HARU

AI-Driven Multimodal Smart Home Manager

Junho Uh
College of Engineering
Hanyang University
Dept. of Information Systems
Seoul, Korea
djwnsgh0248@hanyang.ac.kr

Yeonseong Shin
College of Engineering
Hanyang University
Dept. of Data Science
Seoul, Korea
dustjd2651@gmail.com

Dogyeom Kim
College of Engineering
Hanyang University
Dept. of Computer Science
Seoul, Korea
dogyeom74@hanyang.ac.kr

Donghyun Lim
College of Engineering
Hanyang University
Dept. of Computer Science
Seoul, Korea
limdongxian1207@gmail.com

***Abstract*—Smart home technology has significantly enhanced household convenience through automation; however, it still faces limitations in user personalization and proactive responsiveness. This project proposes an Intelligent Smart Home Agent System that integrates three key functions: (1) user state analysis leveraging biometric and weather data, (2) persona-based prompting grounded in initial user preferences, and (3) an alert-and-call mechanism for anomaly detection. The system establishes an AI persona whose personality and response style are shaped by the user's basic information and lifestyle patterns. Through ongoing interaction and feedback, the system incrementally refines its behavior, adapting to the user's evolving routines and preferences. Furthermore, the system enables bi-directional voice communication between the user and the home AI. Ultimately, the proposed system—HARU—aims to establish a more intelligent and empathetic home environment by implementing a context-aware, self-evolving agent that allows seamless interaction from any location, autonomously manages daily conditions, and continuously learns to align with the user's physical and emotional state.**

TABLE I: Role Assignments

Roles	Name	Task description and etc.
Project Manager, Software Developer (Back-end)	Jun ho Uh	As the representative user and development manager, Jun Ho Uh identifies user requirements and ensures that system features meet real customer needs. He leads overall project planning, timeline management, and quality assurance. Additionally, he supervises the full software lifecycle—from design and implementation to testing—and guarantees alignment between user expectations and system functionality.
AI Developer, User, Consumer	Yeon seong Shin	The AI developer is responsible for designing and implementing intelligent models that interpret user behavior and generate adaptive responses. Yeon Seong Shin develops algorithms for biometric and environmental data analysis, builds multi-agent conversational systems for personalized interaction, and ensures efficient data processing pipelines for continuous learning.
Software Developer (Back-end), Server Manager	Dogyeom Kim	The backend developer is responsible for safety and efficiently handling user data and AI analysis results in HARU platform development. Dogyeom Kim designs and builds FastAPI-based API servers and implements real-time data management systems using MySQL. Based on the cloud environment, it connects various APIs to manage data, and focuses on communication between users and AI core and ensuring the scalability and stability of the system's Backend infrastructure.
Software Developer (Front-end), UI/UX Designer	Dong Hyun Lim	The frontend developer is responsible for designing and implementing the user-facing interface of HARU. Dong Hyun Lim develops interactive dashboards and responsive layouts that visualize user state information, integrates voice and persona interactions, and ensures seamless communication with backend APIs for real-time updates and feedback.

I. Introduction

A. Motivation

1) Development of Smart Homes

Over the past decade, smart home technology has developed rapidly. Initially, it was limited to simple voice-command-based appliance control, but with the spread of IoT technology, the level of inter-device connectivity and automation has gradually increased.

However, despite this progress, current smart home technologies still remain mechanical and function-centered. Systems simply respond and act according to user commands in a repetitive manner. At present, while "intelligent reactions based on commands" are possible, the system still lacks "emotional understanding" of those commands or awareness of the "context in which they occur."

Entering the 2020s, modern users no longer view smart devices as mere tools but as integral parts of their lives. Smartphones, homes, and furniture are expected to exist not merely as "objects that perform functions" but as "companions that understand personal rhythms and emotions." The true evolution of smart homes should therefore move beyond technological convenience toward expanding user experience and emotional depth. Against this backdrop, we propose the "House–Human Relationship" as a new core value of the smart home system.

2) House-Human Relationship

In the past, the house was a physical space and a base for daily living. In modern society, however, it has transformed into more than just a place of residence — it has become a central space for psychological stability, identity, and emotional recovery. With the rise of remote work, personalized services, and generative AI, the home is increasingly evolving into an interactive interface between users and their environment.

Despite this evolution, most current smart homes still fail to consider emotional connection. Emotional states such as fatigue, happiness, loneliness, or tension are not reflected in how systems operate. The paradigm of user experience (UX) is shifting from convenience to empathy. Today's technology focuses less on "what it can do" and more on "how well it understands me."

When a user says "I'm tired," the goal is not merely to respond with "You must be exhausted," but to dim the lighting to reduce eye strain, play soft music to relieve tension, and create an emotionally responsive environment. The home should provide a sense of emotional feedback — transforming itself from a neutral space into a caring and attentive companion.

The house–human relationship thus evolves from a one-way model, where humans set and control the home, into a mutual learning and adaptive relationship in which the home observes and learns about the user. Through repeated interactions in which the home observes, remembers, and reacts to the user's emotions and patterns — and the user, in turn, grows to trust and feel affinity toward the home — the system gradually develops its own persona. This evolution allows users to perceive their homes not as mere objects, but as beings they live together with.

3) Advancement of Generative AI and Self-Supervised Learning

The rise of generative AI and self-supervised learning has opened the possibility for smart home systems to evolve beyond simple data collection and response into autonomously learning and evolving entities. Large language models, in particular, have reached a level where they can understand emotions, intentions, and context from human voice and dialogue, while reinforcement learning helps optimize their behavioral decision-making.

Furthermore, recent AI services are developing not merely in functionality but toward having distinct personalities. Examples such as ChatGPT, Replika, and Character_AI demonstrate how AI systems can learn users' conversational habits, information needs, and behavioral patterns — generating personalized responses unique to each user. These represent early examples of emotional interaction with AI, illustrating how artificial intelligence can be perceived as a relational and empathetic presence rather than a purely functional one.

B. Problem Statement

1) Limitations of Current Smart Home Systems

Most current smart home systems are designed based on a command-driven architecture, in which device operations are primarily governed by event–trigger rules. This approach results in a structure that reacts to individual device-level inputs rather than managing the home as an integrated environment.

Consequently, the system struggles to process multi-variable contextual conditions—for example, the combination of indoor temperature, humidity, and lighting. As a result, when the temperature is high and the humidity is elevated, the system may activate the air conditioner but fail to coordinate with the dehumidifier, leading to inefficient or excessive energy use.

This limitation arises from the lack of a centralized hub or variable device capable of consolidating these factors into a unified decision-making framework. In essence, the actions of one device are not semantically linked to others, and the system cannot preserve or reference contextual information in subsequent interactions.

Another inherent limitation of command-based architectures lies in their inability to understand user context and emotion. Statements such as "I'm a bit tired today" do not correspond to explicit commands directed toward a particular device, and therefore elicit no system response. Moreover, identical phrases can have different meanings depending on context—saying "It's hot" in summer differs significantly

from saying the same in winter, or expressing "It's cold" at 1 p.m. versus at 3 a.m.—yet current systems fail to distinguish these nuances.

This issue originates from the reliance on speech-to-text conversion followed by purely textual analysis, which strips away emotional and situational cues. Additionally, incorporating multimodal contextual factors such as location, time, schedule, lifestyle patterns, recent behavior, and weather conditions requires a more sophisticated computational framework. This makes it difficult for existing systems—designed for lightweight, rule-based operation—to process emotional or contextual information while maintaining low hardware and processing overhead.

2) Information Deficiency

Although current smart home systems have access to a vast amount of user data, they lack the ability to interpret this information contextually or utilize it continuously. Sensors and IoT appliances collect enormous volumes of environmental and behavioral data; however, such information is used primarily for real-time reactions rather than for long-term learning or personalization.

As a result, the system must respond to each user action as if it were new, without reference to accumulated experience, making it impossible to form an adaptive or relational understanding of the user over time.

In most existing architectures, event logs and state data are retained only for a predefined short duration and are subsequently deleted. Consequently, if the system fails to learn from these data within that limited window, the information is never integrated into the AI's internal state, and thus has no influence on its future behavior.

Moreover, because each device typically stores its data on separate, vendor-specific servers, the system as a whole cannot consolidate relationships across devices or evolve a coherent, unified model of the household. This fragmented and short-term data management leads to a fundamental absence of memory, context, and continuity, preventing the smart home from developing a truly personalized and evolving intelligence.

3) Lack of Awareness toward Personal Management Systems

The development of smart home technologies has so far focused primarily on the automation of device control. As a result, most smart home systems operate from a device-centered perspective rather than a human-centered one. Current platforms recognize the user merely as the issuer of control commands, not as an active participant whose physical and emotional states influence the home environment.

Consequently, human-centered factors such as emotion, condition, daily rhythm, and behavioral patterns are rarely considered as core variables in system design. This structural limitation causes smart homes to provide only functional convenience, failing to deliver the psychological stability and lifestyle management that users increasingly expect from intelligent living spaces.

Such a limitation prevents the smart home from evolving into a Personal Management Platform—a system that not only automates tasks but also understands, supports, and manages the user's well-being. For smart homes to mature into truly intelligent systems, they must recognize the user as more than a source of commands—as a subject of management and learning, whose behavioral and emotional data continually shape the home's adaptive intelligence.

C. Solution

1) Persona-Based User Understanding and Relationship Formation

The most fundamental issue with current smart home systems is that they recognize the user merely as a command generator. To address this, we introduce an AI Persona System designed to learn from and remember all interactions with the user.

The persona continuously evolves based on the user's preferences, lifestyle patterns, conversational style, and emotional expression. When a user says, "I'm having a tough day," the system doesn't just process that statement. Instead, it comprehensively considers the environmental settings the user preferred in similar situations, the corresponding time of day and weather, and the user's satisfaction afterward.

Unlike traditional systems that delete data after a set period, this approach accumulates all experiences as part of the system's personality and knowledge.

As a result, the user gains a life partner that understands them deeply, without needing to repeat the same settings each time. This forms the core foundation for the smart home's evolution from a simple device-control platform into a Personal Management Platform.

2) Voice-Based Contextual Understanding and Real-Time Communication

When the system detects unusual events in the home—such as a fire alarm or the sound of a fall—it immediately calls the user to confirm the situation. During this call, it analyzes the user's tone, speech rate, and intonation to determine whether a real emergency is occurring.

Furthermore, the system distinguishes and responds differently to the same phrase—for example, "It's a bit cold"—depending on whether it's spoken with a trembling voice at 3 a.m. or casually at 1 p.m.

This approach goes beyond simple text-based command processing, advancing to a level that integrates the user's emotional state and situational context. Consequently, the system can respond appropriately to implicit expressions like "I'm feeling worn out today," allowing users to create the desired environment through natural conversation rather than precise commands—fulfilling the role of a genuine life companion.

The system leverages LG's existing home cameras for real-time detection and voice call functions. If a camera detects abnormal sounds—such as a fall, glass breaking, or unusual pet behavior—it stores a 10–20 second pre-buffer and calls the user to confirm the situation. The user can immediately respond during the call with "I'm okay" or "I need help," and this feedback continuously improves the model's accuracy.

All recorded data is automatically deleted upon user confirmation, with only the short buffer segment temporarily stored to ensure privacy protection.

3) User-Centric Environment Optimization Through Multi-Dimensional Data Integration

A major inefficiency in current smart homes is that they control temperature, humidity, and lighting on a per-device basis—often leading to contradictions such as the air conditioner and dehumidifier running simultaneously. These systems also tend to react solely to sensor data, regardless of the user's actual comfort level.

By integrating and analyzing biometric data and weather information, the proposed system can understand the user's real condition and recommend appliance operations accordingly.

The model predicting the user's condition doesn't just process the fact that "the temperature is 28°C." Instead, it interprets the context: "the user feels uncomfortable due to 28°C and high humidity while feeling tired." Based on this, it comprehensively adjusts the air conditioner, dehumidifier, and lighting brightness to create an environment optimized for recovery.

The process of confirming the system's predictions with the user enables continuous improvement, overcoming the "single-use data" problem in existing systems.

This signifies a shift from a device-centric to a user-centric paradigm. The decision-making center is no longer which appliance to turn on but what the user needs right now.

As a result, the user experiences an environment optimized for their condition without complex manual settings, while the system operates efficiently without wasting energy.

4) Predictive Lifestyle Management Through Habit Pattern Learning

Existing smart homes rely on fragmented information—such as whether the user is currently home or not—failing to grasp the broader context or continuity of daily actions. The User Routine Tracking System analyzes travel routes, dwell times, and recurring lifestyle patterns to understand the user's full day. It combines this with biometric and weather data to enable predictive lifestyle management.

For example, the system might determine that the user is returning home later than usual, had high travel activity, and shows signs of fatigue based on biometric data.

In response, it proactively creates an environment for recovery by dimming the lights, lowering the indoor temperature, and preparing calming music before the user arrives.

Conversely, if the system detects an unusual outing pattern on a weekend morning, it suggests environmental settings suitable for an active day.

This establishes a system that manages and cares for the user's overall life, moving beyond simple automation. It transitions from a reactive structure (event occurs $\rightarrow$ immediate response) to a proactive one (pattern learning $\rightarrow$ situation prediction $\rightarrow$ preemptive suggestion).

Consequently, the user experiences having what they need prepared even before issuing a command, and the smart home functions as a true Personal Management Platform.

Moreover, long-term pattern learning can detect subtle changes in the user's life rhythm, offering meaningful insights for health management and lifestyle stability.

D. Research on related software

1) LG ThinQ

LG ThinQ is an integrated smart home platform that connects and controls all LG appliances within the household through a centralized hub architecture. It provides the highest level of interoperability and system stability within the LG ecosystem.

In particular, the ThinQ API enables precise monitoring and control of appliance states such as those of washing machines, air conditioners, and air purifiers. Furthermore, the UP System adopts a modular design that supports continuous enhancement of appliance functionality through over-the-air (OTA) updates, allowing devices to evolve even after deployment.

These technical features constitute a core operational foundation for this project, enabling the behavioral policies generated by the AI persona to be executed reliably at the appliance level. In other words, LG ThinQ serves as the most essential interface that ensures the accurate and dependable realization of the persona's decisions within the physical smart home environment.

2) Samsung SmartThings

Samsung SmartThings is a representative multi-brand smart home platform that enables users to control devices from various manufacturers within a unified application environment. At the hub level, it employs Edge Driver technology to support local device control, thereby minimizing network latency and instability while reducing dependence on cloud processing.

In addition, its Energy Management System provides real-time visualization and control of power consumption for individual appliances, achieving a high level of completeness in terms of energy efficiency management.

This high degree of interoperability and data integration capability makes SmartThings particularly effective for this

project, which aims to manage and unify appliances from multiple brands under a single policy framework.

Moreover, since the SmartThings ecosystem is natively integrated with the Matter standard, it provides a strong infrastructural foundation for expanding the proposed system's policies and control mechanisms into a universally compatible smart home environment.

3) Google Home (Assistant)

Google Home is Google's smart home platform, developed through the integration of Nest devices and Chromecast, and is characterized by its deep connectivity with the broader Google ecosystem—including Google Account, Calendar, and Maps. Through this contextual integration, the platform can directly incorporate the user's schedule, location, and habitual behaviors into automated routines and scenario generation.

In particular, Google Assistant's advanced speech recognition and natural language processing (NLP) technologies have reached a level where the system can not only interpret explicit commands but also understand conversational expressions in context and translate them into corresponding automated actions.

This natural language–based automation mechanism closely aligns with the conversational engine structure of our proposed system, serving as a direct reference for designing models that convert dialogue-based instructions into policy-level inputs. Furthermore, by linking with calendar and location data, Google Home can provide valuable contextual and emotional cues, enhancing the system's ability to accurately perceive the user's current state and situational context.

4) openHAB

openHAB is an open-source home automation hub designed to minimize dependence on cloud infrastructure, specializing in local control and on-premise automation. Because it can be executed directly on a user's own server or single-board computer, openHAB allows data to be securely stored within the household, offering strong advantages in terms of data privacy and user sovereignty.

At the same time, it maintains compatibility with major cloud-based platforms such as Google Assistant, Amazon Alexa, and Apple HomeKit, thereby supporting a flexible hybrid operating environment.

Currently, openHAB provides integration with over 3,000 global brands and devices, including Qualcomm AllPlay, Android TV, BenQ, and AIRTHINGS. In addition to its extensive interoperability, the platform enables local learning and inference, making it a valuable experimental base for testing reinforcement learning (RL) models or persona-driven policy frameworks within a controlled, privacy-preserving environment.

This combination of openness, scalability, and on-device intelligence positions openHAB as a practical research platform for developing and validating next-generation smart home systems.

5) Apple HomeKit

Apple Home is a smart home platform deeply integrated with the iOS ecosystem, built upon an architecture that prioritizes security and privacy above all else. Through the HomeKit protocol, only certified and authenticated devices are permitted to connect to the network, ensuring a highly reliable and secure environment.

More recently, Apple Home has adopted support for the Matter standard and Thread networking, significantly enhancing the stability and interoperability of low-power devices such as sensors and lighting systems.

In addition, Apple's characteristic consistent iOS user experience (UX) and the seamless integration of the Siri voice assistant provide users with high accessibility and an intuitive automation setup process. These features directly align with our project's design philosophy, which emphasizes on-device learning and local data protection.

In particular, HomeKit's local inference architecture serves as a concrete reference model for developing the project's on-device persona intelligence strategy, where personalized behaviors and contextual reasoning are executed securely within the user's own environment.

6) Amazon Alexa

Amazon Alexa is a voice-assistant platform built around smart speaker technology and is currently one of the most widely adopted voice interfaces for smart home ecosystems. Alexa offers thousands of extensions (Skills) and automation scenarios (Routines) that enable a broad range of functions, including appliance control, music playback, and information retrieval.

In recent developments, Alexa has integrated large language model (LLM)-based natural language routine generation, allowing users to configure automation simply through conversational expressions such as "When this happens, do that."

This advancement in voice user experience (UX) and affective language processing represents one of the most mature commercial implementations of an emotionally responsive home environment. In this regard, Alexa serves as a practical reference for the vision of our system—a home that understands and responds to the user's emotions and contextual needs, bridging natural conversation with intelligent, adaptive automation.

7) Matter

Matter is a global standard protocol designed to ensure interoperability among smart home devices, allowing them to operate seamlessly regardless of brand or platform.

By utilizing Thread networking technology, Matter enables mesh communication among low-power devices such as sensors and switches, while its simplified commissioning process enhances both the scalability of smart home environments and the consistency of user experience.

