Universiteit Leiden

Master Computer Science

Social Robotics System for Senior Care Using Large Language Models and Raspberry Pi

Name:	Van Nguyen
Student ID:	s3726266
Date:	September 25, 2025
Specialisation:	Computer Science - Data Science
1st supervisor:	Prof.dr. Marco R. Spruit
2nd supervisor:	Dr. B.M.A. van Dijk MSc

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Declaration of Authorship

I, Van NGUYEN, declare that this thesis titled, "Social Robotics System for Senior Care Using Large Language Models and Raspberry Pi" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: 26-09-2025

Date: [illegible]

“One must imagine Sisyphus happy”

Albert Camus

LEIDEN UNIVERSITY

Abstract

The Leiden Institute of Advanced Computer Science
Faculty of Science

Master of Science

Social Robotics System for Senior Care Using Large Language Models and Raspberry Pi

by Van NGUYEN

The rapid growth of aging populations places increasing pressure on healthcare systems, especially in nursing homes facing staff shortages. Socially assistive technologies can help, but existing solutions often depend on costly hardware and cloud services, raising concerns about scalability, latency, and most importantly data privacy.

This thesis presents the design and evaluation of a modular, privacy-conscious social robotics system that enables real-time voice interaction for older adults. A Raspberry Pi client captures speech and plays synthesized responses, while a local server hosts speech-to-text (FastWhisper), large language model inference (Mistral via Ollama), and text-to-speech synthesis (Coqui-TTS). All components are containerized for reproducibility and flexible substitution.

Evaluation under local network conditions demonstrated median end-to-end latencies of 5.9 s (short queries) and 7.3 s (medium queries), with 95th-percentile values under 10 s. The system achieved a 98.2% success rate across interactions. A personalization case study showed high recall but low precision (F1 = 0.54), indicating feasibility with room for refinement. Resource usage remained moderate, and packet capture verified that all communication stayed within the local network.

The results demonstrate that open-source, local-first conversational AI can provide reliable and ethically aligned voice assistance in senior care, while highlighting key areas for optimization.

Acknowledgements

Firstly, I would like to express my sincere gratitude to my supervisors, Prof. dr. Marco R. Spruit and Dr. Bram van Dijk, for granting me the opportunity and the freedom to explore this topic. Their guidance, encouragement, and invaluable advice were essential throughout the course of this work.

To my brother, who encouraged me to go above and beyond and to never settle, and to my parents, who ventured into unknown lands to provide for their children: I stand on the foundation of their hard work, sacrifices, and unrelenting belief in their son. This accomplishment is as much theirs, if not more, than it is mine.

Contents

List of Figures

List of Tables

List of Abbreviations

AI	**A**rtificial **I**ntelligence
ASR	**A**utomatic **S**peech **R**ecognition
CPU	**C**entral **P**rocessing **U**nit
E2E	**E**nd-to-**E**nd
GPU	**G**raphics **P**rocessing **U**nit
LAN	**L**ocal **A**rea **N**etwork
LLM	**L**arge **L**anguage **M**odel
NLP	**N**atural **L**anguage **P**rocessing
RAM	**R**andom **A**ccess **M**emory
SAR	**S**ocially **A**ssistive **R**obot
STT	**S**peech-**t**o-**T**ext
TTS	**T**ext-**t**o-**S**peech
WS	**W**eb**S**ocket
GDPR	**G**eneral **D**ata **P**rotection **R**egulation
DB	**D**ata**b**ase
ORM	**O**bject-**R**elational **M**apping
JSON	**J**avaScript **O**bject **N**otation
API	**A**pplication **P**rogramming **I**nterface
URL	**U**niform **R**esource **L**ocator

Chapter 1

Introduction

1.1 Introduction

The global demographic landscape is undergoing a rapid transformation due to the aging population. According to the World Health Organization (WHO, 2022), "by 2030, one in six people in the world will be aged 60 years or over," increasing from 1 billion in 2020 to 1.4 billion, and expected to double to 2.1 billion by 2050. This demographic shift raises significant pressure on existing healthcare and social support systems, especially in nursing home facilities already facing workforce shortages and capacity limitations (Sapci and Sapci, 2019).

Beyond the physical and logistical challenges, older individuals are increasingly vulnerable to social isolation, cognitive decline, and emotional neglect factors heighten by limited human interaction and shortage of caregivers. These shifts highlight the urgent need for scalable, cost effective, and empathetic technological interventions that can supplement human care and promote mental and emotional well-being among the senior citizens.

Therefore, Conversational Artificial Intelligence (AI) has emerged as a promising solution. Systems powered by Large Language Models (LLMs) offer capabilities in natural language understanding, response generation, and adaptive dialogue management. When combined with voice interfaces, such system can support social robotics applications designed to function as virtual companions capable of fluent conversations, and even identifying signs of emotional or cognitive distress based on interaction patterns.

However, deploying such systems in real-world care for older people contexts presents some challenges: they must be responsive, personalized, non-intrusive, and maintain a high level of privacy and data security. Furthermore, they must be deployable on low-power edge devices to ensure accessibility and cost-effectiveness.

1.2 Problem statement

Despite rapid advancements in LLMs and voice-based interaction systems, conversational AI remains underutilized in care for older people. Research by Miner et al. (Miner et al., 2016) emphasizes the critical need for "intelligent assistive technologies" in aging societies. Especially those that can provide remote monitoring, context-aware interaction, and emotional support (Fitzpatrick, Darcy, and Vierhile, 2017). Current commercial solutions (e.g., Alexa, Google Assistant) are often not suitable for older users. They lack personalization and most importantly raise concerns about data privacy and surveillance (Floridi, Cowls, et al., 2018, Voigt and Bussche, 2017).

Additionally, existing social robots often rely on expensive hardware and cloud-based processing alongside having limiting scalability and introduce latency and privacy risks. Few systems implement a closed-loop conversational pipeline—integrating Speech-to-Text (STT), LLM-based reasoning, Text-to-Speech (TTS), and context tracking—in a resource-aware, modular, and adaptable architecture.

Figure 1.1 (Sapci and Sapci, 2019) summarizes key dimensions used in evaluating assistive technologies in care homes. These include privacy, usability, context-awareness, and robotic and AI-based approaches. The proposed system addresses multiple axes at the same time. Most notably Robotic Technologies, AI Assistive Technologies, Privacy Consideration, and Context-Aware Frameworks—positioning it as a multi-dimensional solution aligned with current research priorities in gerontechnology.

This project addresses these limitations by proposing a fully modular social robotics system tailored for nursing homes or other older people care facilities. By using open-source tools and deployable on affordable hardware. The system enables real-time, voice-based interaction via a Raspberry Pi client, supported by server-side STT, LLM, and TTS modules, coordinated via a WebSocket-based pipeline. Unlike cloud-dependent alternatives, the system focuses on local processing, logging for personalization, and low-latency interaction.

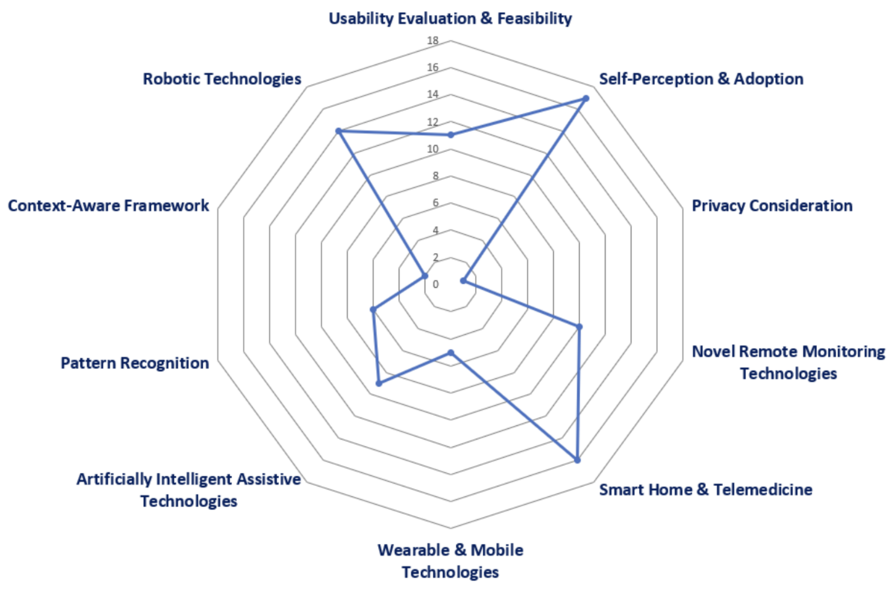


FIGURE 1.1: Usability evaluation and feasibility

1.3 Objectives

This thesis proposes the design and development of scalable and privacy-conscious robotics system for older people care facilities leveraging:

1. **Deploy a lightweight client**: Use a Raspberry Pi to capture voice input and play back synthesized responses.
2. **Implement real-time conversational pipeline**: Leverage server-side modules for STT (FastWhisper), LLM-based reasoning (via Ollama), and TTS.
3. **Log interactions for personalization**: Store STT transcriptions and LLM replies in a PostgreSQL database to enable future personalization and analysis of usage patterns.
4. **Ensure scalable communication**: Use FastAPI with WebSockets to stream audio data bi-directionally with minimal latency.
5. **Enable offline or semi-offline deployment**: Use open-source models and tools to support deployment in resource-limited or privacy-sensitive environments.

1.4 Contributions

This thesis makes both technical and conceptual contributions to the fields of conversational AI and social robotics with the intention of attending older people.

1.4.1 Real-Time Voice Interaction System for older Users

A complete real-time voice interaction pipeline was designed and implemented to enable low-latency, natural voice communication between older users and an LLM-based system, using lightweight client hardware and modular server-side services. The pipeline integrates:

1. FastAPI WebSocket communication for bi-directional audio streaming.
2. Fast Whisper for speech-to-text transcription.
3. Local inference of large language models using the Ollama runtime environment.
4. TTS for voice synthesis
5. Full orchestration via Docker Compose

This contribution demonstrates the feasibility of deploying real-time conversational AI with edge devices and open-source models, overcoming typical limitations of cloud dependency and resource usage.

1.4.2 Modular and Open-Source Architecture for Conversational AI

The system architecture follows a microservice-oriented design, allowing individual components (STT, LLM, TTS) to be developed, tested, and replaced independently. This modular architecture supports rapid experimentation with different models and simplifies long-term maintenance of the system. This modularity enables:

1. Easy substitution of models (e.g., swapping TTS/STT engines/models).
2. Independent scaling of services.
3. Simplified debugging and evaluation.

All core components are containerized, promoting reproducibility and ease of deployment across platforms.

1.4.3 Real-Time Voice AI with Local Processing and No Cloud Reliance

Unlike commercial voice assistants that require cloud access and raise privacy concerns, this system is designed with data minimization and privacy-by-design principles. By hosting LLM and TTS components locally and limiting external communication, the architecture supports:

1. On-premise or LAN-based deployment (especially useful in nursing homes or private residences),
2. Full control over data retention, supporting compliance with ethical standards in context of taking care of older people.

