PocketHome: AI-driven Environmental Optimization Through Multi-User Preference Mediation

Siwoong Lee
Dept. of Information System
Hanyang University
Seoul, Korea
bluewings02@hanyang.ac.kr

Jaebeom Park
Dept. of Information System
Hanyang University
Seoul, Korea
tony0604@hanyang.ac.kr

Yuanjae Jang
Dept. of Information System
Hanyang University
Seoul, Korea
semxe123@gmail.com

***Abstract*—'PocketHome' is an AI-based automated control system that creates an optimal environment for multiple people at once. It works by combining user preferences, observed behaviors, and real-time environmental data from sensors in the space. Using this information, the system automatically adjusts shared appliances like heating/cooling systems, air purifiers, and lighting. The AI's main task is to find a balance that keeps the largest number of people comfortable and satisfied. The system's core technology, reinforcement learning, treats any manual adjustments by users as feedback, allowing it to continuously improve how it operates. This learning process reduces the need for people to make changes themselves.**

***Index Terms*—Reinforcement Learning, Optimal Environment, Multi User, User Satisfaction**

I. Role Assignments

TABLE I
Role Assignments

Role	**Name**	**Task Description**
User	Siwoong Lee	The primary end-user who provides their unique User ID and personal environmental preferences (temperature, humidity, illumination) via the mobile app. Their preference data and any manual overrides are the core inputs for the AI model.
Customer	Siwoong Lee	The sponsoring entity (e.g., building owner, facility operator) who funds the project. Defines the key objectives (occupant satisfaction, energy efficiency) and approves the project scope and its integration with the building's infrastructure.
Software Developer	Jaebeom Park	The technical expert responsible for building the complete 'PocketHome' system as defined in the requirements, including the mobile app (iOS/Android), central database, synchronization APIs, and local presence-detection protocols.
Development Manager	Yuanjae Jang	The project lead responsible for managing the development plan, team, and risks. They ensure the 'Data Utilization (AI Core)', including MOP and RL models, is successfully implemented to meet the Customer's objective of balancing multi-user comfort.

II. Introduction

A. Motivation

At home, we effortlessly control temperature, humidity, and lighting via smartphones. Yet, we lose this personalized control in shared spaces like offices and libraries. This raises a key question: “Why can we manage our home environment meticulously, but not the public spaces where we spend most of our time?” This discrepancy highlights a significant technological gap in which individual comfort is sacrificed for one-size-fits-all management.

This environmental mismatch becomes a personal burden. Individuals in shared spaces often resort to passive measures, such as adding clothing or changing seats, when uncomfortable. This directly impacts not only their comfort but also their focus and productivity. We must therefore move beyond the current, unresponsive centralized control model to a new paradigm that can intelligently accommodate diverse and individual needs.

B. Problem Statement

The fundamental flaw in shared environmental control is the requirement of enforcing a single setting on multiple individuals. This one-size-fits-all system cannot resolve the conflicting preferences that inevitably arise. Current feedback mechanisms, such as manual complaints, are inefficient, slow, and do not mediate these diverse needs. This leads directly to decreased comfort, reduced productivity, and energy waste. The core problem is the absence of an automated mechanism to intelligently mediate these conflicting demands and derive a balanced, optimal environment in real time.

C. Related Service

Siemens Comfy: A leading workplace experience platform. It enables employees to adjust temperature and lighting in their immediate vicinity through an app. It also includes features like booking meeting rooms or desks. Its acquisition by Siemens highlights the trend of merging with smart building technology.

HqO: An integrated app platform that building owners provide to their tenants. It extends beyond simple environmental control to improve the overall 'work-life experience,' including booking building amenities, visitor registration, and viewing local information and announcements.

Honeywell Forge: An enterprise-level solution from Honeywell for optimizing building operations. Integrates data from various systems (HVAC, security, fire safety) for AI analysis to enable prediction of energy consumption and predictive maintenance. Its focus is more on operational efficiency than on employee-facing convenience.

LG Electronics B2B Integrated Solutions: LG provides B2B smart office solutions focused on central air conditioning (HVAC), digital signage, and meeting technologies. The key distinction is that these solutions have not yet integrated LG's consumer home appliances to provide personalized convenience for individual employees.