For the proposed system, Matter serves as a foundational technological framework that ensures the persistence and portability of learned behaviors. When the AI persona generates automation scenarios based on its learned behavioral policies, Matter allows these policies to be transferred and maintained across newly added or replaced devices.

This interoperability provides the essential infrastructure for realizing a continuously evolving and universally compatible intelligent home environment.

8) Hume AI

Hume AI is an affective artificial intelligence research company that quantifies users' emotional states by analyzing subtle variations in vocal tone, speed, intensity, and prosody. Beyond simple positive–negative sentiment classification, Hume AI's models identify underlying vocal patterns to estimate complex emotional indicators such as stress, fatigue, and arousal levels.

Furthermore, the company's technology can infer emotional states using microphone input alone, without requiring visual data from cameras, making it particularly effective and privacy-compliant in environments where visual information collection is sensitive or restricted.

9) Replika

Replika is an emotionally adaptive chatbot that continuously learns from users' conversational data to develop individualized personalities and emotional responses. By analyzing each user's linguistic patterns, emotional expressions, and conversational history, Replika maintains relational consistency while gradually evolving its persona over time.

This architecture provides a direct conceptual foundation for the design of smart home agents that move beyond simple command execution—enabling systems that continuously optimize their behavioral policies and response styles through ongoing user feedback and long-term interaction.

10) Affectiva

Affectiva is a pioneer in emotion recognition (Emotion AI) technology that analyzes facial expressions and vocal cues to interpret human emotions with high precision.

Its research and commercial applications span multiple industries—including automotive systems, robotics, and advertising analytics—focusing on designing interactive systems that adapt their responses based on users' emotional states.

This approach offers valuable insights for the development of multimodal persona architectures in smart home agents, enabling them to interpret and respond to emotional signals not only from voice data but also from visual inputs, thereby enhancing the depth and empathy of human–AI interaction.

11) Twilio

Twilio provides a comprehensive cloud-based voice communication infrastructure through its Programmable Voice API, enabling applications to initiate, receive, and manage phone calls programmatically.

The platform supports multiple communication protocols, including WebRTC, SIP (Session Initiation Protocol), and PSTN (Public Switched Telephone Network), allowing the implementation of reliable bidirectional voice communication between the home AI system and the user—whether the user calls the home AI or the home AI initiates a call to the user.

Twilio's API further supports real-time speech recognition (STT), DTMF input detection, and IVR (Interactive Voice Response) flow control, while the use of TwiML (Twilio Markup Language) allows developers to model detailed call scenarios directly at the code level.

When integrated with the Model Context Protocol (MCP), this architecture enables the AI model to dynamically manage call flows, process real-time call events as contextual inputs, and perform conversational decision-making within an ongoing voice interaction.

12) Vonage

Vonage is a global communication platform centered on its Voice API, supporting bidirectional voice streaming through WebRTC and SIP-based protocols. The Vonage API allows real-time transmission of audio streams to external AI endpoints during a call, making it highly suitable for AI-integrated call processing tasks such as speech recognition, emotion analysis, and intent understanding.

In particular, the Voice Connectors feature enables direct linkage with the Model Context Protocol (MCP) via a WebSocket interface, allowing the AI model to interpret user utterances in real time and generate immediate, context-aware responses during an ongoing conversation.

For instance, in a Vonage Voice API–enabled call, if the user's voice tone rises, the MCP can recognize this as a change in emotional state, triggering an emergency response mode, or prompting the conversational persona to respond in a calmer, more empathetic tone.

In this way, Vonage transforms call audio from a mere sound stream into a contextual communication medium, providing a crucial technological foundation for implementing the persona-based bidirectional communication framework proposed in this project.

13) Google Maps Timeline

Google Maps Timeline is a location-tracking service that records users' movement paths, visited places, transporta-

tion modes, dwell times, and travel distances based on a combination of GPS and Wi-Fi signal data. By integrating additional contextual information from Google's web and app activity as well as photo metadata, the system constructs a comprehensive log of the user's daily mobility and behavioral patterns.

This multi-source dataset provides valuable contextual signals that, when combined with biometric and weather data, significantly enhance the accuracy of lifestyle and condition analysis models—enabling fine-grained estimation of user activity levels, fatigue, and rest cycles.

Within the proposed system, the location and activity data from Google Maps Timeline are incorporated as contextual inputs for adaptive decision-making. For instance, if the system detects higher-than-usual mobility or a delayed return home, it can infer potential fatigue and proactively suggest personalized environmental adjustments, such as "Would you like to dim the lights?" or "Shall I activate sleep mode?"

To ensure privacy, raw location data are anonymized and aggregated into grid-based representations stored locally on the device, while users maintain full control over the scope and duration of timeline recording and sharing preferences.

E. Contributions

This project presents HARU, an adaptive smart-home automation system that integrates wearable physiological sensing, multi-modal context awareness, and LLM-based reasoning. The major contributions of this work are summarized as follows:

1) HRV-Based Real-Time Wellness Estimation:

We design a lightweight yet effective fatigue–stress scoring model based on Apple Watch HRV (RMSSD, SDNN), heart rate, sleep stages, and activity intensity. Unlike rule-based systems, HARU incorporates temporal patterns (TimeSlot) and environmental cues to refine wellness estimation.

2) LLM-Driven Natural-Language Home Automation:

We propose a novel pipeline where GPT-based language models interpret the user's physiological state and contextual factors to generate natural-language appliance policies. These policies are parsed into structured control commands for lighting, HVAC, and humidity systems.

3) Context-Aware Multi-Modal Integration:

HARU unifies diverse data streams—including weather forecasts, GPS-based indoor/outdoor inference, and historical appliance usage—into a cohesive decision-making framework. This enables proactive and personalized adjustments beyond traditional sensor-triggered automation.

4) Adaptive User Preference Learning:

We develop an iterative preference-learning mechanism that updates user-specific comfort profiles (e.g., preferred temperature, lighting level) based on historical behavior and fatigue level. This allows HARU to autonomously evolve toward personalized routines.

5) End-to-End Smart Health–Home Ecosystem:

We implement a fully operational end-to-end prototype across iOS/watchOS, FastAPI backend, Supabase PostgreSQL, OpenAI GPT, and Sendbird. This validates the feasibility of a health-driven smart-home system in real-world environments.

II. Requirement

A. User Management

1) Sign Up

The registration process requires the following information:

- Mobile number: Verified through carrier authentication and later used as the login ID
- Password: Must be at least 8 characters long and include three of the following: uppercase letters, lowercase letters, numbers, and special characters. When requirements are met, indicators turn green; when unmet, they turn red. Password input is masked for security
- Name: Required field; used as the default nickname upon first login and for ID recovery
- Date of birth: Required field; triggers birthday pop-up notification annually and used for ID recovery
- Email: Used for account recovery, additional authentication, and receiving important notices

2) Sign In

The system supports two authentication methods:

- Local login: Users enter their ID (mobile number) and password. If credentials match, the system redirects to the main page; otherwise, a "Member does not exist" popup is displayed.
- SNS login: Users can authenticate via Apple, Google, Kakao, or other social platforms. After consent, the system connects to the same user profile based on mobile key linkage.

B. Device Management

1) Register Home Appliance

The system provides two registration methods:

- QR scan (requires camera permission)
- Device search via Wi-Fi/BLE scan or manual input of model and serial number

Import from LG ThinQ: After OAuth consent, the system imports registered appliances, room layout, and smart routines. Imported data is used exclusively for control and automation suggestions.

C. Data Collection and Privacy

1) Data Sources and Permissions

The system collects data from multiple sources with explicit user consent

- Biometric and activity data: Collected from wearable devices and phone sensors, including heart rate, activity levels, sleep patterns, and step counts. Data collection is minimal and requires explicit consent.
- Weather and environment data: Obtained from external weather APIs and indoor sensors measuring temperature, humidity, CO2 levels, and other environmental factors.
- Required permissions: The system requests notifications, location (for weather data), and health/activity data access. The purpose and retention period for each permission are clearly disclosed to users.

2) RLHF Feedback Loop and Continuous Optimization

These AI-generated suggestions are presented to the user in a card interface, allowing quick responses such as Run now, In 30 min, or Skip today. All user interactions feed back into a Reinforcement Learning from Human Feedback (RLHF) loop, where approval rates, adjustment patterns, and contextual variables are used to continuously refine prompt templates, decision thresholds, and model confidence levels. Over time, the system evolves into a personalized predictive assistant that learns from each interaction and aligns closely with the user's preferences and habits.

D. Apple Watch Integration

1) GPS and Location Awareness

The Apple Watch integration enables real-time tracking of the user's geographical context through GPS data. By continuously monitoring location, the system determines whether the user is near home or traveling. This spatial awareness allows the agent to adapt its behavior — for instance, adjusting home device readiness when the user is approaching or providing location-based reminders. Additionally, accumulated GPS traces are visualized on an adaptive map, supporting routine pattern analysis such as daily commuting routes, time spent outdoors, and regional activity trends.

2) Health Data Synchronization

Through Apple's HealthKit framework, the agent securely retrieves physiological and activity data including heart rate, sleep duration, step count, and exercise sessions. These metrics are continuously analyzed to infer the user's physical condition, stress levels, and lifestyle patterns. Based on the analysis, the system can generate personalized insights — such as recommending rest after elevated stress detection or adjusting environmental conditions (e.g., temperature or lighting) following high activity levels.

3) Behavioral Mapping and Wellness Insights

By combining GPS-derived mobility data with real-time health indicators, the agent constructs a comprehensive behavioral map of the user's daily life. This integration allows it to identify correlations — for example, linking reduced step count with poor sleep quality or detecting stress patterns associated with long commutes. Ultimately, the Apple Watch module transforms raw sensor data into actionable wellness intelligence, enhancing personalization, safety, and long-term health awareness.

E. Personalization

1) Persona Configuration

The personalization framework is centered on the concept of an AI persona configured during the onboarding phase. Upon initial setup, the user selects the agent's tone and communication style — for instance, friendly, professional, or concise — and provides basic profile information such as name, date of birth, occupation, sleep–wake cycle, and weekly exercise frequency. This data directly influences the tone of notifications, phrasing of suggestions, and communication style in voice or text interfaces.

2) Adaptive Personalization and Behavioral Learning

As the system observes real user behavior and feedback, it gradually increases personalization depth. For example, if a user consistently approves "cooler settings after exercise," the agent autonomously learns to trigger air conditioning suggestions post-workout. Similarly, recurring actions such as "dim lights at night" become embedded as personal automation rules, minimizing the need for repeated confirmations.

3) Safety and Transparency

High-risk or low-confidence actions (e.g., oven or gas control) always require explicit user confirmation. Every automation remains overrideable at any time, ensuring user safety, transparency, and trust. Through continuous learning, the agent transitions from a static rule-based assistant to an adaptive, user-aware smart companion.

F. Proactive Automation

1) Autonomous Appliance Adjustment

Instead of waiting for user confirmation, the system autonomously adjusts appliances based on real-time user condition analysis. When the AI detects significant fatigue, elevated stress, or temperature discomfort, it immediately configures connected devices such as lighting, air conditioning, or air purifiers to match the predicted comfort range without requiring prior user input. This enables seamless environmental adaptation that aligns with the user's physiological and contextual state.

2) User Feedback and Re-adjustment

After an automatic adjustment is applied, users can freely modify or reject the new settings. For instance, if the user changes the air conditioner temperature or disables lighting adjustments, the system interprets this as a negative feedback signal. All such interactions are recorded as behavioral feedback data and used to update the internal preference model.

3) Adaptive Learning Loop

Each user correction or rejection is treated as a learning instance. The reinforcement loop continually analyzes these signals to fine-tune decision thresholds, preferred control parameters, and contextual mappings. Over time, this process transforms the automation from a static ruleset into a personalized predictive controller that autonomously optimizes appliance behavior based on accumulated feedback and evolving user preferences.

4) Safety and Transparency

For safety-critical or high-impact actions (e.g., oven, gas, or electrical control), the system enforces reconfirmation or multi-factor authentication. All adjustments remain transparent and reversible; users can override or reset automated actions at any time, ensuring reliability, trust, and control in the automation process.

G. Communication System

1) Communication Channels

Push notifications serve as the default communication method, with SMS and email available as alternatives. For urgent alerts or complex issues, the system can initiate voice calls to establish direct agent-user phone connection through Voice call (VoIP/ARS). In addition, Users can configure do-not-disturb hours, priority levels, guardian designation, and language preferences through contact settings.

H. Security and Compliance

1) Privacy, Security, and Consent

The system implements comprehensive privacy and security measures:

Data handling: Minimal data collection principle is enforced, with encryption applied during both transmission and storage. Personally identifiable information (PII) is separated from telemetry data.

ThinQ integration: Minimal permission scope is requested from LG ThinQ, with prior disclosure of all retrieved items.

User rights: Users can download or delete their data, unlink ThinQ integration, and deactivate their account at any time.

Audit trail: Every automation execution and contact attempt is logged for audit purposes.

2) Reliability, Error Handling, and Offline Support

The system implements robust error handling mechanisms:

Device availability: Offline devices are detected and indicated, with recovery guidance provided. Non-critical commands are queued for later execution.

Conflict resolution: When ThinQ and local status information conflict, the system presents a resolution prompt or applies a single-source policy.

Schema compatibility: Core controls are limited to finalized schemas, with older values displayed as legacy indicators.

III. Development Environment

A. Choice of Software Development Platform

1) Development Platform

a) macOS Development Environment

Fig. 1: macOS

macOS served as the primary development environment for iOS and watchOS applications. All implementations were carried out using **Xcode 16.0** on **macOS Sonoma 15.0**, running on an Apple MacBook Pro (M2, 16 GB RAM). This platform ensured full compatibility with Apple's SDKs and provided integrated tools for debugging, WatchKit simulation, and real-device testing. The macOS environment also enabled seamless SwiftUI preview and build automation through Apple's native developer ecosystem.

b) Windows Environment for Backend Testing

Fig. 2: windows11

Windows 11 was utilized as a secondary development environment to verify backend communication and data exchange between the FastAPI server and client devices. Docker containers were configured to host FastAPI and PostgreSQL instances, providing a consistent testbed across both macOS and Windows. This setup ensured

that the backend could be validated independently from the Apple development environment and maintained cross-platform compatibility.

c) iOS Platform (iPhone)
The iOS application served as the main interface for collecting health, GPS, and contextual data. Developed using **Swift 5.10** and **SwiftUI**, it integrates Apple frameworks such as `HealthKit`, `CoreLocation`, and `MapKit`. Testing and optimization were performed on an **iPhone 14 (iOS 18.0)** device, ensuring stable background synchronization and low-latency data transfer through the `WCSession` framework. The iOS platform also enabled direct visualization of health metrics and movement trajectories within the app's dashboard.

d) watchOS Platform (Apple Watch)

Fig. 3: WatchOS

The watchOS component was responsible for collecting real-time biometric and activity data through the Apple Watch sensors. An **Apple Watch SE (2nd generation, watchOS 10.1)** was paired with the iPhone for continuous synchronization of heart rate, HRV, and exercise sessions via the `HealthKit` and `WatchConnectivity` frameworks. The lightweight watchOS design enabled background data capture with minimal power consumption, ensuring continuous monitoring without user intervention. All collected data were serialized and securely transmitted to the iOS host app, forming the foundation of the system's health monitoring pipeline.

2) Language/Framework

a) Docker
Docker is an open-source containerization platform that enables developers to package applications and their dependencies into lightweight, portable containers. By isolating software components from the host system, Docker ensures consistent performance across development, testing, and production environments. It simplifies deployment pipelines and promotes scalability through container orchestration tools such as Docker Compose

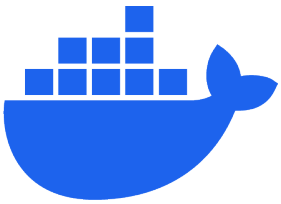

Fig. 4: Docker

and Kubernetes. Docker's image-based architecture allows reproducible builds, rapid scaling, and seamless collaboration among distributed teams.

b) PostgreSQL

Fig. 5: PostgreSQL

PostgreSQL is a widely used open-source relational database management system based on Structured Query Language (SQL). It provides reliable data storage, indexing, and querying capabilities for web and enterprise applications. PostgreSQL ensures data integrity through ACID compliance and supports multi-user access with transaction control and authentication mechanisms. In this project, PostgreSQL was used to manage structured data efficiently, offering strong scalability and compatibility with Docker-based deployments.

c) Mermaid

Fig. 6: Mermaid

Mermaid is a JavaScript-based diagramming and visualization tool that converts text definitions into dynamic charts and diagrams. It supports flowcharts, sequence diagrams, entity-relationship diagrams (ERD), and Gantt charts, enabling engineers to describe complex architectures in a simple, markdown-style syntax. Within this project, Mermaid was employed to visualize database schemas and backend architecture, enhancing documentation clarity and maintainability.

d) Amazon Web Services
Amazon Web Services(AWS) is a comprehensive suite of cloud computing services provided by Amazon. It

Fig. 7: Amazon Web Services

offers scalable infrastructure for computing, storage, networking, and machine learning applications. AWS enables developers to deploy and manage containerized applications using services such as Cloud Run, Compute Engine, and Kubernetes Engine. In this project, AWS was used to host backend APIs and manage containerized environments with high scalability and reliability.

e) FastAPI

Fig. 8: FastAPI

FastAPI is a modern, high-performance web framework for building APIs with Python 3.7+ based on standard type hints. It supports asynchronous I/O and automatic request validation using Python type annotations, making it both efficient and developer-friendly. FastAPI automatically generates interactive API documentation using OpenAPI and Swagger UI, improving testing and maintainability. In this project, FastAPI served as the backend framework responsible for connecting user interfaces, databases, and external services such as the Korea Meteorological Administration (KMA) API.

f) KMA OpenAPI

Fig. 9: Korea Meteorological Administration

The KMA OpenAPI provides official weather data released by the Korea Meteorological Administration. It offers various endpoints including ultra-short-term observation, ultra-short-term forecast, and village forecast, which provide weather information at intervals of 10 minutes to 1 hour based on a 5 km × 5 km grid (nx, ny) system. This API enables real-time access to weather conditions such as temperature, humidity, precipitation, wind speed, and sky condition across South Korea. In this project, the KMA API was integrated into the backend system to collect localized weather data, which was later combined with user health and location information to enhance contextual analysis and AI-driven stress prediction.

g) Swift / SwiftUI

Fig. 10: Swift / SwiftUI

Swift is Apple's modern programming language optimized for safety, performance, and expressiveness. In this project, Swift 5.10 and SwiftUI were used to implement both the iPhone and Apple Watch interfaces. SwiftUI's declarative syntax simplified reactive UI updates in response to changing health and GPS data, while seamless state synchronization between watchOS and iOS enabled real-time visualization of sensor readings. The combination of Swift and SwiftUI improved maintainability and reduced boilerplate code, allowing faster iteration and debugging throughout the development cycle.

