, a system was developed where a customer can make a phone call, ask a question, and receive a response — all processed locally, without an internet connection or external data handling. The locally run solution proved to be technically feasible and offers significant advantages, especially in terms of data security and cost control. Challenges primarily related to performance and component integration, but these were successfully resolved during the development process.

Keywords: artificial intelligence, language models, speech recognition, customer service, programming

SISÄLLYS

1 JOHDANTO 5

2 LAAJAT KIELIMALLIT (LLM) JA NIIDEN TOTEUTUSTAVAT 6

2.1 Transformer-arkkitehtuuri 6

2.2 Toteutustavat 7

3 KÄYTETTÄVÄT TEKNOLOGIAT JA TYÖKALUT 9

4 PAIKALLISEN LLM:N TOTEUTUS 17

4.1 Valmistelut ja ympäristön asennus 18

4.2 LLM-mallin lataus ja testaus 19

4.3 Sovelluksen arkkitehtuuri 21

4.4 FastAPI 22

4.5 Langchain 25

4.6 Twilio 27

4.7 Haasteet ja ratkaisut 30

4.8 Tiivistetty ohje 32

5 PÄÄTÄNTÖ 34

LÄHTEET 36

1 JOHDANTO

Tekoälyn kehitys on viime vuosina edennyt merkittävästi, ja erityisesti laajojen kielimallien (Large Language Models, LLM) rooli luonnollisen kielen käsittelyssä on kasvanut. Nämä mallit mahdollistavat monimutkaisten tekstipohjaisten tehtävien, kuten asiakaspalvelun automatisoinnin, toteuttamisen. Suurin osa tällaisista ratkaisuista on kuitenkin pilvipohjaisia, mikä aiheuttaa haasteita erityisesti tietoturvan, latenssin ja kustannusten näkökulmasta.

Tämän opinnäytetyön tarkoituksena on tutkia ja käytännön tasolla toteuttaa paikallisesti ajettavan laajan kielimallin (LLM) integrointi asiakaspalvelusovellukseen. Tarkoitus on tehdä tuote kansainvälisille markkinoille, joten kielimallien kielenä käytetään englantia. Työssä käsitellään sekä teoreettisia että teknisiä näkökulmia, joiden avulla voidaan ymmärtää, miten LLM voidaan ajaa paikallisesti ja mitä etuja tämä tuo verrattuna pilvipohjaisiin ratkaisuihin.

Opinnäytetyö jakautuu kahteen pääosioon. Ensimmäiseksi tarkastellaan teoreettisemmin laajojen kielimallien toimintaperiaatteita ja erityisesti Transformer-arkkitehtuuria, joka on keskeinen teknologia LLM-mallien kehityksessä. Tämän jälkeen vertaillaan paikallisesti ajettavia ja pilvipohjaisia LLM-malleja, jotta voidaan hahmottaa niiden keskeiset erot ja soveltuvuus eri käyttötilanteisiin. Toisessa osiossa käsitellään myös käytettäviä teknologioita ja työkaluja. Tämä sisältää kehitysympäristöt, ohjelmointikielet ja muita käytettyjä teknologioita.

Kolmannessa osiossa keskitytään käytännön toteutukseen. Tässä vaiheessa käydään läpi vaiheittain, kuinka paikallisesti ajettava LLM saadaan toimintaan, mitä ohjelmistoja ja työkaluja tarvitaan sekä miten API-rajapinnat voidaan toteuttaa.

2 LAAJAT KIELIMALLIT (LLM) JA NIIDEN TOTEUTUSTAVAT

Laajat kielimallit ovat edistyneitä tekoälyjärjestelmiä, jotka on kehitetty käsittelemään ja tuottamaan luonnollista kieltä. Ne perustuvat neuroverkkopohjaisiin arkkitehtuureihin, erityisesti transformer-malleihin, ja ne opetetaan laajoilla tekstiaineistoilla. LLM-mallit kykenevät suorittamaan monenlaisia luonnollisen kielen käsittelytehtäviä, kuten tekstin kääntämistä, tiivistämistä, kysymyksiin vastaamista ja keskustelun ylläpitämistä. Näiden mallien kehitystä on vauhdittanut laskentatehon kasvu, laajojen koulutusaineistojen saatavuus sekä kehittyneet oppimismenetelmät. (Naveed ym. 2024, 1)

2.1 Transformer-arkkitehtuuri

Transformer-arkkitehtuuri on luonnollisen kielen käsittelyyn kehitetty neuroverkkopohjainen malli, joka korvaa perinteiset rekursiiviset hermoverkot (RNN) ja konvoluutioneuroverkot (CNN). Se hyödyntää itsehuomiointimekanismia (self-attention), jonka avulla malli pystyy käsittelemään koko tekstikontekstia samanaikaisesti. Tämä eroaa perinteisistä kielimalleista, joissa sanat käsitellään järjestyksessä, mikä voi tehdä laskennasta hitaampaa ja vähemmän tehokasta. (Wang ym. 2019, 1)

Transformer-mallit, kuten GPT ja BERT, ovat osoittaneet tehokkuutensa erilaisissa luonnollisen kielen käsittelytehtävissä. GPT-malli perustuu monikerroksiseen Transformer-dekooderiin, jossa jokainen kerros koostuu itsehuomiointiyksiköistä ja täysin kytketyistä kerroksista. BERT puolestaan hyödyntää monikerroksista kaksisuuntaista Transformer-enkooderia, joka pystyy käsittelemään sanojen merkityksiä niiden ympäristön perusteella. (Wang ym. 2019, 2)

Vaikka Transformer-arkkitehtuuri on mullistanut luonnollisen kielen käsittelyn, sillä on myös haasteita. Esimerkiksi nykyiset itsehuomiointimekanismit eivät aina kykene tehokkaasti hyödyntämään sanajärjestystä, mikä on tärkeää ennustettaessa seuraavaa sanaa tekstissä. Tämän vuoksi on tutkittu menetelmiä, joissa Transformer-rakenteeseen lisätään LSTM-kerroksia, jotka voivat parantaa mallin kykyä hahmottaa sanojen keskinäisiä suhteita. (Wang ym. 2019, 2–3)

Transformer-arkkitehtuuri on tuonut merkittäviä parannuksia luonnollisen kielen käsittelyyn tarjoamalla tehokkaan, rinnakkaistettavan ja monikäyttöisen mallin. Sen soveltamiseen liittyy kuitenkin myös haasteita, joita pyritään ratkaisemaan yhdistämällä sitä perinteisempiin malleihin kuten LSTM-verkkoihin.

2.2 Toteutustavat

Laajojen kielimallien hyödyntäminen yleistyy nopeasti eri toimialoilla. Organisaatioilla on käytännössä kolme vaihtoehtoa LLM-teknologian käyttöönottoon: paikallinen (on-premises), pilvipohjainen (cloud-hosted) ja ulkoisten LLM-palveluntarjoajien hyödyntäminen, kuten OpenAI:n ChatGPT:n tai Anthropicin Claude:n käyttö. Näillä toteutustavoilla on olennaisia eroja kustannuksissa, hallinnassa, tietoturvassa ja käyttötapauksissa.

Paikallinen (on-premises)

Paikallisesti asennettu LLM antaa organisaatiolle täyden kontrollin mallin toimintaan, dataan ja resurssien käyttöön. Tämä on erityisen tärkeää silloin, kun käsitellään arkaluontoista tietoa, kuten potilastietoja tai luottamuksellista yritysdataa (Alka ym. 2024, 17). Paikallinen malli voidaan myös räätälöidä ja hienosäätää omaan käyttöön sopivaksi. Haasteina ovat suuret alkuinvestoinnit laitteistoon, vaativa ylläpito sekä asiantuntemuksen tarve (Dell Technologies 2024, 4).

Pilvipohjainen (cloud-hosted)

Pilvipohjaiset LLM-ratkaisut tarjoavat skaalautuvuutta ja joustavuutta. Ne sopivat erityisesti organisaatioille, jotka haluavat päästä nopeasti liikkeelle ilman suuria laitehankintoja. Pilvessä toimivan mallin voi kouluttaa tai hienosäätää omalla datalla, mutta kustannukset voivat kasvaa jatkuvan käytön myötä – erityisesti, jos käyttövolyymi on suuri tai jatkuvaa. (Principled Technologies 2024) Pilviratkaisun tietoturva ja vaatimustenmukaisuus on arvioitava tarkasti etenkin toimialoilla, joilla on tiukkoja säädöksiä.

Ulkoiset LLM-palveluntarjoajat

Kolmas vaihtoehto on käyttää valmiita ulkoisia kielimalleja, kuten OpenAI:n ChatGPT, Google Gemini tai Anthropicin Claude. Näissä ratkaisuissa ei tarvitse huolehtia infrastruktuurista, mallin kouluttamisesta tai skaalautuvuudesta – palveluntarjoaja hoitaa kaiken (Bommasani ym. 2021, 11). Tämä on erittäin kustannustehokas ratkaisu monissa käyttökohteissa, kuten asiakaspalvelussa, automaattisessa sisällöntuotannossa tai sisäisessä viestinnässä.

Ulkoisten palvelujen heikkous on tietosuojan hallinnan puute. API-pohjaisessa käytössä data siirtyy ulkoiselle palveluntarjoajalle, eikä organisaatiolla ole täydellistä näkyvyyttä siihen, miten tietoa käsitellään. Vaikka tunnetut toimijat tarjoavat yritystason tietoturvaratkaisuja, sensitiivisten tietojen kanssa toimiminen voi silti muodostaa riskin. (Frontiers in Big Data 2020) Lisäksi mallien hallinta, hienosäätö ja käyttäytymisen muokkaaminen on rajoitettua verrattuna omiin tai pilvessä koulutettuihin malleihin.

Kustannusmielessä ulkoiset LLM-rajapinnat ovat usein houkuttelevia, sillä ne toimivat käyttöön perustuvalla hinnoittelulla. API-hinnoittelu on erityisen kannattavaa, jos käyttömäärät ovat maltillisia ja LLM:n tarve ei ole jatkuvaa. Kuitenkin suurivolyymisessa käytössä tämäkin voi muodostua kalliiksi, jolloin oma tai pilvipohjainen LLM voi tulla halvemmaksi pitkällä aikavälillä. (Antematter 2025)

Organisaation kannattaa valita LLM-toteutustapa käyttötarkoituksen, datan luonteen, tietoturvavaatimusten, kustannusten ja osaamisen perusteella. Paikalliset mallit tarjoavat suurimman hallinnan, mutta vaativat paljon resursseja. Pilvipohjaiset mallit ovat joustavia ja tehokkaita, mutta niiden hallinta ja kustannukset on arvioitava huolellisesti. Ulkoiset LLM-palvelut ovat nopeita ja helppokäyttöisiä, mutta voivat tuoda mukanaan yksityisyys- ja hallittavuushaasteita.

3 KÄYTETTÄVÄT TEKNOLOGIAT JA TYÖKALUT

Tämän opinnäytetyön kehittämisosuudessa hyödynnettiin useita keskeisiä teknologioita, joiden valinta perustui niiden yhteensopivuuteen, avoimuuteen, skaalautuvuuteen ja erityisesti mahdollisuuteen toteuttaa ratkaisu paikallisesti ilman riippuvuutta pilvipalveluista. Työssä yhdistettiin nykyaikaisia tekoälyratkaisuja ja puheentunnistustekniikoita, minkä mahdollisti monipuoliset, mutta hyvin keskenään integroitavissa olevat teknologiat. Seuraavassa kuvataan tarkemmin, miten eri teknologiat palvelivat työn tavoitteita ja mitä hyötyjä niiden käytöstä oli.

