SOFTWARE ENGINEERING (CSE-321)
ODD SEMESTER - 2025-26

<u>PROJECT REPORT</u>

META-LEARNING FOR EDGE-IOT DEVICES

SUBMITTED BY:
MANAS YADAV (23UCS637)
NAMAN JAIN (23UCS653)
VARUN KRISHNAKUMAR (23UCS732)
VEDANT JAISWAL (23UCS736)

UNDER THE GUIDANCE OF:
DR. ANUBHAV SHIVHARE
DR. ASHISH KUMAR DWIVEDI

ABSTRACT

This project addresses the challenge of efficiently training and deploying machine learning models on resource-constrained Edge-IoT devices by developing a Human Activity Recognition (HAR) system as a proof of concept. The model leverages Meta-Learning, specifically Prototypical Networks, to enable rapid on-device adaptation and continuous learning with minimal data transfer. The system was designed and implemented using modern software engineering principles, following a structured software development lifecycle that includes detailed requirements analysis, modular architecture, and well-justified design decisions. TensorFlow and React Native were used for model development and application integration, complemented by comprehensive testing and automated deployment workflows. The final system achieved a significant reduction in model training time and data usage, demonstrating the significance of meta-learning and robust engineering practices.

TABLE OF CONTENTS

INTRODUCTION

The rapid growth of the IoT has intensified the demand for intelligent systems capable of performing real-time inference and personalized adaptation. HAR is a key enabler in domains such as smart homes, healthcare monitoring, assisted living, and fitness applications. However, deploying machine learning models directly on resource-constrained Edge-IoT devices remains challenging due to their limited processing power, memory, and battery capacity.

This project investigates the potential of Meta-Learning techniques, especially the Prototypical Networks to produce models that could adapt effectively in such constrained environments. Since full on-device deployment is outside the scope of this semester’s timeframe, the trained model runs on a standalone backend and communicates with a mobile application through lightweight API calls, enabling real-time inference.

The goal of this work is to develop a proof-of-concept HAR system that demonstrates the feasibility of using meta-learning by combining advanced learning techniques with robust software engineering methodologies.

CHAPTER 1 - REQUIREMENTS ANALYSIS AND PLANNING

1.1 Problem Statement

Problems:

- Edge IoT devices (smartphones) have limited resources, so running heavy models locally is not feasible.
- Existing systems lack an explicit feedback mechanism for continuous refinement.

Need:

- A centralized HAR model that can serve multiple users.
- A lightweight client-server communication setup where the smartphone only collects and sends data.
- A feedback-driven approach that fine-tunes the central model.

Proposed Solution:

- Deploy a meta-learning–based HAR model on a central web server.
- Collect smartphone sensor data (accelerometer, gyroscope) through API calls.
- Perform activity classification on the server and return results to the mobile app.
- Gather user feedback on prediction accuracy to improve the model.

1.2 Project Scope

The system is intended to support real-time human activity recognition on data from edge computing devices, such as smartphones. It utilizes standard datasets (UCI HAR) for training and validation.

Since the project timeline is only one semester long, there is not enough time to learn the intricacies of multiple edge IoT devices to implement a standardized deployed model setup. Additionally, such devices are resource-constrained and not very computationally powerful, making local deployment even more difficult. Thus, a central model will be trained and deployed on a web server using FastAPI and communicate through API calls with a mobile application on smartphones developed with React Native for cross-platform (Android and iOS) compatibility.

Potential real-world applications of a fully developed system include personalized fitness analytics, continuous patient activity monitoring in healthcare settings, fall-detection and assisted-living systems, smart-home automation that adapts to user behavior, and intelligent industrial IoT monitoring where rapid, few-shot adaptation is essential.

1.3 Functional Requirements

1.3.1 Data Acquisition

- Collect raw motion sensor data from accelerometers and gyroscopes.
- Validate and sanitize input and reshape it into (128x6) format where each of the six sensors (accelerometer(x, y, z) and gyroscope(x, y, z)) take 128 sensor motion values.

1.3.2 Meta-Learning Training

- Train base models using Prototypical Networks on benchmark datasets.
- Host pre-trained models on a web back-end to process data coming from edge devices.

1.3.3 Activity Recognition

- Classify human activities (walking, walking-downstairs, walking-upstairs, laying, standing).
- Provide predictions with low latency for real-time usage.

1.3.4 Feedback and Personalization

- Enable users to give feedback on classifications.
- Update the model incrementally to improve accuracy.

1.4 Non-Functional Requirements

1.4.1 Reliability

The system shall achieve maximum uptime under normal operating conditions on supported devices.

1.4.2 Usability

The user interface shall provide an intuitive experience, requiring no more than three steps for users to perform any action.

1.4.3 Portability

The software shall be deployable on Android and iOS.

1.4.4 Maintainability

The codebase shall follow modular design principles, allowing updates to meta-learning algorithms without impacting other components.