B. Software in Use

1) Apple Fitness+ / Apple Fitness (Apple)

Apple Fitness+ and its companion Apple Fitness app provide structured workout programs, personalized recommendations, and motivational tracking features based on data collected through the Apple Watch. The service offers real-time visualization of workout metrics—such as heart rate, calories burned, and activity ring progress—and synchronizes seamlessly across iPhone, iPad, and Apple TV. However, its core focus remains on promoting exercise engagement rather than integrating multi-dimensional data sources like GPS trajectories or meteorological context. In contrast, the proposed system expands upon this foundation by combining health, spatial, and environmental data to support adaptive and contextual behavior analysis.

2) Fitbit-based Health Tracking Platforms

Fitbit's health tracking ecosystem is among the most established wearable platforms, enabling users to monitor daily activity, sleep, and heart rate through connected devices and a cloud-based mobile interface. Its APIs and SDKs facilitate health research, allowing correlation studies on movement, sleep quality, and stress management. Nonetheless, Fitbit's data model primarily concentrates on individual domains—such as physical activity or sleep—without capturing contextual environmental data. Our system advances this approach by integrating real-time GPS and weather data, constructing a unified schema that

relates health trends with environmental and behavioral factors.

3) AI-driven Wearable Data Recommendation Systems

Recent AI-powered fitness systems, such as the "Privacy-Preserving Personalized Fitness Recommender System (P3FitRec)," employ deep learning to analyze multi-sensor data and generate personalized exercise recommendations. These systems extract temporal dependencies from physiological signals like heart rate and activity duration, while preserving user privacy through federated learning and encrypted communication. However, their focus largely remains on recommendation accuracy rather than contextual integration. The present project enhances this paradigm by incorporating geolocation, environmental, and biometric data streams into a single inference pipeline, enabling the analysis of stress and wellness dynamics beyond exercise routines.

4) Figma

Fig. 11: figma

Figma is a collaborative cloud-based design platform optimized for UI/UX design, prototyping, and team collaboration. In this project, Figma was used to design and prototype both the iPhone and Apple Watch interfaces with pixel-perfect precision. Figma's component system and auto-layout features enabled consistent design patterns across different screen sizes, while real-time collaboration capabilities allowed seamless feedback exchange between designers and developers throughout the design process. The combination of Figma's prototyping tools and developer handoff features improved design-to-code accuracy and reduced implementation time, allowing faster validation of user interactions and visual refinements during the development cycle.

5) Visual Studio Code

Visual Studio Code is a lightweight source code editor optimized for cross-platform development, extension customization, and multi-language support. In this project, VSCode was used for editing configuration files, writing documentation in Markdown, and managing Git operations with an intuitive interface. VSCode's extensive extension marketplace and integrated terminal enabled streamlined

Fig. 12: Visual Studio Code

workflows for linting, formatting, and version control without switching contexts, while workspace settings allowed consistent coding standards across team members. The combination of VSCode's IntelliSense and multi-cursor editing improved productivity for repetitive tasks and code navigation, allowing efficient handling of non-Swift files and supporting scripts throughout the development cycle.

6) Xcode

Fig. 13: Xcode

Xcode is Apple's integrated development environment optimized for building, testing, and debugging iOS, watchOS, and macOS applications. In this project, Xcode was used as the primary IDE for Swift development, Interface Builder integration, and device-specific testing on iPhone and Apple Watch simulators. Xcode's Instruments profiling suite and live preview functionality enabled real-time performance monitoring and UI debugging without repeated builds, while built-in HealthKit and CoreLocation frameworks simplified integration with native sensor APIs. The combination of Xcode's code completion and refactoring tools improved development speed and code consistency, allowing rapid prototyping with immediate visual feedback throughout the development cycle.

7) Github

Fig. 14: Github

GitHub is a cloud-based version control platform optimized for collaborative software development, code review, and CI/CD integration. In this project, GitHub was used to manage source code repositories, facilitate pull request reviews, and automate testing pipelines for Swift codebases. GitHub's branching strategy and merge conflict resolution enabled parallel feature development without disrupting the main codebase, while Actions workflows automated build verification and deployment processes for both iOS and watchOS targets. The combination of GitHub's code review tools and issue linking improved code quality and traceability, allowing rapid iteration with confidence and maintaining a clean commit history throughout the development cycle.

8) Jira

Fig. 15: Jira

Jira is a project management platform optimized for agile development, issue tracking, and sprint planning. In this project, Jira was used to manage user stories, track bugs, and coordinate development tasks across iOS and watchOS implementations. Jira's customizable workflow and sprint board visualization enabled clear prioritization of features and real-time progress monitoring, while integration with GitHub allowed automatic status updates based on pull request activities. The combination of Jira's reporting dashboards and backlog management improved team velocity and accountability, allowing data-driven sprint planning and faster identification of blockers throughout the development cycle.

9) Notion

Notion is an all-in-one workspace platform optimized for documentation, knowledge management, and team

Fig. 16: Notion

collaboration. In this project, Notion was used to centralize project documentation, meeting notes, and design specifications in a single accessible repository. Notion's flexible database system and nested page structure enabled organized tracking of feature requirements and research findings, while cross-linking capabilities allowed seamless navigation between related documents and resources. The combination of Notion's collaborative editing and commenting features improved knowledge sharing and reduced information silos, allowing team members to stay aligned on project goals and technical decisions throughout the development cycle.

10) OpenAI

Fig. 17: OpenAI

OpenAI's GPT-5 Nano API is a lightweight language model optimized for efficient natural language processing with minimal latency and resource consumption. In this project, the GPT-5 Nano API was integrated to provide intelligent health insights and personalized workout recommendations based on user activity data. The API's low-latency response time and compact model size enabled real-time text generation on-device without compromising battery life, while fine-tuning capabilities allowed customization for health and fitness domain-specific responses. The combination of GPT-4 Nano's natural language understanding and cost-effective pricing improved user engagement with contextual suggestions, allowing scalable AI-powered features without significant infrastructure overhead throughout the deployment cycle.

11) Overleaf

Overleaf is a collaborative cloud-based LaTeX editor optimized for academic writing, technical documentation,

Fig. 18: Overleaf

and publication-ready typesetting. In this project, Overleaf was used to author the research paper, format technical diagrams, and maintain consistent academic citation standards. Overleaf's real-time collaborative editing and version history enabled simultaneous contributions from multiple authors without file conflicts, while rich LaTeX template library and integrated compiler provided professional formatting for figures, equations, and bibliographies. The combination of Overleaf's track changes feature and comment system improved peer review efficiency and writing quality, allowing seamless revision cycles and faster preparation of camera-ready manuscripts throughout the publication process.

C. Task Distribution

1) Project Manager - Junho Uh

A Project Manager orchestrates the development and delivery of the smart health-home integration system within specified constraints of scope, time, and budget. Their role involves developing comprehensive project plans, establishing critical milestones, and implementing Agile methodologies to ensure efficient delivery. They coordinate resource allocation and timeline management using Jira for issue tracking, Notion for documentation, and GitHub for version control.

Critical responsibilities include managing deliverables across iOS/watchOS health monitoring, backend infrastructure, LLM-based home automation logic, and smart appliance integration. They ensure seamless coordination between health data collection, AI-driven state prediction, and home environment optimization through connected devices. Their success is measured through project completion metrics, team performance, and the system's ability to accurately predict user states and automatically adjust home conditions for optimal comfort and wellness.

2) Frontend Developer - Donghyun Lim

Frontend developers utilize Swift 5.10 and SwiftUI to create iOS and watchOS applications that bridge health monitoring with smart home control. They design interfaces for real-time health data visualization (heart rate, HRV, sleep patterns) alongside home appliance status and control panels. The development integrates HealthKit for biometric data collection, CoreLocation for spatial context, and custom APIs for communicating with smart home devices.

They implement the WCSession framework for seamless data synchronization between Apple Watch and iPhone, ensuring that health metrics captured on the wearable trigger appropriate home automation responses. The role requires expertise in creating intuitive control interfaces where users can monitor their predicted wellness state and manually override or customize automated home adjustments. They work closely with UI-UX designers to ensure the health-to-home automation workflow is accessible and engaging, while coordinating with backend developers to implement real-time bidirectional communication between health sensors, prediction models, and connected appliances.

3) Backend Developer - Dogyeom Kim, Junho Uh

Backend developers design the database schema and API architecture that connects health data collection, AI prediction models, LLM-based decision making, and smart home device control. They manage a PostgreSQL database analyzing relationships among user health states, environmental conditions, location contexts, and home appliance configurations. Using FastAPI, they create RESTful endpoints that receive health samples and GPS data from iOS/watchOS applications, process them through prediction models, and generate LLM prompts that determine optimal home states.

They integrate OpenAI's GPT API to translate predicted user conditions (stress levels, fatigue, sleep readiness) into natural language commands for controlling temperature, lighting, humidity, and other smart appliances. The backend utilizes Docker for containerization and Google Cloud Platform for scalable deployment, while implementing secure protocols for IoT device communication. They design the feature-engineering pipeline that aggregates time-series health data with environmental factors to feed both prediction models and LLM context, ensuring the system maintains data consistency throughout the health-to-home automation workflow.

4) UI-UX Designer - Donghyun Lim

UI-UX designers, using Figma, determine how the integrated health monitoring and smart home control interface is presented across iPhone and Apple Watch platforms. They create a unified design system that seamlessly blends health data visualization with home automation controls, ensuring users can easily understand their current wellness state and corresponding home environment adjustments.

The design process includes creating mockups that display predicted user conditions alongside automated appliance responses, with intuitive override controls and customization options. Designers must balance information density between health metrics (heart rate, sleep quality, stress indicators) and home status displays (temperature, lighting, air quality) within the limited screen space of wearable devices. They leverage Figma's component system to maintain consistent design patterns across health

monitoring dashboards and appliance control interfaces. Once designs are finalized, they communicate specifications to frontend developers through Figma's developer handoff features, ensuring pixel-perfect implementation of the health-to-home automation user experience.

5) AI Developer - Yeonseong Shin, Junho Uh

AI developers build machine learning pipelines and LLM integration systems that predict user wellness states and generate optimal home automation commands. They collect and preprocess multi-modal data from HealthKit (heart rate, HRV, sleep, activity), CoreLocation (GPS, location patterns), and environmental sensors to extract features for stress prediction and fatigue detection models. The AI workflow implements machine learning algorithms that analyze correlations between physiological patterns and contextual factors, then feeds these predictions into OpenAI's GPT API to generate natural language commands for smart home devices.

They design the integration pipeline where predicted states (e.g., "user is stressed and approaching home") trigger LLM prompts that determine appropriate environmental adjustments (dimmed lighting, cooler temperature, calming music). AI developers train and evaluate models using metrics such as prediction accuracy and user comfort scores, while optimizing the LLM prompt engineering to ensure reliable and contextually appropriate appliance control. They ensure the GPT API's responses are parsed correctly and translated into specific device commands, maintaining low latency between state prediction and home automation execution to provide seamless, anticipatory environmental optimization based on real-time health intelligence.

6) Server Manager - Dogyeom Kim

The Server Manager is responsible for ensuring stable backend deployment, scalable infrastructure, and reliable system operation. They manage the AWS EC2 instance hosting the FastAPI backend, configure NGINX as a reverse proxy, and maintain Docker-based container orchestration.

Their responsibilities include setting up secure connections, monitoring system performance, and configuring logs using system tools such as CloudWatch or Docker logs. They also implement CI/CD pipelines using GitHub Actions, automate Docker image builds, and maintain environment variables and version management across development and production environments.

They ensure the backend remains stable under real-time streaming from iOS/watchOS devices, manage API rate limits for OpenAI and KMA, and monitor failures or latency issues across LLM-based automation tasks. Their work ensures that the HARU system can reliably process biometric streams, execute real-time predictions, and deliver seamless home-automation responses.

IV. App Specification

A. Splash Page Page

Fig. 19: Splash Page

When the application is launched, a splash page is displayed for approximately 1–2 seconds. This prevents a blank loading state from appearing while the application initializes essential components such as session tokens, device bindings, and background synchronization tasks. By presenting a stable visual entry point, the splash page ensures a smooth and aesthetically consistent startup experience.

B. Login Page

1) Login Interface

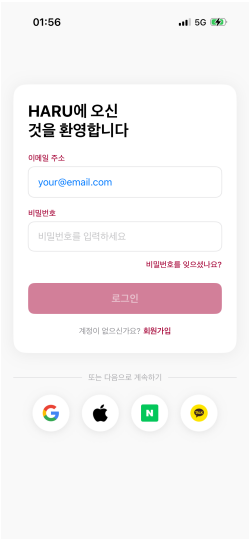

Fig. 20: Login Page

Users can authenticate through multiple login methods, including email–password login, Google, Apple, Naver, and Kakao. Email and password fields include real-time validation feedback, and incorrect formatting prevents login attempts. A “Forgot Password” button directs users to Supabase’s password reset workflow. Upon successful authentication, a user session is created both in the iOS client and the backend, enabling synchronization of health data, devices, and preferences.

C. Sign Up Page

1) Sign Up – Basic Information

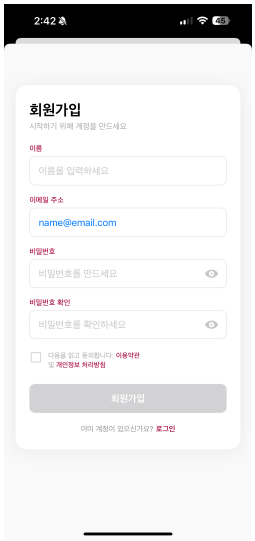

Fig. 21: Signup Page

The Sign Up page collects essential user data, including username, email address, and password. All fields include structured validation to ensure proper formatting and compliance with minimum security requirements.

2) Sign Up – Password Validation

Password requirements include a minimum length and a combination of character types. Visual indicators provide real-time feedback, turning green when conditions are satisfied and red when they are not. The “Sign Up” button becomes active only when all constraints are fulfilled.

3) Sign Up – Consent Agreement

Users must acknowledge the Terms of Service and Privacy Policy. The system prevents account creation unless all required consents are explicitly confirmed.

4) Sign Up – Registration Completion

Once validated, the system creates a Supabase user record and initializes preset configurations on the backend. These include default appliance preferences, persona profiles, and personalized environment parameters essential for downstream inference tasks.

D. Home Page

1) My Home Page

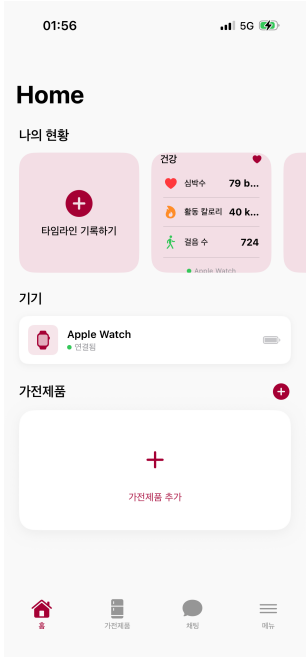

Fig. 22: Home Page1

Fig. 23: Home Page2

The Home page provides an overview of the user’s health and contextual data. This includes the timeline widget for movement summaries, HRV-based wellness indicators, and the currently active AI personas.

2) Connected Devices

Paired devices such as Apple Watch and iPhone are displayed with real-time connection status, battery levels, and last synchronization timestamps.

3) Home Appliances Overview

All registered appliances are displayed as horizontally scrollable cards showing key parameters such as power state, temperature, and humidity. Each card includes quick toggles and navigational access to detailed controls.

E. Appliances Page

1) Appliance Home Page

The appliance home page displays all devices registered under the user’s account. Devices are automatically grouped by type—air conditioner, lighting, air purifier, humidifier, dehumidifier, and television. Each appliance is represented as a card component showing:

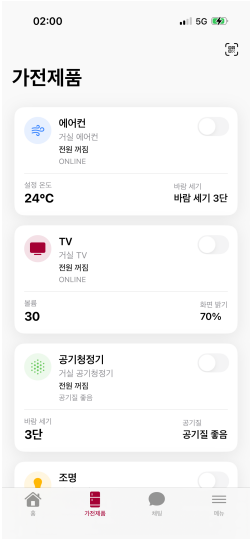

Fig. 24: Appliance Home Page

- **Current Status:** ON/OFF indicator with color-coded visual feedback.
- **Primary Parameter:** e.g., temperature, brightness, or humidity.
- **Room Location:** device placement such as “Living Room” or “Bedroom.”
- **Operation Mode:** e.g., Auto, Sleep, Turbo, or Manual.

Users can perform quick toggles directly on each card for immediate control, while the interface applies optimistic UI updates to maintain responsiveness. A pull-to-refresh action retrieves the latest state from the backend, and tapping on any appliance card opens the detailed control interface.

2) Appliance Detail Page

Fig. 25: Appliance Detail Page

Each appliance type has a dedicated detailed control page that exposes adjustable operational parameters. All settings are synchronized in real time with the backend FastAPI controller. Supported device types and available control functions include:

- **Air Conditioner:** mode (Cooling, Heating, Auto), temperature (16–30°C), and fan speed levels (1–5).
- **Lighting:** brightness (0–100%), color temperature (2700K–6500K), and scene presets (Reading, Relax, Night).
- **Air Purifier:** fan speed control, air quality indicator (PM10, PM2.5), and mode selection (Auto, Turbo, Sleep).
- **Humidifier / Dehumidifier:** humidity target setting and mist/fan intensity.
- **TV:** volume, input source, and page brightness.

Each change triggers an API request to update device states on the backend, and a confirmation response is reflected immediately in the frontend through state binding. In case of communication failure, the system rolls back to the previous state to ensure interface consistency and prevent user confusion.

