Internet of Things
Final Research Project

MediSense Smart Medication Cabinet System Analysis and Development Document

Prepared by:
Sina Toprak Güleç

Course: MISY4461.1 Internet of Things

Instructor: Dr. Şahin Aydın

Date: 17.06.2026

Content

1. Materials

The MediSense system was developed using both hardware and software components required for an IoT-based smart medication cabinet solution. On the hardware side, the system includes an ESP32 microcontroller, four independent load cell sensors, four HX711 amplifier modules, a custom MDF cabinet structure with separate compartments (MORNING, NOON, EVENING, NIGHT/EVERY-MEAL), a Mean Well RD-65A dual-output switching power supply, a relay module, four 12V electromagnetic solenoid locks, an HC-SR04 ultrasonic distance sensor, a K83 infrared (IR) proximity sensor, a DHT22 temperature and humidity sensor, an active buzzer, status LED indicators, and Wi-Fi connectivity. The load cells measure the weight of the medication containers within each compartment, while the HX711 modules convert the analog sensor signals into digital data that can be processed by the microcontroller. The ESP32 reads this data, manages touchless intent verification via the ultrasonic and IR proximity sensors, triggers the corresponding 12V solenoid locks via the relay module, and sends telemetry values and hardware events to the backend system through a wireless network connection. On the software side, the project uses a Python-based FastAPI backend, SQLAlchemy and SQLModel ORM, a PostgreSQL database, and a Flutter and Dart-based mobile application. The backend processes incoming sensor readings, evaluates active dose windows to log adherence status, automatically fires temporal task routines via APScheduler, manages asynchronous caregiver invitation workflows by sending HTML verification emails via aiosmtplib over secure SMTP TLS Port 465, stores profile, cabinet, medication, schedule, event, and notification records, and provides this information to the frontend application via RESTful endpoints and real-time WebSockets. These materials work together to create a real-time, sensor-based monitoring system that helps patients manage their medication routines and improves traceability for caregivers more efficiently.

2. System Architecture and Design

This chapter explains the architectural structure of the MediSense system. The system is designed as a multi-layer IoT solution that includes a physical sensing layer, backend processing layer, database layer, and frontend presentation layer. Each layer has a specific responsibility within the complete monitoring workflow. The hardware layer collects real-time weight and event data from the cabinet compartments, the backend layer processes telemetry and adherence logic, the database layer stores clinical and device-related records, and the frontend layer presents monitoring information to caregivers. The following sections describe these components and their relationships in detail through architectural diagrams and system design explanations.

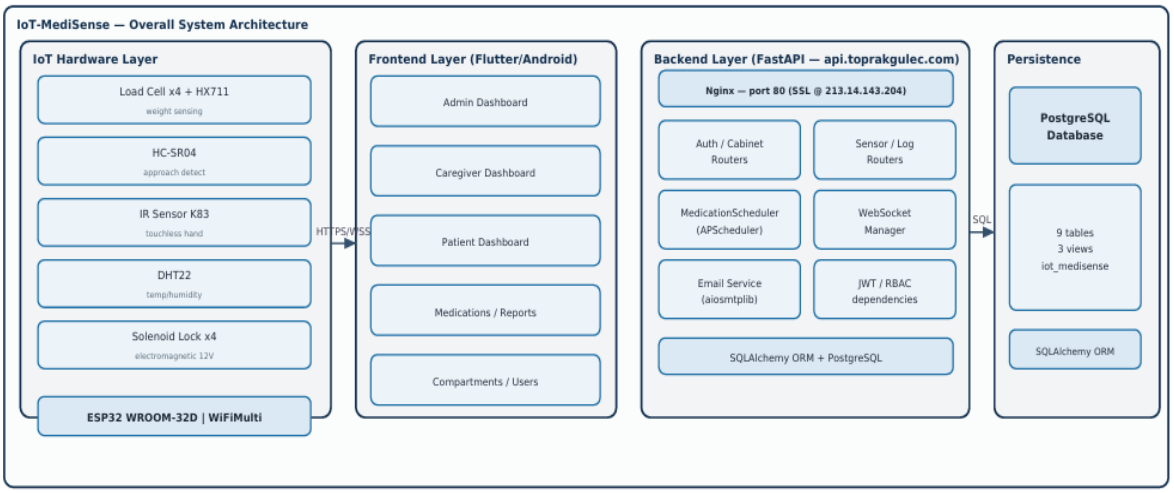

Figure 1 - MediSense Smart Medication Cabinet System Architecture

2.1 System Context Diagram

The system context diagram shows the IoT-MediSense platform as the central system that communicates with caregivers, patients, system administrators, the ESP32 cabinet, and the PostgreSQL database. Caregivers monitor compartment status, manage medications, and view compliance reports through the Flutter app at https://api.toprakgulec.com . Patients view their schedule and take doses from the cabinet. System administrators manage the cabinet configuration, compartment labels, and user invitations. The ESP32 cabinet sends telemetry, events, and logs using the SERVICE_ROLE_KEY . SSL is terminated at the Plesk reverse proxy (213.14.143.204) before traffic reaches the Nginx vhost on the VPS (10.0.60.20:80).

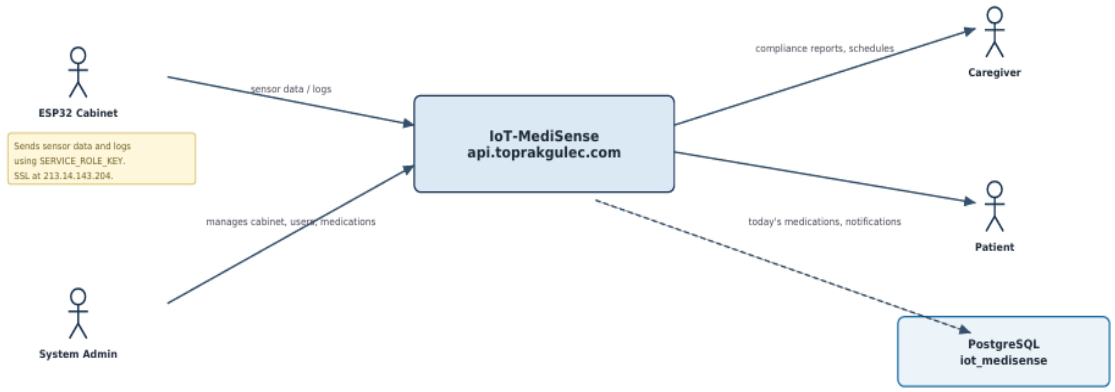

Figure 2 - MediSense System Context Diagram

2.2 Overall System Architecture

The overall architecture of the IoT-MediSense system is divided into four main layers: the IoT hardware layer, frontend layer, backend layer, and persistence layer. The IoT hardware layer is responsible for collecting physical weight and event data from the cabinet compartments. The frontend layer allows caregivers, patients, and administrators to interact with the system through dashboard, compartment, medication, report, and settings screens. The backend layer receives requests from both the frontend and the IoT device, processes business logic, evaluates dose windows, and generates notifications. The persistence layer stores all system data using SQLAlchemy ORM and a relational PostgreSQL database. This layered structure improves maintainability, separates responsibilities, and allows each part of the system to be developed and tested independently.

2.3 Hardware Architecture

The hardware architecture consists of four medication compartments, four load cell sensors, four HX711 amplifier modules, an ESP32 microcontroller, a dual-output power supply, and Wi-Fi connectivity, complemented by an HC-SR04 distance sensor, an IR hand sensor, a DHT22 environmental sensor, solenoid locks, buzzers, and status LEDs. Each compartment is mounted on a load cell; as pills are removed, the measured weight decreases. The HX711 module amplifies and digitizes the signal from the load cell. The ESP32 reads this digital value, formats it as telemetry data, and sends it to the backend API through a wireless network connection.