1.5 Research questions

1. How can a modular, privacy-conscious voice assistant using open-source LLMs and edge devices be designed and deployed to support social interaction and well-being in nursing homes?
2. How effective is a Raspberry Pi–based voice interface, integrated with a server-hosted LLM pipeline, in enabling natural, responsive, and secure conversation for socially assistive applications?

Chapter 2

Literature Review

2.1 Introduction to Literature Review

The aim of this research on literature is to reflect on past studies in all directions: caregiving social robots for elders, dialog artificial intelligence, speech processing technology and implementation centered on privacy. Of relevance is identification of prior capabilities and limitations by current care-supported senior citizen systems using natural language interactions. The review depicts significance of recent breakthroughs by Large Language Models (LLMs) and real-time voice processing.

2.2 Social robotics for taking care of older people

The aging global population presents growing challenges for healthcare systems and social support infrastructures worldwide. Senior individuals are especially vulnerable to cognitive decline, social isolation, and limited access to personalized care. Especially as caregiver-to-patient ratios decline. In response, socially assistive robots (SARs) have been proposed as a technological solution to supplement human care and enhance emotional well-being.

Several commercial examples of SARs have entered the market, including *PARO*, a therapeutic seal robot designed for emotional support, *ElliQ*, a semi-autonomous tabletop companion that engages users in conversation and reminders, and *Pepper*, a humanoid robot developed for broader human-robot interaction contexts. These systems aim to provide companionship, cognitive stimulation, and health-related support through spoken dialogue and interactive behaviors.

These systems, while promising, face limitations in terms of accessibility and deployment models. For example, ElliQ requires an upfront payment and subscription (*ElliQ* n.d.), which may be prohibitive in many healthcare contexts. Most of these platforms rely heavily on cloud-based infrastructure, which raises concerns about data privacy, offline accessibility, and reliability in low-connectivity environments such as residential care facilities. These challenges do not diminish their value in offering companionship or stimulation, but they demonstrate that current SARs are often optimized for user experience rather than modularity, openness, or local deployment (Sapci and Sapci, 2019).

Beyond commercial systems, academic prototypes have also been explored. Conversational agents with focus for older people. For example healthcare-oriented chatbots with features such as adjustable text-size, voice control, and appointment scheduling (e.g., Khamaj, 2025), or frameworks such as *EchoVoice* (Xu et al., 2025), which adapt to seniors' unique vocal patterns through age-aware speech synthesis and persona-consistent memory management. These projects highlight the research community's emphasis on inclusivity and accessibility but they typically remain domain-specific prototypes rather than deployable platforms.

This thesis takes inspiration from both commercial and academic systems but pursues a different focus.Rather than offering new forms of embodiment or therapeutic companionship, it investigates how open-source, modular, and privacy-preserving backends can support conversational agents for senior care. This approach complements existing efforts by addressing technical foundations that could make future SARs more affordable, adaptable, and deployable in privacy-sensitive environments.

2.3 Conversational AI systems in Assistive Contexts

Conversational AI has advanced rapidly through the development of transformer-based Large Language Models (LLMs) such as GPT-3, LLaMA, Bloom, and Mistral. These models offer fluent and contextually rich responses and can emulate social conversation patterns. Systems like *Woebot Health* (Fitzpatrick, Darcy, and Vierhile, 2017) and *Ada Health* (Miner et al., 2016) demonstrate how conversational agents can be leveraged for mental health monitoring and preliminary diagnosis.

However, commercial conversational AI systems typically fall short in certain aspects when considered for senior care:

1. They rely on cloud infrastructure, limiting their suitability in privacy-sensitive contexts.

2. Their personalization mechanisms are often opaque or limited.

3. Few support offline or LAN-based deployments, which are valuable in resource-constrained environments such as care homes.

Academic chatbot prototypes tailored for seniors partly address these issues, for example by focusing on speech accessibility or cognitive support. Yet they are usually not designed for scalability or maintainability, and rarely function beyond small pilot studies. This thesis addresses a complementary gap: the integration of open-source LLMs (e.g., Mistral via Ollama) with a modular pipeline for end-to-end voice interaction. Emphasizing local deployment, reproducibility, and privacy-conscious design.

2.4 Speech Technologies for Natural Interaction

Natural voice interaction is central to the usability and perceived intelligence of conversational agents. This section reviews the state of the art in speech-to-text (STT) and text-to-speech (TTS) technologies, with emphasis on their applicability in real-time, privacy-conscious, and resource-constrained systems for senior care.

2.4.1 Speech-to-Text (STT)

Accurate and low-latency transcription of spoken language is essential for enabling natural dialogue with voice-based agents. Traditional STT systems such as *CMU Sphinx* and the *Google Speech API* laid foundational work in this area, but their limitations in terms of model openness, privacy, and offline usability make them suboptimal for use for older people.

Recent advances have produced more capable open-source models. For example, **Whisper** by OpenAI demonstrates state-of-the-art performance in multilingual and noisy environments, making it robust for real-world speech input (Radford et al., 2022). However, Whisper's computational demands and GPU reliance make it impractical for real-time interaction on low-power devices.

To address these constraints, **FastWhisper** offers a quantized and latency-optimized variant, enabling inference on CPU-only systems without significant degradation in recognition quality. Which makes it highly suitable for LAN-based or edge deployments, where privacy and responsiveness are critical, especially in environments like care homes with limited computational resources.

2.4.2 Text-to-Speech (TTS)

The synthesis of natural, intelligible, and expressive speech is equally critical for user engagement, especially for senior users who may have hearing impairments or cognitive challenges that affect speech comprehension (Choi and Agichtein, 2020). TTS systems must therefore balance *clarity*, *expressiveness*, and *latency*.

Coqui-TTS (**CoquiTTS**), an open-source speech synthesis toolkit, fulfills these needs by offering customizable voice models, real-time synthesis, and support for local deployment. Unlike cloud-dependent services such as *Amazon Polly* or *Google Cloud TTS*, Coqui enables on-device inference, preserving user privacy and ensuring responsiveness in offline scenarios.

A research by Choi and Agichtein (Choi and Agichtein, 2020) demonstrates that variations in *intonation* and *speech pacing* significantly influence user satisfaction and perceived understanding in conversational systems. These findings highlight the importance of not only accurate but also emotionally resonant synthetic speech. Coqui-TTS supports such speech cadence control through fine-tuned models, aligning well with the system requirements of this thesis.

By adopting FastWhisper for STT and Coqui-TTS for synthesis, this thesis project combines open, efficient, and privacy conscious speech technologies that support natural interaction between senior user and the system.

2.5 Comparative Analysis of Related Systems and Technologies

To highlight the limitations in current socially assistive technologies and their underlying voice processing components, this section provides two comparative tables. The first summarizes common socially assistive robots and conversational agents in terms of their target audience, embodiment, and architectural properties. The second focuses on speech technology components relevant to natural language interaction in monitoring of older people.

2.5.1 Socially Assistive Systems

System	Older Adult Focused	Embodied	Cloud-Free	Personalization	Real-Time
Woebot Health	Yes	No	No	Yes	No
ElliQ	Yes	Yes	No	Limited	No
Pepper (Softbank)	Yes	Yes	No	Limited	Yes
PARO	Yes	Yes	Yes	No	No
Ada Health	Partially	No	No	No	No
Khamaj et al.	Yes	No	Partial	Yes	No
EchoVoices	Yes	No	No	Yes	No

TABLE 2.1: Comparison of Socially Assistive Robots and Academic Chatbots for Older Adults.

2.5.2 Speech Technologies for Voice Interaction

Tool	Open Source	Offline Capable	Real-Time	Edge Deployable
Whisper (Radford et al., 2022)	Yes	Partially	No	No
FastWhisper (Klein, 2023)	Yes	Yes	Yes	Yes
Coqui-TTS	Yes	Yes	Yes	Yes
Google TTS	No	No	Yes	No
Amazon Polly	No	No	Yes	No

TABLE 2.2: Comparison of Speech Technology Tools

As shown in Table 2.1, many socially assistive robots and conversational agents targeted at senior users are cloud-dependent, offer limited personalization, and often lack real-time interaction capabilities. While some systems provide physical embodiment to enhance social presence (e.g., PARO, Pepper) (Shibata and Wada, 2011), they tend to be costly, inflexible, and unsuitable for scalable deployment in privacy sensitive environments such as residential care homes.

Table 2.2 highlights the capabilities and limitations of current speech technologies. Unlike commercial solutions like Google TTS and Amazon Polly which are dependant on cloud services, open source tools like FastWhisper and Coqui are available offline and offer real-time voice interaction. These findings reinforce the need for integrated, privacy conscious, and modular systems that combine the strengths of modern speech technologies with deployment models suited for taking care of older people.

Chapter 3

System Design and Methodology

3.1 Introduction

This chapter describes the system's overall architecture, design choices made, and the technologies selected to meet the functional, technical, and ethical requirements established in previous chapters. It also outlines the interaction flow and modularity that enable the system's scalability, privacy, and real-time performance.

3.2 Deployment and Architecture considerations

Healthcare and support for senior citizens applications demand strict adherence to data minimization, user control, and offline capability. Especially in compliance with regulations such as the General Data Protection Regulation (GDPR) and broader ethical frameworks (Voigt and Bussche, 2017). These requirements are often incompatible with cloud-based AI services, which raise concerns about privacy, surveillance, and loss of user autonomy.

To address these challenges, edge computing and on-device inference have gained prominence. Devices such as the Raspberry Pi offer sufficient computational power to perform tasks like local audio capture, real-time streaming to server-side modules, and speech playback while being energy efficient and cost effective hardware platform.

Recent studies such as the work by Husom et al. (Husom et al., 2025) demonstrate that with careful model selection, quantization, and resource management, it is increasingly feasible to run LLM inference and TTS locally on such constrained devices. These findings highlight the potential for fully offline, self-contained conversational agents in the near future.

While this thesis does not deploy the full AI pipeline directly on the Raspberry Pi, it validates a hybrid architecture where only audio processing is handled on the client, while heavier computation is offloaded to a local server. This design demonstrates a realistic and modular path toward full on-device deployment. While keeping in mind that future optimizations may further reduce hardware dependencies and system cost without sacrificing performance or privacy.

Most academic prototypes in conversational AI are not designed for deployability or maintainability. This thesis distinguishes itself by using a microservice architecture built around Docker Compose, where STT, LLM, and TTS components are containerized independently. This allows:

1. Rapid replacement of components (e.g., swap Coqui-TTS for Piper)
2. Parallel scaling (e.g., multiple LLM backends)
3. Simple startup via orchestration and environment configuration

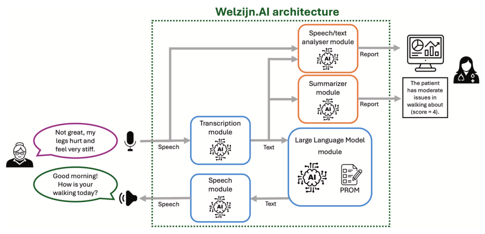

Welzijn.AI architecture
Speech/text analyser module
Summarizer module
Report
Report
The patient has moderate issues in walking about (score = 4).
Not great, my legs hurt and feel very stiff.
Good morning! How is your walking today?
Speech
Transcription module
Text
Large Language Model module
PROM
Speech module
Text
Speech

FIGURE 3.1: Welzijn modules and interaction graph

Furthermore, WebSocket communication between client and server allows low latency streaming, crucial for natural interaction. Compared to REST APIs, WebSockets are better suited for bidirectional audio data exchange, reducing overhead and enabling pipelined processing.