F. Chat Page

1) General User Conversation

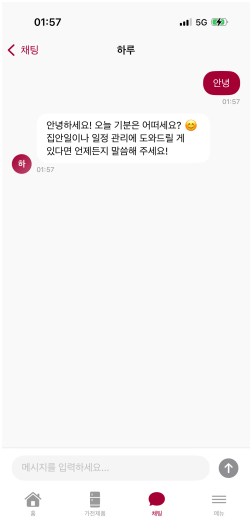

Fig. 26: General User Conversatiol Page

The Chat page lists all AI personas registered by the user. Each persona is displayed with a nickname, adjective tags describing its personality, and the time of the last interaction. When the user selects a persona, a real-time conversation window opens, powered by Sendbird and an LLM-based dialogue engine. In this mode, the user can ask general questions, receive lifestyle or wellness guidance, and engage in free-form conversation without necessarily controlling any devices.

2) Appliance Control via user

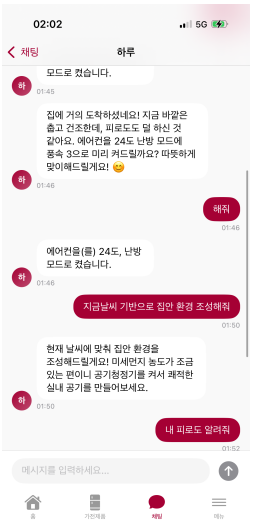

Fig. 27: Appliance Control Via User Chat Page

Beyond casual conversation, the chat interface allows the user to control home appliances using natural language commands. For example, the user may request, “Make the living room cooler” or “Turn off the bedroom lights at 11 p.m.” The LLM interprets the intent, maps it to specific appliance actions (device type, location, target value), and generates a structured control request. This request is transmitted to the backend smart-control API, which updates the corresponding appliance states. Confirmation messages are then returned to the chat window so that the user can verify which devices were changed and how.

3) Trigger-Based Notifications and Suggested Control

G. Timeline Page

The Timeline page visualizes the user’s daily movement combined with physiological metrics. Its main features include:

- GPS polylines representing movement routes,
- Checkpoints with heart rate, HRV, steps, and calorie data,
- Weather annotations such as temperature, humidity, and particulate matter levels,
- Mini-map previews for past records, and
- Start/Stop Recording controls.

This page integrates Apple Watch health data with Core-Location routes for a comprehensive behavioral overview.

H. Persona Page

The Persona page allows the creation and management of AI personas. Users can:

- Add new personas by selecting nickname and character traits,
- Activate up to five personas simultaneously,

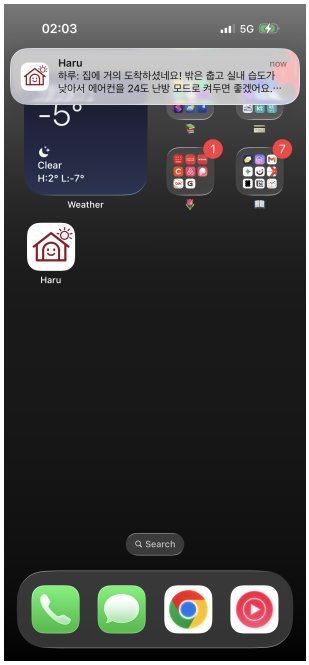

Fig. 28: Trigger-Based Notification

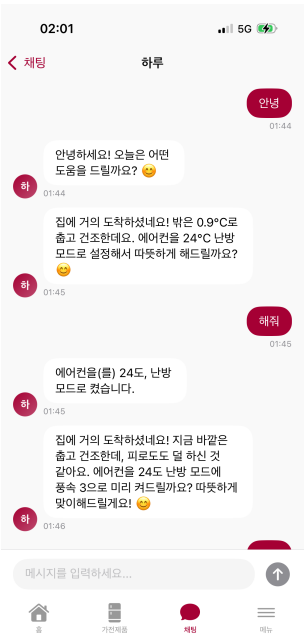

Fig. 29: Suggested Control Chat Page

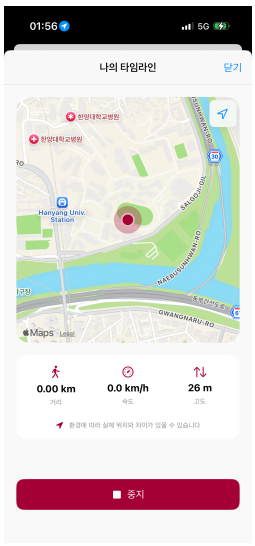

Fig. 30: Timeline Page

- Edit or delete existing personas, and
- Use personas as conversational agents for controlling appliances.

I. Menu Page

The Menu page serves as the settings and profile management hub. It includes:

- User profile details (name, email),
- Account settings and general preferences,
- Font size settings,
- Notification configuration,

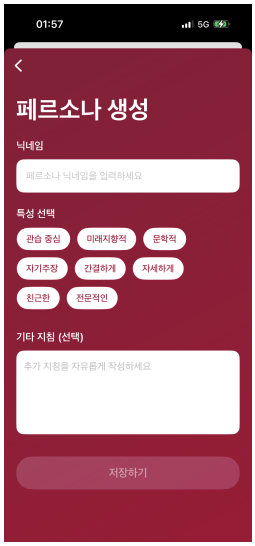

Fig. 31: Persona Page

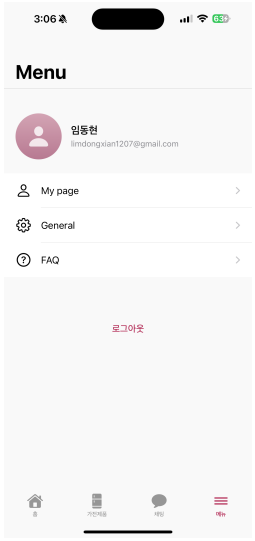

Fig. 32: Menu Page

- Do-Not-Disturb scheduling,
- Auto-tracking toggle, and
- Logout button.

Logging out clears all client-side session data and disconnects active messaging channels.

J. Settings Page

The Settings page provides global configuration options such as:

- Font scaling options,
- Notification settings,
- Do-Not-Disturb settings, and
- Auto-tracking permissions.

User preferences persist across sessions using HARU's settings manager.

Fig. 33: Settings Page

V. Specification

A. Backend Data Management

1) User and Device Information Management

The backend data system functions as an integrated database schema that analyzes the relationships among users' health states, weather conditions, and spatial contexts. Data are categorized into four domains—user, health, location, and weather—each interconnected through a unified identifier hierarchy.

User and device information constitute the fundamental entities of the service, serving as the top-level reference keys for all subsequent data. The User table stores each user's primary information, while a user may register multiple devices through the Device table. Each device record contains model name, platform type (e.g., iOS watch, iOS phone), and registration time, and links to the DeviceModel table to trace hardware and OS-level capabilities. The Consent table records the scope of user consent for health, location, and weather data collection, along with the timestamps of consent and revocation (granted_at, revoked_at), thereby ensuring ethical and legal transparency in data access.

2) Health Data Schema and Time-Series Management

Health-related data are stored in a per-user time-series structure, designed to capture continuous physiological measurements. The HealthSample table stores quantitative biosignals such as heart rate, heart rate variability(HRV), blood oxygen saturation, and respiratory rate. Each record is uniquely identified by a composite key of user_id, type_code, observed_at, and device_id, forming a natural unique constraint to prevent redundant uploads from the same device and timestamp.

To analyze long-term physiological patterns, dedicated tables for sleep and workout sessions are defined. The SleepSession table records the start and end time of each sleep episode, durations of deep, REM, and core sleep, and overall sleep efficiency. The WorkoutSession table manages high-intensity exercise sessions, logging activity type (running, cycling, strength, etc.), average heart rate, and active energy expenditure. In addition, the ActivitySummary table aggregates daily activity features—such as step count, exercise minutes, active calories, and standing hours—which later serve as inputs for the DailyFeature table used in model training.

This hierarchical structure is compatible with Apple's HealthKit data model and is automatically normalized and stored via the FastAPI-based backend service.

3) Environmental Context Integration

To represent users' physical and environmental contexts, the system integrates location and weather information in a unified schema. The LocationFix table stores raw GPS samples collected at regular intervals, including latitude, longitude, altitude, speed, true heading, and both horizontal and vertical accuracy values. These raw samples are clustered to identify frequently visited places (Place), each defined by its center coordinates, radius, confidence level, and visit count. Each place is also mapped to the 5-km grid system (nx, ny) provided by the Korea Meteorological Administration(KMA) for efficient weather retrieval. Address and category data (e.g., home, office, gym) are stored in a features JSON field obtained through Apple Maps reverse geocoding, enabling higher-level behavioral analysis.

Weather data are organized into WeatherTile and WeatherObservation tables. The WeatherTile table defines regional grid cells based on KMA's standardized nx, ny coordinates, while WeatherObservation stores hourly or 10-minute-interval meteorological measurements. Observed attributes include temperature, apparent temperature, humidity, wind speed, cloud cover, precipitation type (none/rain/snow/sleet), precipitation probability, and air-quality indicators. These data are periodically updated through KMA's ultra-short-term forecast and observation APIs and are linked to each Place via the grid_nx and grid_ny attributes for spatiotemporal environmental analysis.

All tables employ UUID (CHAR(36)) primary keys, and major tables include a foreign key (user_id) to ensure data ownership consistency. Cascade deletion rules (ON DELETE CASCADE) guarantee referential integrity when a user or parent record is removed. High-frequency time-series tables are indexed by (user_id, observed_at DESC) to enable efficient temporal queries, while (tile_id, as_of) indexing optimizes spatiotemporal weather lookups.

This ERD-based schema enables seamless integration of user health, location, and environmental data under a common key structure. By aligning all domains through the unified user_id and temporal fields (observed_at, as_of, recorded_at), the system ensures data consistency throughout the feature-engineering and stress-prediction pipelines. Furthermore, linking KMA's grid-based tile system directly to user places allows efficient three-way integration among weather, location, and health data. This architecture maintains high extensibility for incorporating additional environmental factors (e.g., air quality, ambient noise, indoor conditions), and serves as the core backend infrastructure connecting user health, contextual environment, and AI-driven inference in a unified pipeline.

B. Apple Watch Integration and Frontend Architecture

1) Health and Sensor Data Acquisition

The mobile application interfaces with Apple Watch via the HealthKit and CoreLocation frameworks to collect continuous biometric and spatial data in real time. Through the HealthKit API, the system retrieves categorized health data across multiple domains — including heart rate, heart rate variability (HRV), oxygen saturation, respiratory rate, and sleep analysis. Each data type is defined under Apple's standardized schema (`HKQuantityType`, `HKCategoryType`, and `HKWorkoutType`), ensuring compatibility with the user's privacy settings and consent scope.

Collected metrics are automatically normalized into time-series format before being transmitted to the backend pipeline. This structure allows the system to infer physical conditions such as fatigue, stress, and recovery status. To preserve transparency and user control, data synchronization strictly follows HealthKit's permission model, where the user explicitly authorizes or revokes each category (e.g., heart rate, step count, sleep). The synchronized data are later used to generate contextual insights, such as detecting elevated stress from HRV or reduced activity from step count trends.

2) GPS-Based Context Recognition

The CoreLocation module continuously monitors the user's GPS position to determine spatial context. Raw data points including latitude, longitude, altitude, speed, and heading accuracy — are captured at configurable accuracy levels (`kCLLocationAccuracyBestForNavigation`, `kCLLocationAccuracyNearestTenMeters`, etc.), balancing precision with energy efficiency.

Using these coordinates, the application determines whether the user is near home, commuting, or in a specific habitual zone. This context awareness enables anticipatory automation; for example, preparing home device states when the user is approaching or logging environmental conditions along outdoor routes. Each

location update is timestamped and locally cached until successful transmission to the backend.

3) Map Visualization and Routine Pattern Analysis

The MapKit framework is utilized to visualize spatial trajectories and reconstruct the user's daily movement timeline. GPS samples are rendered as polylines on a map interface, where clusters of frequent stops are highlighted as potential key places (home, workplace, gym, etc.). By aggregating temporal and spatial data, the application generates an interactive timeline that represents not only the user's movement patterns but also correlates them with physiological data such as heart rate and activity levels.

This visual feedback helps users interpret lifestyle trends — for instance, linking increased stress levels with prolonged commuting or identifying sedentary periods during the day. Map-based insights are presented in both static (daily summary) and dynamic (real-time tracking) modes, with user control over privacy and data visibility.

4) Integration Architecture and Data Flow

All frontend modules (HealthKit, GPS, and MapKit) are connected through a unified data pipeline within the iOS app architecture. The WatchKit extension communicates sensor updates via background tasks and delegates, which are received by the iOS host app using the `WCSession` framework. The host app then packages these data points into a consistent JSON schema before sending them to the backend through the FastAPI endpoint.

The entire flow ensures high modularity and real-time responsiveness:

1) Apple Watch collects health and motion data via sensors.
2) Data are serialized and transferred to the paired iPhone through `WCSession`.
3) The iPhone app integrates HealthKit and GPS samples, tagging them with timestamps and user identifiers.
4) Processed data are transmitted to the backend, where normalization and database insertion occur.

This modular integration enables continuous synchronization between the user's wearable, smartphone, and backend analytics system, forming the foundation of personalized health and behavior intelligence.

C. User State Prediction Flow

1) Model Architecture and Design Principles

The user state prediction model employs a hybrid deep learning architecture that integrates time-series health data with contextual environmental information to forecast physiological and psychological conditions. The model outputs four primary state indicators: stress level (0-10 scale), fatigue level (0-10 scale), sleep readiness (0-1 probability), and comfort index (0-10 scale). These predictions serve as the foundation for generating LLM-based commands that automatically adjust smart home appliances to optimize the user's living environment.

The architecture combines Long Short-Term Memory (LSTM) networks for temporal pattern recognition in health metrics with dense neural layers for contextual feature encoding. An attention mechanism identifies critical time periods that significantly influence current user states, while a fusion layer integrates both data streams into a unified representation. Multiple output heads enable simultaneous prediction of different state dimensions, allowing the system to capture complex interdependencies between stress, fatigue, and environmental comfort.

2) Input Feature Engineering

The model processes two distinct feature categories: time-series health data and contextual environmental data. Health features include 24-hour rolling windows of heart rate statistics (mean, standard deviation, maximum), heart rate variability (HRV) as a primary stress indicator, activity metrics (step count, active energy expenditure, exercise duration), and sleep quality measures aggregated over the past seven days (sleep duration, efficiency, deep sleep ratio, REM sleep ratio). Additional physiological signals such as respiratory rate and blood oxygen saturation supplement the primary health indicators.

Contextual features capture temporal, spatial, and environmental dimensions that influence user states. Temporal features include hour of day, day of week, and cyclic encodings using sine and cosine transformations to represent daily and weekly patterns. Spatial context is derived from GPS clustering that identifies location types (home, office, commute, other) and calculates time spent at each location. Weather data retrieved from the Korea Meteorological Administration API provides temperature, humidity, precipitation, and air quality indices (PM10, PM2.5) that correlate with physiological comfort and stress responses. Current smart home device states, including indoor temperature, humidity, and lighting levels, are also incorporated to model feedback loops between environmental conditions and user comfort.

3) Real-Time Inference and LLM Integration

The pipeline executes continuously with configurable prediction intervals, typically ranging from 15 to 60 minutes depending on computational constraints and user preferences. When triggered, the feature engineering module queries the PostgreSQL database to extract recent health samples, GPS locations, and weather observations, constructing the input tensors required by the neural network. The trained model performs forward propagation in under 100 milliseconds on the server, producing state predictions that are immediately formatted into a structured natural language prompt for the OpenAI API.

The LLM prompt includes the predicted stress level, fatigue level, sleep readiness, and current contextual information such as time of day, location type, outdoor weather conditions, and current indoor environment settings. The prompt instructs the agent to generate a JSON-formatted response containing recommended adjustments for temperature, lighting intensity, humidity control, and any additional appliance actions. The LLM's reasoning capability enables contextually appropriate decisions. The parsed JSON commands are then transmitted to the smart home controller for execution, with confirmation feedback sent to the iOS interface for user transparency and manual override capability.

4) *Limitation of ML Model*

Despite the advances in AI models, securing appropriate training datasets remained a major challenge during development. To obtain more reliable data, I contacted the researcher who created PMData, a dataset for fatigue prediction based on diverse biometric signals from participants. Through this inquiry, I confirmed that the experiment was conducted in Oslo, after which I collected Oslo weather data to use for training.

However, because the fatigue scores were subjective ratings on a 1–5 scale, the resulting model performance was inevitably poor. As part of broader experimentation, I trained and evaluated several models—including XGBoost, a tree ensemble model with soft voting, and an LSTM with data transformations—using dummy data. These models reached approximately 0.7 accuracy, but the results were dismissed due to their lack of real-world interpretability.

After multiple rounds of discussion, we established a feedback loop that uses real-time weather information to determine whether home appliances should operate, combined with each user's personalized home-appliance preference values. The user can review the recommended appliances and settings through an LLM, and then accept, modify, or reject them. Based on the user's decisions, both the operating conditions and preference values are continuously updated.

VI. Architecture Design

A. Overall Architecture

The overall system architecture consists of four tightly connected layers— (1) data acquisition on Apple Watch, (2) preprocessing and context integration on the iOS client, (3) multimodal inference and personalization within the FastAPI backend, and (4) natural-language–based appliance control via an LLM agent. Each layer operates asynchronously but synchronizes through a unified data schema centered around the user identifier and temporal index.

1) *Sensor and Context Data Acquisition Layer (Apple Watch & iOS)*

The acquisition layer runs on both the Apple Watch and the paired iPhone. The Apple Watch continuously collects high-frequency physiological signals such as heart rate, heart rate variability, respiratory rate, sleep stages, and workout metrics using HealthKit and motion sensors.

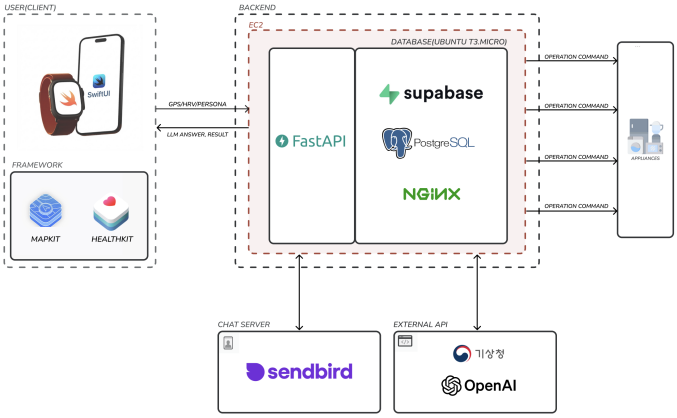

Fig. 34: Architecture

The iPhone aggregates these signals with contextual information from CoreLocation (GPS traces, speed, heading) and ambient environment sources. All data points are normalized into time-series format, timestamped, and enriched with device metadata (device model, OS version, and sample source). A local buffering module ensures reliability by temporarily storing samples when network connectivity is unavailable and flushing them to the backend when the connection is restored.