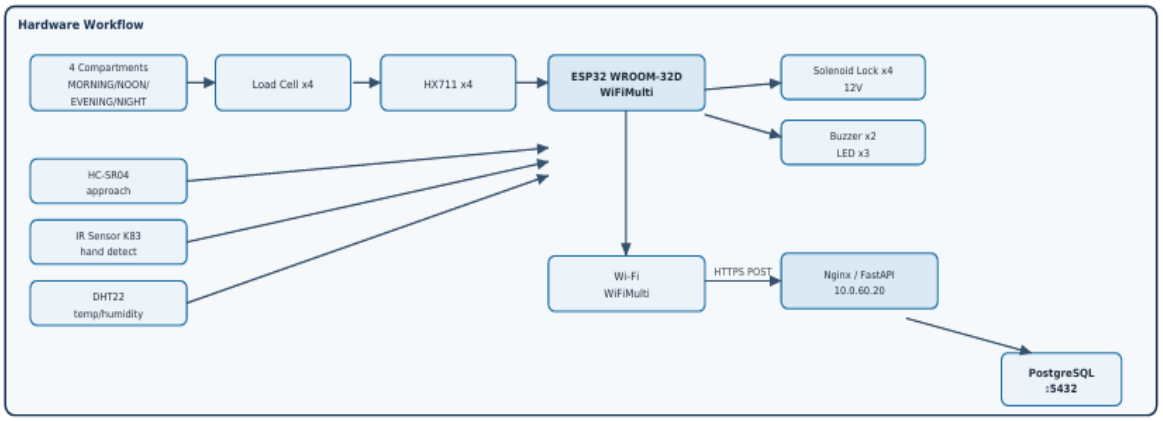

Figure 3- MediSense Hardware Workflow Diagram

2.4 Backend Architecture

The backend architecture follows a clean and layered architecture approach implemented in Python with FastAPI. The API layer exposes RESTful endpoints for frontend clients and IoT devices through nine dedicated routers.

The Service layer contains the email service, APScheduler-based medication alarm scheduler, WebSocket connection manager, and the JWT, bcrypt, and RBAC dependencies.

The Domain layer defines the core entities and enumerations of the system through SQLAlchemy ORM models.

The Persistence layer manages database access through a shared engine, session factory, and PostgreSQL.

The Infrastructure layer contains technical services such as Pillow-based avatar processing, the systemd-managed process, and Nginx reverse proxy integration.

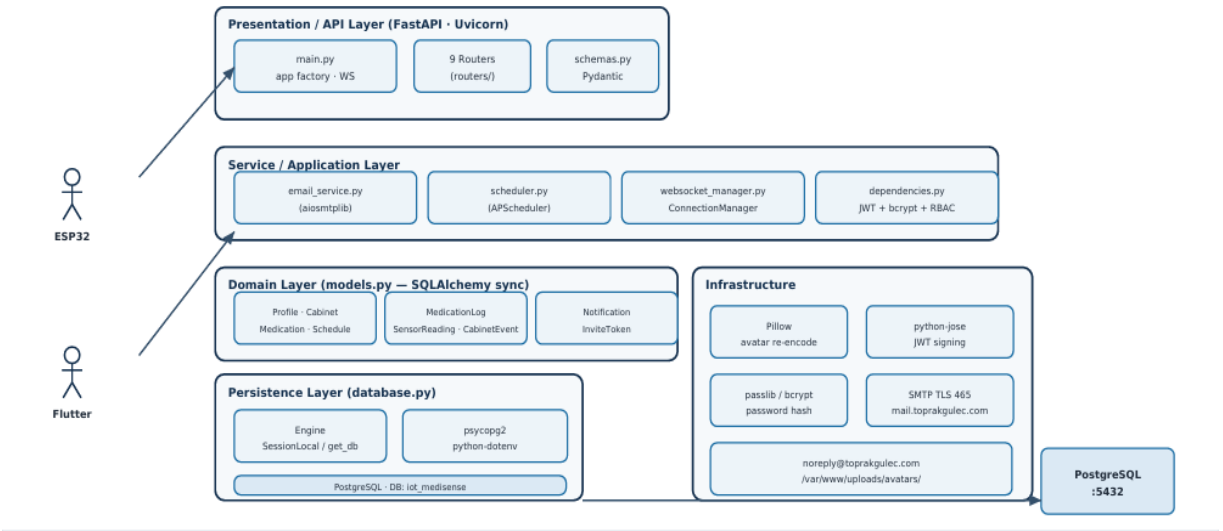

Figure 4- MediSense Backend Architecture

2.4.1 API Layer

The API layer is implemented as the FastAPI application factory in main.py, which mounts nine routers (routers/auth.py, routers/cabinets.py, routers/medications.py, routers/schedules.py, routers/logs.py, routers/sensors.py, routers/events.py, routers/notifications.py, routers/reports.py), registers the WebSocket endpoint at /ws/{cabineid}, mounts StaticFiles at /static -> /var/www/uploads/avatars/, and starts the APScheduler with check_schedules() and a cleanup job. CORS is configured with allow_origins=[“*”]. Swagger UI is accesible aat https://api.toprakgulec.com/docs . Total: 38 HTTP paths + 1 WebSocket.

2.4.2 Application Layer

services/email_service.py sends invitation and reminder emails using aiosmtplib over SMTP TLS port 465 from noreply@toprakgulec.com via mail.toprakgulec.com. Email templates include both plain-text and HTML versions with the invite deep-link landing at https://api.toprakgulec.com/auth/invite/{token} . serviced/scheduler.py uses APSchedular to run check_schedules() every minute; it fetches active schedules, compares scheduled_time against the current time window, and broadcasts medication_reminder events over WebSocket. It also runs a daily cleanup job for expired notifications (30-day TTL). services/websocket_manager.py implements ConnectionManager with cabinet-scoped connections, supporting broadcast(cabinet_id, message) used by both the scheduler and the log/sensor routers.

2.4.3 Domain Layer

The domain layer represents the innermost ring of the architecture and contains the system's core business concepts defined as SQLAlchemy ORM models. It defines the principal entities that model the cabinet environment: Profile, Cabinet, Medication, Schedule, MedicationLog, SensorReading, CabinetEvent, Notification, and InviteToken. These entities capture the full lifecycle of a medication routine — from the registration of a cabinet and its medications, through the creation of dosing schedules, to the recording of logs when a dose is taken or missed. The domain layer also defines the valid states for key concepts through enumerations: LogStatus (taken, missed, late, skipped), CabinetStatus (online, offline), Compartment (MORNING, NOON, EVENING, NIGHT), EventType, and UserRole (admin, caregiver, patient).

2.4.4 Persistence Layer

The persistence layer is responsible for all data access concerns and provides the concrete implementations of the repository interfaces declared in the application layer. It uses SQLAlchemy with a PostgreSQL provider, configured through a shared session factory and engine. Referential integrity rules — such as restricting

deletes on MedicationLog and Notification relationships — are explicitly configured to protect safety-critical data. Schema evolution is managed through idempotent hand-written SQL migration files, and the initial schema, profile extension migrations, and compartment label migrations are tracked historically.

2.4.5 Infrastructure Layer

The Infrastructure responsibilities are distributed across several files. contains get_current_user() (JWT decode using dependencies.py python-jose, HS256), password verification with passlib (bcrypt), the require_role(*roles) RBAC decorator, and the verify_service_key() dependency used by all ESP32 endpoints. Pillow is used in routers/auth.py to re-encode uploaded avatar images (jpg/png/webp, max 2MB) to JPEG and save them to /var/www/uploads/avatars/, stripping potentially malicious EXIF or embedded payloads. The avatar is served statically at https://api.toprakgulec.com/static/avatars/ {filename} via Nginx as a systemd unit (alias /var/www/uploads/avatars/. The entire backend process runs /etc/systemd/system/iot-medisense.service), logging to medisense.log. Configuration is loaded from /var/log/iot /home/toprak/IoT-MediSense/.env via python-dotenv.

2.5 Frontend Architecture

The frontend system is designed as a Flutter-based mobile application following the MVVM architectural pattern, implemented with the Riverpod state-management library. Views are responsible for displaying user interface components, while ViewModels (Riverpod notifiers) manage screen state, API requests, and business related UI logic. API services handle communication with the backend endpoints. This separation makes the frontend easier to maintain, test, and extend.