Although prior projects have explored WebSockets for real-time chat, few combine it with live STT-TTS-LLM cycles. This architecture is designed specifically to bridge the real-time constraints of speech interfaces with modular AI pipelines.

3.3 Overview of System Architecture

The proposed system is built with a focus on modularity, scalability and maintainability with main voice based interaction focus on older people and their well being. It is designed to support real-time natural language interaction using local first deployment with possible extensibility for live deployment. The architecture is divided into two main layers:

- A lightweight Raspberry Pi client, responsible for voice capture, wake word detection, and audio playback
- A server-side microservice pipeline, responsible for processing audio, generating intelligent responses via a Large Language Model (LLM), and synthesizing speech output

3.4 Project inception and explored system iterations

The project is extension of system described in paper called Welzijn.AI: Developing responsible conversational AI for the care of older people through stakeholder involvement (Dijk, Lefebvre, and Spruit, 2025. In the project the authors have proposed a well being system powered by LLM. The interaction are done via chat based interface and offers graph monitoring several aspects of user's well being.

The Figure 3.1 acted as point of inception for this project. However this model architecture needed some modifications to fit this project. Mainly the separation of project into aforementioned two layers.

3.4.1 Iteration I: Matrix Protocol-Based Design

System Architecture

The first version of the system explored the use of the Matrix protocol to support secure and decentralized communication between the user and the assistant. Matrix provides end-to-end encryption, room-based access control, and a federated model, making it attractive for privacy-sensitive applications. The open-source Element client was adopted as the user-facing interface.

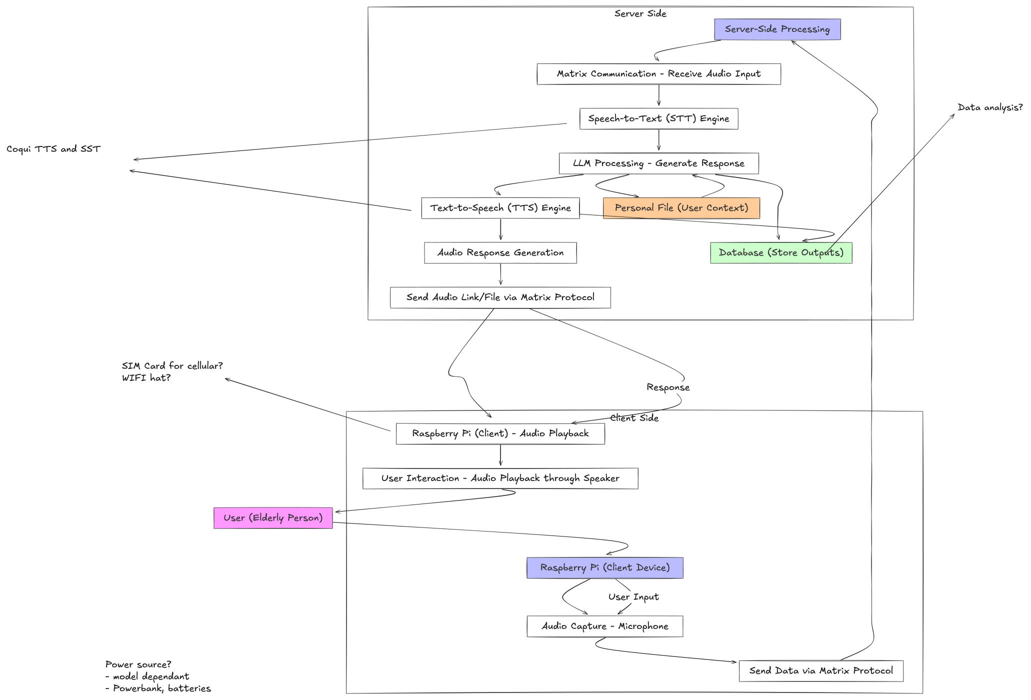
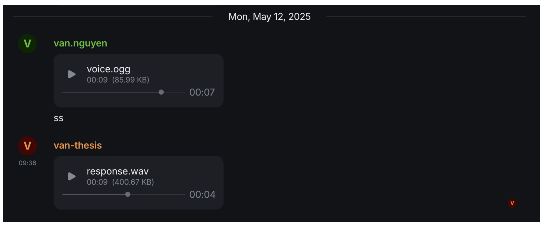

Server Side
Server-Side Processing
Matrix Communication - Receive Audio Input
Speech-to-Text (STT) Engine
Data analysis?
Coqui TTS and SST
LLM Processing - Generate Response
Text-to-Speech (TTS) Engine
Personal File (User Context)
Audio Response Generation
Database (Store Outputs)
Send Audio Link/File via Matrix Protocol
SIM Card for cellular?
WIFI hat?
Response
Client Side
Raspberry Pi (Client) - Audio Playback
User Interaction - Audio Playback through Speaker
User (Elderly Person)
Raspberry Pi (Client Device)
User Input
Audio Capture - Microphone
Send Data via Matrix Protocol
Power source?
- model dependant
- Powerbank, batteries

FIGURE 3.2: Preliminary system design using Matrix protocol

As illustrated in Figure 3.2, users interacted with the system by recording and sending audio messages within a Matrix room. A bot, implemented using the `matrix-nio` library, continuously monitored the room for new events. Upon receiving an audio message, the bot performed the following sequence:

1. Download and decrypt the audio message.
2. Forward the audio to a local Speech-to-Text (STT) service (e.g., Vosk or Fast-Whisper).
3. Pass the transcription to the Large Language Model (LLM) for generating a reply.
4. Synthesize the reply into speech using Coqui-TTS.
5. Upload and return the synthesized response as a Matrix voice message.



Mon, May 12, 2025

van.nguyen
voice.ogg
00:09 (85.99 KB)
00:07

ss

van-thesis
09:36
response.wav
00:09 (400.67 KB)
00:04

FIGURE 3.3: Matrix Element client interface used for user-bot interaction

Role of Iteration I

This architecture was one early proof-of-concept that demonstrated one could assemble an entire conversational pipeline—the STT, LLM, and TTS—in one's own private messaging environment. While it was usable, it revealed some design trade-offs that informed the next iteration. Rather than proceeding along the Matrix-based path, the project advanced to a streaming architecture described in later subsection 3.4.2.

3.4.2 Iteration II – WebSocket-Based Real-Time Pipeline

System Architecture

The second and final version of the system adopts a streaming architecture implemented using **FastAPI WebSockets**. This design separates responsibilities between a lightweight client (Raspberry Pi) and a server hosting modular STT, LLM, and TTS services. The architecture is shown in Figure 3.4.

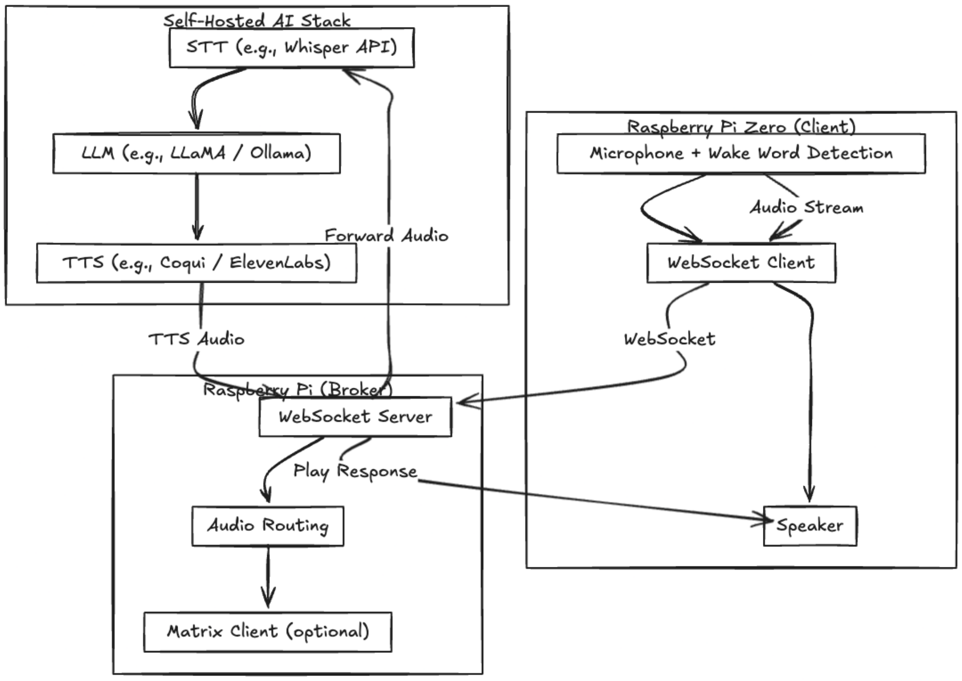


FIGURE 3.4: System architecture using FastAPI and WebSockets

Client Responsibilities

The Raspberry Pi client handles the following tasks:

- **Wake word detection** using Porcupine (Picovoice, n.d.))
- **Voice recording** via a USB microphone (3–60s duration)
- **Streaming** of recorded audio to the server via WebSocket
- **Playback** of synthesized voice responses through USB speakers

Server Responsibilities

The server exposes a WebSocket endpoint for full-duplex audio streaming and coordinates the following services:

1. **STT:** FastWhisper transcribes incoming audio chunks into text.
2. **LLM:** Mistral (served via Ollama) generates context-aware responses.
3. **TTS:** Coqui-TTS synthesizes responses into WAV format.
4. **Database:** PostgreSQL stores STT transcriptions and LLM responses for future personalization.

Each service runs as an independent Docker container, with inter service communication handled via an internal Docker network. The WebSocket server also performs input validation and error filtering (e.g., ignoring prompts below a minimum length).

Interaction Flow

The system interaction cycle is as follows:

1. The user activates the system with a wake word on the Raspberry Pi.
2. Audio is streamed to the FastAPI WebSocket server.
3. Transcription, LLM inference, and speech synthesis are executed sequentially.
4. The synthesized speech is streamed back and played on the client device.
5. Transcriptions and generated responses are logged with timestamps in the database.

Design Rationale

This architecture was chosen to address limitations identified in the preliminary Matrix based design. WebSockets allow continuous, low overhead audio streaming, while containerized microservices provide modularity and scalability. The separation between lightweight client processing and server-side inference also supports privacy preserving deployment in local networks. A detailed comparison between Iteration I and Iteration II, including performance characteristics and user suitability, is presented in Chapter 5.