2) *Data Transmission and Preprocessing Layer (iOS → Backend)*

The client sends health and context samples to the FastAPI backend through a lightweight REST protocol. Each batch upload includes:

- user identifier and device identifier
- time-series physiological measurements
- location window and place-inference results (home/commute/office)
- device state snapshots (lighting, AC, humidity)
- weather tile index for the current GPS coordinate

Before storage, the backend applies validation (type checking, timestamp ordering, duplication removal using composite keys) and stores the data into domain-specific tables aligned with the ERD.

This layer ensures:

- schema normalization,
- deduplication of rapidly updated sensors, and
- efficient time-indexed insertion through dedicated indices.

3) *Backend Inference and Context Integration Layer*

The core intelligence of the system resides in the backend, where multiple data streams are merged into a unified feature vector.

The backend periodically assembles:

- last 24-hour health signal windows,
- matched weather observations (temperature, humidity, rainfall, PM2.5),
- spatial context (home/office/commuting), and
- user baseline statistics (HRV median, resting HR, sleep trend)

These combined features are passed through a hybrid inference module consisting of:

- an LSTM encoder for temporal physiological patterns,
- dense layers for environmental encoding, and
- an attention mechanism for highlight periods that influence the user's present condition.

The model outputs four real-time indicators: stress, fatigue, sleep readiness, and environmental comfort.

The backend maintains a preference memory that updates dynamically using:

- user overrides on appliance settings,
- action outcome logs,
- RLHF-style feedback from approval/skip decisions, and
- historical routines extracted from GPS–time clustering.

This module ensures that recommendations become more personalized over time.

4) LLM-Driven Appliance Control and Decision-making Layer

Once the backend produces the user state indicators, the system composes a structured prompt and sends it to an LLM

The analyzed condition output is passed to a Generative AI module, which converts it into a structured prompt describing the user's current physiological and environmental context. The AI then generates personalized appliance-control commands—for example, "activate dehumidifying mode at 27 °C after exercise," or "set lighting to 50% brightness and 3500 K color temperature for relaxation." Each recommendation includes concise scientific rationale (e.g., HRV recovery facilitation or melatonin preservation) derived from established sources such as WHO, ISO 7730, and AASM sleep guidelines.

5) Execution, Logging, and Closed-loop Feedback Layer

After parsing the LLM output, the backend executes the valid commands through the appliance control interface, using direct API calls (LG ThinQ or custom device controllers).

These logs form the feedback loop that continuously refines model thresholds, user preference embeddings, and the LLM prompt template—building a personalized and adaptive automation agent.

B. Directory Organization

TABLE II: DIRECTORY-ORGANIZATION-IOS-APP

Directory	File Name
space/ (Root)	spaceApp.swift TimelineDataModel.swift FastAPIService.swift DeviceManager.swift FontSizeManager.swift CallHistoryManager.swift WatchConnectivityManager.swift HealthKitManager.swift SupabaseManager.swift LocationManager.swift
space/Views/Auth	LoginView.swift SignUpView.swift SocialLoginButton.swift
space/Views/Chat	CallErrorHistoryView.swift ChatView.swift PhoneCallView.swift
space/Views/Common	ContentView.swift MainTabView.swift PersonalTestView.swift
space/Views/Device	AddItemWidget.swift ApplianceCard.swift ApplianceItemCard.swift ApplianceView.swift DeviceCard.swift DeviceDetailView.swift DeviceView.swift
space/Views/Home	HomeLocationSetupView.swift HomeView.swift LocationTagSheet.swift SplashView.swift
space/Views/Persona	AdjectiveTagView.swift PersonaBubbleWidgetNew.swift PersonaChatView.swift PersonaEditView.swift PersonaListView.swift PersonaWidget.swift
space/Views/Settings	DoNotDisturbView.swift EmergencyCallView.swift FontSizeView.swift GeneralView.swift MenuRow.swift MenuView.swift MyPageView.swift NotificationMethodView.swift SpaceNotificationView.swift
space/Views/Timeline	StateDetailView.swift StateWidget.swift TimelineDetailView.swift TimelineWidget.swift
space/Models	AdjectiveModels.swift TaggedLocation.swift
space/Managers	TaggedLocationManager.swift
space/Repositories	PersonaRepository.swift
space/Services	SupabaseService.swift

TABLE III: DIRECTORY-ORGANIZATION-WATCH-APP

Directory	File Name
space Watch App/ (Root)	space_Watch_AppApp.swift ContentView.swift MapView.swift WatchConnectivityManager.swift WatchHealthKitManager.swift WatchLocationManager.swift
space Watch App/Assets.xcassets	AppIcon.appiconset/ AccentColor.colorset/ Contents.json

TABLE IV: DIRECTORY ORGANIZATION - BACKEND

Directory	File Name
app/ (Root)	__init__.py main.py models.py schemas.py
app/api	__init__.py users.py chat.py hrv.py weather.py location.py appliance.py voice.py voice_realtime.py characters.py sendbird_webhook.py tracking.py
app/models	__init__.py user.py hrv.py appliance.py weather.py location.py info.py tracking.py
app/schemas	__init__.py user.py info.py tracking.py
app/cruds	__init__.py user.py info.py tracking.py
app/services	llm_service.py appliance_rule_engine.py appliance_control_service.py hrv_service.py weather_service.py geofence_service.py fatigue_predictor.py sendbird_client.py voice_service.py realtime_voice_agent.py
app/config	__init__.py db.py env.py sendbird.py
app/migrations	env.py

TABLE V: DIRECTORY ORGANIZATION - AI MODULE

Directory/Component	Description
app/services/ llm_service.py	**LLMService** class: - __init__() - _build_system_prompt() - parse_user_intent() - generate_appliance_suggestion() - detect_modification() - generate_response() - generate_geofence_trigger() **MemoryService** class: - add_message() - get_history() - update_long_term_memory() - get_long_term_memory() **LLMAction** enum: - NONE - CALL - AUTO_CALL
app/api/chat.py	**RESTful Endpoints**: - POST /chat/{user_id}/message - POST /chat/{user_id}/approve - GET /chat/{user_id}/history - DELETE /chat/{user_id}/session **Request/Response Models**: - ChatMessageRequest - ChatMessageResponse - ApplianceApprovalRequest - ApplianceApprovalResponse **Session Management**: - chat_sessions (dict) - get_or_create_session()
app/services/realtime _voice_agent.py	**Voice Integration**: - OpenAI Realtime API WebSocket handler - Audio streaming management - Real-time voice conversation - Geofence-triggered call support
app/api/voice _realtime.py	**WebSocket Endpoint**: - WS /voice/realtime/{user_id} - Bidirectional audio streaming - Integration with llm_service

C. Frontend

1) Purpose

The frontend for HARU is a native iOS application built with Swift and SwiftUI, responsible for providing the user interface for health monitoring, location tracking, smart home control, and AI-powered conversational interactions. The application integrates with Apple Watch for real-time biometric data collection (heart rate, HRV, calories, sleep), implements GPS-based routine tracking with intelligent checkpoint detection, and provides seamless communication with the FastAPI backend for AI-driven appliance automation. The frontend manages user authentication, persona customization, location tagging with proximity notifications, and bidirectional iPhone-Watch data synchronization using WatchConnectivity framework.

2) Functionality

The HARU iOS application provides comprehensive health and automation features across iPhone and Apple Watch platforms. Core functionality includes: (1) Real-time HealthKit data collection from Apple Watch with automatic HRV-based stress calculation and fatigue prediction integration; (2) Intelligent GPS tracking with automatic checkpoint generation triggered by HRV changes, stay detection (15m radius, 5min duration), and periodic fallback (10min intervals); (3) Location tagging system with configurable proximity notifications (default 1000m radius) for important places (home, office, cafe); (4) Multi-persona AI chat system with Sendbird integration for natural language appliance control and conversational interactions; (5) Smart home appliance control interface with real-time status monitoring; (6) Timeline visualization showing complete route history with health metrics correlation; and (7) Apple Watch companion app for autonomous workout tracking and health data collection. The application implements two primary user scenarios mirroring the backend: geofence-triggered proactive voice calls when approaching home, and user-initiated conversational appliance control through chat interfaces.

3) Location of Source Code

https://github.com/pupwchk/SWE-G04-SPACE/tree/main/swift_app_demo/space

4) Class Components

spaceApp.swift: Main application entry point defining the SwiftUI `App` structure with `@main` attribute. Initializes core managers in sequence: WatchConnectivityManager for iPhone-Watch communication, HealthKitManager for biometric data collection with authorization requests, LocationManager for GPS tracking with notification permissions, and TaggedLocationManager for loading saved location tags. Sets ContentView as root view with authentication flow handling.

Views/Common: Directory for core navigation and shared UI components.

- **ContentView.swift**: Root authentication flow manager. Displays SplashView (3-second duration) on launch, then transitions to LoginView (unauthenticated users) or MainTabView (authenticated users) based on SupabaseManager session state.
- **MainTabView.swift**: Four-tab navigation container providing access to Home (dashboard with widgets), Appliance/Device (smart home controls), Chat (messaging and voice calls), and Menu/Settings (user preferences). Implements tab-based navigation with SF Symbols icons.
- **SplashView.swift**: Launch screen displaying HARU logo with 3-second auto-dismiss timer before transitioning to authentication flow.
- **PersonalTestView.swift**: Development/testing view for persona functionality verification.

Managers: Core singleton service classes managing application-wide functionality.

- **HealthKitManager.swift**: Manages all HealthKit data collection from Apple Watch with real-time observer queries and background delivery. Published properties include daily metrics (sleepHours, stressLevel, caloriesBurned), real-time metrics (currentHeartRate, currentCalories, currentSteps, currentDistance, currentHRV), and historical data (weeklySleepData, weeklyStressData, weeklyCaloriesData). Implements HRV-based stress calculation using inverse relationship (higher HRV = lower stress) with SDNN metric. Key methods:
  - **requestAuthorization()**: Requests HealthKit permissions for sleep, heart rate, HRV, calories, steps, distance.
  - **startObservingRealTimeData()**: Establishes background observer queries for continuous health monitoring.
  - **fetchDailyHealthData()**: Retrieves aggregate health metrics for current day.
  - **fetchWeeklyHealthData()**: Computes 7-day trends for sleep, stress, calories.
  - **calculateStress(from:)**: Converts HRV SDNN to stress level (1-4 scale).

  Synchronizes data with Watch app via WatchConnectivityManager, sending health updates every 30 seconds during active tracking.
- **LocationManager.swift**: GPS tracking manager implementing intelligent checkpoint detection with best-for-navigation accuracy (kCLLocationAccuracyBestForNavigation). Provides always-on location updates (background + foreground) with 4-point moving average smoothing to reduce GPS noise. Published properties include current location data (currentLatitude, cur-

rentLongitude, currentSpeed, currentHeading), route tracking (routeCoordinates, totalDistance, speedHistory, timestampHistory), and liveCheckpoints. Implements three checkpoint generation strategies:

  - **HRV jump detection**: Triggers when HRV changes by 15ms or 20% from previous checkpoint.
  - **Stay detection**: Identifies prolonged stops (5+ minutes within 15m radius).
  - **Periodic fallback**: Creates checkpoints every 10 minutes as backup.

  Integrates with HealthKitManager to enrich checkpoints with heart rate, calories, steps, HRV, and stress level context. Manages proximity-based notifications for tagged locations.
- **TaggedLocationManager.swift**: Manages user-defined location tags (home, office, cafe, etc.) with proximity-based notification system. Implements CRUD operations for tagged locations with configurable notification radius (default 1000m) and throttling (1-hour minimum interval between notifications for same location). Supports location tag types: Home, Dormitory, Office, School, Cafe, Custom. Provides primary home location detection for special handling in geofence scenarios. Key methods:
  - **loadTaggedLocations()**: Retrieves saved locations from Supabase database on app launch.
  - **saveLocation()**: Creates new tagged location with user-defined name and tag type.
  - **checkProximity()**: Calculates distance to all tagged locations and triggers notifications when within radius.
  - **updateNotificationSettings()**: Configures radius and enabled state per location.
- **SupabaseManager.swift**: Primary authentication and database manager implementing Supabase client integration. Handles user authentication (sign up, sign in, sign out, session management), database operations (user profiles, settings, personas, tagged locations), and session validation with auto-refresh. Key methods:
  - **signUp(email:password:name:)**: Creates new user account with email confirmation, automatically creates user_profile and default user_settings records.
  - **signIn(email:password:)**: Authenticates user with session token storage in UserDefaults.
  - **signOut()**: Terminates session and clears cached credentials.
  - **getCurrentUser()**: Retrieves authenticated user session with automatic refresh.
  - **getUserProfile()**: Fetches extended profile data (birthday, avatar URL).
  - **updateUserProfile()**: Updates profile fields with optimistic UI updates.
  - **getUserSettings()**: Retrieves app preferences (notification method, DND, font size, emergency settings).
  - **updateUserSettings()**: Persists preference changes to database.

  Delegates persona management to PersonaRepository for clean separation of concerns.
- **WatchConnectivityManager.swift**: Manages bidirectional iPhone-Watch data synchronization using WatchConnectivity framework. Implements three communication methods: (1) sendMessage() for immediate delivery requiring Watch reachability, (2) transferUserInfo() for background queued transfers, (3) updateApplicationContext() for latest state only (overwrites previous). Handles data flows:
  - **iPhone → Watch**: Authentication status (user ID, email), tracking commands (start/stop).
  - **Watch → iPhone**: Location updates (coordinates array with timestamps and health data), health data (heart rate, calories, steps, distance, HRV), checkpoints (mood, stress change, location, health context), tracking status updates.

  Message types: `locationUpdate`, `healthData`, `checkpoint`, `trackingStatus`, `authentication`. Implements WCSessionDelegate for receiving Watch messages with automatic data parsing and manager updates.
- **CallHistoryManager.swift**: Tracks call history with AI-generated summaries. Manages CallRecord entities containing contact name, call type (Text, Voice, Video), timestamp, duration, and AI-generated summary text. Provides CRUD operations for call history with UserDefaults persistence.
- **DeviceManager.swift**: Manages connected smart devices (Apple Watch, AirPods, iPhone). Implements device detection via WatchConnectivity, battery level monitoring, connection status tracking, and device type categorization. Published properties: `connectedDevices` array with real-time updates.
- **FontSizeManager.swift**: Global font size manager for accessibility. Provides font size presets (Small, Medium, Large) with dynamic type scaling. Published property `currentFontSize` triggers UI updates across all views when changed.

Services: Backend integration service layer for external API communication.

- **SupabaseService.swift**: Generic MCP-style service layer providing reusable data access patterns for Supabase REST API. Implements clean separation of concerns with protocol-based design. Methods:
  - **query¡T¿()**: Generic SELECT with PostgREST filters (eq, in, order, limit).
  - **insert¡T¿()**: Generic INSERT with return representation (minimal, full).
  - **update¡T¿()**: Generic PATCH with filters for targeted updates.
  - **delete()**: Generic DELETE with filters for batch deletion.
  - **rpc¡T¿()**: Remote Procedure Call support for custom database functions.

All methods are async/await-based with structured error handling (SupabaseError enum). Handles authentication headers (apikey, Authorization Bearer token) automatically.

- **FastAPIService.swift**: Integration client for Python FastAPI backend AI services. Current endpoints:
  - **registerUser(email:)**: POST /api/users/ for email-based user registration. Returns user ID on success, handles 400 status for duplicate emails.

  Planned extensions: user lookup, AI chat endpoints for persona interactions, health data analysis integration, timeline recommendation services.

Repositories: Repository pattern implementations for clean data access abstraction.

- **PersonaRepository.swift**: Implements repository pattern for AI persona CRUD operations with protocol-based design for testability. Provides persona creation with adjective-based prompt generation, multi-persona selection (up to 5 personas), active persona management, and final prompt composition from adjectives + custom instructions. Key methods:
  - **fetchAllPersonas()**: Retrieves user's personas with adjective details via JOIN query.
  - **createPersona()**: Inserts new persona, generates final_prompt from selected adjectives and custom instructions.
  - **updatePersona()**: Modifies persona fields, regenerates final_prompt if adjectives changed.
  - **deletePersona()**: Removes persona and cleans up selection/active references.
  - **getActivePersona()**: Retrieves currently active persona for conversations.
  - **setActivePersona()**: Updates user_active_persona table.
  - **getSelectedPersonas()**: Fetches up to 5 selected personas with order.
  - **updateSelectedPersonas()**: Batch updates selection with order preservation.
  - **generateFinalPrompt()**: Combines adjective instruction texts with custom instructions using newline concatenation.

  Uses SupabaseService for all database operations, maintaining clean separation from business logic.

Models: Data structures and entity definitions for application state management.

- **User Models** (embedded in SupabaseManager.swift):
  - **User**: Basic authentication info (id, email, name) from Supabase auth.users table.
  - **UserProfile**: Extended profile (user_id, birthday, avatar_url, created_at, updated_at).
  - **UserSettings**: App preferences (user_id, notification_method, dnd_enabled, dnd_start_time, dnd_end_time, emergency_call_enabled, emergency_contact, font_size).
- **Persona Models** (PersonaRepository.swift):
  - **Adjective**: Pre-defined personality trait (adjective_id, adjective_name, instruction_text, category).
  - **Persona**: User-created AI personality (persona_id, user_id, nickname, adjective_ids, custom_instructions, final_prompt, created_at).
  - **ActivePersona**: Currently selected persona reference (user_id, persona_id, updated_at).
  - **SelectedPersona**: Multi-selection support (user_id, persona_id, order, selected_at).
- **TaggedLocation.swift**: Defines saved location entities with notification settings. Fields: location_id, user_id, latitude, longitude, tag (enum: Home, Dormitory, Office, School, Cafe, Custom), custom_name, is_home (bool), notification_enabled, notification_radius, last_notification_at, created_at. Implements Codable for JSON serialization and Identifiable for SwiftUI list rendering.
- **TimelineDataModel.swift**: Defines route tracking data structures:
  - **TimelineRecord**: Complete route record (record_id, start_time, end_time, coordinates array, checkpoints array, total_distance, duration, average_speed, max_speed).
  - **CoordinateData**: GPS point (latitude, longitude, timestamp).
  - **Checkpoint**: Significant event (coordinate, mood enum, stress_change enum, stay_duration, created_at, heart_rate, calories, steps, distance, hrv, stress_level).
  - **CheckpointMood**: Enum (VeryHappy, Happy, Neutral, Sad, VerySad) with emoji representation.
  - **StressChange**: Enum (Increased, Unchanged, Decreased) based on HRV delta.