III. Requirements

A. User Roles

- **General User:** Any individual present in a space where 'PocketHome' is installed. They register their unique ID via the app and transmit their personal environmental preferences to the system in real-time.

B. Account Management

- **User Identification:**
  - *Initial Setup:* Upon first launch, the user enters their unique `User ID` once.
  - *ID Uniqueness:* The `User ID` must be a unique value, verified against the central database in real-time to prevent duplicates.
  - *Session Management:* After registration, the app stores the `User ID` on the device, allowing the user to transmit preferences without logging in each time.

C. Core Features (Single Screen Interface)

1) Preference Input Dashboard:

- **Single Screen UI:** All core functions (ID verification, preference input) are provided on a single main screen.
- **User ID Display:** The user's currently registered `User ID` is clearly displayed.
- **Optional Biometric Data Consent:** A heart shaped icon in the top-left corner opens a pop-up window where users can consent to the collection of optional biometric data.
- **Optional Profile Access:** A + (plus) button in the top-right corner. Tapping this button opens an overlay screen (modal) where users can input or delete their optional 'Personal Profile Attributes' (e.g., MBTI, sensitivity to cold).

2) Environmental Preference Controls:

- **Temperature:**
  - *Control:* Stepper control.
  - *Range:* 18.0°C ˜ 28.0°C.
  - *Increment:* Adjustable in units of 0.5°C.
- **Humidity:**
  - *Control:* 5-step segmented button group.
  - *Steps:* 5 levels (e.g., “Very Dry”, “Dry”, “Neutral”, “Humid”, “Very Humid”).
- **Illumination (Brightness):**
  - *Control:* Continuous slider.
  - *Range:* 0% (Darkest) ˜ 100% (Brightest).

3) Data Synchronization:

- **Real-time Update:** As soon as the user changes any of the three preference values (temperature, humidity, illumination), the new value is immediately sent with the `User ID` to the central database to be updated and stored.

D. Optional Data Integration

1) Biometric Data:

- **Heart Rate Sensor:** With user consent, integrates with the smartphone's heart rate sensor (on compatible devices).
- **Blood Oxygen Sensor (SpO2):** With user consent, integrates with the smartphone's blood oxygen saturation sensor (on compatible devices).
- **Real-time Transmission:** When activated, periodically transmits the real-time measured heart rate data along with the `User ID` to the central database.

2) Personal Profile Attributes:

- **Optional Input:** Allows users to voluntarily provide additional personal attributes, such as MBTI type, self-assessed sensitivity to cold/light, or general work patterns (e.g., "Focus work", "Collaborative work").
- **Data Utilization:** This supplementary, non-biometric data can be used by the AI model as another feature to find correlations and improve the accuracy of its satisfaction predictions (e.g., "Users with MBTI type 'INXX' may correlate with a preference for lower illumination").

E. System & Space Integration

1) Presence Detection:

- **Network Communication:** When a user enters a specific 'PocketHome' enabled zone, their smartphone begins communication with the local network (e.g., Wi-Fi, Bluetooth Beacon).

2) Preference Transmission:

- **Interval-based Communication:** While in space, the smartphone transmits its `User ID` and current preference data to the local space system at regular intervals (e.g., every 5 minutes).
- **Purpose:** This allows the space system to identify in real-time which users (User IDs) with which preferences are currently present in this space.

F. Data Utilization (AI Core)

1) Central AI Model (Multi-Objective Optimization):

- **Data Aggregation:** The central AI system queries the database to retrieve two key data sets: (1) All users' registered preferences (from Data Synchronization) and (2) The list of `User IDs` currently present in a specific zone (from Preference Transmission).
- **Optimization Goal:** The AI model (e.g., using Genetic Algorithms or MORL) processes this aggregated data. Its goal is to compute the single optimal environmental setting (temp, humidity, illumination) that best satisfies a defined objective function (e.g., the 'Max-Min' principle, maximizing the satisfaction of the least-satisfied user).
- **Control Command:** The resulting optimal setting is then sent as a command to the building's control system (e.g., HVAC, lighting controllers) for that specific zone.