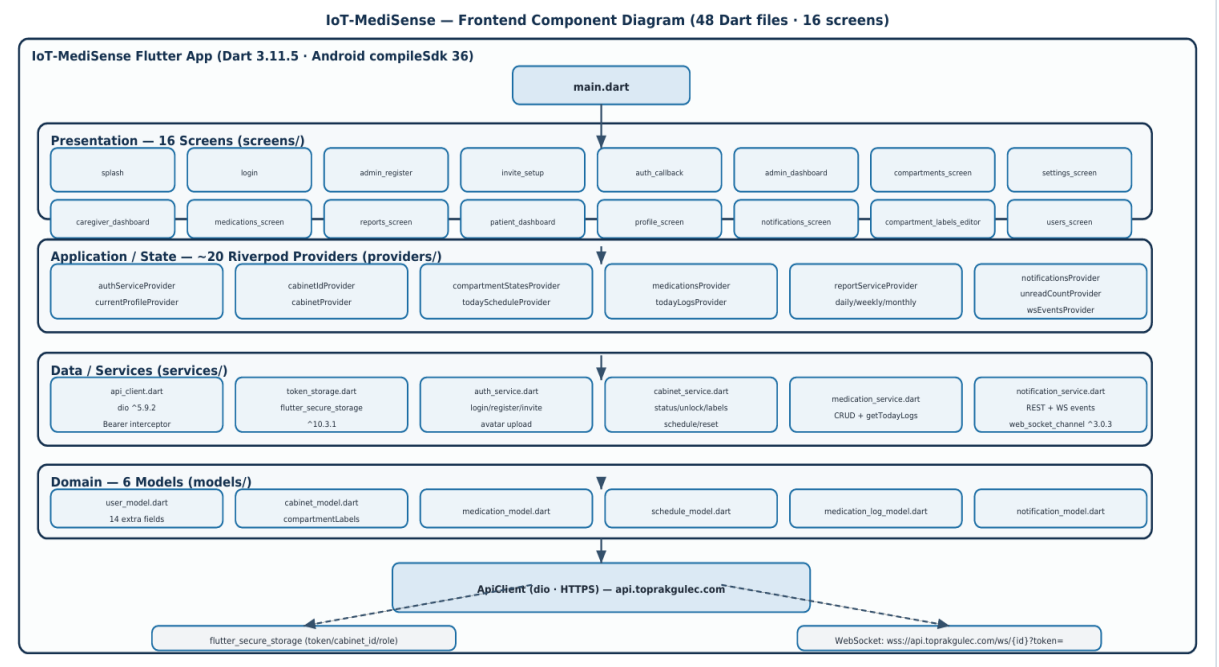

Figure 5- MediSense Frontend Component Diagram

2.5.1 Flutter User Interface

The application is built entirely with Flutter's declarative Material UI targeting Android API 21+. The design system is defined in core/theme/colors.dart (kBackground #0A0F1E, kCard #1A1F35 , kElectricBlue #00D4FF , kPrimaryBlue #1D6FA5 , kSuccess #2ECC71, kDanger #E74C3C, kWarning #F39C12) and core/theme/app_theme.dart (ThemeData). The full screen inventory: splash_screen, notifications_screen, profile_screen (admin/caregiver/patient), auth/login_screen, auth/admin_register_screen (2-step wizard), auth/invite_setup_screen (2-or-3-step dynamic wizard), auth/auth_callback_screen, cabinet/compartment_labels_editor (bottom sheet), admin/admin_dashboard, admin/compartments_screen, admin/settings_screen, admin/ users_screen, caregiver/caregiver_dashboard, caregiver/medications_screen, caregiver/ reports_screen, patient/patient_dashboard. Shared widgets: 9 reusable components including ComplianceChart (fl_chart ^1.2.0), CompartmentCard, shimmer placeholders, and flutter_animate (FadeIn + SlideY, 400ms ease-out) entry animations.

2.5.2 MVVM Pattern

The application follows MVVM: Views observe Riverpod providers (ViewModels) and forward user actions; providers call services; services use ApiClient. The 20 providers include: authServiceProvider, currentProfileProvider, tokenStorageProvider (auth layer); cabinetIdProvider, cabinetProvider, compartmentStatesProvider, cabinetUsersProvider, todayScheduleProvider, compartmentLabelsProvider (cabinet layer); medicationServiceProvider, medicationsProvider, todayLogsProvider (medication layer); reportServiceProvider, daily/weekly/monthlyReportProvider (reports layer); notificationServiceProvider, notificationsProvider, unreadCountProvider, wsEventsProvider (notification layer). The root MyApp (ConsumerStatefulWidget) sets ApiClient.onUnauthorized → router.go('/login') at startup, connects the WebSocket when a cabinet_id is available (_connectIfLoggedIn()), and listens to wsEventsProvider to show snackbars for medication_reminder events.

2.5.3 API Client and Services

services/api_client.dart is a singleton class written with the dio package and covers a total of 38 different HTTP paths by housing all REST endpoint definitions; its base URL is configured as https://api.toprakgulec.com, and it features a flutter_secure_storage repository-based Bearer-token interceptor, along with a 401 error handler possessing the onUnauthorized?.call() function which clears the token and redirects the user to the login screen when triggered. Located on the secure local storage side, the services/token_storage.dart service wraps the flutter_secure_storage package to make critical session data such as auth_token, cabinet_id, and role permanently secure on the device. Among the services forming the functional core of the system, services/auth_service.dart; exposes the login, register, acceptInvite, inviteUser, logout, me, and updateProfile functions as well as the multipart/form-data formatted uploadAvatar function which enables profile picture uploading; meanwhile, services/cabinet_service.dart manages IoT hardware integration processes such as cabinet status, user tracking, remote unlocking (unlock), scheduling (schedule), resetting (reset), and updateCompartmentLabels. The services/medication_service.dart service, which is responsible for medication tracking processes, provides full CRUD capability on medication data while also providing

the getTodayLogs function for up-to-date compliance reports; lastly, the services/notification_service.dart service, alongside standard REST-based notification management, uses the web_socket_channel package to establish an instantaneous and bi-directional WebSocket communication channel over the address wss://api.toprakgulec.com/ws/{cabinet_id}?token={jwt}, thereby synchronizing the live data stream.

2.5.4 Main Frontend Screens

The application is organized around role-specific dashboards: on startup SplashScreen waits 2s then calls getUserRole() — null → /login, admin → /admin, caregiver → /caregiver, patient → /patient. Admin Dashboard: cabinet status (is_online badge), 2×2 compartment grid with lock/unlock controls, quick-access to compartments, users, and settings screens. Caregiver Dashboard: today's schedule status per compartment with color-coded cards (kSuccess/kWarning/kDanger), weekly compliance bar chart and recent notifications panel. Medications Screen: searchable medication list with add (fab) and swipe-to-delete; medication form supports compartment picker, dosage, unit, color. Reports Screen: tabbed daily/weekly/monthly compliance via GET /reports/* . Patient Dashboard: large-type today's medications and countdown timer to next dose (accessible design for elderly users). Deep-link invite flow: invite email opens https://api.toprakgulec.com/auth/invite/{token} (HTML landing) → iot-medisense://auth/callback?token=xxx deep link → AuthCallbackScreen extracts token → InviteSetupScreen (2-or-3-step wizard) → role dashboard.