  Implements Codable for UserDefaults persistence, Identifiable for SwiftUI lists, computed properties for map region calculation and statistics.
- **AdjectiveModels.swift**: Extends Adjective model with UI-specific properties for personality tag display (color, icon, localized descriptions).
- **Device Models** (DeviceManager.swift):
  - **ConnectedDevice**: Device info (device_id, type enum, name, model, battery_level, is_connected, last_seen).
  - **DeviceType**: Enum (Watch, AirPods, iPhone) with SF Symbols icon mapping.
- **Call Models** (CallHistoryManager.swift):
  - **CallRecord**: Call history entry (record_id, contact_name, call_type enum, timestamp, duration, summary).
  - **CallType**: Enum (Text, Voice, Video) with icon and color representation.

Views/Auth: Authentication and onboarding user interfaces.

- **LoginView.swift**: User authentication interface with email/password input fields, form validation, error

message display, navigation to SignUpView, and social login button placeholders (Google, Apple). Calls SupabaseManager.signIn() on submission, transitions to MainTabView on success.

- **SignUpView.swift**: User registration interface with name, email, password, password confirmation fields. Implements client-side validation (email format, password match, minimum length). Calls SupabaseManager.signUp() followed by FastAPIService.registerUser() to synchronize user creation across backend services. Displays email confirmation notice on success.
- **SocialLoginButton.swift**: Reusable button component for social authentication providers with provider icon, customizable colors, and tap handler.

Views/Home: Dashboard and primary user interface.

- **HomeView.swift**: Main dashboard displaying three widget sections: (1) Personal Status with TimelineWidget (latest route), StateWidget (current health metrics), PersonaBubbleWidget (active personas); (2) Connected Devices showing Apple Watch status and battery level; (3) Smart Appliances with home automation controls (mock data). Implements ScrollView for vertical scrolling, lazy loading for performance optimization.
- **HomeLocationSetupView.swift**: Initial home location configuration wizard. Displays Map view for location selection with draggable pin, current location detection button, location tag selector (Home, Dormitory, Office, etc.), custom name input field, and save button. Creates tagged_location record with is_home flag on completion.
- **LocationTagSheet.swift**: Bottom sheet modal for selecting location tag type with icon and color-coded options. Returns selected LocationTag enum to parent view.
- **SplashView.swift**: Launch screen with HARU logo, fade-in animation, and 3-second auto-dismiss timer using onAppear + DispatchQueue.main.asyncAfter.

Views/Chat: Messaging and voice communication interfaces.

- **ChatView.swift**: Multi-mode communication interface with SelectionView (choose text messaging or phone call), PersonaChatListView (list of available personas), and navigation to PersonaChatView or PhoneCallView based on selection. Integrates with Sendbird for real-time messaging backend.
- **PersonaChatView.swift**: Individual persona chat interface with message history display (LazyVStack), text input field with send button, persona name and adjective tags header, and real-time message synchronization via Sendbird SDK. Implements auto-scroll to bottom on new messages, keyboard avoidance with .ignoresSafeArea(.keyboard).
- **PhoneCallView.swift**: Full-screen voice/video call interface displaying contact name, call duration timer, call controls (mute, speaker, end call buttons), and video preview (AVCaptureVideoPreviewLayer integration placeholder). Implements CallKit integration for system call UI (planned).
- **CallErrorHistoryView.swift**: Historical call records list displaying CallRecord entities with call type icon, contact name, timestamp (relative format), duration, and AI-generated summary text. Implements search/filter functionality, swipe-to-delete actions.

Views/Device: Smart device and appliance control interfaces.

- **DeviceView.swift**: Connected device management interface (placeholder for future expansion). Displays list of ConnectedDevice entities with device type icons, connection status indicators, battery levels.
- **ApplianceView.swift**: Smart home device control dashboard with appliance cards (ApplianceCard components), category filters (All, Lighting, Climate, Security), on/off toggles, brightness/temperature sliders, and real-time status updates via FastAPI backend integration (planned).
- **ApplianceCard.swift**: Reusable appliance control card component with device icon, name, status indicator (on/off), control widgets (toggle, slider, buttons), and tap gesture for detailed view navigation.
- **ApplianceItemCard.swift**: Simplified appliance card for list display with compact layout optimized for horizontal scrolling.
- **DeviceCard.swift**: Reusable device status card component displaying device info, connection status, battery level with color-coded indicators.
- **DeviceDetailView.swift**: Detailed device settings view with device information, control options, notification preferences, and removal/reset actions.
- **AddItemWidget.swift**: Add new device/appliance widget with plus icon button, modal sheet for device selection, QR code scanning support (placeholder).

Views/Persona: AI persona management and customization interfaces.

- **PersonaListView.swift**: List of user-created personas with ChatRoomCell components, selection mode for choosing up to 5 personas (checkmark indicators), edit/delete context menu per persona, floating add button for PersonaEditView navigation, and selection count indicator (X/5).
- **PersonaEditView.swift**: Create/edit persona interface with nickname TextField, adjective multi-selector (AdjectiveTagView grid), custom instructions TextEditor (multi-line), final prompt preview (read-only), and save button. Calls PersonaRepository.createPersona() or updatePersona() with automatic final_prompt generation.
- **AdjectiveTagView.swift**: Grid of selectable adjective tags with category grouping (Personality, Tone, Style), multi-selection support (up to 10 adjectives), visual selection indicators (border, background color), and tap gesture handling.
- **PersonaWidget.swift**: Home screen widget displaying selected personas (up to 5) with circular bubble UI (first letter of nickname), background gradient based on

persona ID hash, tap gesture for chat navigation, and horizontal scroll for overflow.

- **PersonaBubbleWidgetNew.swift**: Redesigned persona bubble component with profile image support, online status indicator, unread message badge, and press animation effects.
- **PersonaChatView.swift**: (See Views/Chat section above for details).

Views/Timeline: Route tracking and health metrics visualization.

- **TimelineWidget.swift**: Latest timeline summary card displaying minimap with route polyline, checkpoint markers, statistics (distance, duration, checkpoint count), and tap gesture for TimelineDetailView navigation. Implements MapKit integration with automatic region calculation.
- **TimelineDetailView.swift**: Full route visualization with Map view (polyline overlay, checkpoint annotations with mood icons), health metrics graph (Line chart showing heart rate, stress, HRV over time), detailed statistics panel (distance, duration, average/max speed, start/end times), checkpoint list with timestamps and health context. Implements zoom/pan gestures, timeline scrubbing.
- **StateWidget.swift**: Current health status dashboard displaying real-time metrics: current heart rate (BPM with heart icon), calories burned (kcal with flame icon), steps count (with shoe icon), distance walked (km with location icon). Updates every 30 seconds via HealthKitManager published properties.
- **StateDetailView.swift**: Detailed health metrics view with weekly trend charts (sleep hours, stress level, calories burned), daily history table, average/min/max statistics, and date range selector. Implements Chart framework for data visualization with interactive tooltips.

Views/Settings: User preferences and configuration interfaces.

- **MenuView.swift**: Settings menu navigation with MenuRow components for each category: My Page (profile editing), Space Notification (notification preferences), Do Not Disturb (DND schedule), Emergency Call (emergency contact settings), Font Size (accessibility), General (app info, logout). Implements NavigationStack for hierarchical navigation.
- **MyPageView.swift**: User profile editing interface with avatar image picker, name TextField, birthday DatePicker, email display (read-only), and save button. Calls SupabaseManager.updateUserProfile() with optimistic UI updates.
- **SpaceNotificationView.swift**: Notification method selector (Push, SMS, Email) with toggle for each method, notification history list, test notification button. Persists to user_settings.notification_method.
- **DoNotDisturbView.swift**: DND schedule configuration with enabled toggle, start time picker, end time picker, override for emergency calls toggle. Persists to user_settings.dnd_enabled, dnd_start_time, dnd_end_time.
- **EmergencyCallView.swift**: Emergency contact settings with enabled toggle, contact name/phone input, test call button, call history list. Persists to user_settings.emergency_call_enabled, emergency_contact.
- **FontSizeView.swift**: Font size selector (Small, Medium, Large) with preview text at each size, system dynamic type integration, and immediate UI updates via FontSizeManager.currentFontSize published property.
- **GeneralView.swift**: App information and account management with app version display, privacy policy link, terms of service link, open source licenses, logout button (with confirmation alert), delete account button (with double confirmation).
- **MenuRow.swift**: Reusable settings menu row component with SF Symbol icon, title text, disclosure indicator, and tap gesture navigation.
- **NotificationMethodView.swift**: Detailed notification preferences with granular controls per notification type (geofence, health alert, appliance status), method selector per type, quiet hours configuration.

Watch App: Apple Watch companion application for autonomous health and location tracking.

- **space_Watch_AppApp.swift**: Watch app entry point initializing WatchConnectivityManager.shared on launch. Defines single scene with ContentView.
- **ContentView.swift**: Main Watch UI with authentication-aware display. Authenticated view shows tracking status indicator (red circle when active), iPhone connection status icon, navigation button to MapView, quick stats (distance, coordinate count). Not authenticated view displays lock icon with "Please log in via iPhone app" message.
- **MapView.swift**: Real-time route visualization on Watch with Map view showing current location, route polyline, start/stop tracking button (with workout session integration), live stats during tracking (distance, duration, heart rate), and automatic data sending to iPhone via WatchConnectivityManager.
- **WatchHealthKitManager.swift**: Watch-side HealthKit manager implementing HKWorkoutSession for workout tracking. Manages real-time queries for heart rate, active energy, steps, distance. Sends health data to iPhone every 30 seconds during active workouts. Supports workout types: Walking, Running, Cycling, Outdoor (generic). Key methods:
  - **startWorkout(type:)**: Begins HKWorkoutSession with specified activity type.
  - **endWorkout()**: Terminates workout session and saves to HealthKit.
  - **startHealthObservers()**: Establishes real-time observer queries for live data collection.

  - **sendHealthDataToiPhone()**: Packages current health metrics and transmits via WatchConnectivity.
- **WatchLocationManager.swift**: Watch-side GPS tracking manager with coordinate collection, distance calculation, timestamp recording, and integration with health data per GPS point. Sends location updates to iPhone bundled with corresponding health metrics.
- **WatchConnectivityManager.swift**: Watch-side connectivity manager mirroring iPhone implementation. Handles authentication state from iPhone (receives user ID, email), sends location/health data to iPhone (coordinates array, heart rate, calories, steps, HRV), receives tracking commands from iPhone (start/stop), and implements WCSessionDelegate for message handling with automatic state updates.

5) Technology Stack

Core Frameworks and Libraries:

- **SwiftUI**: Declarative UI framework for all view implementations with modern reactive data binding.
- **CoreLocation**: GPS location tracking with background delivery, geofencing, and region monitoring.
- **HealthKit**: Apple Watch biometric data collection (sleep, heart rate, HRV, calories, steps, distance) with background observer queries.
- **WatchConnectivity**: Bidirectional iPhone-Watch data synchronization with three transfer methods (immediate message, background transfer, application context).
- **MapKit**: Map visualization with polyline overlays, custom annotations, and region calculation.
- **Combine**: Reactive programming framework for observable state management (@Published properties).
- **UserNotifications**: Local notifications for proximity alerts (tagged locations), health warnings.
- **Charts**: Native data visualization framework for health metrics trends (iOS 16+).

Backend Integration:

- **Supabase Swift Client**: Authentication (sign up, sign in, session management), REST API client for database operations (PostgREST query syntax).
  - Authentication: Bearer token with automatic refresh
  - Tables: user_profiles, user_settings, personas, adjectives, user_active_persona, user_selected_personas, tagged_locations, location_notifications
- **FastAPI Integration**: HTTP client for Python backend AI services.
  - Endpoints: POST /api/users/ (user registration)
  - Planned: AI chat, health analysis, timeline recommendations
- **Sendbird SDK**: Real-time chat backend for persona conversations.
  - User initialization on login with Supabase user ID
  - Channel-based messaging (one channel per persona)
  - Message history persistence, read receipts, typing indicators

Data Persistence:

- **UserDefaults**: Local storage for session tokens, timeline records, call history, font size preferences.
- **Supabase PostgreSQL**: Server-side persistence for users, personas, locations, settings.
- **HealthKit Store**: Native iOS health data storage managed by Apple Health app.

Development Requirements:

- **Minimum iOS Version**: iOS 16.0+ (SwiftUI features, Charts framework)
- **Minimum watchOS Version**: watchOS 9.0+
- **Target Devices**: iPhone, Apple Watch (companion app)
- **Language**: Swift 5.9+
- **IDE**: Xcode 15.0+

Required Permissions:

- **Location**: "Always" authorization for background GPS tracking and geofencing.
- **HealthKit**: Read authorization for sleep analysis, heart rate, HRV (SDNN), active energy, step count, walking/running distance, resting heart rate.
- **Notifications**: User authorization for proximity alerts, health warnings.
- **WatchConnectivity**: Automatic pairing requirement for iPhone-Watch data synchronization.

6) Architecture Patterns

Design Patterns:

- **Singleton Pattern**: All manager classes (HealthKitManager, LocationManager, SupabaseManager, WatchConnectivityManager, TaggedLocationManager, DeviceManager, FontSizeManager, CallHistoryManager) implement singleton pattern with `shared` static property ensuring single instance across app lifecycle.
- **Repository Pattern**: PersonaRepository provides clean abstraction layer between business logic and data access, implementing protocol-based design for testability and dependency injection.
- **MVVM Pattern**: ViewModels for complex views (PersonaListViewModel, PersonaChatListViewModel) separate business logic from UI code, implement @Published properties for reactive data binding with SwiftUI views.
- **Observer Pattern**: Combine framework @Published properties enable automatic UI updates when manager state changes, eliminating manual refresh logic.
- **Delegate Pattern**: CLLocationManagerDelegate (GPS updates), HKWorkoutSessionDelegate (workout lifecycle), WCSessionDelegate (Watch connectivity).
- **Service Layer Pattern**: SupabaseService provides generic CRUD operations with type-safe generics, eliminating code duplication across repositories.

Data Flow:

```
User Interaction (SwiftUI View)
    ↓
ViewModel (Business Logic, @Published State)
    ↓
Repository/Manager (Data Access, Singleton)
    ↓
Service (Backend API Client)
    ↓
External Backend (Supabase/FastAPI/Sendbird)
```

State Management:

- **@Published**: Observable properties in managers/view models automatically trigger view re-renders when changed.
- **@StateObject**: View-owned observable objects with lifecycle tied to view.
- **@ObservedObject**: Parent-passed observable objects for child views.
- **@State**: View-local state for UI-only data (text field input, toggle states).
- **@EnvironmentObject**: Globally accessible state injected at root level (SupabaseManager, FontSizeManager).
- **UserDefaults**: Persistent local storage for session tokens, cached data, user preferences.
- **Supabase**: Server-side single source of truth for user data, personas, locations, settings.

Error Handling:

- Custom error enums: AuthError, SupabaseError, TaggedLocationError, WatchConnectivityError, each implementing LocalizedError protocol for user-facing messages.
- Structured error propagation using Swift Result type and async/await throwing functions.
- ViewModel-level error state management with @Published errorMessage properties for UI display.

Asynchronous Operations:

- **async/await**: Modern Swift concurrency for all API calls (Supabase, FastAPI, Sendbird) with structured error handling.
- **Task**: Asynchronous work in SwiftUI views using Task { } blocks for API calls on button taps.
- **@MainActor**: Ensures UI updates execute on main thread, preventing SwiftUI purple runtime warnings.
- **Combine**: Reactive programming for real-time updates from managers (published properties auto-propagate to views).
- **DispatchQueue**: Legacy async for specific use cases (timer-based updates, background Work).

Performance Optimizations:

- **GPS Smoothing**: 4-point moving average filter reduces GPS noise and prevents erratic route polylines.
- **Lazy Loading**: LazyVStack for persona lists, timeline history, call records to render only visible cells.
- **Background Delivery**: HealthKit data collection continues in background via observer queries with enableBackgroundDelivery().
- **Notification Throttling**: 1-hour minimum interval per tagged location prevents notification spam.
- **Session Caching**: Supabase auth token stored in UserDefaults reduces auth API calls on app launch.
- **Debouncing**: HRV checkpoint minimum 3-minute interval prevents excessive checkpoint generation.
- **Batch Transfers**: WatchConnectivity transferUserInfo() queues multiple updates for efficient background synchronization.

Security Measures:

- **Bearer Token Authentication**: Supabase JWT token with automatic refresh before expiration.
- **Anonymous Key**: Supabase anon key for public endpoints (sign up, sign in) with RLS policies.
- **Session Validation**: getCurrentUser() checks session validity on app launch, redirects to login if expired.
- **No Hardcoded Credentials**: All API keys loaded from configuration files excluded from version control.
- **HTTPS Only**: All backend communication uses TLS encryption (Supabase HTTPS, FastAPI HTTPS planned).
- **Keychain Storage**: Secure storage for sensitive tokens (planned migration from UserDefaults).

D. Backend

1) Purpose

The backend for HARU is responsible for managing server-side operations, database management, and orchestrating communication between multiple services. Built with FastAPI and Python, it handles user authentication, biometric data processing (HRV analysis), environmental data integration (weather APIs), smart home appliance control logic, and real-time chat functionality through Sendbird integration. The backend stores and retrieves data using PostgreSQL, ensuring reliable and scalable data management for health-responsive automation and conversational AI features.

2) Functionality

The HARU backend processes user requests from the frontend and chat interfaces, managing user health data (HRV metrics), fatigue prediction, geolocation-based triggers, weather condition monitoring, and appliance control automation. It implements two primary scenarios: (1) Geofence-triggered proactive voice calls when users approach home, and (2) User-initiated conversational appliance control based on environmental complaints. The backend integrates with external services including OpenAI GPT-4o for natural language processing, Sendbird for chat management, OpenWeatherMap for environmental data, and manages virtual appliance states with comprehensive command logging for analytics and preference learning.