2) Reinforcement Learning Feedback Loop:

- **Monitoring for Manual Overrides:** The system must log any manual adjustments made by users (e.g., via a physical thermostat or an admin override) after the AI has set an environment.
- **Model Refinement:** A manual override is treated as strong negative feedback ("penalty"). The AI's reinforcement learning component uses this feedback to retrain its models, refining its understanding of true user preferences and discomfort thresholds to avoid this outcome in the future.

G. Technical Requirements

- **Platform:** Mobile application (Android).
- **Database:** A central, real-time database capable of receiving and updating all user `User IDs` and their changing preference data.
- **Network:**
  - *Central:* Stable internet connection (Wi-Fi or Cellular) for database updates.
  - *Local:* Local communication protocol (Bluetooth LE or Wi-Fi) for the detection and transmission of user presence within a space.
- **Privacy & Consent:**
  - *Initial Consent:* A clear notification and consent process is mandatory upon `User ID` registration regarding the collection of preference and optional data.

IV. Development Environment

A. Software Development Platform

- **Windows**

Fig. 1. Windows

Windows 11 was selected as the primary developer operating system due to its robust and high-performance support for the project's essential toolchain. Its main role is to host Android Studio, enabling the stable compilation, debugging, and emulation of the native Kotlin application. This environment allows for efficient build management via Gradle and simultaneously supports the local development and testing of the Python-based AI Core services before their eventual deployment to the Firebase cloud.

Fig. 2. Kotlin

B. Programming Language

- **Kotlin**
  Kotlin was chosen primarily because it is Google's official, modern language for native Android development, ensuring the best integration with Android Studio and safer, more concise code through features like null safety. Most importantly, it provides the direct and reliable access to the native SDKs essential for this project's core functionality—specifically, the Bluetooth LE (BLE) API for Presence Detection and the Health Connect API for Biometric Data Transmission—guaranteeing a level of performance and hardware compatibility that cross-platform tools cannot reliably offer.

- **Python**

Fig. 3. Python

  Python was chosen to power the server-side Data Utilization (AI Core) because it is the definitive industry standard for the complex AI modeling this project requires. Its primary role is to execute the Multi-Objective Optimization (MOP) and Reinforcement Learning (RL) models that form the system's intelligence. This decision leverages Python's unparalleled ecosystem of mature libraries, such as Pymoo for optimization and TensorFlow Agents for reinforcement learning, which are essential for processing aggregated user data. This stack enables a flexible architecture where heavy model training (using feedback logs) is performed offline, while the resulting optimized models are deployed to a serverless environment (like Google Cloud Run) for efficient real-time inference.

C. Software in Use

- **Android Studio**
  Android Studio was selected as the development environment because its unmatched synergy with Kotlin, Google's official and primary language for the platform, which provides a seamless, optimized development

Fig. 4. Android Studio

  experience that enhances code stability and accelerates production. Critically, this choice guarantees 100% compatibility with essential native Android SDKs, allowing for the reliable, direct integration of core hardware-centric features like the Bluetooth LE (BLE) API for Presence Detection and the Health Connect API for Biometric Data Transmission through its integrated Gradle system, thereby avoiding the limitations often encountered when using cross-platform frameworks.

- **Flutter**

Fig. 5. Flutter

  Flutter was chosen for its ability to deliver a high-fidelity, consistent UI across both Android and iOS from a single codebase. Its rich, declarative widget system allows for deep customization and rapid prototyping. This speed is accelerated by Hot Reload, enabling instant testing and refinement of the Preference Input Dashboard. This method ensures a polished user experience and avoids duplicating design efforts for each platform.

- **Firebase**
  Firebase was selected as the backend stack due to its direct alignment with the project's core requirements and its seamless integration with the native Kotlin app. Its Cloud Firestore component provides the essential real-

Fig. 6. Firebase

time database capability specified in the Data Synchronization requirement, allowing for instantaneous preference updates from the app with minimal latency. Furthermore, Firebase's serverless architecture, through Cloud Functions, is ideal for executing the Data Utilization (AI Core) logic, enabling scalable, event-driven inference for our MOP/RL models. This comprehensive platform, combined with the cost-free prototyping offered by the Spark Plan and a robust Kotlin SDK, makes it the most rapid and efficient solution for development.