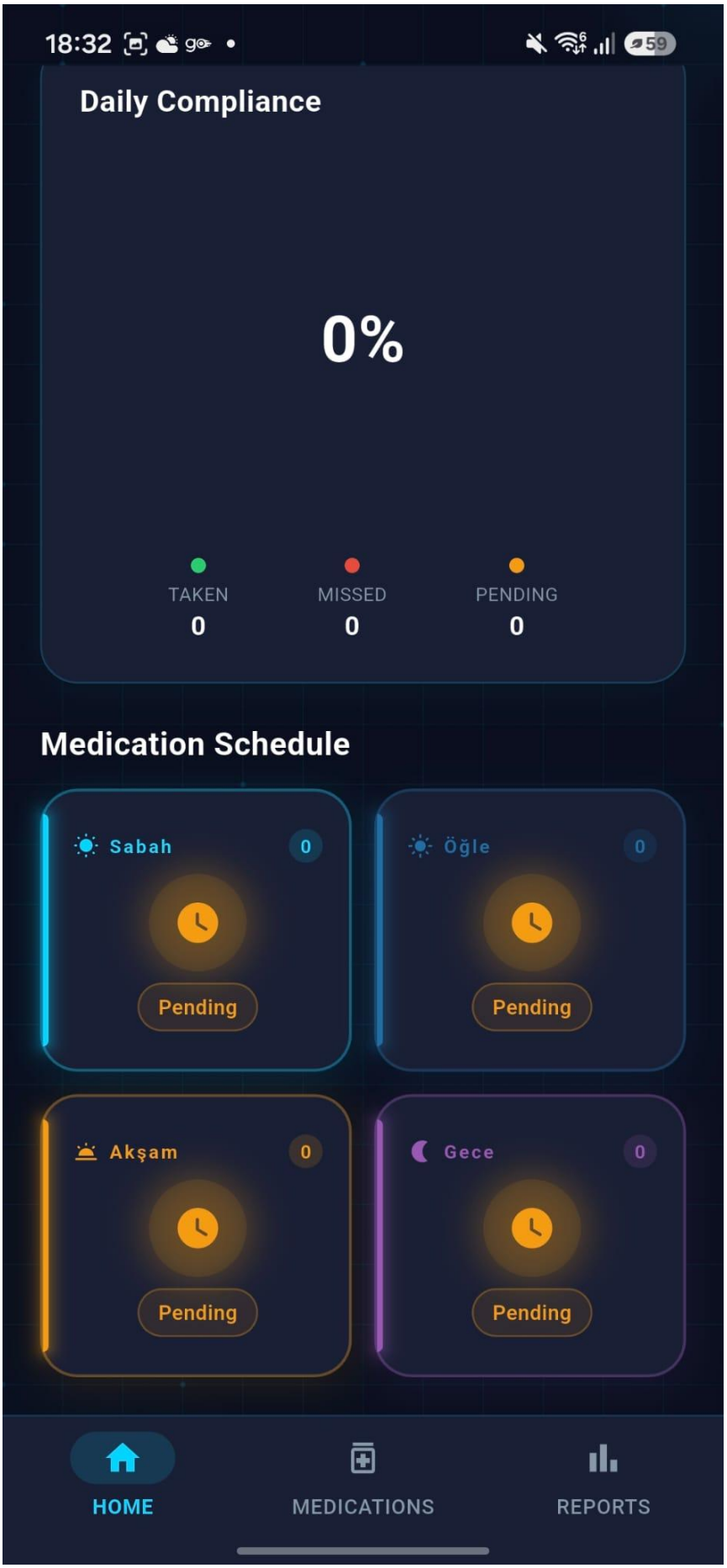
18:32
Daily Compliance
0%
TAKEN
0
MISSED
0
PENDING
0
Medication Schedule
Sabah
0
Pending
Öğle
0
Pending
Akşam
0
Pending
Gece
0
Pending
HOME
MEDICATIONS
REPORTS

Figure 6- MediSense Main Page (Caregiver Dashboard)

2.6 Database Architecture

The database is PostgreSQL, database name iot_medisense, owner medisense_user, encoding UTF-8, running at localhost:5432 (accessible only from the VPS internal network), and the schema comprises nine tables and three SQL views within a relational data model that stores profiles, cabinets, medications, schedules, medication logs, sensor readings, cabinet events, notifications, and invite tokens. Each cabinet is linked to its owner profile and its medications, where each medication is assigned to one of the four compartments and may have one or more schedules. Medication logs are stored with cabinet, medication, schedule, and user references, capturing the full dose intake event including measured weight before and after intake; meanwhile, sensor readings store ambient temperature and humidity over time, while cabinet events record lock, approach, and connectivity activity. Notifications are linked to users and carry a 30-day expiry, and the three SQL views provide aggregated daily, weekly, and monthly compliance rates, establishing a structure that allows caregivers to trace the reason and context of each missed dose or device event.

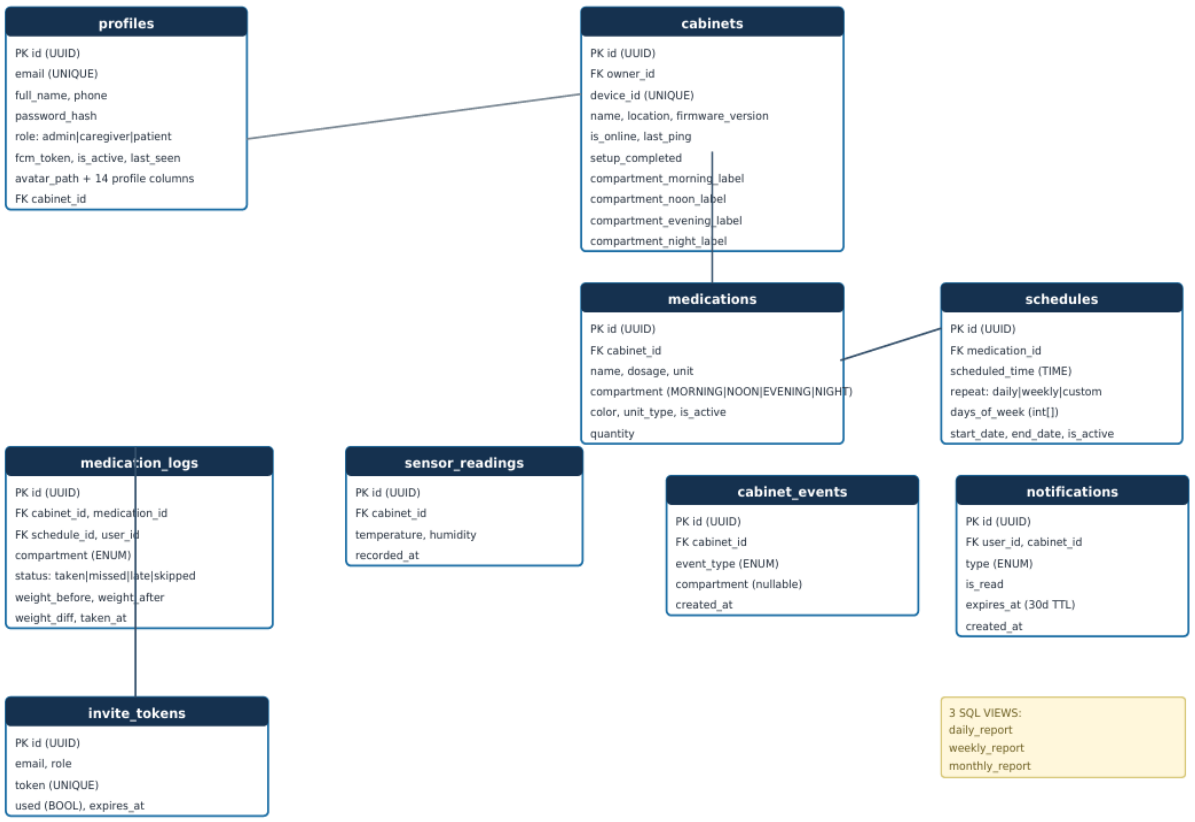

Figure 7- MediSense Domain ER Diagram

2.7 Communication Architecture

SSL is terminated at the Plesk reverse proxy on api.toprakgulec.com (Let's Encrypt). Traffic reaches the VPS (213.14.143.204 / 10.0.60.20) on port 80 via the internal network. Nginx routes /ws/ with WebSocket upgrade headers and all other paths to http://127.0.0.1:8000 . The ESP32 communicates over port 443 (HTTPS) using the SERVICE_ROLE_KEY header. The Flutter app uses JWT Bearer tokens obtained at login (POST /auth/login → access_token 7-day HS256). WebSocket token

is passed as query parameter: wss://api.toprakgulec.com/ws/{cabinet_id}?token={jwt} .