Valitut teknologiat muodostivat kokonaisuuden, joka mahdollisti monipuolisen ja tehokkaan asiakaspalvelusovelluksen rakentamisen. Teknologiat tukivat paikallista ajettavuutta, tietoturvaa, kielellistä saavutettavuutta ja älykkään toiminnallisuuden laajennettavuutta – kaikkia työn tavoitteiden kannalta keskeisiä ominaisuuksia.

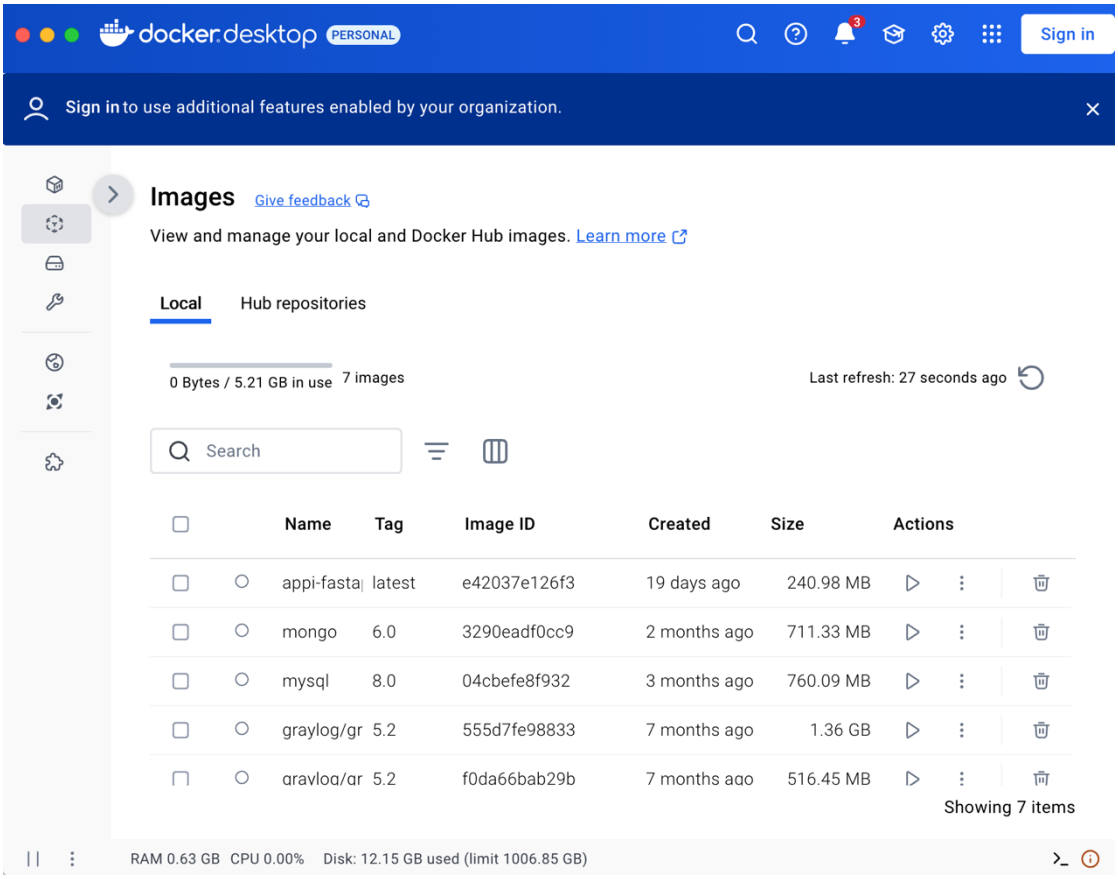
Python

Työn ohjelmointikielenä käytettiin Python-ohjelmointikieltä. Sen käyttö perustui ensisijaisesti sen yksinkertaiseen syntaksiin, laajaan kirjastoekosysteemiin ja erinomaiseen tuettavuuteen tekoälysovellusten rakentamisessa. Python tarjosi valmiit työkalut niin puheentunnistukseen, laajojen kielimallien käyttöön, API-rajapintojen toteuttamiseen kuin puhesynteesiinkin. Sen suosio tekoälyn ja koneoppimisen yhteisössä mahdollisti myös sen, että kaikki työn kannalta keskeiset kirjastot ja teknologiat olivat yhteensopivia Pythonin kanssa. Lisäksi Pythonin asynkroniset toiminnot (async/await) tukivat työn tarvetta käsitellä useita samanaikaisia prosesseja, kuten puhelupyyntöjä ja mallivastauksia.

Docker

Docker (kuva 1) tarjosi tehokkaan ja joustavan tavan hallita riippuvuuksia, eriyttää komponentteja ja varmistaa, että järjestelmä toimii yhdenmukaisesti eri kehitysalustoilla. Koska sovellus koostui useista itsenäisistä osista – kuten FastAPI-rajapinnasta, puheentunnistuksesta, kielimallista ja puhesynteesistä –

oli tärkeää, että jokainen komponentti voitiin määritellä, käynnistää ja hallita erillään, mutta silti osana yhteistä kokonaisuutta.



Kuva 1. Docker Desktop -käyttöliittymä

Dockerin käyttö mahdollisti kehitysympäristön standardoinnin ja toistettavuuden. Kaikki sovelluksen osat voitiin määritellä *Dockerfile*- (kuva 2) ja *docker-compose.yml*-tiedostojen avulla, jolloin järjestelmä oli käynnistettävissä yhdellä komennolla täysin identtisesti eri koneilla. Tämä vähensi merkittävästi kehitykseen liittyviä konfiguraatio-ongelmia ja teki projektin jakamisesta, dokumentoinnista ja testaamisesta huomattavasti helpompaa.

```
FROM python:3.11-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Kuva 2. Esimerkki Dockerfilestä

Konttien käyttö oli erityisen tärkeää silloin, kun sovelluksessa käytettiin kirjastoriippuvaisia tai raskaita komponentteja, kuten puheentunnistusmallia (Whisper) tai paikallisesti ajettavaa LLM-mallia (esimerkiksi Ollama). Näiden työkalujen asentaminen suoraan koneelle olisi voinut aiheuttaa ristiriitoja olemassa olevien järjestelmäkomponenttien kanssa. Dockerin ansiosta kaikki tarvittavat kirjastot, ohjelmistot ja ajoympäristöt voitiin eristää omiin kontteihinsa, jolloin ne eivät vaikuttaneet toisiinsa tai isäntäkoneen asetuksiin.