D. Cost Estimation

- **Software Costs: $0.** All primary development tools are free (Android Studio, Kotlin, Git, VS Code).
- **Cloud Costs (Prototype): $0.** We will utilize the **Firebase (GCP) Free Tier (Spark Plan)**. This plan provides sufficient capacity for the `Cloud Firestore` database (real-time data sync) and `Cloud Functions` (API/AI logic) needed for development and initial testing.
- **Hardware Costs: $0.** Development will use existing laptops (PC/Mac) and Android test devices.

E. Development Environment Details

- **OS:** Windows 11
- **IDE:** Android Studio (Latest stable version)
- **Version Control:** Git (Repository hosted on GitHub)
- **Compile SDK Version:** 36 (API 36.0 ("Baklava";Android 16.0))
- **Backend Stack:** Firebase (GCP)
- **Key Libraries (Kotlin):**
  - `Firebase Kotlin SDK (Firestore, Auth)`: For real-time database synchronization.
  - `Android Bluetooth LE (BLE) API`: For 'Presence Detection' requirement.
  - `Health Connect API`: For 'Optional Biometric Data' integration.
  - `Retrofit2/OkHttp3`: For API communication with the AI Core.

F. Software in Use (Existing Algorithms)

We will use existing, validated software and algorithms. The Kotlin application will be built with Google's Android SDK, Firebase Kotlin SDK, and Health Connect API. For the Python AI Core, we will use libraries like Pymoo (Multi-Objective Optimization) and TensorFlow Agents (Reinforcement Learning), deployed on Firebase Cloud Functions.

V. Specifications

Technology Stack: Kotlin (Android App), Firebase (Backend/DB), Python (AI Core)

A. Account Management

1) ID Uniqueness (New ID Uniqueness Verification):

a) Platform:: Kotlin (Android App) → Firebase (Firestore)

b) Description:: Logic for when the user enters an ID and clicks the 'Register' button.

```
// Called when the user clicks the 'Register' button
FUNCTION onRegisterClick(userID: String):
    IF userID.length < 4:
        showError( ID must be at least 4 characters. )
        RETURN

    // 1. Check if a document with this ID already exists
    db = Firebase.firestore
    userDocRef = db.collection( users ).document(userID)

    userDocRef.get()
        .addOnSuccessListener { document ->
            IF document.exists():
                // 2a. If doc exists, show error
                showError( This ID is already in use. )
            ELSE:
                // 2b. If doc does not exist (unique ID)
                // 3. Create a new user doc
                defaultPrefs = {
                     temperature : 24.0,
                     humidity : 3,  //  Neutral
                     illumination : 80
                }
                userDocRef.set({
                     preferences : defaultPrefs,
                     createdAt : NOW()
                })

                // 4. Save ID locally (Session)
                CALL saveUserIDLocally(userID)

                // 5. Navigate to main dashboard
                navigateToMainDashboard()
            END IF
        }
        .addOnFailureListener { error ->
            showError( Network error: ${error.message} )
        }
END FUNCTION
```

2) Session Management:

a) Platform:: Kotlin (Android App)

b) Description:: Checks for a locally stored ID on app launch to handle auto-login.

```
// (SharedPreferences or DataStore Utility)
FUNCTION saveUserIDLocally(userID: String):
    localStorage = getSharedPreferences( PocketHomePrefs )
    localStorage.edit().putString( USER_ID ,
         userID).apply()
END FUNCTION

FUNCTION loadUserIDLocally(): String?
    localStorage = getSharedPreferences( PocketHomePrefs )
    RETURN localStorage.getString( USER_ID , null)
END FUNCTION

// (In the app's first-run Activity or ViewModel's init)
FUNCTION checkLoginStatus():
    userID = CALL loadUserIDLocally()

    IF userID == null:
        // If no ID is saved, navigate to Registration
        navigateToRegistration()
    ELSE:
        // If an ID is saved, navigate to Main Dashboard
        mainViewModel.setUserID(userID)
        navigateToMainDashboard()
    END IF
END FUNCTION
```