Source	Destination	Communication / Channel	Port
Esp32 Firmware	FastAPI logs, /sensors/, /sensors/event	HTTPS POST · SERVICE_ROLE_KEY header	443 (SSL term.) → 80 → 8000
Flutter App	FastAPI REST endpoints (38 paths)	HTTPS · dio · JWT Bearer header	443 → 80 → 8000
Flutter App	FastAPI /ws/ {cabinet_id}	WSS · web_socket_channel · token query	443 → 80 → 8000
FastAPI	PostgreSQL	SQLAlchemy	5432 (internal)
FastAPI	SMTP mail.toprakgulec.com	aiosmtplib · TLS · invite + reminder mail	465
FastAPI schedule	WebSocket clients	ConnectionManager.broadcast(cabinet_id, event)	internal

Table 1- MediSense Data Flow and Port Table

3. System Analysis and UML Diagrams

This chapter presents the UML diagrams used to analyze the IoT-MediSense system. The diagrams describe system actors, use cases, the data model, runtime workflows, state transitions, deployment structure, and monitoring process, providing a clear technical understanding of how components interact during normal operation.

3.1 Use Case Diagram

The use case diagram identifies the main actors and functional interactions of the system. Caregivers can log in, view dashboard data, manage medications and schedules, view compliance reports, and acknowledge notifications. Patients can view their dashboard and today's medications, and interact with the cabinet to take their doses. System administrators manage users, cabinets, and medications, and can invite users by email with deep-link tokens. ESP32 IoT devices interact with the system by sending sensor readings, cabinet events, and heartbeat data. The use case diagram therefore defines the functional scope of the IoT-MediSense system from the perspective of each actor.

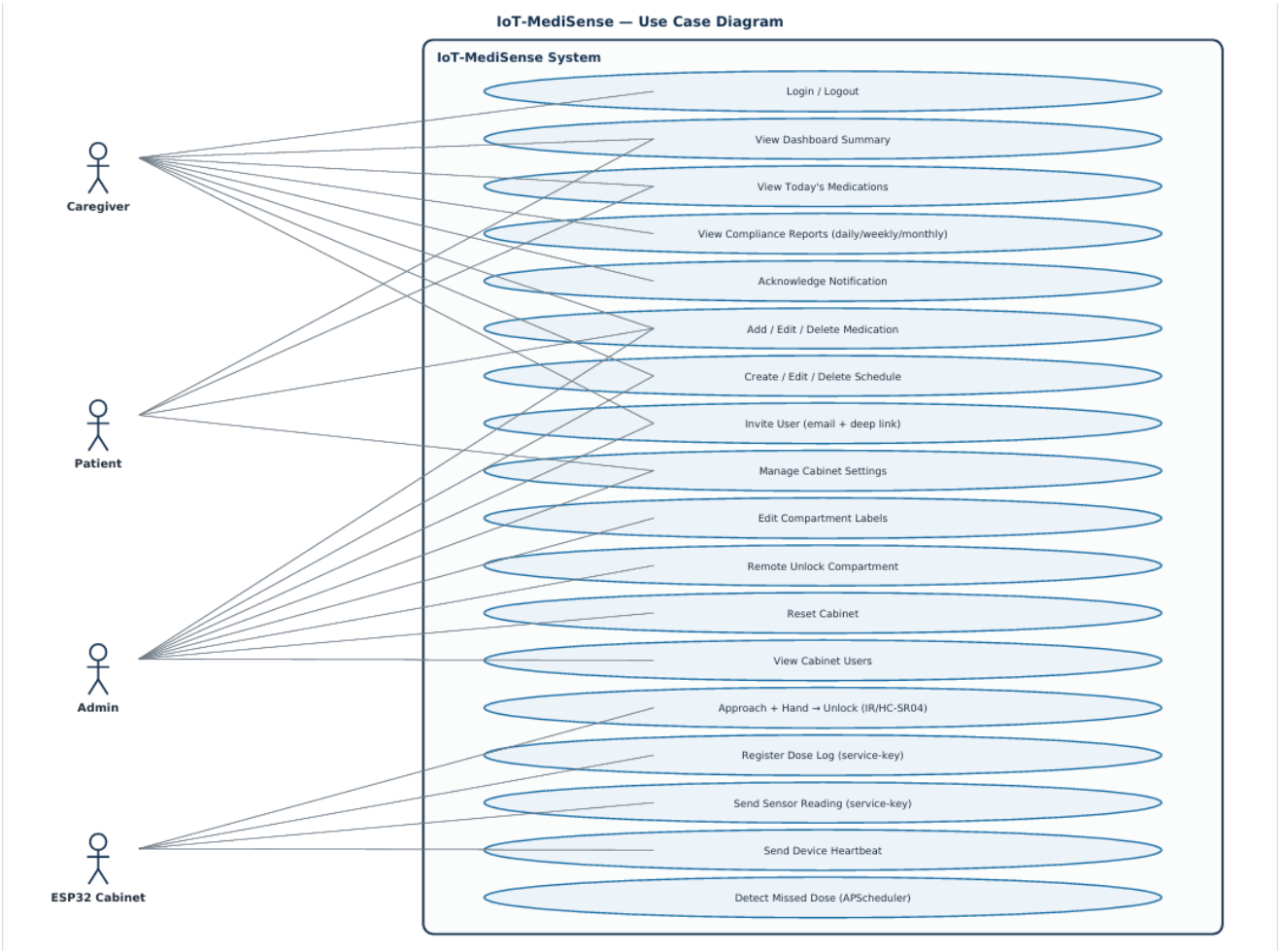

Figure 8- MediSense Use Case Diagram

3.2 Entity Relationship Diagram

The entity relationship diagram defines the main data structure of the system. A profile owns or is assigned to a cabinet. Cabinets contain medications, each assigned to one of four compartments. Medications have one or more schedules that define when and how often a dose is due. Medication logs reference both the cabinet and the schedule, capturing the dose status and measured weight values for each intake event. Sensor readings store real-time environmental telemetry, while cabinet events record physical lock, approach, and connectivity activity. Notifications store missed dose and device-offline messages with a 30-day expiry. Invite tokens manage email based onboarding with role assignment

3.3 End to End Telemetry Sequence Diagram

The end-to-end telemetry sequence diagram explains how data flows from the physical compartment to the frontend dashboard. The compartment weight is measured by the load cell and converted into digital data by the HX711 module. The ESP32 sends this value to the backend API. The backend identifies the cabinet by its device code, finds the active schedule for the compartment, interprets the weight drop as a dose intake, saves the medication log and sensor reading, and creates an alert when a dose is missed. Finally, the WebSocket manager broadcasts the update so the frontend displays the latest adherence status to caregivers

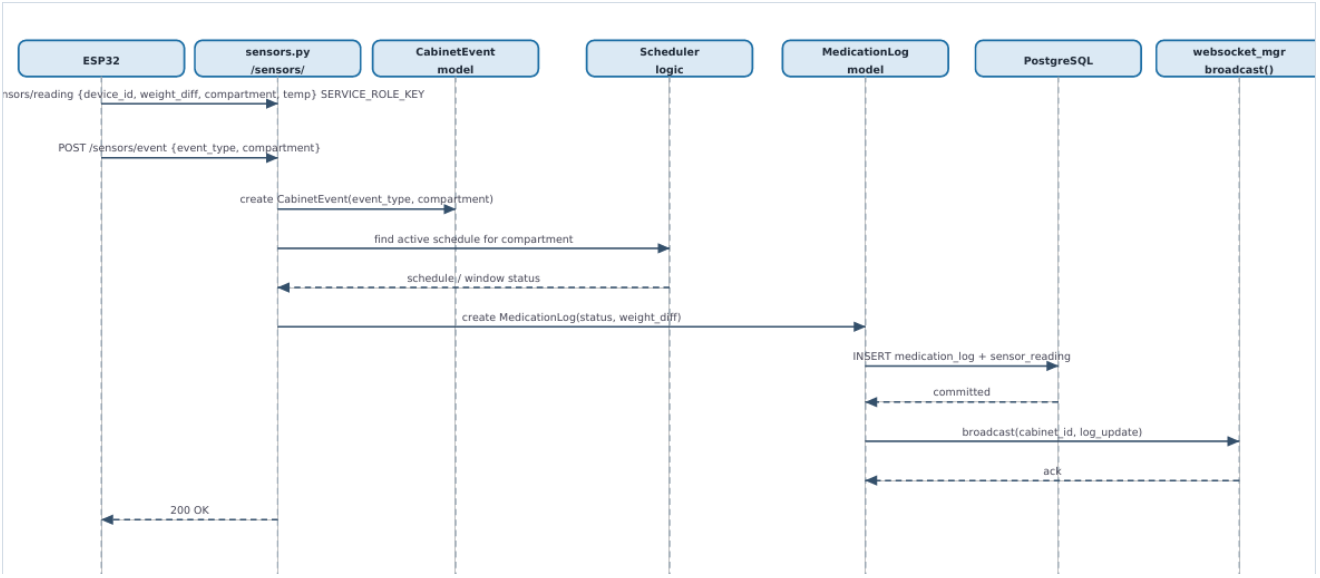

Figure 9- MediSense End to End Telemetry Sequence Diagram

3.4 Add Medication and Schedule Sequence Diagram

The add medication and schedule sequence diagram shows how a caregiver creates a new medication record and dosing schedule. The caregiver selects the medication details and target compartment from the frontend. The frontend sends