3.5 Evaluation Plan

To assess whether the proposed system meets the requirements of real-time responsiveness, deployability, and privacy conscious design, a structured evaluation plan was defined. The evaluation focuses on system-level metrics that directly follow from the research objectives rather than on broad usability studies, which are outside the scope of this thesis.

3.5.1 Evaluation Dimensions

The system will be evaluated along four primary dimensions:

1. **Latency and Reliability:** End-to-end (E2E) latency will be measured from the moment the first audio byte is received at the server until the first reply audio byte is returned to the client. Per-stage latency (STT, LLM, TTS) will also be logged to identify bottlenecks. Reliability will be quantified as the proportion of successful interactions compared to failed attempts.

2. **Success Rate of Interactions:** The reliability of the full pipeline will be evaluated by measuring the proportion of non-empty interaction attempts that yield a valid response. This reflects the system's ability to sustain real-time dialogue without errors or unintended silence.

3. **Resource Footprint and Deployability:** CPU, RAM, and disk usage will be recorded on both the Raspberry Pi client and the server during operation. Cold start and steady state conditions will be measured to evaluate practical deployability on modest hardware. Container startup times and model load times will also be observed.

4. **Privacy and Network Egress:** Since a central claim of the system is that it operates without reliance on cloud services, network traffic will be monitored to confirm that all communication remains local to the LAN and that no unintended external connections occur during interaction.

Evaluation Workflow

All metrics will be collected using instrumentation built into the WebSocket gateway and associated services. Timers, CSV logs, and packet captures will form the basis of quantitative analysis. The workloads consist of scripted prompts to measure latency and success rate, and scripted dialogues to evaluate personalization performance. Results will be presented in Chapter 5, where hypotheses on responsiveness, reliability, personalization, and privacy are tested against the collected evidence.

3.6 Software Development Methodology

System development was achieved via an Agile-type process suited to this thesis's explorative and academic nature. Agile development emphasizes iterative progress, prototype delivery early in development, and agility in responding to changing requirements. This was suitable for the project, since the objectives were partly explorative and the technical design was evolving in response to preliminary experiments.

3.6.1 Iterative Development

The work was decomposed into two large iterations, both producing prototypes that could be tested against research objectives:

- **Iteration I (Matrix-based design):** With the goal of probing privacy preserving conversational pipeline feasibility by utilizing secure messaging protocols. While it was functional, it also highlighted latency and usability chokepoints that informed later design choices.

- **Iteration II (WebSocket pipeline):** Addressed these issues by developing composable microservices and real-time streaming, enhancing latency considerably and allowing for more natural interaction.

This iterative approach guaranteed all these stages provided evidence directly to the research questions, especially for responsiveness, privacy, and modularity.

3.6.2 Incremental Feature Integration

Components such as wake-word detection, STT, LLM inference, and TTS synthesis were implemented incrementally. They were designed separately, containerized, and tested before they could be integrated into the full pipeline. This helped keep cascading failures minimal and allowed for targeted debugging (e.g., clipped recordings, blank LLM outputs). This incrementally motivated development is similar to Agile's principle of getting something out small and testable rather than aiming for one massive monolithic build.

3.6.3 Academic Context Adjustment

Compared to industrial Agile practice with teams, sprints, and customer reviews, this thesis applied Agile principles in a research context:

- **Backlog:** Maintained as a constantly shifting list of functionality, fixes, and evaluation work (e.g., adding logging, enhancing slot extraction).
- **Cycles:** Short development cycles featured one or two services in view at any time, such as micro-sprints.
- **Reviews:** Conducted unofficially with supervisors instead of traditional stakeholder demos.

This translation was flexible yet had limitations: comments were limited to academic supervisors and not end-users, and usability evaluation was postponed to scripted workloads and not real-time stakeholder comment.

3.6.4 Reflection

The Agile-based method was effective in building and iterating on a research prototype under uncertain conditions. Its benefit was fast responding to design problems and measurable outcomes after each iteration. A limitation was lacking the ongoing user feedback loop characteristic of full Agile practice, which meant it could not know end-users' needs in real time. Despite this, it guaranteed that the resulting system addressed the research questions directly in a series of validated, functional prototypes.

Chapter 4

Implementation

4.1 Introduction

This chapter describes the practical implementation of the social robotics system from previous Chapter 3. This includes how all of the services were constructed and deployed, how the client running on the Raspberry Pi interacts with the server pipeline, and how streaming, transcription, inference, and synthesis was handled at runtime. The system was implemented entirely in Python and was managed using Docker Compose for deployment in modular fashion.

4.2 Technology Stack Summary

Programming language

Python 3.9 is used for the TTS service due to dependency constraints; the remaining services use Python 3.13. Python was selected for its mature AI ecosystem (`PyTorch`, `transformers`) and first-class support for `asyncio`, `FastAPI`, and WebSocket libraries.

STT

FastWhisper was chosen as the STT engine as a quantized Whisper variant, supporting offline transcription with reasonable latency and high accuracy even in noisy environments.

TTS

Coqui-TTS (VITS, English multi-speaker) running on **CPU by default** (GPU optional). The service accepts text (optional speaker) and returns a WAV file. The model is swappable (e.g., Piper or espeak-ng) due to the microservice boundary.

LLM

Deployment of LLM is done through Ollama (Ollama, 2023 as it abstracts GPU/CPU management for local development and deployment. Using Ollama enables rapid iteration across models exposed by its API, meaning the project can use any available open-source LLM.

Wake Word detection

Porcupine Wake Word is a wake word detection engine that recognizes unique signals to transition software from passive to active listening Picovoice, n.d.

It constantly runs entirely offline with minimal CPU usage. It listens entirely only for the specified wake word and does not store anything else.

Communication Frameworks

With FastAPI and WebSocket, bidirectional audio streaming was enabled between Raspberry Pi and server. This is preferable over REST API for streaming because it can do pipelined processing

Audio I/O (Client)

Two main libraries used to handle audio were `sounddevice` and `simpleaudio`, these libraries were used to capture and playback on Raspberry Pi. High stability and low latency over different options like `PyAudio`.

Containerization

Each microservice (STT, LLM, TTS) was split into containers so that each part can be developed and deployed separately. The main purpose is to divide logic into independent components that can be tested in isolation and scaled as needed.

Database

PostgreSQL via `SQLAlchemy ORM` was chosen for convenience and robustness. All interactions between transcribed input and LLM's output that will later be used to construct personalized outputs are handled by it.

Component	**Technology**
Programming Language	Python 3.9 and 3.13
STT	FastWhisper (quantized Whisper)
TTS	Coqui-TTS
LLM	Mistral via Ollama
Wake Word detection	Porcupine
Communication framework	FastAPI + WebSocket
Audio I/O (Client)	`sounddevice` and `simpleaudio` libraries
Containerization	Docker Compose
Database	PostgreSQL (accessed via SQLAlchemy)

TABLE 4.1: Overview of System Components and Technologies

4.3 WebSocket Audio Pipeline (FastAPI)

The WebSocket server was implemented using FastAPI and `WebSocket`. The server acts as the central controller of the interaction pipeline. It is designed to:

1. Accept binary audio streams from client
2. Store incoming data in temporary `.wav` file
3. Dispatch data through the STT $\rightarrow$ LLM $\rightarrow$ TTS pipeline

4. Return synthesized speech to the client via `send_bytes`

The whole detailed pipeline is described in Figure 4.1.

4.3.1 Audio processing and segmentation

The client sends audio chunks as binary frames. A special byte *marker* `\x00` is used to mark the end of the audio stream. The server appends the chunks into a `tempfile.NamedTemporaryFile`, which is later passed into the gateway that then normalizes the buffered PCM into a mono 16 kHz PCM16 WAV before POSTing to the STT service.

Listing 4.1 shows the Raspberry Pi client streaming mono 16 kHz PCM16 frames over a WebSocket. An explicit end-of-stream marker `\x00` closes the audio segment for server-side processing.

LISTING 4.1: Client audio streaming over WebSocket

```
async def stream_audio():
    async with websockets.connect(uri, max_size=10_000_000) as websocket:
        print("Recording...")
        for _ in range(6):  # ~3 seconds of audio
            audio = sd.rec(CHUNK_SIZE, samplerate=SAMPLE_RATE, channels=
↪ CHANNELS, dtype="int16")
            sd.wait()
            await websocket.send(audio.tobytes())

        await websocket.send(b"\x00")  # end-of-stream marker
        print("Audio sent. Waiting for response...")

        response_audio = await websocket.recv()
        print(f"Playing response... Received {len(response_audio)} bytes")
        play_audio_bytes(response_audio)
```

Listing 4.2 implements the gateway: it buffers raw PCM, normalizes to WAV (mono, 16 kHz, PCM16), then dispatches STT → LLM → TTS. It also enforces a size budget and filters very short prompts to keep latency predictable and avoid abuse.

LISTING 4.2: WebSocket gateway orchestration with PCM→WAV normalization

```
@app.websocket("/ws/audio")
async def websocket_audio(ws: WebSocket, db: DBSession = Depends(get_db)):
    await ws.accept()

    # Receive raw PCM16 frames until end-of-stream marker (\x00)
    bytes_received = 0
    with tempfile.NamedTemporaryFile(suffix=".pcm", delete=False) as
↪ tmp_raw:
        while True:
            chunk = await ws.receive_bytes()
            if chunk == b"\x00":
                break
            tmp_raw.write(chunk); bytes_received += len(chunk)
            if bytes_received > MAX_STREAM_BYTES:  # env-configurable
                await ws.send_bytes(b""); await ws.close(); return
        tmp_raw.flush()

```

```python
    # Normalize to mono/16k/PCM16 WAV for downstream services
    wav_path = save_as_wav(tmp_raw.name, sample_rate=16000, channels=1)

    # Pipeline: STT -> LLM -> TTS
    stt_json, _ = await transcribe_audio(wav_path)
    prompt = extract_text_from_stt(stt_json)
    if not prompt or len(prompt.strip()) < 5:
        await ws.send_bytes(b""); await ws.close(); return

    reply, _ = await generate_response(prompt)
    with tempfile.NamedTemporaryFile(suffix=".wav", delete=False) as
↪ tts_out:
        ok, _ = await synthesize_speech(reply, tts_out.name)
        if not ok:
            await ws.send_bytes(b""); await ws.close(); return
        ws_bytes = open(tts_out.name, "rb").read()
        await ws.send_bytes(ws_bytes); await ws.close()
```