```
fastapi-app:
  build: .
  ports:
    - "8000:8000"
  environment:
    - OLLAMA_URL=http://ollama:11434
    - DB_HOST=mysql
    - DB_NAME=appdb
    - DB_USER=appuser
    - DB_PASS=apppass
  depends_on:
    - ollama
    - mysql
  volumes:
    - .:/app
```

Kuva 3. Docker compose -esimerkki

Toteutuksessa käytettiin *docker-compose*-konfiguraatiota (kuva 3), jossa määriteltiin erilliset palvelut esimerkiksi FastAPI-sovellukselle, Whisper-palvelulle ja kielimallille. Jokaiselle palvelulle voitiin määrittää omat portit, ympäristömuuttujat ja volyymit tiedostojen jakamista varten. Tämä teki kehitystyöstä modulaarista ja skaalautuvaa: yksittäisen palvelun uudelleenkäynnistäminen,

korvaaminen tai päivittäminen ei vaikuttanut muihin osiin, mikä on tärkeä ominaisuus.

Dockerin käyttö helpotti myös testaamista ja virheiden jäljittämistä. Koska koko ympäristö oli käynnistettävissä uudelleen nopeasti ja identtisesti, pystyttiin palaamaan aiempaan, toimivaan versioon ilman vaaraa, että koneelle jäänyt vanha konfiguraatio aiheuttaisi ongelmia. Lisäksi konttien sisällä tapahtuva lokitus tarjosi arvokasta tietoa jokaisen komponentin toiminnasta ja mahdollisista virhetilanteista, kuten puuttuvista riippuvuuksista tai verkkoyhteyksien katkeamisesta.

FastAPI

Palvelinpuolen toteutuksessa käytettiin FastAPI-kehystä, joka valittiin sen teknisten ominaisuuksien ja soveltuvuuden vuoksi tähän projektiin. FastAPI on Python-kielellä toimiva kehys, joka soveltuu erityisesti REST-tyyppisten ohjelmointirajapintojen rakentamiseen. Sen avulla rajapintojen määrittely oli nopeaa, ja kehys loi automaattisesti OpenAPI-muotoisen dokumentaation, jota voitiin käyttää sekä testaamiseen että rajapinnan rakenteen tarkasteluun. Tämä helpotti kehitystyötä merkittävästi, sillä dokumentaation avulla pystyttiin näkemään, millaisia pyyntöjä palvelin odottaa ja millaisia vastauksia se antaa.

FastAPI:n syntaksi perustuu Pythonin tyyppimerkintöihin, minkä ansiosta koodi oli selkeää ja helposti luettavaa. Kehitystyötä tehtäessä käytettiin Uvicorn-nimistä palvelinta, joka mahdollisti sovelluksen ajamisen asynkronisesti. Tämä oli tärkeää, koska sovelluksessa käsiteltiin puhetta reaaliaikaisesti.

FastAPI toimi myös yhteyden vastaanottajana ulkopuolisilta järjestelmiltä. Esimerkiksi Twilio lähetti saapuvasta puhelusta HTTP-pyynnön, jonka FastAPI vastaanotti webhookin kautta. Sovellus käsitteli tämän pyynnön ja palautti TwiML-muotoisen vastauksen, joka sisälsi ohjeet puhelun käsittelyyn. Näin FastAPI toimi keskeisenä osana sovellusta, sillä sen kautta ohjattiin kaikki toiminnallisuus, kuten puheen tallennus, transkriptio, kielimallin käyttö ja vastausten palauttaminen.

FastAPI yhdisti kaikki sovelluksen osat toimimaan yhdessä. Se teki sovelluksen rakenteesta selkeän ja sitä oli helppo laajentaa myöhemmin. FastAPI osoittautui toimivaksi ja tarkoituksenmukaiseksi ratkaisuksi tämän tyyppisessä sovelluskehityksessä.

Ollama

Työssä käytetty kielimallipalvelu perustui Ollama-nimiseen ohjelmistoon, jonka avulla laajoja kielimalleja voitiin käyttää paikallisesti ilman internet-yhteyttä. Tämä tarkoitti sitä, että mitään tietoa ei tarvinnut lähettää ulkoisiin palveluihin, vaan kaikki käsittely tapahtui omalla koneella. Ollama tarjosi komentorivipohjaisen käyttöliittymän, jonka avulla mallit voitiin ladata ja käynnistää helposti. Lisäksi Ollamassa oli sisäänrakennettu HTTP-rajapinta, jonka kautta mallia voitiin käyttää sovelluksesta käsin.

FastAPI-sovellus voitiin liittää Ollaman tarjoamaan rajapintaan yksinkertaisesti lähettämällä sille HTTP-pyyntöjä, jolloin saatiin takaisin mallin tuottama vastaus. Tämän ansiosta kielimallin käyttö sovelluksessa ei vaatinut erillistä ohjelmointia mallin käynnistämiseen tai hallintaan. Ollaman avulla pystyttiin säilyttämään kaikki puhe- ja tekstidata paikallisesti, mikä paransi tietoturvaa ja yksityisyyden suojaa.

Ollama tuki useita eri kielimalleja. Tässä työssä kokeiltiin muun muassa DeepSeek-Coder -mallia. Mallin suorittaminen onnistui tavallisella tietokoneella, mikä helpotti kehitystyötä. Kokonaisuutena Ollama mahdollisti kielimallin käytön osana sovellusta ilman pilvipalveluita tai monimutkaista asennusta.

Whisper

Puheentunnistuksessa käytettiin OpenAI:n kehittämää Whisper-mallia, joka mahdollistaa puheen muuntamisen tekstiksi. Whisperin merkittävin etu tässä työssä oli sen kyky toimia täysin paikallisesti ilman verkkoyhteyttä. Se oli tarkka erityisesti selkeästi artikuloidussa puheessa ja sieti kohtalaisen hyvin myös taustahälyä. Mallin skaalautuvuus (eri tarkkuustasoja, kuten base, medium ja large) mahdollisti tasapainottelun tarkkuuden ja suorituskyvyn välillä.

Whisper toimi saumattomasti osana puheluprosessia, ja sen tuottama teksti ohjattiin eteenpäin kielimallin tulkittavaksi.

LangChain

Jotta kielimalli saatiin käyttäytymään älykkäämmin ja vastaamaan tarpeisiin, hyödynnettiin LangChain-kirjastoa. LangChain mahdollisti agenttipohjaisen arkkitehtuurin, jossa LLM ei ainoastaan vastannut kysymyksiin, vaan osasi myös itse päätellä, milloin käyttää ulkoisia toimintoja, kuten funktioiden käyttöä. LangChainin tarjoama rakenteellinen lähestymistapa teki LLM:n käyttämisestä järjestelmällisempää ja tarjosi mahdollisuuden rakentaa skaalautuvia ja laajennettavia ketjuja (chains), joissa mallin vastausprosessi oli hallittavissa ja läpinäkyvä. Tämän avulla kielimalli kykeni toimimaan logiikan ja tehtävänjaon kautta älykkäänä järjestelmäagenttina.

Twilio

Yhteys loppukäyttäjään toteutettiin Twilion avulla, joka on pilvipohjainen viestintäalusta. Twilio mahdollisti puheluiden vastaanoton ja niihin vastaamisen ohjelmallisesti, mikä oli olennainen osa työn tavoitetta luoda puhekäyttöliittymä asiakaspalvelusovellukselle. Twilion tarjoama webhook-pohjainen toimintamalli sopi erinomaisesti FastAPI:n kanssa yhteen, ja sen avulla pystyttiin toteuttamaan sekä puheen tallennus että puhevastauksen toistaminen. Twilio oli luotettava ja kehittäjäystävällinen alusta, joka nopeutti toteutusta merkittävästi selkeällä dokumentaatiollaan ja testityökaluillaan.

Coqui TTS

Mallin tuottamien tekstivastausten muuttaminen kuultavaan muotoon toteutettiin Coqui TTS -kirjaston avulla. Coqui TTS on avoimen lähdekoodin tekstistä puheeksi -ratkaisu, joka mahdollistaa puhesynteesin suorittamisen täysin paikallisesti ilman verkkoyhteyttä. Tämä oli erityisen tärkeä ominaisuus työn tavoitteiden kannalta, sillä se mahdollisti kielimallin tuottamien vastausten muuntamisen puheeksi tietoturvallisesti ja riippumattomasti ulkoisista palveluista.

Coqui TTS tukee useita eri kieliä, mukaan lukien suomi. Äänitiedostot tehtiin WAV-muodossa, mikä mahdollisti niiden käytön suoraan osana puhelinvastausta ilman erillistä muuntamista.