3) Location of Source Code

https://github.com/pupwchk/SWE-G04-SPACE/tree/main/BackEnd/fastapi-starter

4) Class Components

main.py: Entry point for launching the FastAPI application, configuring logging and initializing the API router.

app/__init__.py: Initializes the FastAPI application instance with logging configuration.

app/api: Directory for handling incoming API requests and routing them to appropriate services.

- **__init__.py**: Aggregates all API routers for centralized routing configuration.
- **users.py**: Manages user-related endpoints, including registration, profile retrieval, and user settings.
- **chat.py**: Handles conversational AI endpoints for Scenario 2 (user-initiated chat), including message processing, intent parsing, appliance suggestion generation, and approval handling.
- **hrv.py**: Processes heart rate variability data from Apple Watch, stores HRV metrics, and triggers fatigue prediction.
- **weather.py**: Retrieves real-time weather data (temperature, humidity, PM10, PM2.5) from OpenWeatherMap API.
- **location.py**: Manages user location updates and geofence detection for Scenario 1 triggers.
- **appliance.py**: Handles appliance control endpoints, including manual control, status queries, and rule management.
- **voice.py**: Manages voice call endpoints for text-to-speech conversational interactions.
- **voice_realtime.py**: Implements OpenAI Realtime API integration for real-time voice conversations via WebSocket.
- **characters.py**: Manages AI persona/character configurations for personalized dialogue styles.
- **sendbird_webhook.py**: Receives webhook notifications from Sendbird chat service for message events.
- **tracking.py**: Handles activity tracking and wellness metrics logging.

BACKEND DIRECTORY DOCUMENTATION

app/models

Defines all database entity models using SQLAlchemy ORM.

- **user.py**: Defines core user-related entities.
  - **User**: Basic user profile (email, username, created_at).
  - **UserPhone**: Stores user phone information.
  - **UserDevice**: Stores device installation metadata per user.
- **tracking.py**: Stores user activity, health, and contextual tracking information.
  - **Place**: User-defined physical places (home, workplace, gym).
  - **TimeSlot**: Per-hour tracking window referencing a Place.
  - **WeatherObservation**: Cached hourly weather data indexed by (nx, ny, ts_hour).
  - **SleepSession**: Tracks sleep start/end timestamps and quality.
  - **WorkoutSession**: Tracks workout metadata.
  - **HealthHourly**: Hourly HRV/heart rate summary.
- **info.py**: Defines appliance and AI-character-related entities.
  - **Appliance**: Represents a user-owned appliance instance (AC, TV, Purifier, Light, etc.).
  - **AirConditionerConfig**: AC modes, temperature, wind speed settings.
  - **TvConfig**: Channel, volume, power, input source.
  - **AirPurifierConfig**: Air purifier fan levels and modes.
  - **LightConfig**: Light brightness and color modes.
  - **HumidifierConfig**: Mist level and operating mode.
  - **Character**: AI persona attached to each user.

app/schemas

Contains Pydantic models for type-safe request/response validation.

- **user.py**:
  - **UserBase**, **UserCreate**: DTO for creating users.
  - **User**: Response schema for full user entity.
  - **UserWithRelations**: User + phones + devices.
- **tracking.py**:
  - DTOs including **PlaceCreate**, **TimeSlotCreate**, **WeatherObservationCreate**, **SleepSessionCreate**, **WorkoutSessionCreate**, **HealthHourlyCreate**.
- **info.py**:
  - **ApplianceCreate**
  - **AirConditionerConfigCreate**, **TvConfigCreate**, **AirPurifierConfigCreate**, **LightConfigCreate**, **HumidifierConfigCreate**
  - **CharacterCreate**, **CharacterUpdate**, **Character**

app/cruds

Implements DB access patterns and business logic for each model.

- **user.py**:
  - **create_user(db, user_in)**: Register a new user.
  - **get_user(db, user_id)**: Retrieve a user by ID.
  - **get_user_by_email(db, email)**: Fetch user by email.
  - **get_users(db, skip, limit)**: Paginated user listing.
- **tracking.py**:
  - **create_place**, **create_timeslot**
  - **create_weather_observation**
  - **create_sleep_session**
  - **create_workout_session**
  - **create_health_hourly**

  - (Each function inserts the corresponding tracking model.)
- **info.py**:
  Appliance CRUD:
  - **create_appliance**: Create appliance instance.
  - **get_appliance**: Retrieve appliance by ID.
  - **get_appliances_by_user**: Filter by user and type.
  - **delete_appliance**: Remove appliance instance.

  Config Upsert (1:1 per appliance):
  - **upsert_air_conditioner_config**
  - **upsert_tv_config**
  - **upsert_air_purifier_config**
  - **upsert_light_config**
  - **upsert_humidifier_config**
  - (All support update-if-exists / create-if-not.)

  Character CRUD:
  - **create_character**: New AI persona.
  - **get_character**: Single-character lookup.
  - **get_characters_by_user**: List user's personas.
  - **update_character**: Modify nickname or persona.
  - **delete_character**: Remove character profile.
- **user.py**: Handles database operations for user entities.
- **info.py**: Manages character/persona database operations.
- **tracking.py**: Handles tracking data persistence.

app/services: Contains business logic for core application functionality.

- **llm_service.py**: Implements GPT-4o integration for natural language processing tasks:
  - **parse_user_intent()**: Classifies user messages into environment_complaint, appliance_request, or general_chat.
  - **generate_appliance_suggestion()**: Creates natural language suggestions based on weather, fatigue, and recommendations.
  - **detect_modification()**: Parses user approval/rejection/modification responses.
  - **generate_response()**: Generates general conversational responses with persona support.
  - **generate_geofence_trigger()**: Creates automatic voice call prompts for geofence events.
- **appliance_rule_engine.py**: Evaluates fatigue-based conditions to determine which appliances to control:
  - **evaluate_condition()**: Checks if weather data meets activation thresholds.
  - **get_appliances_to_control()**: Queries rules, evaluates conditions, retrieves learned preferences, and returns control recommendations.
  - **create_default_rules()**: Initializes default rules and preferences for new users across all fatigue levels (1-4).
- **appliance_control_service.py**: Executes virtual appliance control commands:
  - **execute_command()**: Updates appliance state, logs commands, handles errors.
  - **execute_multiple_commands()**: Batch execution for multiple appliances.
  - **get_appliance_status()**: Queries current appliance states.
  - **get_command_history()**: Retrieves historical command logs.
- **hrv_service.py**: Processes HRV data and provides fatigue level calculations:
  - **get_latest_fatigue_level()**: Retrieves most recent fatigue prediction (1-4 scale).
  - **calculate_hrv_metrics()**: Computes RMSSD and SDNN from raw HRV data.
- **weather_service.py**: Integrates with OpenWeatherMap API:
  - **get_combined_weather()**: Fetches temperature, humidity, PM10, PM2.5 data.
  - Implements caching to reduce API calls.
- **geofence_service.py**: Calculates distance to home and triggers Scenario 1 events:
  - **calculate_distance()**: Haversine formula for GPS distance calculation.
  - **check_geofence()**: Detects when user enters 100m radius.
- **fatigue_predictor.py**: Loads XGBoost model to predict fatigue levels from HRV metrics and weather data.
- **sendbird_client.py**: Wrapper for Sendbird Platform API for chat management.
- **voice_service.py**: Implements text-to-speech using external TTS APIs.
- **realtime_voice_agent.py**: Manages OpenAI Realtime API WebSocket connections for real-time voice conversations.

app/config: Configuration files and dependency injection.

- **db.py**: SQLAlchemy database engine and session management.
- **env.py**: Environment variable loading and configuration.
- **sendbird.py**: Sendbird API credentials and configuration.

app/migrations: Alembic database migration scripts.

- **env.py**: Alembic migration environment configuration.
- **versions/**: Individual migration files for schema evolution:
  - 673442797dd9_init_user_tables.py: Initial user table creation.
  - 81c03ade5e0d_add_hrv_appliance_weather_and _geofence_.py: Adds HRV, appliance, weather, and location tables.
  - 3ab07a8e1cd4_add_action_settings_priority_to_.py: Enhances appliance rules with action, settings, and priority fields.
  - add_fatigue_prediction_table.py: Adds fatigue prediction result storage.
  - e0bbb0017a7b_add_weather_cache_table.py: Adds weather data caching table.

test/: Integration and manual testing scripts.

- **test_scenario2_integration.py**: End-to-end test for conversational appliance control.
- **test_with_real_user.py**: Tests with actual user data.
- **test_hrv_manual.py**: Manual HRV data submission and fatigue prediction testing.
- **setup_appliance_rules.py**: Script to initialize default rules for test users.

E. AI

1) Purpose

The AI module serves as the intelligent core of the HARU system, implementing LLM-based dialogue management for natural language understanding and generation. Its primary purpose is to process user inputs in Korean and English, analyze conversational intent, generate context-aware appliance control suggestions based on real-time environmental and health data, and maintain adaptive learning through preference storage. The AI module leverages OpenAI's GPT-4o model exclusively, eliminating the need for custom dialogue datasets through carefully engineered prompt templates and structured JSON output enforcement.

2) Functionality

The AI module performs four core natural language processing tasks: (1) Intent Recognition - classifying user messages into environment_complaint, appliance_request, or general_chat categories; (2) Context-Aware Recommendation - combining weather data, fatigue levels, and learned preferences to generate appliance control suggestions; (3) Natural Language Generation - creating personalized, context-aware response messages in Korean with persona support; and (4) Modification Detection - parsing user approval, rejection, or modification responses to adjust control parameters. All LLM interactions use structured JSON output format to ensure reliable parsing and action execution. The module integrates seamlessly with the backend's FastAPI services through the LLMService class, providing sub-2-second response times for complete conversational flows.

3) Location of Source Code

https://github.com/pupwchk/SWE-G04-SPACE/tree/main/BackEnd/fastapi-starter/app/services

(Note: Unlike the Spring Boot + Kotlin backend in the referenced template, HARU uses a unified FastAPI architecture where AI services are integrated directly within the backend codebase under `app/services/llm_service.py`)

4) Class Components

app/services/llm_service.py: Core LLM integration service implementing all natural language processing functionality.

- **LLMService**: Main service class for GPT-4o API interactions.
  - **__init__()**: Initializes OpenAI AsyncClient with API key and model configuration (gpt-4o).
  - **_build_system_prompt(persona: Optional[Dict])**: Constructs system prompts with optional persona integration for personalized dialogue styles. Supports role definitions, response format specifications, and behavioral guidelines.
  - **parse_user_intent(user_message: str, context: Optional[Dict])**: Classifies user messages using GPT-4o with temperature=0.3 for consistent intent recognition. Returns JSON with `intent_type`, `issues` array (temperature/humidity/air_quality conditions), `needs_control` flag, and summary text. Handles Korean informal speech variations ("hot", "fire", "melting").
  - **generate_appliance_suggestion(appliances: List, weather: Dict, fatigue_level: int, user_message: str, persona: Optional[Dict])**: Creates natural Korean language suggestions incorporating current environmental conditions, recommended appliances, and user fatigue state. Uses temperature=0.7 for varied, natural responses. Returns plain text (non-JSON) formatted as "Present [environment explanation]. [Appliance Suggestion]?".
  - **detect_modification(original_plan, user_response)**: Parses user approval/rejection/modification responses with temperature=0.2 for precise parsing. Returns JSON with `approved` (bool), `has_modification` (bool), `modifications` (dict mapping appliance types to setting overrides), and `reason` (string). Handles responses like "good" (approve), "AC to 24" (modify), "Stop" (reject).
  - **generate_response(user_message, conversation_history, persona, context)**: Generates general conversational responses with action classification (NONE/CALL/AUTO_CALL). Maintains conversation history (last 10 messages) and injects real-time context. Returns structured JSON with action type and response text.
  - **generate_geofence_trigger(user_id: str, distance: float, context: Optional[Dict])**: Creates automatic voice call prompts for Scenario 1 geofence events. Generates user-friendly messages explaining automatic call trigger and provides pre-call notification text.
- **LLMAction**: Enum class defining action types:
  - **NONE**: Standard text response.
  - **CALL**: User-requested phone call.
  - **AUTO_CALL**: GPS-triggered automatic call.
- **MemoryService**: In-memory conversation state management (temporary implementation).
  - **add_message(user_id: str, role: str, content: str)**: Appends messages to conversation history, maintaining last 50 messages per user.
  - **get_history(user_id: str, limit: int)**: Retrieves recent

conversation context for LLM prompts.
  - **update_long_term_memory(user_id: str, key: str, value: Any)**: Stores persistent user information for future context.
  - **get_long_term_memory(user_id: str)**: Retrieves user-specific long-term memory.

app/api/chat.py: RESTful API endpoints for conversational appliance control (Scenario 2).

- **POST /chat/{user_id}/message**: Processes user messages through multi-step pipeline:
  1) Calls `llm_service.parse_user_intent()` for intent classification.
  2) If `needs_control=true`, queries weather data, fatigue level, and user preferences.
  3) Invokes `appliance_rule_engine.get_appliances_to_control()` for recommendations.
  4) Calls `llm_service.generate_appliance_suggestion()` for natural language response.
  5) Stores suggestions in session state for approval step.

  Returns `ChatMessageResponse` with AI response text, intent type, control flag, suggestions array, and session ID.
- **POST /chat/{user_id}/approve**: Handles user approval/modification responses:
  1) Calls `llm_service.detect_modification()` to parse user response.
  2) If approved, applies modifications to original suggestions.
  3) Executes commands via `appliance_control_service.execute_command()`.
  4) Saves learned preferences to `UserAppliancePreference` table.

  Returns `ApplianceApprovalResponse` with approval status, modifications, execution results, and confirmation message.
- **GET /chat/{user_id}/history**: Retrieves conversation history from session storage (last 20 messages).
- **DELETE /chat/{user_id}/session**: Clears conversation session and pending suggestions.

app/api/voice_realtime.py: WebSocket endpoint for OpenAI Realtime API integration.

- **app/services/realtime_voice_agent.py**: Manages bidirectional WebSocket connections for real-time voice conversations.
  - Handles audio streaming to/from OpenAI Realtime API.
  - Integrates with LLMService for context-aware voice responses.
  - Supports Scenario 1 geofence-triggered automatic calls.

Prompt Templates: Embedded within `llm_service.py` methods as f-string templates:

- **System Prompt (General Chat)**: Defines AI assistant role, capabilities (smart home control, wellness monitoring, proactive calling), and JSON response format specifications. Includes persona description when provided.
- **Intent Parsing Prompt**: Provides classification criteria with Korean examples:
  - environment_complaint: ”hot”, ”cold”, ”dry”, ”wet”, ”stuck”
  - appliance_request: ”turn on the AC”, ”turn on the light”
  - general_chat: ”Bye”, ”Thx”

  Enforces `needs_control=true` for environmental complaints and appliance requests.
- **Suggestion Generation Prompt**: Provides current weather data (temperature, humidity, PM10), fatigue level (1-4 scale), and recommended appliances. Requests natural Korean response format: ”Present [environment explanation]. [Appliance control recommandation]?”. Specifies plain text output (not JSON).
- **Modification Detection Prompt**: Shows original plan and user response, with examples:
  - ”good”, ”yes” → approved: true, has_modification: false
  - ”AC to 24” → approved: true, has_modification: true, modifications: {”AC”: {”target_temp_c”: 24}}
  - ”Stop” → approved: false
- **Geofence Trigger Prompt**: Includes user distance to home, requests AUTO_CALL action with geofence trigger, voice message for call, and pre-call notification text.
- **OpenAI Client**: Initialized with `AsyncOpenAI` for non-blocking API calls.
- **Model**: GPT-4o (`gpt-4o`) exclusively.
- **Temperature Settings**:
  - Intent Parsing: 0.3 (consistent classification)
  - Modification Detection: 0.2 (precise parsing)
  - Suggestion Generation: 0.7 (natural variation)
  - General Chat: 0.7 (conversational responses)
- **Response Format**: `{"type": "json_object"}` for all tasks except suggestion generation (plain text).
- **API Key**: Loaded from environment variable `OPENAI_API_KEY`.

VII. Use Case

VIII. App Specification

A. Loading

When the application is launched, an initial loading screen appears. This screen is shown only while the system loads essential components and prepares user-specific services such as authentication tokens, device bindings, and background

Fig. 35: Initial Loading Screen

data pipelines. After initialization, the application automatically transitions to either the Sign Up page or the Login page depending on the user's authentication status.

B. Sign Up

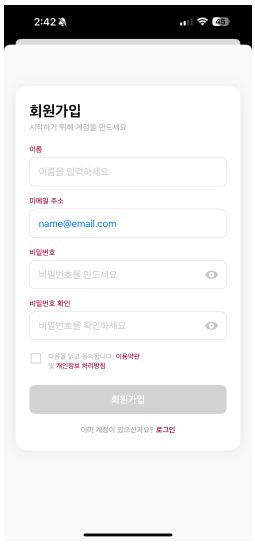

Fig. 36: Sign Up Page

On the Sign Up page, the user enters their name, password, birth date, and verifies their mobile number through a carrier-based authentication process. Identity verification is handled via a one-time code sent to the registered phone number. Once verification is complete, the user can create an account and proceed to log in. The verified phone number is stored as the primary login ID and linked to subsequent persona, device, and health data records.

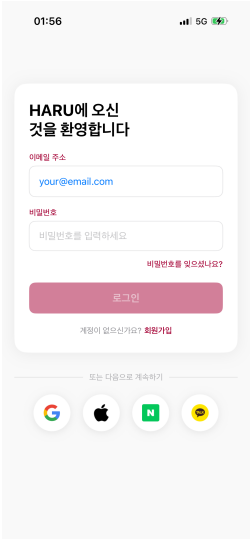

Fig. 37: Login Page

C. Login

Users log in using their registered mobile number and password. Upon successful authentication, the system checks whether the user has already configured at least one persona. If no persona exists, the application redirects to the persona setup flow. Otherwise, the user is taken directly to the main page, where daily status, devices, and shortcuts are displayed.