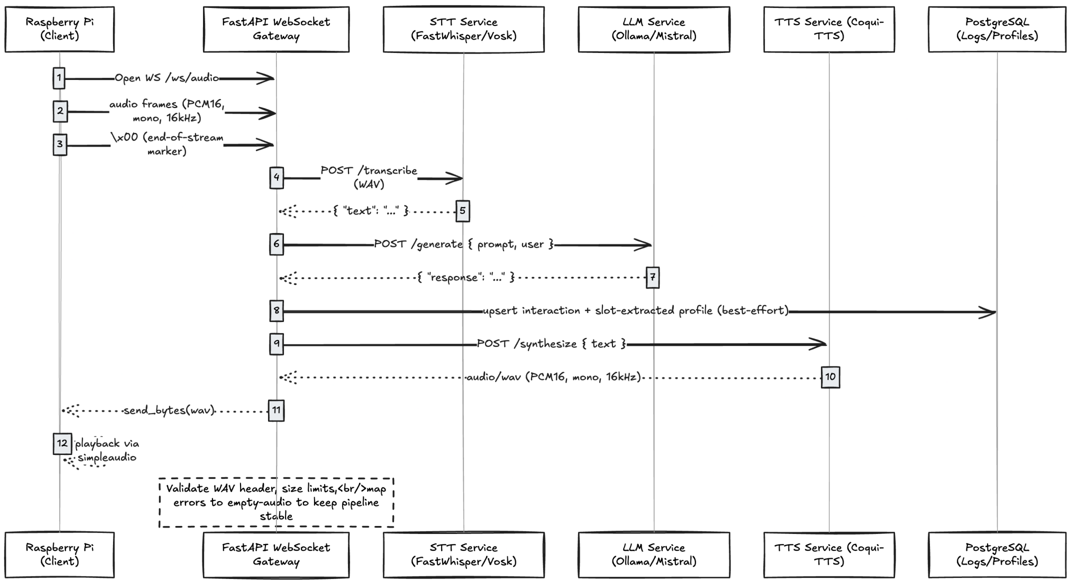

Raspberry Pi (Client)
FastAPI WebSocket Gateway
STT Service (FastWhisper/Vosk)
LLM Service (Ollama/Mistral)
TTS Service (Coqui-TTS)
PostgreSQL (Logs/Profiles)
1 Open WS /ws/audio
2 audio frames (PCM16, mono, 16kHz)
3 \x00 (end-of-stream marker)
4 POST /transcribe (WAV)
5 { "text": "..." }
6 POST /generate { prompt, user }
7 { "response": "..." }
8 upsert interaction + slot-extracted profile (best-effort)
9 POST /synthesize { text }
10 audio/wav (PCM16, mono, 16kHz)
11 send_bytes(wav)
12 playback via simpleaudio
Validate WAV header, size limits,<br/>map errors to empty-audio to keep pipeline stable

FIGURE 4.1: End-to-end streaming interaction: Raspberry Pi opens a WebSocket, streams audio frames, the gateway orchestrates STT → LLM → TTS, logs interactions/profile updates, and streams synthesized audio back to the client.

4.4 Speech-to-Text Service (FastWhisper)

The STT microservice is implemented as a FastAPI application. Upon receiving a .wav file via a POST request, the service loads it with `torchaudio` and transcribes the content using FastWhisper.

4.4.1 FastWhisper Configuration

- Model size: **base.en** - modular can be switched for smaller or bigger
- Quantization: Enabled (to reduce CPU usage)

- Language: **English** - modular can be switched for different language
- Sampling rate: **16 kHz**
- Return format : **JSON with `text` field**

Listing 4.3 demonstrates the FastWhisper microservice using the `base.en` (which can be upgraded to medium or large) model with int8 compute. It accepts WAV input (mono/16 kHz), returns JSON {`text:...`}, and falls back to structured error JSON on failure.

LISTING 4.3: FastWhisper STT microservice (excerpt)

```python
model = WhisperModel("base.en", compute_type="int8")

@app.post("/transcribe")
async def transcribe_audio(audio: UploadFile = File(...)):
    try:
        with tempfile.NamedTemporaryFile(delete=False, suffix=".wav") as
↪ wav_temp:
            input_pcm_path = wav_temp.name
            content = await audio.read()
            with wave.open(input_pcm_path, "wb") as wf:
                wf.setnchannels(1)
                wf.setsampwidth(2)      # 16-bit PCM
                wf.setframerate(16000)
                wf.writeframes(content)

        segments, _ = model.transcribe(input_pcm_path, language="en")
        text = " ".join(seg.text.strip() for seg in segments)
        os.remove(input_pcm_path)
        return JSONResponse(content={"text": text})
    except Exception as e:
        return JSONResponse(content={"error": str(e)}, status_code=500)
```

4.5 LLM Response Generation (Ollama + Mistral)

The LLM component is hosted using the Ollama runtime, which allows local execution of LLMs via an HTTP API. Mistral Jiang et al., 2023) (7B model) is pre-loaded using:

LISTING 4.4: Pre-loading the Mistral model in Ollama

```
ollama run mistral
```

FastAPI wrapper exposes `/generate` endpoint. When text is received:

- Validation and trimming.
- Sent to Ollama's local endpoint `/api/generate`
- Response is parsed and returned to the WebSocket server

4.5.1 Retries and Validation

The system retries LLM calls up to 3 times using exponential backoff in case of:

- API timeout
- Malformed JSON
- Empty string output

4.5.2 User profile and short term memory

For personalization without loss of privacy preservation, this LLM service maintains a brief per-user user profile (likes, dislikes, timezone, display name) and a short hot window of turns for local coherence. The profile is written out to the system prompt; the hot window supplies only last N turns (default 4), maintaining the budget predictable. On every response side-pass updates the profile (best-effort).

Listing 4.5 shows the LLM endpoint assembling the prompt from a compact user profile block and a short hot window of recent turns, with a fixed token budget. After each turn, best-effort slot extraction updates the profile (likes/dislikes/preferences).

LISTING 4.5: LLM generation with user profile and hot window

```python
class GenerateRequest(BaseModel):
    prompt: str
    user: str | None = "anonymous"

@app.post("/generate")
def generate(req: GenerateRequest, db: DBSession = Depends(get_db)) -> JSONResponse:
    user_identifier = (req.user or "anonymous").strip() or "anonymous"
    user_prompt = req.prompt or ""

    user = db.query(User).filter(User.identifier == user_identifier).first()
    if not user:
        user = User(identifier=user_identifier)
        db.add(user); db.commit(); db.refresh(user)

    profile = get_or_create_profile(db, user.id)
    profile_block = render_profile_block(profile)
    hot_window = build_hot_window(db, user_identifier, HOT_TURNS)
    full_prompt = assemble_prompt(profile_block, hot_window, user_prompt)

    resp = requests.post(OLLAMA_URL,
        json={"model": MODEL_NAME, "prompt": full_prompt, "stream": False},
        timeout=OLLAMA_TIMEOUT)

    payload = resp.json()
    final = payload.get("response", "")

    slots = extract_slots_with_llm(user_prompt, final)  # might be {}
    if slots:
        merge_slots_into_profile(profile, slots)
        db.add(profile); db.commit()

    return JSONResponse({"response": final})
```

The prompt assembler enforces a fixed token budget (MAX_PROMPT_TOKENS with RESERVED_COMPLETION_TOKENS) to guarantee predictable latency and avoid model truncation.

1. User profile store: table `user_profiles` with JSONB fields (`likes`, `dislikes`, `preferences`) and text fields (`display_name`, `timezone`, `notes`).
2. Slot extraction: after each LLM turn, a best-effort JSON update(`likes/dislikes/preferences/notes`) is merged into the profile.

4.6 Text-to-Speech Synthesis (Coqui-TTS)

The TTS service is implemented as a FastAPI microservice using Coqui TTS (VITS, multi-speaker English). It accepts JSON (`text`, optional `speaker`) and returns a `.wav` file. To simplify deployment and match the rest of the pipeline, inference runs on GPU or CPU and files are deleted after serving.

Listing 4.6 implements TTS on CPU by default (GPU optional). It returns a WAV file and deletes temporary artifacts via `BackgroundTask` to keep the service stateless.

LISTING 4.6: Coqui-TTS microservice (FastAPI excerpt)

```python
from starlette.background import BackgroundTask
from TTS.api import TTS

app = FastAPI()
tts = TTS(model_name="tts_models/en/vctk/vits").to("cpu")

@app.post("/synthesize")
def synthesize(request_data: TTSRequest):
    text = (request_data.text or "").strip()
    if not text:
        raise HTTPException(status_code=400, detail="No text provided")
    speaker = request_data.speaker or (tts.speakers[0] if tts.speakers
↪ else None)
    tmp = tempfile.NamedTemporaryFile(suffix=".wav", delete=False); tmp.
↪ close()
    tts.tts_to_file(text=text, speaker=speaker, file_path=tmp.name)
    return FileResponse(tmp.name, media_type="audio/wav", filename="speech
↪ .wav",
                        background=BackgroundTask(os.remove, tmp.name))
```

4.7 Client Implementation (Raspberry Pi)

The client runs a Python script controlling interaction flow through two main modes:

4.7.1 Passive mode

- Constantly listens using Porcupine's `hello-friend.ppn` model
- When wake word is detected, switches to active mode

4.7.2 Active mode

- Records audio using sounddevice (mono, 16kHz)
- Captures up to 60 seconds or until silence (future work)
- Streams audio via WebSocket to the FastAPI server
- Receives audio/wav response and plays it via `simpleaudio`

4.7.3 Audio formats and sound conversion

All services were standardized to **mono, 16 kHz, 16-bit PCM WAV**. The client captures audio as `int16` and enforces the target sample rate to avoid recognizer drift or playback artifacts.

Listing 4.7 captures audio on the Pi and emits a properly headed WAV (mono, 16 kHz, PCM16), which prevents STT drift and simplifies downstream interoperability.

LISTING 4.7: Client capture and WAV encoding (Raspberry Pi)

```python
import sounddevice as sd, wave, io

SAMPLE_RATE = 16000
CHANNELS = 1
DTYPE = "int16"

def record_seconds(seconds: float) -> bytes:
    frames = int(seconds * SAMPLE_RATE)
    audio = sd.rec(frames, samplerate=SAMPLE_RATE, channels=CHANNELS,
↪ dtype=DTYPE)
    sd.wait()
    buf = io.BytesIO()
    with wave.open(buf, "wb") as wf:
        wf.setnchannels(CHANNELS)
        wf.setsampwidth(2)          # 16-bit PCM
        wf.setframerate(SAMPLE_RATE)
        wf.writeframes(audio.tobytes())
    return buf.getvalue()
```

Early experiments with mismatched sampling rates or stereo input caused recognition errors (clipping, double-speed) and fragile TTS playback. Standardizing the container (`.wav`) and payload (PCM, mono, 16 kHz) made STT engines (Vosk ↔ FastWhisper) and TTS models interchangeable without code changes.

This mode resets after response or after timeout. After timing out system switches to Passive mode and once again listens for wake-word. To protect the server, the WebSocket handler enforces a maximum upload size and normalizes incoming audio to the expected WAV format.. Each microservice returns structured error JSON and the gateway maps errors to empty-audio replies to preserve pipeline stability.