Coqui TTS:n käyttöönotto oli teknisesti suoraviivaista, ja sen tarjoamat API-ratkaisut toimivat hyvin yhteen Python-pohjaisen sovelluksen kanssa. Äänen luonnollisuus ja ääntämisen tarkkuus olivat riittävällä tasolla prototyyppikäyttöä varten, ja mallin paikallinen ajettavuus tuki kokonaisarkkitehtuurin perusperiaatetta: kaikki toiminnallisuus toteutetaan ilman pilviyhteyksiä. Näin ollen Coqui TTS tarjosi tehokkaan ja tietoturvallisen tavan palauttaa kielimallin muodostamat vastaukset asiakkaalle kuultavassa muodossa puhelun aikana.

Ngrok

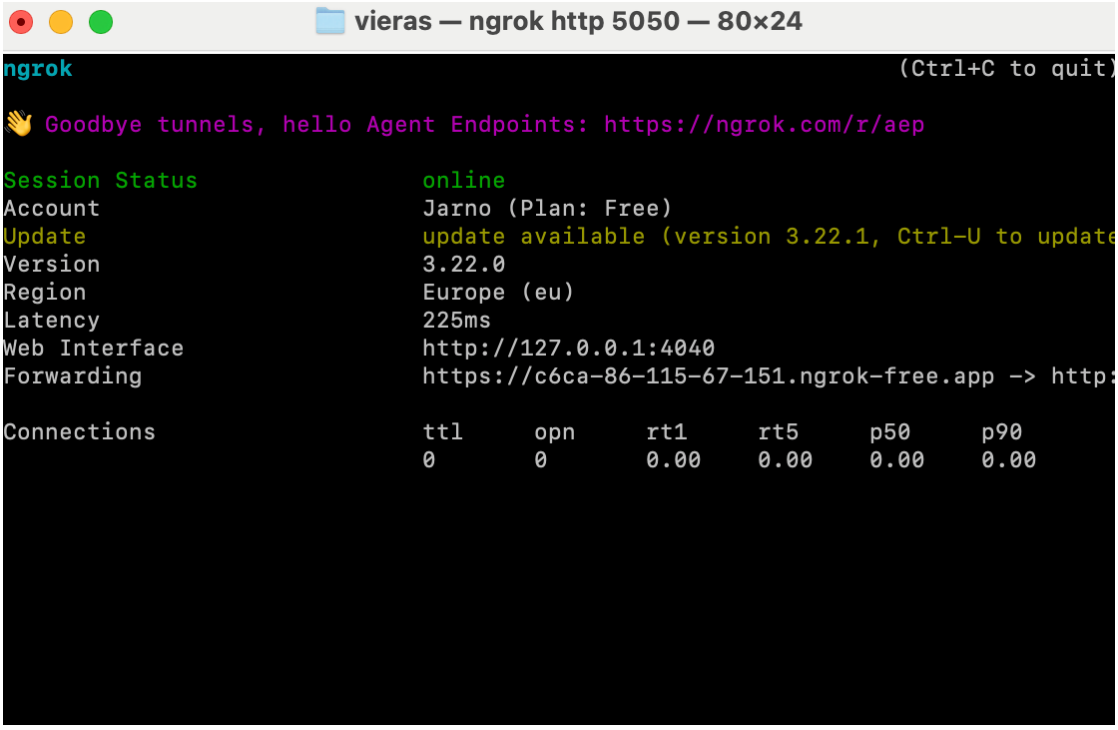
Kehitystyön aikana käytettiin lisäksi *Ngrok*-nimistä työkalua (kuva 4), joka mahdollisti paikallisen FastAPI-sovelluksen julkaisemisen väliaikaisesti internetiin turvallisen HTTPS-tunnelin kautta. Ngrok loi julkisesti saatavilla olevan URL-osoitteen, johon ulkopuoliset palvelut – kuten Twilio – pystyivät lähettämään webhook-kutsuja. Tämä oli kehitystyön kannalta korvaamaton apuväline, sillä sen avulla pystyttiin simuloimaan tuotantoympäristöä ilman, että tarvittiin erillistä verkkopalvelinta, DNS-konfiguraatioita tai pilvipohjaista infrastruktuuria.

Ngrokin käyttö oli erityisen hyödyllistä tilanteissa, joissa tarvittiin nopeita testisyklejä tai haluttiin kokeilla muutoksia järjestelmän eri osiin ilman raskasta käyttöönottoprosessia. Kun esimerkiksi FastAPI-sovellukseen tehtiin muutos, voitiin uusi versio käynnistää paikallisesti ja heti yhdistää ulkomaailmaan uu-

den Ngrok-linkin avulla. Tämä mahdollisti sen, että Twilion webhookit tavoittivat sovelluksen viiveettä, ja koko vuorovaikutusketju (puhelu, puheentunnistus, kielimallivastaus, puhesynteesi) voitiin testata realistisessa tilanteessa.

Tekninen toteutus oli yksinkertainen: Ngrok asennettiin paikalliseen kehitysympäristöön ja se käynnistettiin komentoriviltä komennolla, joka loi julkisen HTTPS-osoitteen porttiin 8000 osoittavalle paikalliselle palvelimelle.

Tämä osoite liitettiin Twilion hallintapaneeliin webhook-osoitteeksi, jolloin saapuvat puhelut reitittyivät suoraan kehityksessä käytettävään sovellukseen. Kehitystyön aikana Ngrokin web-käyttöliittymä tarjosi myös hyödyllistä tietoa kutsujen sisällöstä, HTTP-vasteista ja mahdollisista virhetilanteista, mikä helpotti virheiden paikantamista ja korjaamista.



Kuva 4. Ngrok ajettuna päätteessä

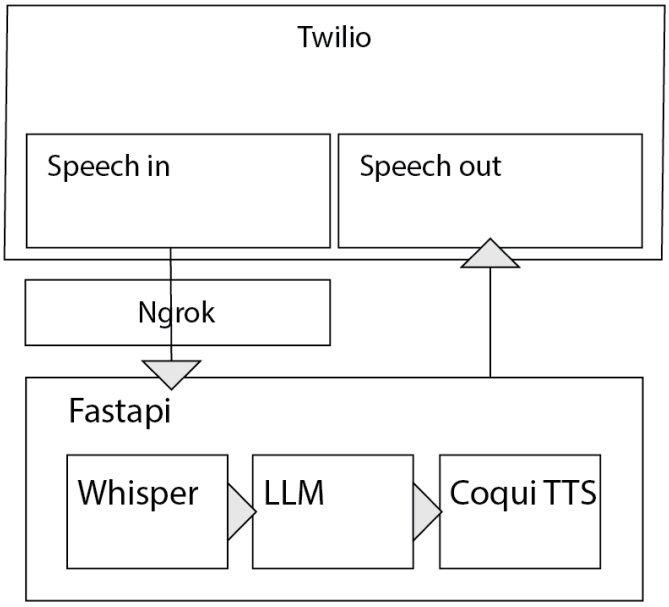
Ngrok toimi erityisesti kehityksen alkuvaiheessa kriittisenä linkkinä ulkoisen järjestelmän ja paikallisen ympäristön välillä. Sen ansiosta integraatiot voitiin toteuttaa vaiheittain ja iteratiivisesti ilman suuria infrastruktuurimuutoksia. Lisäksi se mahdollisti etäesittelyt ja demonstroinnit, joissa paikallisesti kehitetty

sovellus voitiin jakaa ulkopuolisille testaajille tai ohjaajille helposti ja turvallisesti.

Vaikka Ngrokin käyttö on luonteeltaan tilapäistä ja soveltuu parhaiten kehitysympäristöön, niin sen rooli tässä projektissa oli keskeinen: se mahdollisti koko järjestelmän testauksen aidossa käyttötilanteessa ilman tarvetta julkaista sovellusta ennen kuin se oli valmis. Ngrok nopeutti kehitystyötä, vähensi esteitä ja teki kokeilusta ja oppimisesta sujuvaa – mikä on erityisen arvokasta innovatiivisten, kokeellisten ohjelmistoprojektien yhteydessä.

4 PAIKALLISEN LLM:N TOTEUTUS

Tässä luvussa kuvataan kehitystyön keskeisin osa: paikallisesti ajettavan laajan kielimallin käyttöönotto ja sen integrointi puheohjattuun asiakaspalvelusovellukseen. Kuvassa 5 on esitettynä kaavio toteutuksesta.



Kuva 5. Toteutuksen kaavio

Luvussa esitellään sovelluksen arkkitehtuuri ja sen eri komponentit, kuten ohjelmointirajapinta (FastAPI), älykäs agenttirakenne (LangChain), puhekäyttöliittymä (Twilio) sekä puheentunnistus- ja puhesynteesiratkaisut.

4.1 Valmistelut ja ympäristön asennus

Ennen sovelluksen varsinaista kehittämistä oli tarpeen luoda tekninen ympäristö, jossa eri komponentit voisivat toimia yhdessä tehokkaasti ja luotettavasti. Valmisteluvaiheessa asennettiin tarvittavat ohjelmistot, määriteltiin riippuvuudet ja varmistettiin, että järjestelmä tukee sekä paikallisesti ajettavaa kielimallia että web-pohjaista sovelluslogiikkaa.

Kielimallin käyttöön liittyvä infrastruktuuri rakennettiin siten, että se olisi mahdollisimman kevyt ja helposti hallittavissa. Mallin suorittamiseen käytettävä ohjelmisto asennettiin suoraan käyttöjärjestelmään ja sen jälkeen suoritettiin alustavat testit mallin latauksen ja toimivuuden varmistamiseksi. Lisäksi määriteltiin, kuinka ohjelmaresurssit, kuten suoritin- ja muistinkäyttö optimoidaan erityisesti ilman grafiikkasuoritinta toimivassa ympäristössä.

Ympäristön hallinnan helpottamiseksi käyttöönotettiin konttipohjainen ratkaisu, jonka avulla eri komponenttien kuten kielimallipalvelun ja sovelluslogiikan asennus ja käyttö voitiin eriyttää toisistaan. Konttiteknologian avulla kehitystyötä pystyttiin jatkamaan riippumattomasti käyttöjärjestelmästä ja varmistamaan, että sovellus toimii samanlaisesti eri koneilla. Tämä mahdollistaisi myös kehitystyön jatkamisen tai jakamisen toisille kehittäjille ilman tarvetta yksityiskohtaiselle ympäristökonfiguraatiolle.

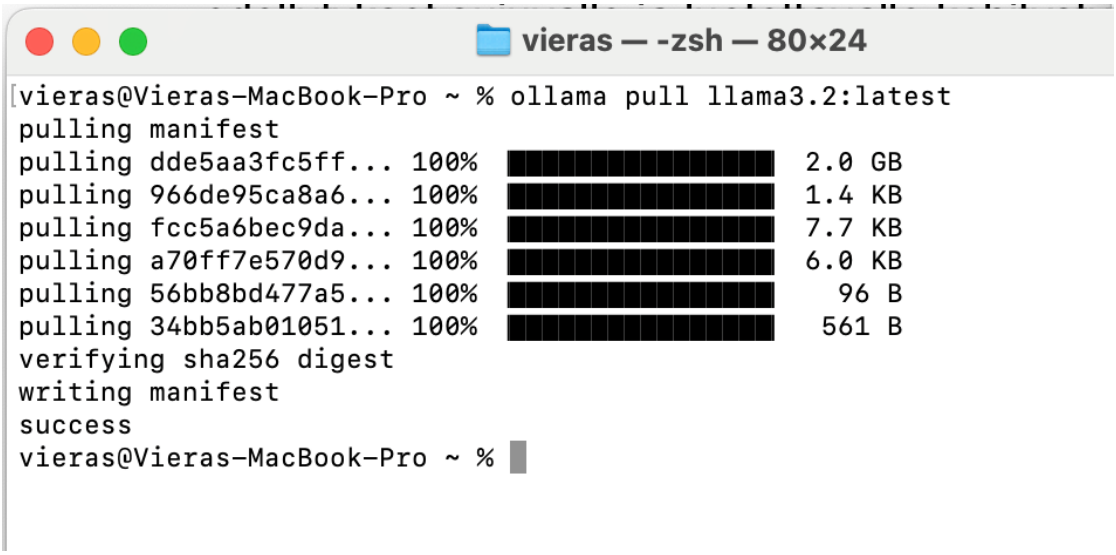
Samalla luotiin ohjelmointirajapinnan käyttöön tarvittava runko, johon sisällytettiin kaikki olennaiset kirjastot ja konfiguraatiot sovelluksen jatkokehitystä varten. Virtuaaliympäristöön lisättiin vain kehittämistyössä käytettävät riippuvuudet. Tällä varmistettiin, että ympäristö pysyi selkeänä ja helposti hallittavana, ja että sovelluksen komponentit kommunikoivat keskenään odotetulla tavalla.

Kun tekninen perusta oli valmis, voitiin siirtyä kielimallin varsinaiseen lataamiseen ja soveltamiseen osana asiakaspalveluprosessia. Valmisteluvaihe loi edellytykset sujuvalle ja luotettavalle kehitystyölle, jossa sekä toiminnallisuus että siirrettävyys voitiin toteuttaa hallitusti.

4.2 LLM-mallin lataus ja testaus

Teknisen ympäristön valmistelun jälkeen seuraava vaihe kehittämistyössä oli laajojen kielimallien käyttöönotto ja niiden toimivuuden varmistaminen. Sovelluksen ytimenä toimi paikallisesti ajettava kielimalli, jonka tuli vastata käyttäjän kysymyksiin nopeasti ja johdonmukaisesti ilman ulkoisia verkkopalveluja. Tämän vuoksi mallin oikea valinta ja huolellinen testaus olivat keskeisessä roolissa.

Ensimmäiseksi valittiin käyttöön soveltuva malli, joka tukee englannin kieltä, tuottaa johdonmukaisia vastauksia ja on riittävän kevyt paikalliseen ajoon ilman erillistä grafiikkasuoritinta. Mallin lataus toteutettiin komentorivityökalun kautta, ja prosessi oli teknisesti suoraviivainen (kuva 6). Malli tallennettiin paikallisesti, jolloin sitä voitiin käyttää ilman jatkuvaa verkkoyhteyttä, mikä vastasi työn tavoitteita tietoturvan ja käytettävyyden näkökulmasta.



Kuva 6. LLM-mallin lataaminen koneelle

Mallin toiminta testattiin ensin suoraan komentoriviltä. Käyttäjän syöttämät kysymykset käsiteltiin mallin kautta, ja vasteen muodostumista sekä sisällön laatua arvioitiin sekä teknisesti että kielellisesti (kuva 7).