D. Setting Persona

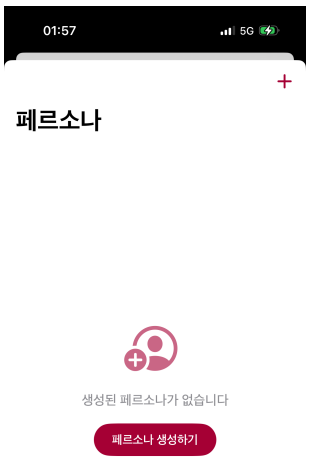

Fig. 38: Persona Creation Prompt

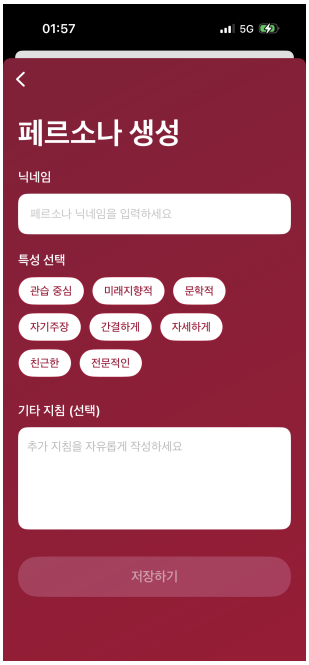

Fig. 39: Persona Selection Screen

Upon first login, users are prompted to create their personal smart-home persona. They can customize attributes such as

persona name, personality traits, conversational tone, and additional notes for personalization. Once the configuration is saved, the system automatically creates a dedicated chat room for the persona and links it with the user's devices and contextual data. Subsequent logins allow users to choose among existing personas or create additional ones as needed.

E. Import My Appliance

Fig. 40: Import Appliance Entry from Home

Users can import their home appliances into the system from the home screen or dedicated device menu. The application supports two importing methods: linking with the manufacturer's cloud service (if available) or manually registering individual devices. During registration, users specify the device type, location, and optional nickname. After importing, the appliances become available for monitoring and control throughout the application, including the main dashboard and chat-based automation.

F. Register My Wearable Device

Users can connect wearable devices such as smartwatches to synchronize biometric and activity data. Once registration is complete, the wearable continuously provides real-time metrics—including heart rate variability, stress indicators, and activity logs—via the paired smartphone. This data is integrated into the user's timeline, health metrics, and persona context, enabling advanced health-driven personalization and automated environment control.

G. Main Page

The main page serves as the central dashboard where users can view their daily activity, health status, and connected devices at a glance. At the top of the screen, a real-time map or timeline widget displays the user's current route, pace, and distance based on GPS tracking. Next to the map, a health card summarizes key biometric indicators—including sleep duration, stress level, and calories burned—collected from connected wearable devices.

Below the health overview, the page presents quick-access sections for managing connected devices. Users can add or view their wearable devices (e.g., Apple Watch, AirPods Pro) and register home appliances. Each device is represented as a card with its icon and name, allowing users to check status or navigate to detailed control pages.

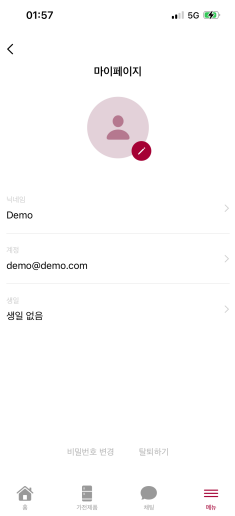

Fig. 41: Wearable Registration Page

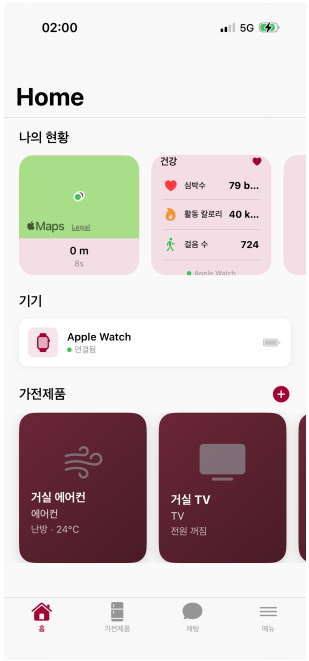

Fig. 42: Main Page – Activity Overview

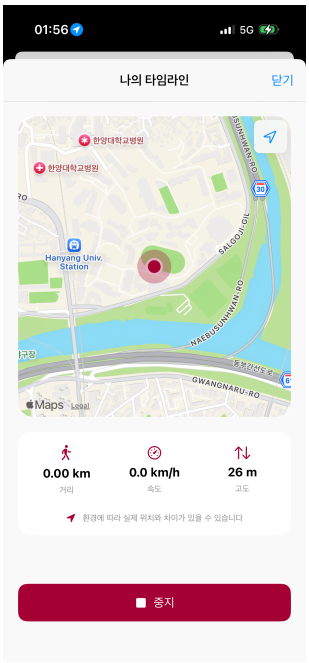

Fig. 43: Main Page – Timeline Focus

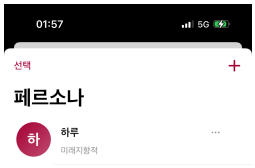

Fig. 44: Add Persona Entry from Main Page

1) Add Persona

From the main page, users can tap the *Add Persona* button to create additional personas. After customizing the persona's profile and saving, the system creates a new chat room and associates it with the same devices and contextual data as existing personas. This enables multi-agent interaction, where different personas can specialize in distinct roles such as relaxation coaching, productivity support, or home energy management.

H. Chat Room

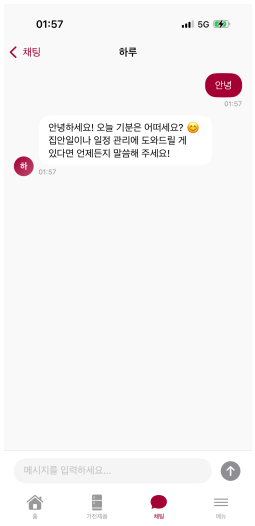

Fig. 45: General Chat Room

The chat room enables natural, continuous interaction between the user and their persona. Users can ask questions, request automation actions, receive condition summaries, and control connected devices through conversational input. The persona generates responses using multimodal context—including biometric signals, environmental data, and daily routines—retrieved from the backend. All proactive and user-generated messages are stored within the chat room, maintaining a continuous interaction history between the user and their persona.

1) Proactive Messaging via Geofencing

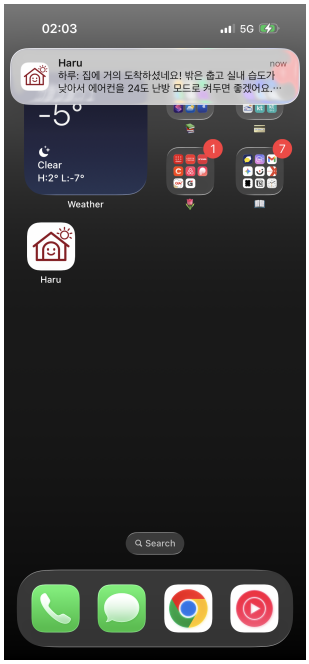

Fig. 46: Trigger-Based Notification

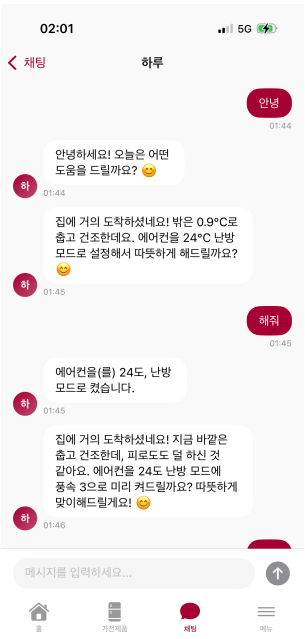

Fig. 47: Suggested Control Chat Page

The chat room supports proactive messaging triggered by geofencing events. When the system detects that the user has arrived home or is approaching a predefined boundary, a personalized message is automatically generated and sent by the persona. This feature enables the persona to greet the user, provide timely condition-based suggestions, and offer relevant automations—such as adjusting lighting, temperature, or initiating relaxation routines—without requiring explicit input from the user.

I. Devices

1) Device List

The Devices menu allows users to manage all registered home appliances. The device list page displays connected appliances organized by category, such as air conditioner, lighting, or air purifier. Each entry shows the device icon, room location, power state, and connectivity status, providing a quick overview of the home environment.

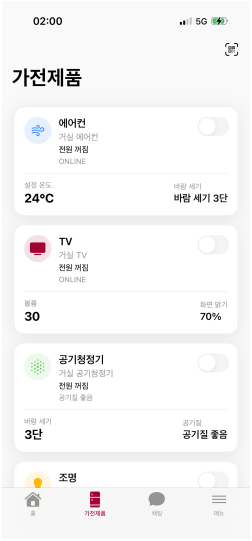

Fig. 48: Device List Page

Fig. 49: Device Detail Page

2) Device Details

From the device list, users can navigate to the detail page for each appliance. This view exposes advanced controls and status indicators, including operation mode, target temperature or humidity, brightness, and recent activity logs. Users can modify settings directly from this page, and any changes are immediately synchronized to both the physical device and the persona's contextual model.

J. Timeline

The timeline provides an overview of the user's daily movement and physiological condition. Using wearable and mobile sensor data, the system visualizes walking routes, time spent at various locations, and stress patterns throughout the day.

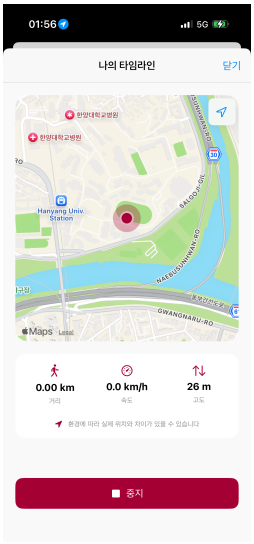

Fig. 50: Timeline Overview Page

1) Walking Log & State Point

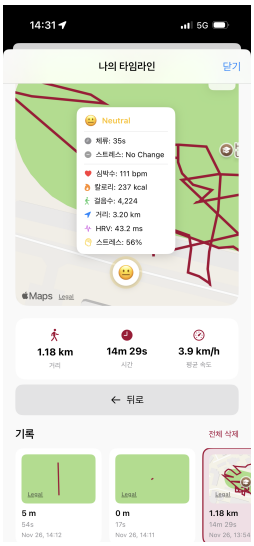

Fig. 51: State Point Page

The system tracks walking activity and highlights contextual state points that summarize the user's condition—such as fatigue, stress, and time spent in each location. Emoticons and color-coded markers are used to provide an intuitive understanding of the user's daily condition.

2) Home Location Setting for Proximity Triggers

The application provides a home location setting feature that allows users to register and store their primary home address as a geofenced point of interest. When the user saves their home location on the map, the coordinates and radius are persisted in the backend and linked to their profile. This stored home zone is then used as a trigger condition:

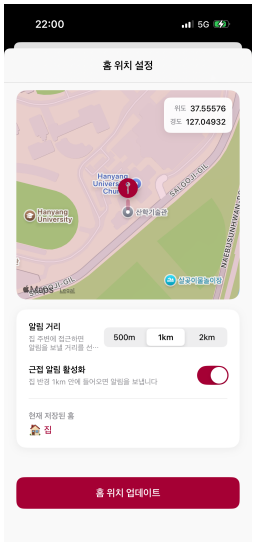

Fig. 52: Home Settings Page

whenever the system detects that the user is approaching or entering the defined home area, a proximity event is generated and a notification is delivered through the persona chat or push notification channel. This mechanism enables context-aware automation, such as suggesting appliance controls or welcome routines, based on the user's arrival at home.

3) Walking History

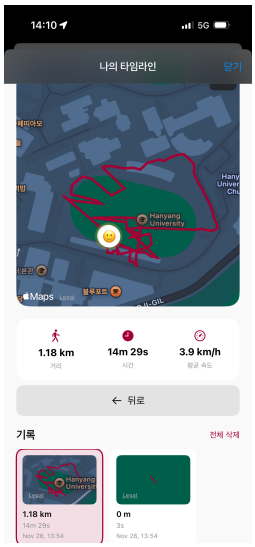

Fig. 53: History Page

The timeline history view allows users to browse past walking data by date. Selecting a specific date reveals all activities recorded on that day, along with corresponding routes, state points, and summarized condition labels.

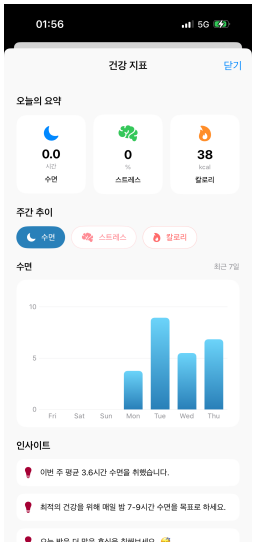

Fig. 54: Health Metrics / Stress Overview

K. Health Metrics

The Health Metrics page aggregates biometric data and provides personalized insights. Metrics include heart rate variability (HRV), heart rate patterns, sleep analysis, and physical activity levels collected from connected wearable devices.

1) Weekly Trends & Insights

Fig. 55: Weekly Trend and Insight View

Weekly trend charts visualize long-term changes in HRV stability, fatigue buildup, and recovery cycles. These visualizations help users understand whether their habits are improving or degrading their physiological resilience over time.

AI-generated insights provide personalized recommendations based on observed patterns. For example, if elevated

stress persists over several days, the system may suggest relaxation routines such as breathing exercises, short walks, or guided meditation, and can optionally propose environment adjustments through connected appliances.

L. Settings

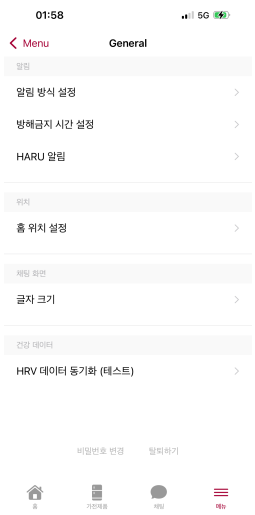

Fig. 56: Settings Overview Page

The Settings page allows users to customize general preferences related to notifications, typography, and quiet hours. These options control how and when the system communicates with the user and how information is visually presented across the interface.

1) Notification Method

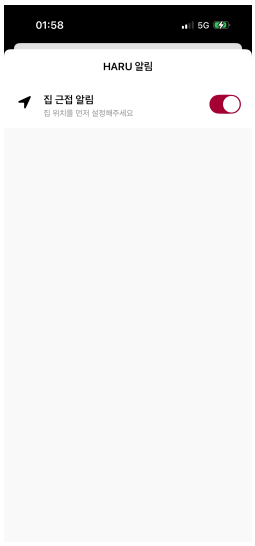

Fig. 57: Notification Method Settings

Users can choose how alerts are delivered, selecting between sound, vibration, or a combination of both. Notification preferences apply to health alerts, automation suggestions, and system messages from personas.

2) Font Size

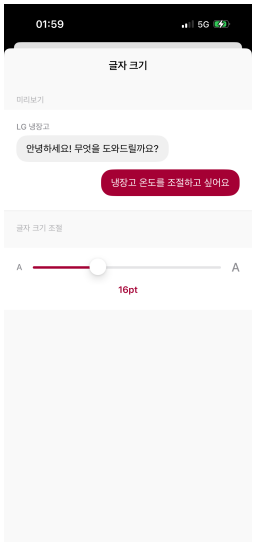

Fig. 58: Font Size Settings

Font size can be adjusted for better readability across the application. The selected size is applied globally to chat messages, dashboard widgets, and configuration pages while preserving layout consistency.

3) Do Not Disturb Time

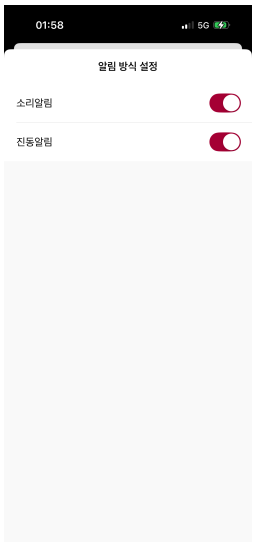

Fig. 59: Do Not Disturb Settings

Users can configure time intervals during which notifications are muted to prevent interruptions. During Do Not Disturb periods, only critical alerts (if any) are allowed through,

and routine health summaries or automation suggestions are deferred until the quiet window ends.

IX. Chat Logic

A. LLM-Based Conversational Smart Home Control System

1) Introduction

This system provides an intelligent, dialogue-driven smart home control interface that integrates natural language understanding with context-aware automation. Its core innovation lies in combining multiple data sources—biometric indicators (HRV), environmental conditions (weather), and learned user preferences—to enable personalized conversational appliance management.

The system addresses three major challenges in modern smart home automation: Natural Interaction, Users can express discomfort naturally (e.g., “It’s hot in here”) without issuing explicit control commands. Context Awareness, The AI evaluates fatigue levels, weather, and past behavior to recommend appropriate appliance actions. Adaptive Learning. User modifications are continually incorporated to refine future recommendations.

2) System Architecture

The system operates through two main scenarios:

Scenario 1: Geofence-Triggered Proactive Assistance GPS detects that the user is within a 100m radius of their home. The system automatically initiates a voice call via the OpenAI Realtime API. The user may request pre-arrival actions (e.g., “Turn on the AC before I get home”). Commands are executed immediately to prepare the home environment.

Scenario 2: User-Initiated Conversational Control The user expresses discomfort through text. The AI analyzes intent and queries current weather and fatigue conditions. The system proposes personalized appliance actions. The user may approve, modify, or reject the plan. Approved commands are executed and used to update user preference data.

3) Intent Recognition System

The intent parser uses GPT-4o to classify messages into three categories which are Intent Types environment complaint, Temperature - ”It’s hot”, Humidity - ”dry”, ”humid”, Air quality - ”stuck”, appliance request, Direct control requests - ”Turn on the AC”, Specific settings - ”Set the temp to 24”, general chat, Greetings or small talk - ”How are you?” Output Format includes intent type, issues, condition, needs condition, needs control, summary.

The engine integrates multiple sources to determine optimal appliance operations. Weather API Data consists of Temperature, humidity, PM10, PM2.5. Fatigue Level(from HRV analysis) consists of Level 1 - Good, Level 2 - Normal, Level 3 - Bad, Level 4 - Very Bad. User Preferences are Learned from previous modifications and stored per fatigue level and appliance type. Preference Priority System has Priority Order UserAppliancePreference and Default ApplianceConditionRule.

Natural Language Generation Template is ”Present [weather description]. How about [recommended appliance actions]?” Example Output is ”Present temperature is 28 degrees, and your fatigue level is 3. Would you like me to turn on the air conditioner at 23°C and also activate the air purifier?”

We used GPT-4o Primary Model and is used for all tasks due to: excellent multilingual support, reliable JSON-structured outputs, fast inference, no fine-tuning required. The system eliminates the need for fine-tuning or annotated datasets.