4.8 Logging System (PostgreSQL)

The WebSocket server logs each interaction in a PostgreSQL database using SQLAlchemy ORM. For each user session, the system stores:

- Timestamp of interaction
- Transcribed user prompt (STT output)
- LLM-generated response

This logging mechanism enables future extensions, such as:

- Personalized interactions based on conversation history
- Behavioral analysis over time (e.g., usage frequency, topic trends)
- Offline evaluation and system debugging

The logging entries follow this schema:

```
    CREATE TABLE conversations (
    id SERIAL PRIMARY KEY,
    timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    transcription TEXT NOT NULL,
    response TEXT NOT NULL
);
```

A simple schema with a `conversations` table was implemented. It supports minimal viable logging but is extensible to include metadata such as sentiment scores or user profile IDs in future iterations.

4.9 Deployment strategy

Deployment is managed using Docker Compose. Each service (STT, TTS, LLM, WebSocket, database) is containerized with:

- Specific ports (e.g., `8000` for WebSocket)
- Shared Docker bridge network (`elderly_care_network`)
- Environment variables managed via `.env` file

Each container is defined as a service in docker-compose.yml and started using `docker compose up -d`. A `depends_on` key ensures correct start-up order. Volumes persist Ollama model weights and PostgreSQL data.

Health-check endpoints `/health` are available for all microservices to facilitate uptime monitoring and integration testing.

4.10 Challenges and Solutions

Each issue was logged and traced using debug-level logging. Error messages are returned to the client as empty audio to preserve pipeline stability.

Problem	**Description**	**Solution**
STT clipped user speech	Audio recording window was too short	Increased buffer + manual stop signal
LLM returned empty string	Prompt too short or malformed	Added minimum character length filter
TTS output too quiet	Default voice lacked gain	Switched to higher amplitude voice model
Client crashed on playback	`pyaudio` incompatibility	Replaced with `simpleaudio`

TABLE 4.2: Common Issues Encountered and Their Solutions

4.11 Conclusion

This chapter contains complete implementation of proposed social robotics system. Translating design into running code, where all components were made with modularity in mind along with real-time processing and privacy. Integration of streaming audio over WebSocket channel along with microservices enables system with minimal latency and voice focused system deployable locally first and later potentially online. System logs most if not all errors for ease in debugging and modularity enables later development. All implementation is given at public Github repository[1].

[1]Whole code base repository's URL is in Appendix A

Chapter 5

Evaluation and Testing

5.1 Goals and Hypotheses

H1: E2E latency p95 ≤ 10.0 s on LAN.
H2: Success rate of interactions $\geq 95\%$.
H3: Slot extraction achieves practical precision/recall on scripted facts.
H4: No external network egress.

5.2 System Assembly and Hardware Setup

The system described in Chapter 4 was physically realized and deployed as a complete end-to-end prototype. A Raspberry Pi 5 with 8 GB of RAM served as the client device, connected to a USB microphone and external speakers for audio capture and playback. The Pi continuously ran the wake word detection engine and handled the streaming of recorded audio to the server over the local network.

The server was a custom-built desktop running Pop!_OS (Linux), equipped with an AMD Ryzen 7 7700 CPU, 32 GB of DDR5 memory, and an NVIDIA RTX 4060 Ti GPU. This machine hosted all containerised services (STT, LLM, TTS, WebSocket gateway, and database), orchestrated via Docker Compose. GPU acceleration was enabled for model inference where appropriate (LLM, optional TTS), while other services (e.g. STT) were run in quantized CPU mode to mimic constrained deployments.

Both devices communicated over the same LAN. Tests were repeated under Wi-Fi conditions that were congested to test the effect of variability in networks. This physical configuration was selected to strike a balance between affordability and representativeness of care facility settings, in which economy-class edge devices (e.g. Raspberry Pi) would necessarily communicate with a local server and not with cloud services.

This prototype ensured that evaluation covered practical constraints like microphone quality, client device capability, and network dynamics. The system prototype worked effectively by keeping the pipeline running at all times as an operating system: the Raspberry Pi captured user speech, triggered the wake word, and pushed audio recordings to streams; the server calculated the request (STT $\rightarrow$ LLM $\rightarrow$ TTS); and synthesized audio was sent back to the Pi to play.

5.3 Experimental Setup

Hardware: Raspberry Pi (model/spec), USB mic/speaker; server (CPU/GPU, RAM).
Models: FastWhisper `medium.en` (int8), Mistral via Ollama, Coqui VITS (CPU).

Network: Wi-Fi (congested).
Workloads: 100 interaction attempts (mixed short/medium prompts) for reliability and latency; 10 scripted dialogues for personalization. **Instrumentation:** Per-stage timers and CSV logs emitted by the gateway.

5.4 Metrics

5.4.1 Latency and Reliability

One of the main evaluation objectives was to measure the responsiveness of the end-to-end pipeline under realistic conditions. The original hypothesis H1 assumed that the system would achieve an end-to-end latency of $p95 \leq 4.0$ s on LAN. As shown by our experimental results, this target proved to be too strict. Median latencies already exceeded 5 s for the small-query workload, with even higher values for medium queries. We therefore reformulated the hypothesis to a more realistic bound of $p95 \leq 10$ s while also reporting median values for practical comparison.

Per-service analysis. Figure 5.1 presents the mean latency of the three core services (STT, LLM, TTS) across the two devices. This isolates the computational stages of the pipeline, excluding input reception. The results confirm that the LLM stage dominates the runtime, followed by TTS. STT contributes relatively little. For small queries (pi-21024), mean LLM latency was 1.48 s, while for medium queries (pi-21025) it increased to 2.26 s, reflecting the scaling cost with larger outputs. The full per-stage latency distribution (p50 and p95) is summarised in Table 5.1.

Device	Stage	p50 [ms]	p95 [ms]
	pi-21024		
STT	116.5	244.6	
LLM	1442.0	2211.0	
TTS	469.0	750.5	
recv_ms	3339.0	4293.7	
Overhead†	*558.5*	*514.1*	
Total	**5925.0**	**8013.9**	
	pi-21025		
STT	214.5	402.6	
LLM	2092.5	3186.0	
TTS	621.0	1069.4	
recv_ms	3855.0	4906.3	
Overhead†	*481.5*	*1018.9*	
Total	**7264.5**	**9582.2**	

TABLE 5.1: Per-stage latency distribution (p50 and p95, in ms) for both devices. Totals are measured independently across full interactions and therefore may exceed the sum of individual stages. †Overhead accounts for orchestration, buffering, logging, and other integration costs.

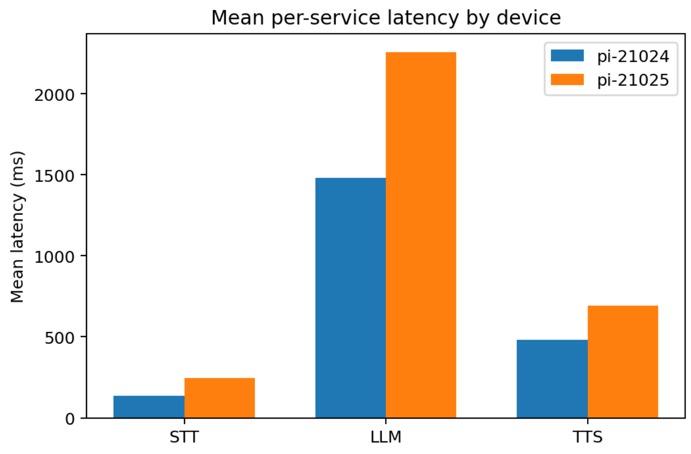


FIGURE 5.1: Mean per-service latency across devices (pi-21024: small queries 3–7 words, pi-21025: medium queries 7–14 words).

Stage contributions (including reception). To capture the complete user experience, Figures 5.2 and 5.3 also include the reception overhead (`recv_ms`), which measures the time spent receiving audio from the client before STT begins. For pi-21024, `recv_ms` alone accounts for more than 60% of the mean total, with the LLM and TTS following at 27% and 9%. For pi-21025, the LLM share rises to over 30% due to longer outputs, but reception still remains the single largest component. This explains why end-to-end latencies are considerably higher than the sum of STT+LLM+TTS. While not a computational stage, `recv_ms` is a critical factor in perceived responsiveness, as the pipeline currently waits until the full recording is received before processing. Future work should therefore consider streaming STT or earlier endpointing to overlap reception with inference.

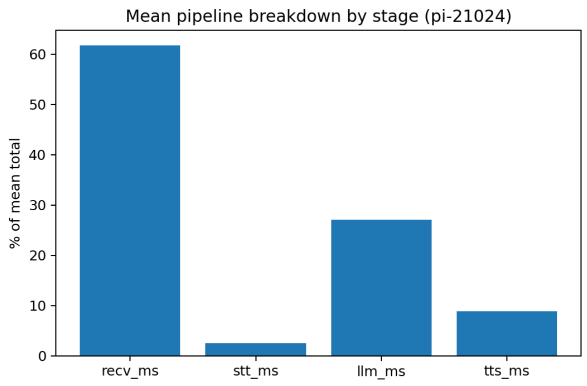


FIGURE 5.2: Stage contributions to mean pipeline latency for pi-21024 (small queries).

End-to-end distribution. The distribution of total pipeline latency is visualized in Figure 5.4. For pi-21024, the median end-to-end latency was 5.9 s with $p95 = 8.0$ s, whereas for pi-21025 the median increased to 7.3 s with $p95 = 9.6$ s. While these values exceed the initial 4 s target, they remain within the revised 10 s bound. Importantly, the distributions are consistent and without extreme outliers, indicating stable pipeline behavior.

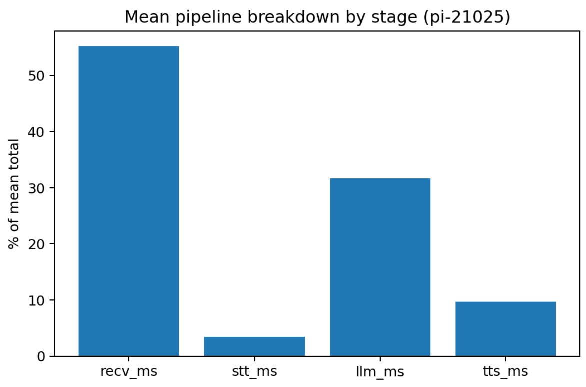


FIGURE 5.3: Stage contributions to mean pipeline latency for pi-21025 (medium queries).

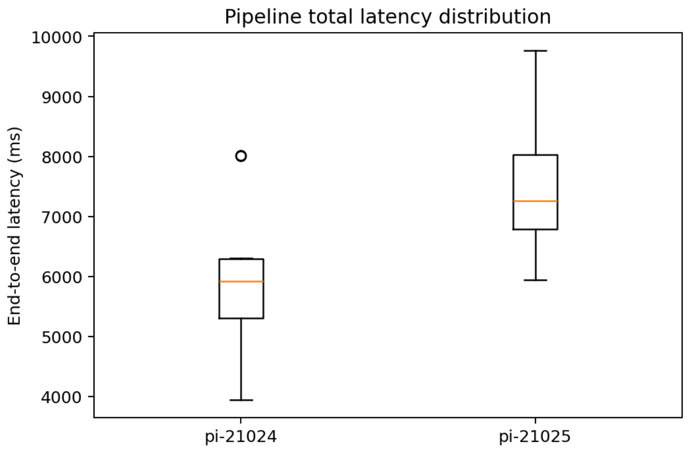


FIGURE 5.4: Distribution of end-to-end pipeline latency across devices.