```
vieras@Vieras-MacBook-Pro ~ % ollama run llama3.2:latest "what is e?"
"E" can have different meanings depending on the context. Here are a few
possible interpretations:

1. **Element**: In chemistry, "e" stands for the element Einsteinium, a
radioactive metal with the atomic number 99.
2. **Electricity**: In electrical engineering and physics, "e" is often
used to represent the unit of electric charge, called the elementary
charge (approximately equal to 1.602 x 10^-19 Coulombs).
3. **Mathematics**: In mathematics, "e" is a fundamental constant, known
as Euler's number, approximately equal to 2.71828. It's a fundamental
element in calculus and appears in many mathematical formulas.
4. **Programming**: In programming languages like C and Java, "e" can be
used to represent the symbol for exponentiation (e.g., x^e).
5. **Other meanings**: E can also stand for other things, such as
"envelope," "email," or a specific chemical formula.

Without more context, it's difficult to determine which interpretation is
most relevant. If you have any additional information or clarification,
I'd be happy to try and provide a more specific answer!
```

Kuva 7. LLM-mallin testaaminen päätteessä

Tämän testauksen avulla varmistettiin, että mallin tuottamat vastaukset olivat loogisia, selkeitä ja käyttökelpoisia asiakaspalvelun kontekstissa. Erityistä huomiota kiinnitettiin siihen, kuinka hyvin malli ymmärsi yksinkertaisia ja rakenteeltaan vaihtelevia kysymyksiä, sekä siihen, oliko vasteaika käytännön tilanteissa hyväksyttävällä tasolla.

Kun mallin perustoiminta oli varmistettu, sitä alettiin käyttää osana sovelluksen ohjelmointirajapintaa. Mallia voitiin kutsua ohjelmallisesti lähettämällä sille pyyntöjä HTTP-rajapinnan kautta. Tämä mahdollisti sen integroinnin muihin sovelluksen osiin, kuten puheentunnistukseen, vastauslogiikkaan ja puhesynteesiin. Tällöin mallin toiminta ei ollut enää sidottu komentorivipohjaiseen vuorovaikutukseen, vaan se voitiin liittää osaksi kokonaisvaltaista asiakaspalveluprosessia.

Mallin testauksessa havaittiin, että sen suorituskyky riippuu merkittävästi koneen resursseista. Kevyemmillä malleilla saavutettiin riittävän nopeita vasteaikoja myös ilman GPU-tukea. Lisäksi todettiin, että yksinkertaisilla kysymyksillä saavutettiin johdonmukaisempi laatu, kun taas monitulkintaiset kysymykset saattoivat toisinaan johtaa tarpeettoman pitkiin tai epäolennaisiin vastauksiin. Näiden havaintojen perusteella tehtiin jo tässä vaiheessa alustavia rajauksia siitä, millaisia käyttötapauksia sovelluksella kannattaisi jatkossa ensisijaisesti tukea.

Kielimallin onnistunut käyttöönotto ja alustava testaus antoivat kuitenkin hyvän pohjan jatkokehitykselle. Näin ollen seuraavaksi siirryttiin sovelluksen arkkitehtuurin rakentamiseen ja eri komponenttien välisen tiedonkulun suunnitteluun.

4.3 Sovelluksen arkkitehtuuri

Kun tekninen ympäristö ja kielimallin käyttöönotto oli saatu valmiiksi, siirryttiin kehittämistyön seuraavaan vaiheeseen, eli sovelluksen arkkitehtuurin rakentamiseen. Tavoitteena oli luoda selkeästi jäsennelty kokonaisuus, jossa eri komponentit toimivat itsenäisesti mutta saumattomassa yhteistyössä. Sovelluksen arkkitehtuuri suunniteltiin modulaariseksi, jotta sen osia olisi mahdollista kehittää, korvata tai laajentaa ilman, että koko järjestelmää tarvitsee muuttaa. Tämä mahdollistaa sovelluksen soveltamisen myös tulevissa projekteissa tai muissa käyttökonteksteissa.

Sovellus koostuu useasta erillisestä osasta, joista kukin vastaa yhdestä toiminnallisesta vaiheesta prosessissa. Ensimmäinen osa on puheluiden vastaanotto, joka toteutettiin kolmannen osapuolen, eli Twilion, avulla. Twilio toimii ikään kuin välittäjänä asiakkaan ja sovelluksen välillä ohjaten saapuvan puhelun rajapintaan, joka on toteutettu FastAPI-kehyksellä. Kun puhelu vastaanotetaan, käyttäjän puhe tallennetaan äänitiedostona, joka toimitetaan edelleen sovellukselle käsiteltäväksi.

Tallennettu puhe analysoidaan seuraavaksi puheentunnistuksen avulla. Tässä vaiheessa sovellus käyttää transkriptiotyökalua, joka muuntaa puhutun sisällön tekstimuotoon. Transkripti toimii kielimallille syötettävänä aineistona, jonka perusteella järjestelmä muodostaa vastauksen. Kielimallin tehtävänä on tuottaa mahdollisimman luonnollinen, looginen ja asiayhteyteen sopiva tekstivastaus asiakkaan esittämään kysymykseen.

Mallin muodostama vastaus toimitetaan vielä ennen palauttamista puhesynteesille, joka muuntaa tekstin takaisin puheeksi. Tämä tehdään, jotta vastaus voidaan toistaa asiakkaalle puhelun aikana. Lopullinen puhevastaus palautetaan Twilion kautta takaisin asiakkaan linjalle, jolloin asiakas saa suullisen

vastauksen kysymykseensä ilman, että hänen tarvitsee koskaan olla suoraan vuorovaikutuksessa ohjelmiston teknisten osien kanssa.

Sovellusarkkitehtuuri perustuu palvelupohjaiseen ajatteluun, jossa jokaisella komponentilla on selkeä vastuualue ja rajapinta muiden osien kanssa. Tällainen rakenne mahdollistaa eri teknologioiden hyödyntämisen rinnakkain, mutta myös yksittäisten komponenttien vaihtamisen toisiin tarpeen mukaan. Esimerkiksi puhesynteesiratkaisun voi halutessaan vaihtaa laadukkaampaan äänenmuodostukseen erikoistuneeseen palveluun, tai kielimallin voi myöhemmin päivittää tehokkaampaan versioon ilman, että muun sovelluksen rakennetta tarvitsee muuttaa.

Arkkitehtuuriratkaisuissa painotettiin erityisesti tiedonkulun loogisuutta ja yksinkertaisuutta. Järjestelmässä ei käsitellä käyttäjätietoja, eikä säilytetä tietoja pysyvästi, mikä helpottaa tietoturvaan liittyviä kysymyksiä. Sovelluksen eri osat kommunikoivat toistensa kanssa pääosin HTTP-rajapintojen välityksellä, mikä tekee järjestelmästä helposti testattavan ja laajennettavan. Kehitetyn kokonaisuuden voidaan katsoa muodostavan teknisesti ja toiminnallisesti selkeän perustan puheohjatulle asiakaspalvelusovellukselle.

4.4 FastAPI

Yksi sovelluskehityksen keskeisimmistä päätöksistä oli valita FastAPI ohjelmointirajapinnan (API:n) toteutukseen. FastAPI on moderni, Python-pohjainen web-kehys, joka on suunniteltu erityisesti REST-tyyppisten HTTP-rajapintojen rakentamiseen. Sen vahvuuksia ovat korkea suorituskyky (Starlette- ja Uvicorn-pohjainen toteutus), sisäänrakennettu tuki asynkroniselle ohjelmoinnille sekä selkeä, Pythonin tyyppimerkintöihin perustuva syntaksi. Valintaan vaikutti myös FastAPI:n erinomainen yhteensopivuus muiden käytettyjen teknologioiden – kuten Twilion, Whisperin ja Ollaman – kanssa.

```
vieras@Vieras-MacBook-Pro twilio_voice % python3 -m venv venv
vieras@Vieras-MacBook-Pro twilio_voice % source venv/bin/activate
(venv) vieras@Vieras-MacBook-Pro twilio_voice % pip3 install fastapi uvicorn httpx pydantic
python-multipart
```

Kuva 8. Esimerkki virtuaaliympäristön ja riippuvuuksien asentamisesta ilman Docker ympäristöä

Kehitystyö aloitettiin perustamalla Pythonin virtuaaliympäristö ja asentamalla riippuvuudet, kuten fastapi, uvicorn, https, pydantic ja python-multipart (kuva 8). Tämän jälkeen luotiin projektirakenne, jossa eroteltiin loogisesti routes (kuva 9), services, utils ja main.py

```python
from fastapi import FastAPI
from routes import twilio_routes

app = FastAPI()
app.include_router(twilio_routes.router)
```

Kuva 9. Esimerkki routes-käytöstä

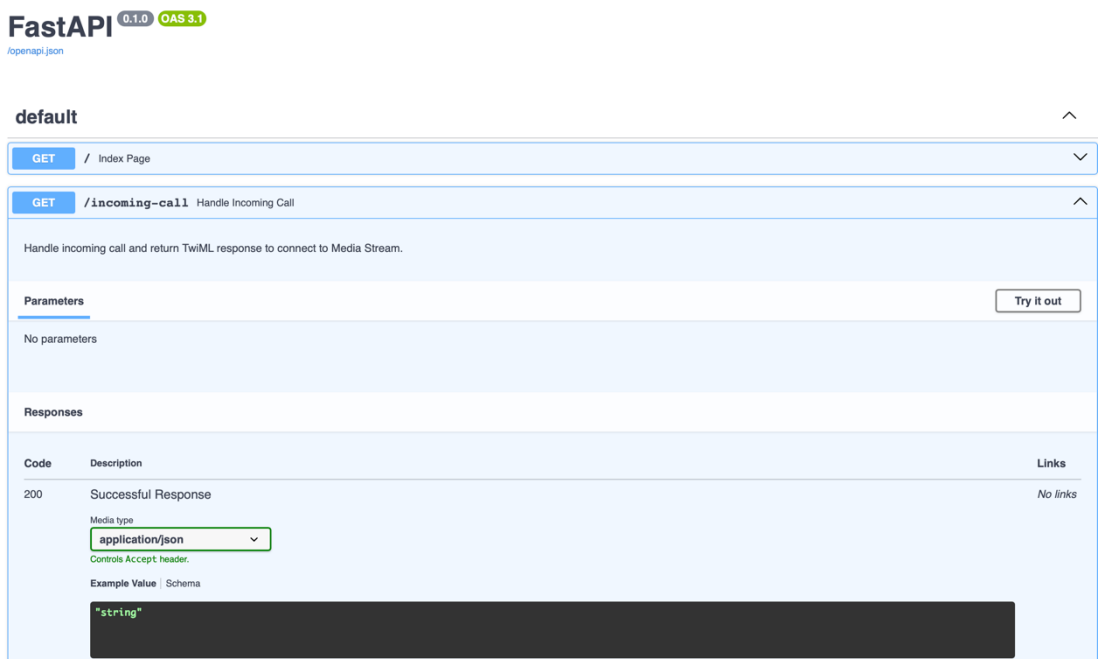
Yksi tärkeimmistä FastAPI-reiteistä oli webhook, johon Twilio lähettää puhelukutsun saapuessa HTTP POST -pyynnön. Tämä määriteltiin erillisenä reittinä (kuva 10):

```python
@app.api_route("/incoming-call", methods=["GET", "POST"])
async def handle_incoming_call(request: Request):
    response = VoiceResponse()
    host = request.url.hostname
    connect = Connect()
    connect.stream(url=f'wss://{host}/media-stream')
    response.append(connect)
    return HTMLResponse(content=str(response), media_type="application/xml")
```

Kuva 10. Esimerkki endpointista

TwiML (Twilio Markup Language) -vastaus koostettiin joko manuaalisesti XML:nä tai Twilio-kirjaston avulla, riippuen käyttötavasta. Tässä vaiheessa tuli huomioida Twilion vaatimukset, kuten HTTPS-osoite, oikeat otsikot ja vastemuoto.

FastAPI:n automaattisesti generoima Swagger-dokumentaatio osoittautui erityisen hyödylliseksi. Se mahdollisti rajapintojen testaamisen selainkäyttöliittymässä (kuva 11) ja helpotti rajapintojen kehittämistä, kun pyyntöjen ja vastausten rakenteet näkyivät graafisesti.



Kuva 11. Swagger-selainkäyttöliittymä

Tämän avulla testattiin esimerkiksi, mitä parametreja webhook tai äänenkäsittelypalvelu odottaa ja miltä vastauksen JSON-muoto näyttää.

FastAPI mahdollistaa syötteiden automaattisen validoinnin Pydantic-mallien avulla (kuva 12). Tätä käytettiin mm. puheentunnistuksen vastauksissa ja käyttöliittymän tietoturvallisessa käsittelyssä:

```
class TranscriptionResult(BaseModel):
    text: str
    confidence: float
```

Kuva 12. Pydantic-malli

Tämän avulla voitiin suojautua virheellisiltä syötteiltä ja tarjota informatiivisia virheilmoituksia. Lisäksi otettiin käyttöön try/except-lohkoja, jotka lokittivat virheet esimerkiksi Whisperin tai LLM:n vastauksissa.

FastAPI:n reitit toimivat järjestelmän keskeisenä ohjauspisteenä, jonka kautta kulki kaikki sovelluksen sisäinen liikenne. Kun puhelu saapui käyttäjältä, FastAPI vastaanotti Twilion lähettämän webhook-kutsun ja palautti siihen

XML-pohjaisen TwiML-ohjeen, joka määritteli puhelun jatkokäsittelyn. Kun käyttäjä aloitti puhumisen, puhe striimattiin WebSocketin kautta palvelimelle, missä FastAPI ohjasi äänen tallennettavaksi ja lähetti tiedoston Whisper-pohjaiseen puheentunnistuspalveluun. Kun puheesta tuotettu transkriptio saatiin, FastAPI siirsi sen edelleen LLM-mallille, joka muodosti kysymykseen vastaavan tekstin. Lopuksi FastAPI ohjasi tämän vastauksen puhesynteesipalveluun kuten Coqui TTS, ja valmis puhevastaus palautettiin asiakkaalle takaisin puhelun aikana. Tämän kokonaisuuden ansiosta FastAPI toimi koko sovelluksen arkkitehtonisena ytimenä, joka yhdisti kaikki keskeiset komponentit yhdeksi toimivaksi järjestelmäksi.

FastAPI:n käyttöön liittyi myös useita huomionarvoisia teknisiä erityispiirteitä. Koska järjestelmä käsittelee reaaliaikaista puhedataa, oli tärkeää käyttää asynkronisia async def -funktioita ja await-kutsuja sujuvan suorituskyvyn takaamiseksi. Sovelluksen rakenne suunniteltiin modulaariseksi siten, että eri toiminnot, kuten webhookien käsittely, puheentunnistus ja kielimallin kutsuminen, oli eriytetty omiin moduuleihinsa. Tämä paransi koodin ylläpidettävyyttä ja helpotti yksittäisten osien testausta.

Lisäksi Pydantic-kirjaston hyödyntäminen mahdollisti tehokkaan ja luotettavan datan validoinnin erityisesti tilanteissa, joissa järjestelmä vastaanotti epätarkkaa tai rakenteeltaan vaihtelevaa syötettä, kuten puheentunnistuksen tuottamaa tekstiä. Kun sovellusta testattiin esimerkiksi Ngrokin avulla, oli tärkeää varmistaa, että CORS-asetukset sekä HTTPS-viestintä toimivat moitteettomasti. Yksi keskeinen huomio liittyi myös Twilion kanssa käytettävään TwiML-kieleen: pienetkin virheet XML-rakenteessa saattoivat estää koko puhelun käsittelyn, joten TwiML-vastausten oikeellisuus oli aina tarkistettava huolellisesti ennen käyttöönottoa.

4.5 Langchain

Yksi kehittämistyön mielenkiintoisimmista osa-alueista liittyi paikallisesti ajettavan kielimallin älykkyyden laajentamiseen LangChain-kirjaston avulla. Vaikka kielimalli pystyy itsessään vastaamaan monenlaisiin kysymyksiin, sen toimintaa voidaan merkittävästi laajentaa antamalla sille mahdollisuus käyttää ulkoi-

sia toimintoja. Esimerkiksi hakea ajankohtaista säätietoa tai pyytää suorittamaan laskutoimituksia. Tätä varten käyttöön otettiin LangChain, joka on nimenomaan laajojen kielimallien soveltamiseen tarkoitettu kehys.

LangChain tarjoaa rakenteen, jonka avulla voidaan rakentaa niin sanottuja agentteja. Agentti on kielimallin laajennettu versio. Se ei pelkästään muodosta vastauksia annettuun syötteeseen, vaan kykenee myös päättämään milloin ja miten se käyttää erikseen määriteltyjä työkaluja, kuten funktioita tai API-kutsuja. Tämä antaa sovellukselle aivan uudenlaista toiminnallista joustavuutta. Esimerkiksi kun käyttäjä kysyy säätilaa, pelkkä kielimalli tuottaa arvauksen, mutta agentti sen sijaan voi ohjata kysymyksen säätietopalvelulle ja palauttaa oikean vastauksen.