this information to the backend API. The backend validates the request against role-based access rules, stores the medication and schedule in the database, and registers the corresponding APScheduler alarm job. After the records are created, the frontend updates the medication list.

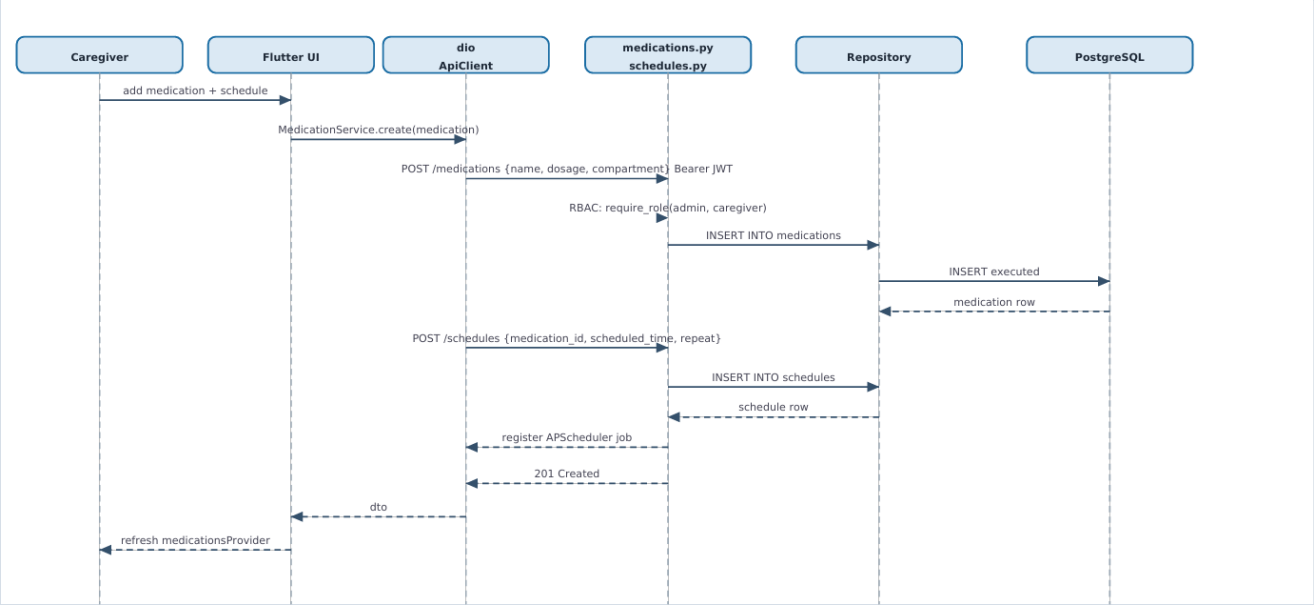

Figure 10- MediSense Add Medication and Schedule Sequence Diagram

3.5 Compartment Unlock and Dose Intake Sequence Diagram

The compartment unlock and dose intake sequence diagram describes how the system handles a scheduled dose event. When the medication time is reached, the APScheduler service triggers the buzzer and dispatches a WebSocket notification to the Flutter app. The patient approaches the cabinet, detected by the HC-SR04 sensor, and presents a hand to the IR sensor. The ESP32 unlocks only the compartment matching the active schedule. After the patient removes the pill, the load cell registers a weight drop, the ESP32 posts the log to the backend, and the backend records the medication log with the measured weight difference.

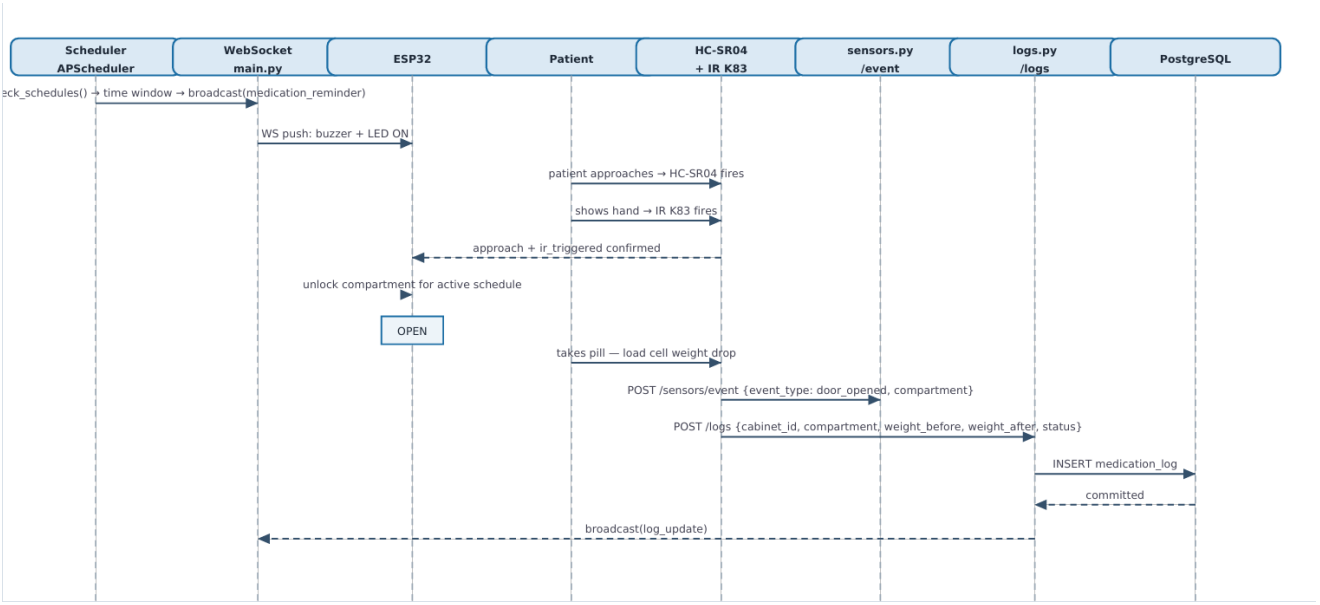

Figure 11- MediSense Compartment Unlock and Dose Intake Sequence Diagram

3.6 System State Diagram

The state diagram defines the possible states of compartments, doses, and the cabinet device. A compartment may be locked or unlocked depending on the schedule and the approach and hand detection events. A scheduled dose may be pending, taken, or missed depending on whether intake is confirmed within the scheduled window. The cabinet device may be online or offline depending on whether it sends heartbeat or telemetry data. These states help define the lifecycle of the main system objects.