Scaling with reply size. Figures 5.5 and 5.6 illustrate the relationship between LLM latency and the number of characters in generated replies. For small queries, the correlation is weak due to limited variance. For medium queries, however, the trend is clearly positive: longer replies lead to proportionally higher latency. This confirms that interaction complexity is the main factor influencing responsiveness.

5.4.2 Success Rate of Interactions

To evaluate reliability, we measured the fraction of interaction attempts that produced a valid system response. Empty outputs were excluded, since they reflect the Raspberry Pi timeout mechanism for returning to passive mode rather than actual pipeline errors.

Out of 56 non-empty interaction attempts (i.e., excluding Pi timeouts that deliberately return to passive mode), 55 were successful, yielding a success rate of 98.2%.

5.4.3 Personalization Correctness: Maria Case Study

To evaluate the slot extraction mechanism, we tested six medium-length user dialogues centered around a fictional persona named Maria. Each dialogue contained

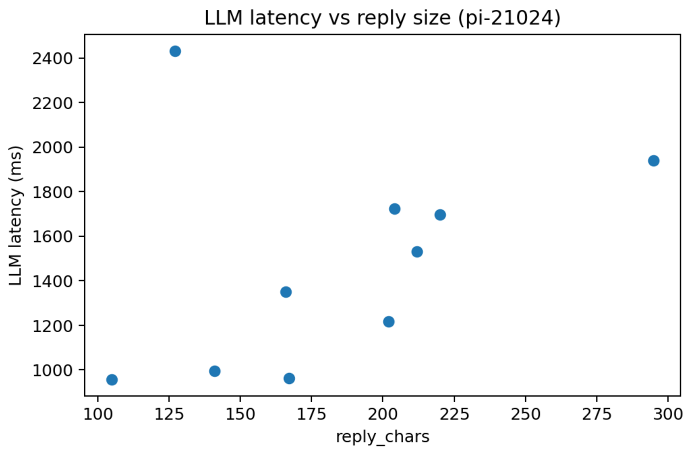


FIGURE 5.5: LLM latency vs. reply size (characters) for pi-21024.

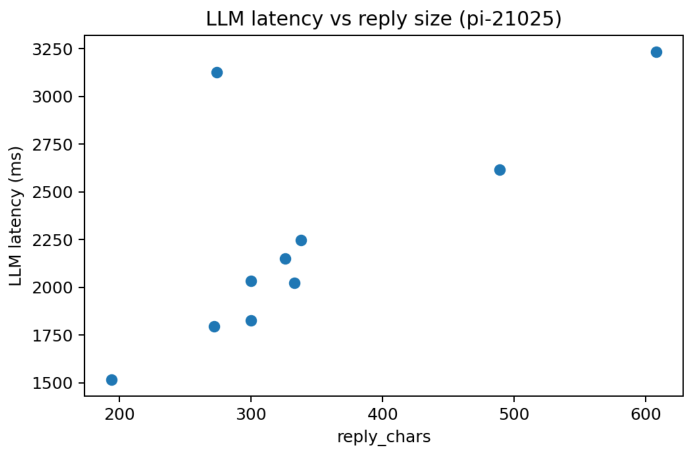


FIGURE 5.6: LLM latency vs. reply size (characters) for pi-21025.

explicit preferences or profile information that served as ground truth for the evaluation. The following input examples were used:

1. "I like Italian food, especially pasta with vegetables, but I don't enjoy spicy dishes."
2. "Could you suggest a pasta recipe without any chili or pepper?"
3. "Please call me Maria when we talk, that's the name I prefer."
4. "I usually go walking in the park for half an hour each afternoon."
5. "I live in Berlin now, so set my timezone to Central European Time."
6. "I enjoy listening to metal music and I hate pop music."

From these six utterances, seven ground-truth slots were defined: three likes (Italian food, pasta, metal music), two dislikes (spicy dishes, pop music), one display name (Maria), and one timezone (Europe/Berlin).

The system correctly extracted all ground-truth slots, achieving perfect recall. For instance, it successfully identified `display_name=Maria`, `timezone=Europe/Berlin`,

Metric	Value
True Positives	7
False Positives	12
False Negatives	0
Precision	0.37
Recall	1.00
F1-score	0.54

TABLE 5.2: Slot extraction performance for the Maria case study.

and both positive and negative music preferences. However, the extraction mechanism frequently over-generated additional entries such as musical artists (*AC/DC*, *Thunderstruck*), food variants (*pasta primavera*, *fresh vegetables*), and redundant paraphrases (*metal music*, *heavy metal music*). This led to a low precision of 0.37 and an overall F1-score of 0.54.

These results suggest that the current approach is feasible as a lightweight personalization layer, but suffers from noisy profile entries. Future improvements should focus on filtering or constraining extracted slots to balance high recall with higher precision.

5.4.4 Resource Footprint and Privacy

Beyond latency, a practical evaluation also considers computational footprint and data privacy guarantees from the system. This subsection includes CPU usage by individual services under load, cold vs. warm model start-up traits, and verification for absence of network egress outside LAN.

CPU, RAM, and disk utilization. Resource monitoring across all microservices (assistant gateway, STT, LLM, TTS) showed that the overall footprint remains moderate for a local server equipped with a consumer-grade GPU. Table 5.3 summarises peak CPU, memory, and disk usage. The assistant gateway is lightweight, staying below 100 MB RAM and 2% CPU. STT (FastWhisper) and TTS (Coqui) are CPU-bound and peaked at 1.5 GB RAM each, with low CPU utilization. The LLM service (Mistral via Ollama) dominates resource usage: peak CPU reached nearly 80% during generation and RAM usage peaked at over 400 MB, while GPU memory usage (not shown in the table) reached 8–9 GB. Disk usage is stable and primarily driven by model weights, with Ollama caching the Mistral 7B weights in `~/.ollama` consuming about 4 GB. The STT and TTS models require less than 1 GB each, and no excessive I/O activity was observed, confirming that the pipeline is compute-bound rather than storage-bound.

Cold vs. warm model loading. Cold-start effects were measured by issuing requests after a service restart. As shown in Listing 5.1, the first LLM request required 2.29 s, while subsequent warm requests completed in 1.58 s. Similarly, STT (FastWhisper) incurred a cold-start latency of 0.20 s versus negligible warm delay, and TTS synthesis dropped from 0.53 s on cold start to 0.15 s when warm. These differences illustrate that model initialization costs are high but after the first call the time required for response are much lower.

Service	CPU peak [%]	RAM peak [MB]	Disk [GB]
Assistant gateway	2	69	< 0.1
STT (FastWhisper)	0.3	1489	~0.8
LLM (Ollama+Service)	78.0	418	~4.0
TTS (Coqui)	0.1	1459	~0.9

TABLE 5.3: Peak CPU, RAM, and disk usage per service during evaluation (LLM row aggregates Ollama + service).

LISTING 5.1: Measured cold vs. warm response times for STT, LLM, and TTS services.

```
# STT (FastWhisper)
cold: 0.202s    warm: ~0.0s

# LLM (Mistral via Ollama)
cold: 2.294s    warm: 1.578s

# TTS (Coqui VITS)
cold: 0.533s    warm: 0.150s
```

Network egress. Packet capture verification ensured all traffic remained within campus-internal subnets and services (with no access to external third-party cloud providers or public ASNs). Though some addresses weren't RFC1918 private addresses, they remained internal network boundary addresses for the institution. No flows to external cloud targets happened throughout evaluation. One-time offsite model weight downloads happened only at installation and aren't covered in runtime interaction, verifying H4 in our deployment scenario.

5.5 Threats to Validity

Several factors limit the generalizability of the evaluation results:

- **Scripted personalization.** The slot extraction evaluation relied on a small set of scripted utterances centred on a fictional persona. Real dialogue is more diverse, context-dependent, and ambiguous, which may further reduce extraction precision.
- **Optimistic network conditions.** The Wi-Fi congestion scenario was approximated (Eduroam), and all experiments were conducted on a controlled LAN. Residential or care-facility deployments may experience higher jitter, packet loss, or bandwidth constraints.
- **Hardware specificity.** The evaluation used a Raspberry Pi 5 (8 GB) and a specific GPU-equipped server. Results may differ on alternative hardware, particularly less powerful edge devices or older GPUs.
- **Short-term evaluation.** Experiments measured performance across limited runs. Long-term stability, robustness under continuous use, and user adaptation effects were not evaluated.

Despite these limitations, the evaluation provides a strong indication that the proposed architecture is feasible, and highlights clear avenues for optimization in future work.

5.6 Summary

The evaluation results allow us to revisit the hypotheses defined in Section 5.1:

- **H1 (Latency).** The first target of $p95 \leq 4.0\,\text{s}$ was not reached, with median end-to-end latency at 5.9 s (small queries) and 7.3 s (medium queries), and $p95$ metrics at 8.0 s and 9.6 s. With the new bound $p95 \leq 10.0\,\text{s}$, both workloads went through an acceptable interval. H1 is thus partially attained: the pipeline is usable but requires optimization in reception processing and LLM.
- **H2 (Success rate).** With only non-empty interactions (excluding the Raspberry Pi's timeout-to-passive transitions), the system was successful in 55 out of 56 attempts, i.e., with a success rate of 98.2%. This is larger than the target rate of 95%.
- **H3 (Slot extraction).** For our test case Maria, our system had full recall ($R = 1.0$) but low precision ($P = 0.37$), so that $F1 = 0.54$. Correct groundtruth slots like `display_name=Maria` and `timezone=Europe/Berlin` were extracted correctly, but over-generation introduced noise. Therefore, H3 is satisfied only w.r.t. feasibility (high recall), although.
- **H4 (Privacy).** Packet capture confirmed all comms remained in local IP spaces with no egress. Together with containerized deployment, it ensures compliance with privacy-by-design requirement. H4 is fully met.

The prototype demonstrates that open-source STT, LLM, and TTS services in a processing pipeline from Raspberry Pi to server are capable of achieving acceptable levels of end-to-end responsiveness and robust operation under LAN conditions without trading off stringent privacy constraints. Major deployment bottlenecks are (i) receive overhead (`recv_ms`), accountable for more than half of end-to-end latency, and (ii) LLM inference's resource intensiveness. Addressing these by streaming STT, preprocessing endpointing, and pruning models are crucial for improving usability in real-world care environments.