```
from sqlalchemy.orm import Session
from models import Business
from database import get_db

@tool
def get_opening_hours(place: str, db: Session = get_db()) -> str:
    """Fetches opening hours of a business from the database."""
    business = db.query(Business).filter(Business.name == place).first()

    if business and business.opening_hours:
        return f"{place} is open: {business.opening_hours}"
    else:
        return f"Opening hours for {place} could not be found."
```

Kuva 13. Langchain-työkalu, joka hakee aukioloajat tietokannasta

Käytännön toteutuksessa määriteltiin joukko yksinkertaisia työkaluja, joita agentti voi käyttää. Työkalu on käytännössä Python-funktio, jolla on nimi, toiminnallinen kuvaus ja varsinainen toteutus. Nämä työkalut rekisteröitiin LangChainin-agenttirakenteeseen (kuva 13). Tällä tavalla kielimalli sai käyttöönsä keinon ohjata toimintaa ulospäin määritellyissä tilanteissa. Agentin toimintaa ohjataan ohjeistuksella, joka kuvaa mallille missä tilanteissa mitäkin työkalua tulisi käyttää. Tämän ansiosta mallin käyttäytyminen voidaan säätää käyttötarkoitukseen sopivaksi.

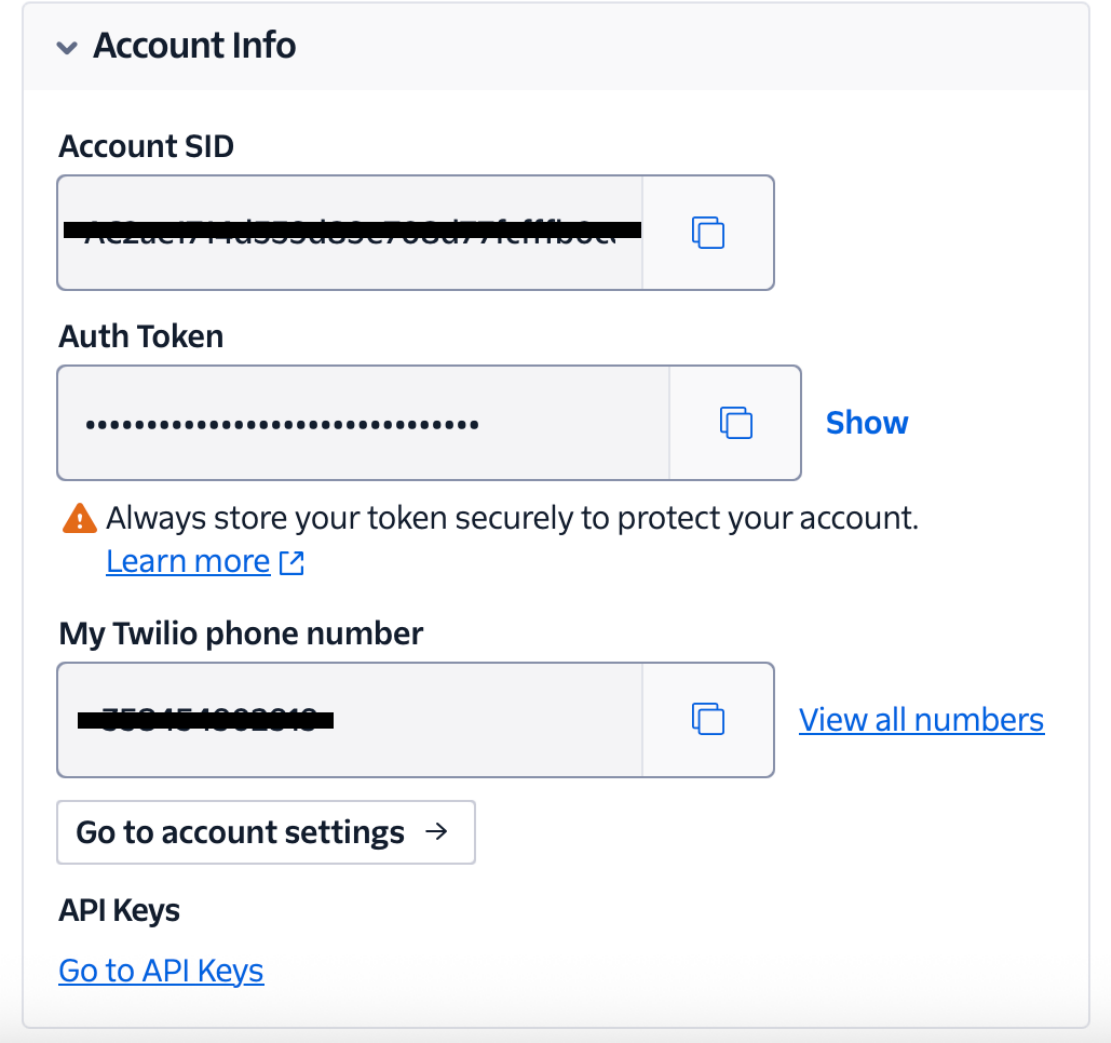
Integraatio paikallisesti ajettavaan kielimalliin tapahtui LangChainin Ollama-luokan kautta. Se mahdollisti agentin ajamisen ilman pilviyhteyksiä. Tällä tavalla säilytettiin järjestelmän tietoturvaperiaatteet ja vähennettiin riippuvuutta ulkopuolisista palveluista.

LangChainin käyttöönotto toi sovellukseen lisää älyä ja käytännön hyötyä. Käyttäjän näkökulmasta agentti toimi luontevasti ja joustavana. Se pystyi muodostamaan vastauksia, jotka perustuivat osittain sen omaan kielimalliosaamiseen ja osittain ulkopuolisesta lähteestä haettuun tietoon. Tämänkaltaista hybridimallia voidaan hyödyntää monissa asiakaspalvelusovelluksissa, joissa tarvitaan sekä luonnollista vuorovaikutusta että tarkkoja ajantasaisia vastauksia.

LangChainin avulla voitiin myös organisoida sovelluksen toimintalogiikkaa selkeämmäksi. Kun eri toiminnot, kuten mallin kutsuminen, työkalujen käyttö ja vastausten palauttaminen eroteltiin loogisiksi osakokonaisuuksiksi, kehitystyö helpottui ja sovelluksen laajentaminen tulevaisuudessa on helpommin hallittavissa. Näin ollen LangChain ei toiminut pelkästään kielimallin kyljessä, vaan koko järjestelmän rakenteellisen ajattelun mahdollistajana.

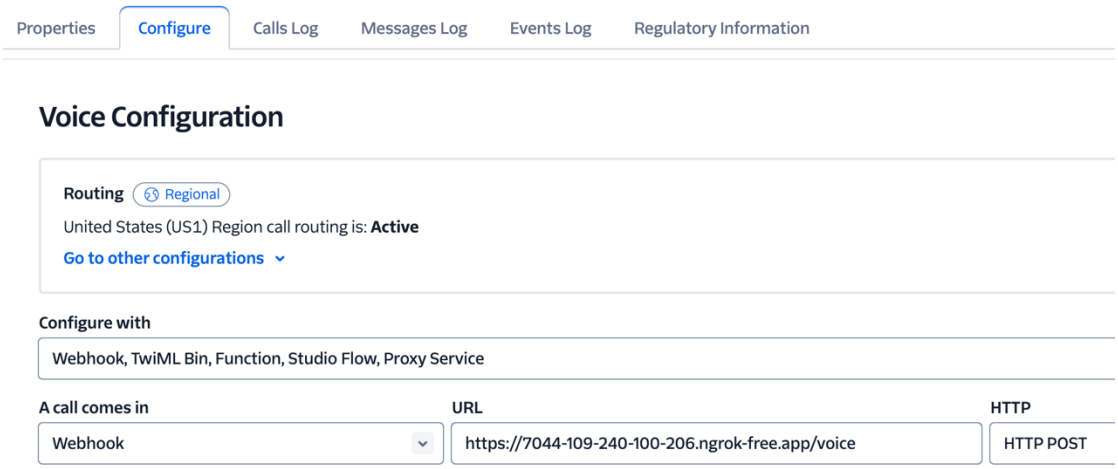
4.6 Twilio

Sovelluksen puhekäyttöliittymän toteuttaminen edellytti teknistä ratkaisua, jonka avulla asiakkaan puhelut voitiin vastaanottaa, ohjata jatkokäsittelyyn ja palauttaa vastaus puheena saman puhelun aikana. Tätä varten valittiin Twilio, laajasti käytetty pilvipohjainen viestintäalusta, joka tarjoaa ohjelmointirajapinnat muun muassa puheluiden ja viestien hallintaan. Twilion hyvä dokumentaatio, testityökalut sekä yhteensopivuus FastAPI:n kanssa helpottivat käyttöönottoa merkittävästi. Twilio mahdollisti, että asiakas voi soittaa sovellukseen tavallisella puhelimella, jonka jälkeen järjestelmä käsittelee ja vastaa puheeseen ohjelmallisesti – ilman ihmiskontaktia.



Kuva 14. Twilio-tilin tiedot

Ensimmäiseksi luotiin kehittäjätili Twilioon (kuva 14) ja aktivoitiin ilmainen testikrediitti. Twilion hallintapaneelista hankittiin virtuaalinen puhelinnumero, johon tulevat puhelut voitiin ohjata sovellukseen. Numero valittiin siten, että se tuki ääniominaisuuksia (voice capabilities) ja oli yhteensopiva webhook-pohjaisen ohjauksen kanssa.



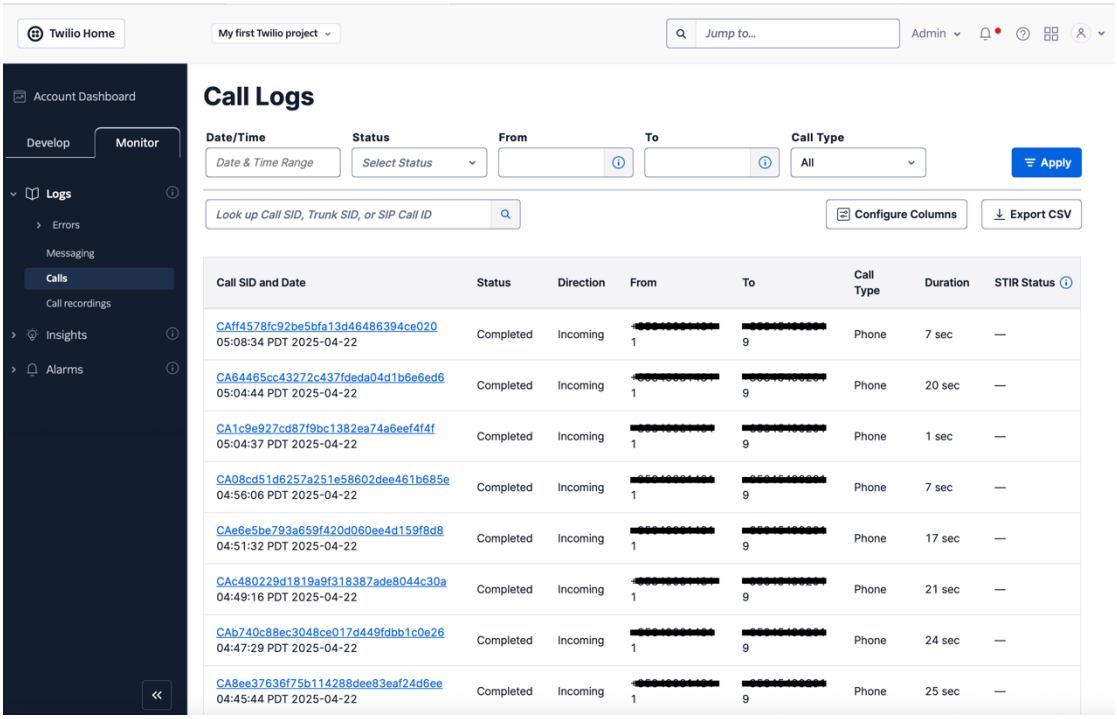
Kuva 15. Twilio-kutsun konfiguraatio

Twilion toimintalogiikka perustuu siihen, että saapuvasta puhelusta syntyy HTTP POST -kutsu, eli webhook, joka toimitetaan määritettyyn URL-osoitteeseen. Tässä projektissa webhookiksi määriteltiin FastAPI-sovelluksen reitti, jossa puhelun logiikka käsiteltiin (kuva 15). FastAPI palautti vastauksena Twilion odottaman XML-pohjaisen ohjeen (TwiML), jossa voitiin esimerkiksi määritellä puheentallennuksen käynnistyminen tai viestin toistaminen asiakkaalle.

Koska FastAPI-sovellus toimi paikallisesti, se ei oletuksena ollut saavutettavissa ulkomaailmasta. Twilio vaatii kuitenkin julkisesti näkyvän HTTPS-osoitteen webhook-kutsuja varten. Tämän vuoksi otettiin käyttöön Ngrok, jonka avulla luotiin väliaikainen julkinen HTTPS-tunneli paikalliseen kehityspalvelimeen. Ngrokin komentorivikäskyllä ngrok http 8000 luotiin osoite, joka liitettiin Twilion webhook-asetuksiin. Tämä mahdollisti sovelluksen testaamisen aidossa puhelintilanteessa ilman tuotantopalvelinta.

Kun asiakas soitti numeroon, Twilio ohjasi puhelun FastAPI-sovellukseen määritellyn webhookin kautta. Sovellus palautti TwiML-vastauksen, jossa määrättiin muun muassa puheen striimaus WebSocketin yli sovellukselle. Tämän jälkeen puhe ohjattiin järjestelmän seuraavaan vaiheeseen: puheentunnistukseen. Kuvassa 16 näkyy Twillon puheluloki.

Twilion <Stream> -komennolla puhe ohjattiin WebSocket-protokollan kautta FastAPI-sovellukselle, joka tallensi äänen äänitiedostoksi. Tämä reaaliaikainen striimaus mahdollisti sen, että puhetta ei tarvinnut tallentaa kokonaisuudessaan etukäteen, vaan se voitiin lähettää ja käsitellä dynaamisesti.



Kuva 16. Twilion loki

Tallennettu äänitiedosto analysoitiin Whisperin avulla, minkä jälkeen transkriptio siirrettiin paikalliselle LLM-mallille, joka muodosti vastauksen. Vastaus muunnettiin takaisin puheeksi Coqui TTS -kirjastolla, ja lopullinen äänivastaus palautettiin asiakkaalle takaisin Twilion kautta osana puhelua. Näin muodostui suljettu, reaaliaikainen vuorovaikutusketju.

4.7 Haasteet ja ratkaisut

Kehittämistyön aikana kohdattiin useita haasteita, jotka liittyivät niin teknisiin yksityiskohtiin kuin eri teknologioiden yhteensovittamiseen. Moni näistä haasteista oli odotettavissa, sillä projekti yhdisti useita vaativia osa-alueita, kuten puheentunnistuksen, kielimallien ohjelmallisen käytön sekä puheluinfrastruktuurin hallinnan. Osa ongelmista puolestaan ilmeni vasta käytännön testaamisen yhteydessä, mikä osoittaa, kuinka tärkeää on edetä kehittämisessä vaiheittain ja jatkuvasti testaten.

Yksi merkittävä haaste liittyi paikallisesti ajettavan kielimallin suorituskykyyn erityisesti ympäristössä, jossa ei ollut käytettävissä grafiikkasuoritinta. Vaikka pienikokoiset mallit toimivat yllättävän hyvin myös pelkällä suorittimella, ha-