B. *Core Features (Single Screen Interface)*

1) *Data Synchronization (Real-time Preference Sync):*

a) Platform:: Kotlin (Android App) $\rightarrow$ Firebase (Firestore)

b) Description:: Updates preferences in the Firestore DB in real-time as the user interacts with UI.

```
// 1. (Read) Listen for real-time changes from Firestore
FUNCTION startListeningForPreferences(userID: String):
    db.collection( users ).document(userID)
        .addSnapshotListener { snapshot, error ->
            // This block executes automatically
            // whenever the DB value changes
            IF snapshot != null AND snapshot.exists():
                newPrefs =
                     snapshot.toObject(Preferences.class)
                // Update LiveData or StateFlow
                // to automatically reflect changes in the
                     UI
                _preferenceStateFlow.value = newPrefs
            END IF
        }
END FUNCTION

// 2. (Write) Called when user stops sliding temp slider
FUNCTION onTemperatureChanged(userID: String,
                               newTemp: Double):
    // (Debounce logic recommended for performance)
    db.collection( users ).document(userID)
        .update( preferences.temperature , newTemp)
END FUNCTION

// 3. (Write) Called when user clicks a humidity button
FUNCTION onHumidityChanged(userID: String,
                            newHumidityStep: Int):
    db.collection( users ).document(userID)
        .update( preferences.humidity , newHumidityStep)
END FUNCTION
```

2) *Optional Biometric Data Consent:*

a) Platform:: Kotlin (Android App)

b) Description:: Asks for consent when the heart icon is clicked and saves the choice locally.

```
FUNCTION onHeartIconButtonClick():
    currentConsent = loadBiometricConsent() // Read local

    showDialog(
        title= Biometric Data Collection Consent ,
        message= Allow 'PocketHome' to use heart rate and
             +
                 SpO2 data to improve the AI model? ,
        onConfirm = {
            saveBiometricConsent(TRUE)
            CALL startBiometricService() // Consented
        },
        onDeny = {
            saveBiometricConsent(FALSE)
            CALL stopBiometricService() // Denied
        }
    )
END FUNCTION
```

C. *Optional Data Integration*

1) *Biometric Data Transmission:*

a) Platform:: Kotlin (Android App - Background Service)

b) Description:: If consented, periodically (e.g., every 5 min) measures and transmits biometric data to Firestore.

```
SERVICE BiometricDataWorker:
    FUNCTION doWork(): // Runs periodically
        // 1. Re-check consent status
        consent = loadBiometricConsent()
        IF consent == FALSE:
            RETURN Result.failure()

        // 2. Read data from Health Connect API
        userID = loadUserIDLocally()
        heartRate = HealthConnect.readLatestHeartRate()
        spO2 = HealthConnect.readLatestSpO2()

        // 3. Add to a separate log collection
        logData = {
             userID : userID,
             heartRate : heartRate,
             spO2 : spO2,
             timestamp : NOW()
        }
        db.collection( biometric_logs ).add(logData)

        RETURN Result.success()
    END FUNCTION
END SERVICE
```

D. *System & Space Integration*

1) *Presence Detection & Preference Transmission:*

a) Platform:: Kotlin (Android App - Background Service)

b) Description:: Detects current zone via BLE scan and periodically updates user's presence/preferences to the `presence_logs` collection.

```
SERVICE PresenceDetectionWorker:
    currentZone = null

    FUNCTION startService():
        // 1. Start BLE scan for 'PocketHome_BEACON_UUID'
        bleScanner.startScan(
            filters=[ PocketHome_BEACON_UUID ],
            onScanResult
        )

        // 2. Schedule 'transmitPresence' job (e.g., 1 min)
        scheduler.schedulePeriodicWork(
            1_MINUTE, CALL transmitPresence
        )
    END FUNCTION

    // 3. BLE Scan Callback
    FUNCTION onScanResult(beaconID: String): // e.g.,
          Zone_A
        currentZone = beaconID
    END FUNCTION

    // 4. Function that runs every 1 minute
    FUNCTION transmitPresence():
        userID = loadUserIDLocally()

        IF currentZone == null: // User is not in a zone
            // (Optional: log out event)
            db.collection( presence_logs )
              .document(userID).delete()
            RETURN
        // 5. Update my info in the current zone's log
        // (Note: This is the core data for the AI)
        currentPrefs =
             mainViewModel.preferenceStateFlow.value
        presenceData = {
             zone : currentZone,
             preferences : currentPrefs,
             lastSeen : NOW()
        }
        // Overwrite(set) doc with {userID} as the ID
        db.collection( presence_logs )
          .document(userID).set(presenceData)
    END FUNCTION
END SERVICE
```

E. Data Utilization (AI Core)

Note: This section is Python code running on the Google Cloud backend, NOT in the Kotlin app.