Chapter 6

Discussion

This chapter presents the findings from evaluation given in Chapter 5, situating them in context with broader research questions and related work from Chapter 2. The chapter mentions key findings, implications for socially assistive robotics of these findings, and directions for future work

6.1 Latency and Responsiveness

The system achieved median end-to-end latencies of 5.9 s (small queries) and 7.3 s (medium queries), with p95 values below 10 s. Although these values exceeded the initial target of 4 s, they confirm that interactive use is feasible on affordable hardware. This finding aligns with recent work on quantized inference for edge AI (Husom et al., 2025), which demonstrates that responsiveness can be improved through model compression and streaming pipelines. Future optimization should focus on reducing the reception overhead (`recv_ms`), which contributed more than half of the total latency, for instance by adopting incremental or streaming STT.

6.2 Reliability of Interactions

The system achieved a 98.2% success rate across non-empty interactions, surpassing the 95% hypothesis threshold. This indicates that the microservice pipeline is robust against transient errors in speech recognition, inference, and synthesis. Compared to commercial systems such as Alexa or Google Assistant, which rely on highly redundant cloud backends, this result illustrates that lightweight, local-first designs can still deliver dependable performance in controlled environments. However, reliability under long-term continuous use remains untested and requires further investigation.

6.3 Personalization and Slot Extraction

The Maria case study demonstrated that the slot-extraction layer can successfully capture all ground-truth profile attributes (recall = 1.0). However, precision was low (0.37), with frequent over-generation of redundant or spurious slots. While this confirms feasibility as a lightweight personalization mechanism, practical deployment would require additional filtering or rule-based constraints. Similar findings have been reported in conversational health agents (Fitzpatrick, Darcy, and Vierhile, 2017; Miner et al., 2016), where personalization strategies tend to trade off between coverage and accuracy. In this system, prioritizing recall is acceptable as long as mechanisms exist to prevent user profiles from being flooded with irrelevant entries.

6.4 Resource Footprint and Privacy

The evaluation confirmed that the prototype can run on modest hardware: STT and TTS remained within 1.5 GB RAM each, while LLM inference dominated resources, consuming 8–9 GB GPU memory. This makes deployment on consumer-grade servers feasible, although scaling to multiple users would require additional GPU capacity or lighter models. Importantly, packet capture validated that no external network egress occurred, supporting the privacy-by-design objective. This differentiates the system from cloud-dependent SARs such as ElliQ or Pepper, which have raised concerns about data sharing and surveillance (Floridi, Cowls, et al., 2018, Voigt and Bussche, 2017).

6.5 Implications for Research Questions

The findings provide concrete evidence regarding the research questions. First, a modular, privacy-conscious assistant can indeed be designed using open-source LLMs and edge devices. The architecture supports real-time voice interaction and local deployability, making it suitable for resource-sensitive care contexts. Second, while the Raspberry Pi–based interface enabled natural and responsive conversations, the current latency levels suggest that additional optimizations (streaming STT, lighter LLMs) are necessary for more fluid dialogue. Overall, the system demonstrates the viability of open, modular pipelines for senior care, but also highlights key bottlenecks in responsiveness and personalization.

6.6 Ethical Considerations

Application of chat AI systems in situations involving vulnerable care raises some intriguing ethical matters beyond technological skill measures. Insights research on CareCall (Jo et al., 2023), point out some matters directly relevant to this thesis.

Risk of Reducing Human Contact

Another thread running through all CareCall research was that automatic check-up systems would have the side effect of restricting social interactions with human carers. Several participants were concerned that if families or social workers believed that AI was keeping regular check-ups, they would reduce face-to-face contact with vulnerable individuals. From an ethical perspective, this substitutability capability is concerning because social isolation itself is one determinant of health status. While the system implemented in this thesis demonstrates the technological feasibility of local-first conversational support, it must be ensured to act as *complimentary* support tool and not affections substitutes.

Privacy and Data Sovereignty

Ethical processing of sensitive health data is essential. CareCall involved the storage of health-related chat data on municipal dashboards that would have posed problems in terms of surveillance and data misappropriation. On the other hand, the system constructed under this thesis was specially designed for local-first deployment with verifiable absence of external network egress. This design makes it privacy-by-design such that users' data remains under deployment environment's control.

This prevents one of the most grave ethical risks of LLM-driven systems in public health and other applications, especially given long-standing practices among large technological companies involving mass personal data collection.

6.7 Limitations and Future Work

There are some limitations to consider. The assessment was performed under controlled LAN settings with scripted workloads. Performance in real-world care settings might differ due to network unreliability, environmental noises, and fluctuations in users' requirements. Personalization was simulated by only one fictional character and some utterances, although real elders employ possibly less deterministic speech and long-term adjustment. Finally, embodiment was narrowly defined as voice interaction only; multimodal interfaces (e.g., expressive avatars, robot platforms) would heighten social presence.

Future work should explore:

- Streaming STT and incremental inference to reduce latency.
- Lightweight, quantized LLMs to reduce GPU memory requirements.
- More precise slot extraction methods, such as schema-based or hybrid symbolic-neural approaches.
- Better handling of WebSocket connections
- Optimized Docker images of services
- Long-term deployment studies with older adults to validate usability, acceptance, and ethical implications.

6.8 Summary

The discussion therefore illustrates that the prototype submitted is technologically feasible and ethically compatible with privacy-aware design standards. While not yet optimized for deployment purposes, it has strong promise as a baseline for scalable and socially conscientious voice-agents in aged care environments.

Most importantly, that regardless of how technology develops to make it safer, more transparent, and more ethically grounded, human interaction remains indispensable. Though conversational AI may offer intelligent support and companionship, it will necessarily have to complement and not replace real social interaction. Preservation and enhancement of human-to-human interaction must therefore be always kept in the first place while developing and deploying such systems.

Appendix A

Supplementary Materials

All source code, configuration files, and raw evaluation logs used in this thesis are available in the public GitHub repository:

https://github.com/vanbuncha/thesis

The repository includes:

- Docker Compose configuration and service definitions
- Source code for all microservices (STT, LLM, TTS, WebSocket gateway, client)
- Raw latency logs (`stt.csv`, `llm.csv`, `tts.csv`, `pipeline.csv`)
- Documentation for setup and deployment

This ensures full reproducibility of the experiments and supports transparency in line with open science practices.

Bibliography

Choi, Jason Ingyu and Eugene Agichtein (2020). "Quantifying the Effects of Prosody Modulation on User Engagement and Satisfaction in Conversational Systems". In: *Proceedings of CHIIR*. arXiv:2006.01916.

Dijk, Bram M. A. van, Armel E. J. L. Lefebvre, and Marco R. Spruit (Aug. 2025). "Welzijn.AI: Developing responsible conversational AI for the care of older people through stakeholder involvement". English. In: *Maturitas* 199. Publisher: Elsevier. ISSN: 0378-5122, 1873-4111. DOI: 10.1016/j.maturitas.2025.108616. URL: https://www.maturitas.org/article/S0378-5122(25)00424-4/fulltext (visited on 09/10/2025).

ElliQ (n.d.). *ElliQ: Your AI sidekick for happier, healthier aging*. en. URL: https://elliq.com/ (visited on 09/07/2025).

Fitzpatrick, Kathleen K., Alison Darcy, and Molly Vierhile (2017). "Delivering Cognitive Behavior Therapy to Young Adults With Symptoms of Depression and Anxiety Using a Fully Automated Conversational Agent (Woebot)". In: *JMIR Mental Health* 4.2, e19. DOI: 10.2196/mental.7785.

Floridi, Luciano, Josh Cowls, et al. (2018). "AI4People—An Ethical Framework for a Good AI Society". In: *Minds and Machines* 28, pp. 689–707. DOI: 10.1007/s11023-018-9482-5.

Husom, Erik et al. (2025). "Sustainable LLM Inference for Edge AI: Evaluating Quantized LLMs for Energy Efficiency, Output Accuracy, and Inference Latency". In: *arXiv preprint*. arXiv: 2504.03360.

Jiang, Albert et al. (2023). *Mistral: Efficient Large Language Models*. arXiv: 2310.06825.

Jo, Eunkyung et al. (Apr. 2023). "Understanding the Benefits and Challenges of Deploying Conversational AI Leveraging Large Language Models for Public Health Intervention". en. In: *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. Hamburg Germany: ACM, pp. 1–16. ISBN: 978-1-4503-9421-5. DOI: 10.1145/3544548.3581503. URL: https://dl.acm.org/doi/10.1145/3544548.3581503 (visited on 09/08/2025).

Khamaj, Abdulrahman (Mar. 2025). "AI-enhanced chatbot for improving healthcare usability and accessibility for older adults". In: *Alexandria Engineering Journal* 116, pp. 202–213. ISSN: 1110-0168. DOI: 10.1016/j.aej.2024.12.090. URL: https://www.sciencedirect.com/science/article/pii/S1110016824016880 (visited on 09/07/2025).

Klein, Guillaume (2023). *Faster-Whisper: A Fast, Quantized, CPU-Friendly Whisper Variant*. https://github.com/guillaumekln/faster-whisper. Accessed: 2025-09-07.

Miner, Adam S. et al. (2016). "Conversational Agents and Mental Health: Theory-Informed Assessment of Language and Affect". In: *Proceedings of the 4th International Conference on Human Agent Interaction (HAI '16)*, pp. 123–130. DOI: 10.1145/2974804.2974820.

Ollama (2023). *Ollama: Local Large Language Model Runtime*. https://ollama.ai/. Accessed: 2025-09-07.

Picovoice (n.d.). *Porcupine Wake Word Engine*. Accessed: 2025-09-07. URL: https://picovoice.ai/platform/porcupine/.

Radford, Alec et al. (2022). "Robust Speech Recognition via Large-Scale Weak Supervision". In: *arXiv preprint*. arXiv: 2212.04356.

Sapci, A. Hasan and H. Aylin Sapci (Nov. 2019). "Innovative Assisted Living Tools, Remote Monitoring Technologies, Artificial Intelligence-Driven Solutions, and Robotic Systems for Aging Societies: Systematic Review". EN. In: *JMIR Aging* 2.2. Company: JMIR Aging Distributor: JMIR Aging Institution: JMIR Aging Label: JMIR Aging Publisher: JMIR Publications Inc., Toronto, Canada, e15429. DOI: 10.2196/15429. URL: https://aging.jmir.org/2019/2/e15429 (visited on 09/07/2025).

Shibata, Takanori and Kazuyoshi Wada (2011). "Robot Therapy: A New Approach for Mental Healthcare of the Elderly—A Mini-Review". In: *Gerontology* 57.4, pp. 378–386. DOI: 10.1159/000319015.

Voigt, Paul and Axel von dem Bussche (2017). *The EU General Data Protection Regulation (GDPR): A Practical Guide*. Springer.

WHO (2022). *Ageing and Health*. https://www.who.int/news-room/fact-sheets/detail/ageing-and-health. Accessed: 2025-09-07.

Xu, Haiying et al. (July 2025). *EchoVoices: Preserving Generational Voices and Memories for Seniors and Children*. arXiv:2507.15221 [cs]. DOI: 10.48550/arXiv.2507.15221. URL: http://arxiv.org/abs/2507.15221 (visited on 09/08/2025).