vaittiin vasteajoissa vaihtelua etenkin silloin, kun käyttäjän kysymys oli monimutkainen tai pitkä. Tähän reagoitiin optimoimalla mallin syötteiden pituutta ja valitsemalla vastauksiin sellaiset muodot, jotka vaativat mahdollisimman vähän tulkintaa. Lisäksi harkittiin vaihtoehtoisesti mallin suorittamista kevyemmässä konfiguraatiossa tilanteissa, joissa nopeus oli erityisen kriittinen.

Puheentunnistuksen osalta ilmeni, että tunnistustarkkuus vaihteli riippuen puheen selkeydestä, taustahälystä ja äänityksen laadusta. Ongelmia aiheuttivat erityisesti tapaukset, joissa käyttäjän puheessa esiintyi epäselvää artikulointia tai teknisiä termejä, joita ei ollut mallin koulutusdatassa. Ratkaisuksi päätettiin käyttää tarkempaa versiota puheentunnistusmallista ja ohjeistaa käyttäjää puhumaan rauhallisesti ja selkeästi.

Kolmas haaste liittyi Twilion ja paikallisesti toimivan sovelluksen yhteensovittamiseen. Twilio edellyttää, että webhook-osoite on julkisesti saatavilla, eikä suoraan pysty kommunikoimaan paikallisesti ajetun sovelluksen kanssa ilman välikerrosta. Tämä ratkaistiin ottamalla käyttöön Ngrok, joka mahdollisti turvallisen HTTPS-tunnelin luomisen paikalliselle palvelimelle. Ratkaisu toimi hyvin kehitystyössä ja tätä ongelmaa ei ole, kun siirrytään tuotantoympäristöön.

Haasteita syntyi myös äänitiedostojen hallinnassa. Kun käyttäjän puhe tallennettiin palvelimelle käsiteltäväksi, oli tärkeää huolehtia siitä, ettei levytila täyttynyt ja että tiedostot säilöttiin ja myöhemmin poistettiin asianmukaisesti käytön jälkeen. Tähän rakennettiin automatisoitu siivousprosessi, jossa tiedostot poistettiin järjestelmästä ennalta määritetyn ajan jälkeen. Tämä paitsi paransi suorituskykyä, myös tuki tietoturvan näkökulmasta hyvää käytäntöä, kun käyttäjädataa ei säilytetty tarpeettomasti.

Puhesynteesin toteutuksessa jouduttiin tekemään kompromisseja äänen luonnollisuuden ja toteutuksen keveyden välillä. Käyttöön otettu Coqui TTS tarjosi nopean ja kielituetun tavan tuottaa puhetta suomeksi, mutta sen äänenlaatu ja luonnollisuus eivät täysin vastanneet ammattimaisten synteesiratkaisujen tasoa. Työ kuitenkin osoitti, että Coqui TTS riittää prototyyppivaiheessa ja sen käyttö voidaan korvata myöhemmin laadukkaammalla synteesiratkaisulla.

Viimeinen keskeinen haaste liittyi LangChain-kirjaston toimintaan paikallisesti ajettavan kielimallin kanssa. Vaikka LangChain tukee monipuolisesti eri mallityyppejä, suurin osa sen valmiista rakenteista ja esimerkeistä on suunniteltu pilvipalveluiden, kuten OpenAI:n API:en, käyttöön. Tämän seurauksena osa toiminnoista vaati manuaalista säätöä, erityisesti työkalujen (tools) ja agenttien konfiguroinnissa. Lopulta tarvittavat mukautukset saatiin tehtyä ja agenttirakenne saatiin toimimaan suunnitellulla tavalla myös paikallisessa ympäristössä.

Kehitystyöhön liittyvät haasteet olivat luonnollinen osa kokeellista ja soveltavaa ohjelmistokehitystä. Moni ongelma opetti samalla keskeisiä asioita järjestelmäarkkitehtuurista, teknologioiden yhteentoimivuudesta ja siitä, kuinka tärkeää on testata jokainen komponentti käytännössä mahdollisimman varhaisessa vaiheessa. Työn onnistumisen kannalta oli keskeistä, että haasteisiin pystyttiin reagoimaan ketterästi ja että jokainen ratkaisu vei järjestelmää eteenpäin kohti toimivaa kokonaisuutta.

4.8 Tiivistetty ohje

Yksi tämän työn tavoitteista oli tuottaa konkreettinen esimerkki ja lähtökohta myös muiden tahojen käyttöön, jotka ovat kiinnostuneita hyödyntämään laajoja kielimalleja tai rakentamaan puhekäyttöliittymiä paikallisesti ajettavien komponenttien avulla. Koska kehittämistyö perustui avoimen lähdekoodin teknologioihin ja Twilion alustaa lukuunottamatta toteutettiin ilman pilvipalveluriippuvuuksia, se on erityisen hyvin siirrettävissä erilaisiin ympäristöihin ja tarkoituksiin.

Työn perusrakenne koostuu useista itsenäisistä osista, jotka muodostavat kokonaisuuden, jossa asiakas voi puhua puhelimessa, järjestelmä ymmärtää puheen, muodostaa vastauksen kielimallin avulla ja palauttaa sen takaisin puheena. Sovellus on rakennettu Pythonilla, ja sen ympärille liitettiin työkalut kuten FastAPI, Whisper, Ollama ja puhesynteesiratkaisu. Ainoana ulkoisena työkaluna toimi Twilio, jonka avulla käyttäjän puhelu ohjautuu järjestelmään.

Sovelluksen käyttöönotto alkaa teknisen ympäristön pystyttämisestä. Tämä tarkoittaa käytännössä Dockerin ja Pythonin asennusta, virtuaaliympäristön

luomista ja tarvittavien kirjastojen asentamista. Tämän jälkeen ladataan ja käynnistetään kielimalli Ollaman avulla. Mallin voi valita oman käyttötarkoituksen mukaan – esimerkiksi DeepSeek-Coder on kevyt mutta tehokas malli teknisissä kysymyksissä. Sovelluksen rajapinta määritellään FastAPI:n avulla, ja sitä ajetaan paikallisesti esimerkiksi Uvicorn-palvelimella. Puheentunnistus toteutetaan Whisper-mallilla, joka voidaan asentaa suoraan Pythonin kautta, ja puhesynteesi voidaan toteuttaa aluksi vaikkapa Coqui TTS -kirjastolla, joka muuntaa tekstin suomenkieliseksi puheeksi.

Twilion käyttöönotto edellyttää kehittäjätilin luomista ja puhelinnumeron rekisteröintiä, jonka jälkeen voidaan määritellä webhook-osoite, johon puhelut ohjautuvat. Koska paikallisesti ajettava FastAPI-sovellus ei ole näkyvissä ulkomaailmaan, tarvitaan työkalu kuten Ngrok, joka muodostaa väliaikaisen, julkisesti saatavilla olevan HTTPS-tunnelin sovelluksen testikäyttöä varten. Tämä mahdollistaa sen, että puhelut voidaan ohjata paikalliselle palvelimelle ja sovelluksen toiminta voidaan testata aidossa käyttötilanteessa.

Käytännössä käyttäjä, joka haluaa toteuttaa vastaavanlaisen sovelluksen, voi edetä vaiheittain: ensin ympäristö, sitten puheentunnistus, kielimalli ja lopuksi vastauslogiikka ja palautus puhelun kautta. Jokainen osa voidaan rakentaa ja testata itsenäisesti ennen siirtymistä seuraavaan. Kehitystyössä on suositeltavaa testata mahdollisimman pienissä osissa ja hyödyntää runsaasti lokeja ja virheenkäsittelyä, jotta mahdolliset ongelmat voidaan havaita ajoissa.

Tämä käyttöohje toimii siis paitsi ohjeena toteutukseen, myös pohjana omalle sovelluskehitykselle. Koska teknologiat kehittyvät nopeasti, voi osa käytetyistä työkaluista saada jatkossa uusia ominaisuuksia tai parempia vaihtoehtoja. Siksi on suositeltavaa pysyä ajan tasalla sekä kielimallien että puheentunnistuksen ja -synteesin kehityksestä. Silti tämän työn pohjalta rakennettu ratkaisu toimii hyvänä esimerkkinä siitä, kuinka nykyaikaista tekoälyä voidaan hyödyntää käytännön sovelluksessa – kokonaan paikallisesti, tietoturvallisesti ja ilman ulkopuolisia palveluriippuvuuksia.