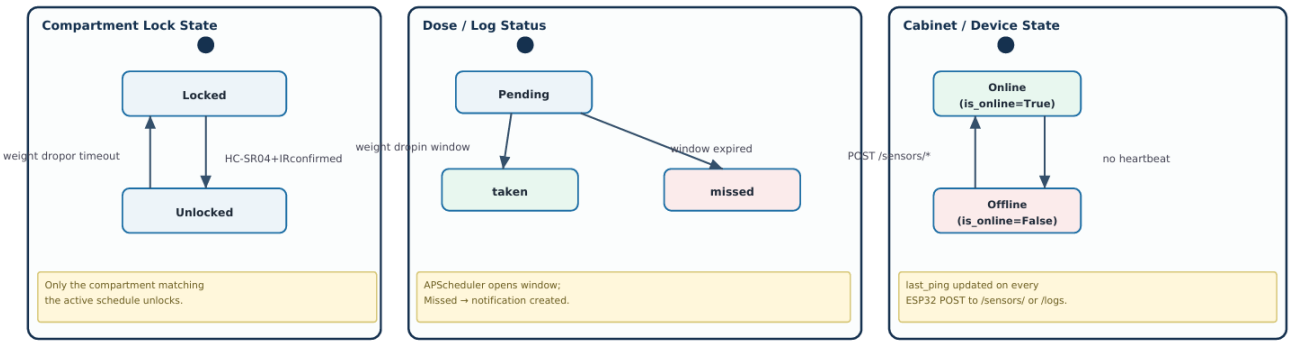

Figure 12- MediSense System State Diagram

3.7 Deployment Diagram

The deployment diagram shows how the system is physically and logically deployed. The cabinet contains the four compartments, load cells, HX711 modules, sensors, locks, and the ESP32 device. Caregivers and patients access the Flutter application from an Android device. The Nginx reverse proxy and FastAPI backend, together with the APScheduler, WebSocket manager, and email services, run on an application server managed as a systemd service, while the PostgreSQL database runs on the same server accessible only from the internal network. Communication between the ESP32, frontend, backend, and database occurs over the local Wi-Fi or internet connection.

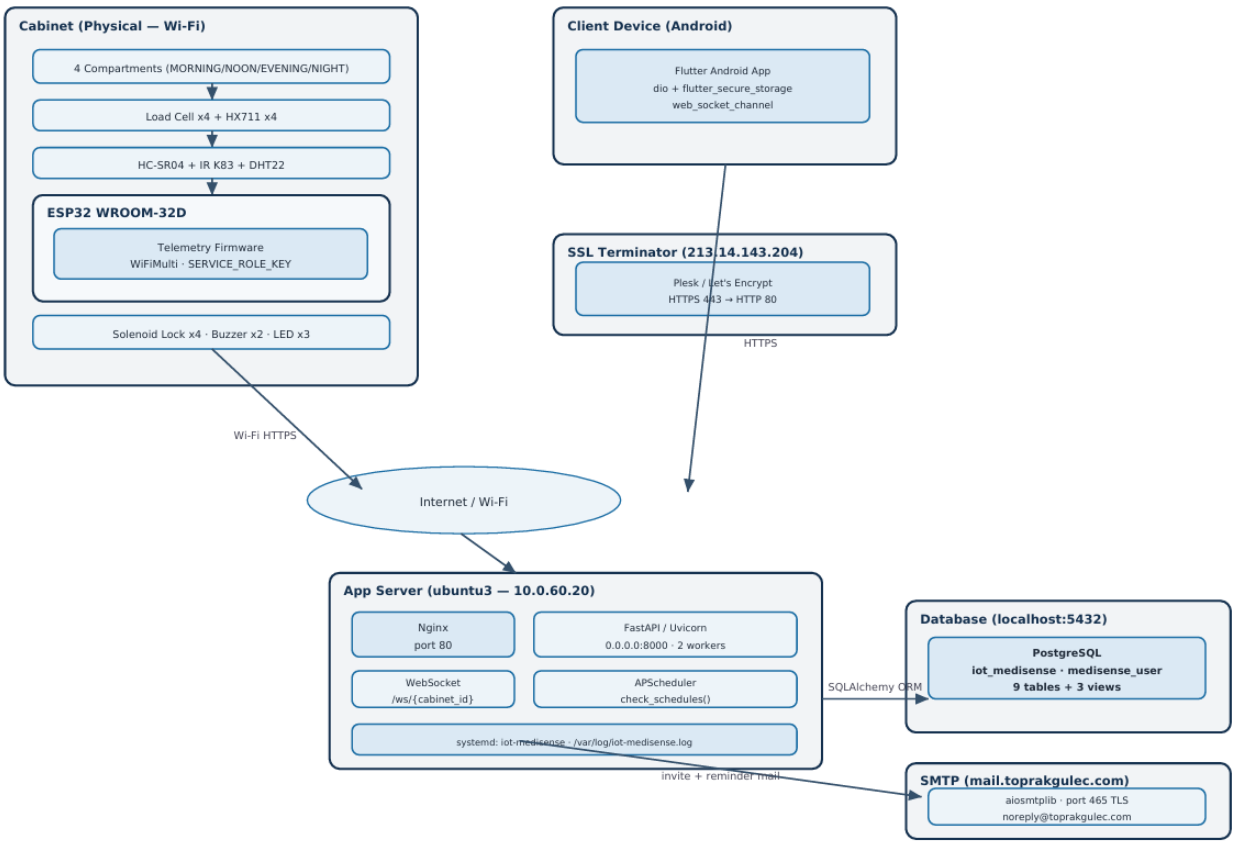

Figure 13- MediSense Deployment Diagram

3.8 Complete Monitoring Workflow Activity Diagram

The activity diagram summarizes the complete monitoring workflow from system setup to alert handling. First, cabinet data, users, medications, and schedules are registered. Then the APScheduler begins monitoring medication windows. The ESP32 is alerted at medication time and activates the buzzer. The patient approaches, the compartment unlocks, and the patient takes the pill. The load cell detects the weight drop, the ESP32 posts the log to the backend, and the backend calculates the dose status and checks thresholds. If the dose is taken within the window it is marked taken; otherwise it is marked missed and a notification is generated. The frontend displays the updated compliance status and the caregiver acknowledges the alert and takes necessary action.