1) Central AI Model (MOP Inference):

a) Platform:: Python (Google Cloud Run / Cloud Function)

b) Description:: Triggered every 5 minutes. Aggregates preferences of users in a zone, 'infers' the optimal setting using a pre-trained model, and sends the command to the HVAC system.

```
# FUNCTION calculate_optimal_environment(event, context):
def calculate_optimal_environment(zoneID= Zone_A ):
    # 0. Load the MOP model trained locally
    model = load_model( mop_model.pkl )

    # 1. Aggregate preferences of users in 'Zone_A'
    db = firestore.client()
    users_in_zone = db.collection( presence_logs )
                        .where( zone ,  == , zoneID).get()

    user_preferences = []
    # e.g., [{ temp : 22}, { temp : 24}]
    for doc in users_in_zone:
        user_preferences.append(
            doc.to_dict()[ preferences ]
        )

    IF len(user_preferences) == 0:
        RETURN # No one is in the zone

    # 2. 'Infer' the optimal solution
    optimal_setting = model.predict(user_preferences)
    # e.g., { temp : 23.5,  humidity : 3}

    # 3. Send command to the building API
    HVAC_API.set_temperature(
        zoneID, optimal_setting[ temp ]
    )
    LIGHTING_API.set_brightness(
        zoneID, optimal_setting[ illumination ]
    )

    # 4. (For feedback loop) Record AI's setting in DB
    db.collection( zone_status ).document(zoneID).update({
         ai_setting : optimal_setting,
         timestamp : NOW()
    })
END FUNCTION
```

2) Reinforcement Learning Feedback Loop:

a) A. Feedback Collection (Kotlin App): **Platform:** Kotlin (Android App)
Description: When the user performs a 'Manual Override', this event is sent to Firestore as a 'penalty' log.

```
// When user 'forces' a value different from AI's
FUNCTION onManualOverride(userForcedTemp: Double):
    userID = loadUserIDLocally()
    zone = presenceService.currentZone

    // 1. Read the current AI setting from the DB
    aiSetting = db.collection( zone_status )
                    .document(zone).get().ai_setting

    // 2. Record a 'penalty log' to Firestore
    feedbackLog = {
         userID : userID,
         zone : zone,
         ai_setting : aiSetting, // The value AI proposed
         user_override : userForcedTemp, // The value user
             forced
         timestamp : NOW()
    }
    db.collection( override_logs ).add(feedbackLog)

    // 3. (Optional) Immediately send override value
    HVAC_API.set_temperature(zone, userForcedTemp)
END FUNCTION
```

b) B. Model Retraining (Python - Local PC): **Platform:** Python (Developer's Local PC)
Description: Due to Spark Plan limitations, the developer periodically downloads 'penalty' logs, 'retrains' the RL/MOP model locally, and 'deploys' the new model file to Cloud Run.

```
# FUNCTION train_rl_model_offline():
def train_rl_model_offline():
    # 1. Download all 'penalty_logs' from Firestore
    logs = download_collection_from_firestore(
         override_logs
    )

    # 2. 'Retrain' the RL model based on the logs
    # (State: ai_setting, Action: user_override,
    #  Reward: -abs(ai - user))
    # ... (RL.train(logs) ...

    rl_model.save( new_rl_policy_v2.pkl )

    # 3. Update the MOP model with the new RL policy
    mop_model.update_policy(rl_model)
    mop_model.save( mop_model_v2.pkl )

    # 4. Manually deploy new model to Cloud Run
    # (gcloud run deploy ...

    PRINT( New model deployed successfully! )
END FUNCTION
```