5 PÄÄTÄNTÖ

Tämän opinnäytetyön tavoitteena oli selvittää, miten laajaa kielimallia (Large Language Model, LLM) voidaan hyödyntää asiakaspalvelusovelluksessa paikallisesti ajettavana versiona ilman riippuvuutta pilvipalveluista. Työssä tutkittiin teoreettista taustaa kielimallien ja erityisesti Transformer-arkkitehtuurin osalta, vertailtiin paikallisten ja pilvipohjaisten LLM-ratkaisujen eroja ja arvioitiin niiden soveltuvuutta asiakaspalvelukäyttöön. Lisäksi kehitettiin ja toteutettiin toimiva prototyyppi, joka mahdollisti puheohjatun asiakaspalvelun lähes kokonaan paikallisessa ympäristössä.

Teoreettisessa osuudessa käytiin läpi laajojen kielimallien kehityshistoriaa, teknisiä perusteita ja käytön erityispiirteitä. Erityistä huomiota kiinnitettiin Transformer-arkkitehtuurin rooliin ja sen tuomiin etuihin verrattuna aikaisempiin neuroverkkoratkaisuihin, kuten RNN- ja CNN-verkkoihin. Lisäksi selvitettiin paikallisesti ajettavien ja pilvipalveluihin perustuvien LLM-ratkaisujen keskeiset erot kustannusten, tietoturvan, skaalautuvuuden ja hallittavuuden näkökulmista.

Käytännön kehitystyössä rakennettiin paikallisesti ajettava asiakaspalvelusovellus, joka yhdisti puheentunnistuksen, kielimallin, puhesynteesin ja puhelinrajapinnan saumattomaksi kokonaisuudeksi. Projektissa hyödynnettiin laajasti avoimen lähdekoodin teknologioita, kuten Python-ohjelmointikieltä, FastAPI-kehystä, Whisper-puheentunnistusmallia, Ollama-alustaa kielimallin ajamiseen, LangChain-kirjastoa agenttirakenteiden toteuttamiseen, Coqui TTS -puhesynteesiä ja Twilio-alustaa puheluiden hallintaan. Kehitystyön eteneminen osoitti, että näiden teknologioiden avulla voidaan rakentaa joustava ja suorituskykyinen sovellus, joka täyttää nykyaikaisen asiakaspalvelun vaatimukset.

Projektin aikana kohdattiin lukuisia haasteita, kuten suorituskykyongelmia CPU-pohjaisessa ympäristössä, puheentunnistuksen tarkkuuden vaihtelua sekä haasteita paikallisen ja ulkoisen viestintäinfrastruktuurin yhteensovittamisessa. Nämä haasteet ratkaistiin teknisin keinoin, kuten valitsemalla kevyempiä kielimalleja, optimoimalla puheentunnistusasetuksia, ja hyödyntämällä Ngrok-työkalua turvallisten yhteyksien luomiseen kehitysvaiheessa. Haasteet

ja niiden ratkaisut toivat esiin sen, kuinka tärkeää on suunnitella järjestelmä modulaariseksi ja testata jokaista komponenttia iteratiivisesti.

Opinnäytetyön keskeinen johtopäätös on, että paikallisesti ajettavat LLM-ratkaisut tarjoavat merkittäviä etuja tietyissä käyttötilanteissa verrattuna pilvipohjaisiin palveluihin. Erityisesti silloin, kun tietoturva, yksityisyyden suoja ja kustannusten hallinta ovat keskeisiä prioriteetteja, paikallinen ratkaisu voi olla ylivoimainen vaihtoehto. Vaikka paikallisen ratkaisun alkuinvestoinnit voivat olla korkeammat ja ylläpito vaatii enemmän teknistä osaamista, pitkällä aikavälillä saavutettavat hyödyt — kuten datan täysi hallinta ja ennakoitavat kustannukset — tekevät siitä houkuttelevan vaihtoehdon monille organisaatioille.

Kehitystyön tuloksena syntynyt sovellus toimii myös lähtökohtana jatkokehitykselle. Tulevaisuudessa sovellusta voidaan laajentaa esimerkiksi integroimalla siihen useampia kielimalleja, tukemalla useampia kieliä, parantamalla puhesynteesin laatua tai hyödyntämällä agenttipohjaisia rakenteita entistä älykkäämpään vuorovaikutukseen. Myös kielimallien jatkokoulutus (fine-tuning) asiakaskohtaisten aineistojen avulla tarjoaa mielenkiintoisen mahdollisuuden palvelun laadun ja relevanssin parantamiseen.

Opinnäytetyö osoittaa, että tekoälyratkaisujen kehittäminen paikallisesti ei ole ainoastaan mahdollista, vaan myös käytännöllistä nykyaikaisessa sovelluskehityksessä. Se tarjoaa vaihtoehdon nykyiseen vahvasti pilvipalveluvetoiseen kehityssuuntaan ja antaa mahdollisuuden rakentaa kestävämpiä, hallittavampia ja tietoturvallisempia järjestelmiä. Tämän työn kokemukset ja ratkaisut voivat toimia esimerkkinä muille kehittäjille ja organisaatioille, jotka etsivät vaihtoehtoja pilvipalveluriippuvuudelle tekoälysovellusten toteutuksessa.

LÄHTEET

Alka, L., Mahesh, R. & Chattopadhyay, A. 2024. Privacy and Security Implications of Cloud-Based AI Services: A Survey. PDF-dokumentti. Saatavissa: https://arxiv.org/abs/2402.00896 [viitattu 2.5.2025].

Antematter. 2025. Scaling Large Language Models: Effective Strategies for Cost and Performance. WWW-dokumentti. Saatavissa: https://antematter.io/blogs/llm-scalability [viitattu 4.5.2025].

Bommasani, R., Hudson, D.A., Adeli, E., et al. 2021. On the Opportunities and Risks of Foundation Models. Stanford Center for Research on Foundation Models. PDF-dokumentti. Saatavissa: https://arxiv.org/abs/2108.07258 [viitattu 3.5.2025].

Dell Technologies. 2024. Understanding the Total Cost of Inferencing Large Language Models. ESG Analyst Paper. PDF-dokumentti. Saatavissa: https://www.delltechnologies.com/asset/en-in/solutions/business-solutions/industry-market/esg-inferencing-on-premises-with-dell-technologies-analyst-paper.pdf [viitattu 4.5.2025].

Frontiers in Big Data. 2020. Securing Machine Learning in the Cloud: A Systematic Review of Cloud Machine Learning Security. Frontiers in Big Data 3, 587139. WWW-dokumentti. Saatavissa: https://www.frontiersin.org/articles/10.3389/fdata.2020.587139/full [viitattu 2.5.2025].

MonsterAPI. 2025. Cloud vs. On-Premises: Choosing the Best Deployment Option for LLMs. WWW-dokumentti. Saatavissa: https://blog.monsterapi.ai/cloud-vs-on-premises-hosting/ [viitattu 4.5.2025].

Naveed, H., Khan, A.U., Qiu, S., Saqib, M., Anwar, S., Usman, M., Akhtar, N., Barnes, N. & Mian, A. 2024. *A comprehensive overview of large language models*. PDF-dokumentti. Saatavissa: https://arxiv.org/abs/2307.06435v10 [viitattu 29.4.2025].

Principled Technologies. 2024. A Cost-Benefit Analysis of Dell On-Premises vs. AWS and Azure Deployments. WWW-dokumentti. Saatavissa: https://www.principledtechnologies.com/A-cost-benefit-analysis-of-Dell-on-premises-vs-AWS-and-Azure%E2%80%AF-deployments [viitattu 3.5.2025].

Wang, C., Li, M. & Smola, A. J. 2019. Comparative analysis of on-premises and cloud hosting solutions. *DiVA Portal*. PDF-dokumentti. Saatavissa: https://www.diva-portal.org/smash/get/diva2%3A1887940/FULLTEXT01.pdf [viitattu 30.4.2025].