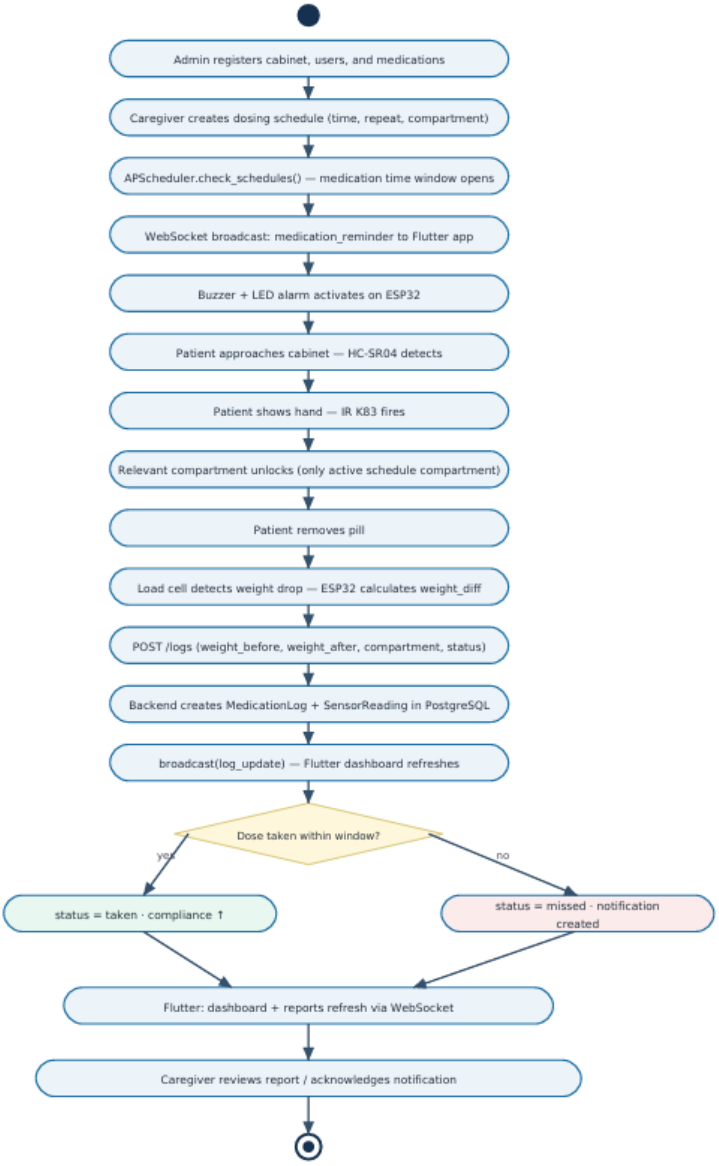

Figure 14- MediSense Complete Monitoring Workflow Diagram

4. Example

To illustrate the practical operation and real-time data orchestration of the MediSense system, a complete end-to-end operational scenario can be examined. In this scenario, a primary caregiver logs into the Flutter-based mobile application to configure a daily cardiovascular medication routine for an elderly patient. Using the user-friendly interface, the caregiver schedules a blood pressure medication to be administered from the physical cabinet's "MORNING" compartment precisely at 08:00 AM every day. The mobile application captures this event, and the custom ApiClient handles the authentication layer by injecting the signed 7-day expiration JSON Web Token (JWT) into the request header. This structured payload is transmitted via a secure HTTPS POST request to the Python-based FastAPI backend deployed on the Virtual Private Server behind an Nginx reverse proxy. Upon receiving the payload, the backend validates the data schema, maps it using the SQLAlchemy and SQLModel Object-Relational Mapping (ORM) toolkit, and commits the records directly into the Schedule table within the local PostgreSQL database (iot_medisense).

At exactly 08:00 AM, the asynchronous background task engine managed by APScheduler triggers its minute-by-minute evaluation loop (check_schedules()). The system instantly identifies that the designated dose window for the patient's morning medication has opened. The backend immediately dispatches a real-time bi-directional telemetry packet via an open WebSocket pipeline managed by the server's ConnectionManager class. This state mutation simultaneously broadcasts a push alert to the patient's mobile application and transmits a command string to the physical hardware. The local ESP32 WROOM-32D microcontroller, which maintains an active wireless connection via the WiFiMulti library, captures this incoming command. It immediately initiates the local alert sequence, causing the active buzzer to sound a rhythmic notification and triggering the synchronized status LEDs to blink aggressively, signaling to the patient that it is time to take their scheduled dose.

When the patient physically walks over to the custom MDF cabinet to retrieve their medication, the spatial environment sensors actively track their physical presence. The integrated HC-SR04 ultrasonic distance sensor registers a sharp decrease in the proximity path, confirming that a user is standing directly in front of the device. To bypass mechanical contact and maintain strict hygiene, the

patient executes a touchless hand-wave gesture in front of the K83 infrared (IR) proximity sensor. The ESP32 processes this local proximity event, verifies that the user intent matches the active dose window, and drives a physical relay module to channel a +12V DC current from the Mean Well RD-65A dual-output switching power supply. This electrical impulse activates the electromagnetic solenoid lock dedicated exclusively to the "MORNING" compartment, allowing the cabinet door to spring open while the remaining three compartments stay securely locked. Because the lojik circuits of the MCU are isolated on a separate +5V DC rail of the RD-65A PSU, the high-current inductive load generated by the solenoid latch does not cause a brownout reset on the microcontroller.

Once the compartment is unlatched, the patient removes the specific medication container from the floating tray. The continuous gravitational force exerted on the physical compartment changes, causing a strain gauge variation in the local load cell sensor. The associated HX711 24-bit analog-to-digital converter (ADC) module digitizes this raw microvolt-level physical telemetry into a precise weight value and hands it over to the ESP32. The microcontroller calculates the weight differential, packages the numeric telemetry alongside a unique hardware event token, and executes an asynchronous HTTP POST request to the backend's ingestion path, validating its identity using a hardcoded SERVICE_ROLE_KEY authentication header.

Upon receiving the hardware telemetry payload, the FastAPI backend routes the data straight into the central Adherence Engine. The engine compares the incoming weight drop timestamp against the active scheduling window and verifies that the container was successfully removed within the allowed timeframe. It then logs the medication status as taken inside the MedicationLog table of the PostgreSQL database, which automatically updates the daily compliance statistics aggregated by the pre-compiled database SQL Views. Finally, the backend pushes a live confirmation payload via WebSockets to the caregiver's mobile dashboard, instantly rendering a success chart component via the fl_chart visualization framework. This comprehensive workflow demonstrates how the integrated layers of the MediSense system successfully transform an unmonitored home health routine into a highly traceable, automated, and secure medical adherence environment.

5. Limitations and Future Work

This chapter explains the current limitations of the IoT-MediSense system and identifies possible future improvements. Since the project is designed as an IoT course project, the current implementation focuses on proving the main adherence monitoring concept with the ESP32, load cells, backend API, database, and frontend visualization. However, several improvements would be necessary before using the system in a real care environment.

5.1 Current Limitations

The primary technical and operational limitations of the current prototype revolve around physical sensing, environmental dependencies, and integration constraints. On a hardware and mechanical level, the system relies on load-cell readings that are highly susceptible to calibration quality, vibration, mounting instability, and external forces, which can skew the detection of a dose intake event. The current model assumes a consistent per-pill weight and does not account for partial doses or varied pill sizes within a single compartment. Because the hardware is designed as an academic demonstration rather than a certified medical-grade device, it lacks the electrical safety certifications, clinical validation, and regulatory approvals required for active patient care.

Operationally, the system is strictly bounded by network dependencies, meaning that temporary Wi-Fi signal loss on the ESP32 directly interrupts real-time monitoring. Additionally, the system lacks per-device authentication — all ESP32 devices share a single service-role key — and there is no offline data buffering on the firmware side. The notification pipeline currently relies exclusively on WebSocket, which requires the Flutter application to be open; Firebase Cloud Messaging integration is planned but not yet implemented. The frontend does not yet cache data offline, there is no automated test suite, and the deployment pipeline is entirely manual. Finally, the platform operates in isolation, lacking integrations with pharmacy systems, electronic health records, or emergency contact services.

5.2 Future Improvements

To enhance data accuracy, responsiveness, and hardware resilience, future iterations of the IoT-MediSense system will focus on technical and physical optimizations. Introducing offline data buffering on the ESP32 will allow local caching during Wi-Fi dropouts, ensuring seamless synchronization once connectivity is restored. Per-device authentication using signed certificates or individual API keys will replace the shared service key. A calibration interface will allow administrators to record per compartment load-cell calibration factors directly into the database using known weights. Furthermore, completing the Firebase Cloud Messaging integration will enable push notifications to reach caregivers even when the application is not in the foreground, and fixing the known WebSocket handshake issue will ensure stable real time connections.

Expanding beyond the core tracking capability, the system's analytical depth, security stratum, and interoperability will be systematically upgraded. The alert engine will transition toward predictive logic that analyzes historical patterns and patient risk profiles to forecast potential non-adherence before it occurs. Security protocols will be tightened through rate-limiting on authentication endpoints, refresh-token support, and stricter CORS policies. An automated CI/CD pipeline and database migration tooling will replace the current manual deployment process. Ultimately, the standalone platform will scale into a more complete solution by establishing interoperability with pharmacy refill services, electronic health records, and family or emergency notification channels.

6. Biography

Sina Toprak Güleç is a Management Information Systems student at Işık University. His academic interests include the Internet of Things (IoT), software development, backend systems, database management, mobile application development, cloud computing, cyber security, and networking. He possesses a strong passion for architecting and engineering projects across both Linux and Windows environments, spanning from polished graphical user interfaces (UI) to low-level console applications within black terminal screens. In this project, he worked on the comprehensive design and development of the MediSense system, an IoT-based smart medication cabinet solution. The project allowed him to seamlessly combine hardware components such as the ESP32 microcontroller, load cell sensors, HX711 modules, a Mean Well RD-65A dual-output power supply, and 12V solenoid locks with advanced software technologies including a Python-based FastAPI backend, SQLAlchemy and SQLModel ORM, a PostgreSQL database, and a Flutter and Dart-based mobile application. Through this work, he gained practical, end-to-end engineering experience in deploying a secure, real-time healthcare monitoring environment that successfully connects physical IoT devices, microservices, robust relational databases, and dynamic user interfaces within a unified architecture.