1.4.5 Scalability

The system shall support extensions to additional activities and datasets without major redesign.

1.5 Other Requirements

1.5.1 Regulatory and Compliance Requirements

The system shall comply with applicable data protection and privacy regulations (Digital Personal Data Protection Act (DPDPA) 2023).

1.5.2 Platform Requirements

Minimum platform specifications: Android 9/iOS 13 or higher.

1.5.3 Licensing and Open-Source Dependencies

Any third-party libraries or frameworks (PyTorch, Expo) shall comply with open-source licenses.

1.6 Tech Stack

1.6.1 Mobile App

- React Native (with TypeScript) - Cross Platform Compatibility.
- Expo Framework - Reduces boilerplate and provides structure.

1.6.2 Web Backend

- FastAPI (with Python) - Lightweight and easy-to-use.
- MongoDB - Simple NoSQL database.

1.6.3 Model

- PyTorch - Rich ecosystem and flexibility.

1.7 Software Process Model

Considerations:

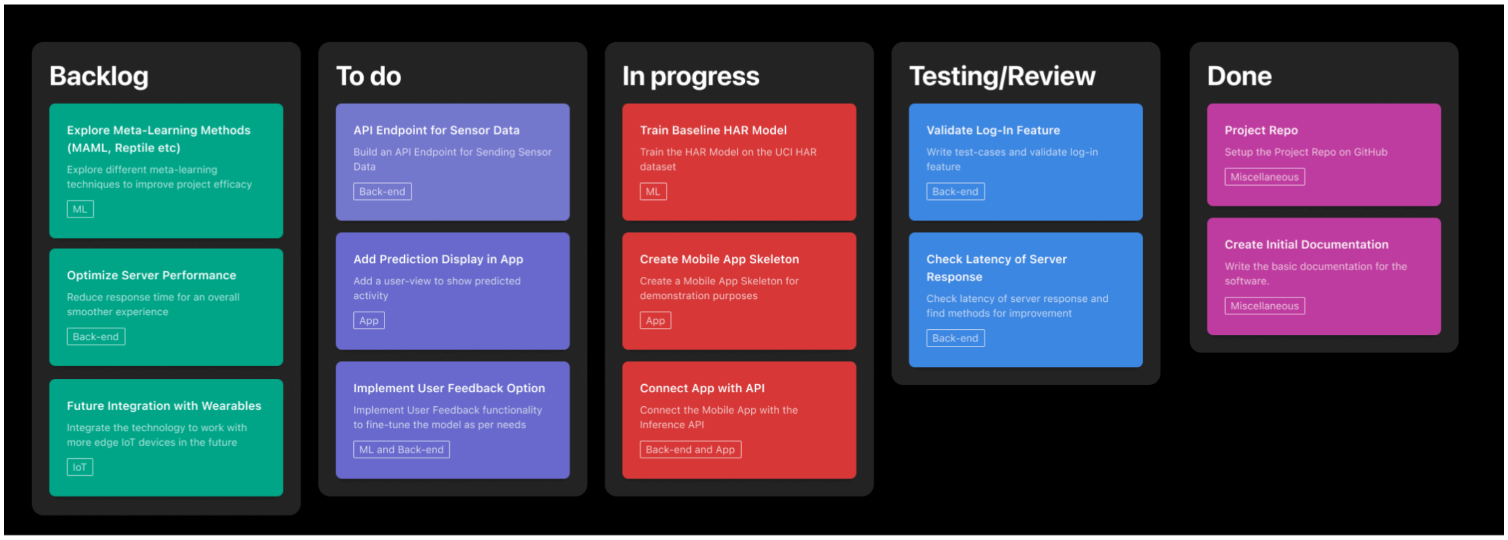
- **Duration:** Small-scale, short-duration university project. A lightweight process model should be preferred.
- **Clarity of Requirements:** Requirements may need to be adapted for certain features (such as model tuning and feedback). A flexible process model would be ideal.

- **Complexity:** Since Meta-Learning and HAR personalization involve frequent experimentation and testing, a model that supports frequent evaluation and adjustment is preferred.

Chosen Process Model: Agile Framework (KANBAN)

Justification:

- Visual, lightweight and flexible.
- The project is exploratory with continuous improvements and experimentation.
- Avoids overhead of Scrum due to a less rigid and more adaptable structure.
- Ensures the project remains flexible, user-driven and deliverable within limited time.



Backlog

Explore Meta-Learning Methods (MAML, Reptile etc)
Explore different meta-learning techniques to improve project efficacy
ML

Optimize Server Performance
Reduce response time for an overall smoother experience
Back-end

Future Integration with Wearables
Integrate the technology to work with more edge IoT devices in the future
IoT

To do

API Endpoint for Sensor Data
Build an API Endpoint for Sending Sensor Data
Back-end

Add Prediction Display in App
Add a user-view to show predicted activity
App

Implement User Feedback Option
Implement User Feedback functionality to fine-tune the model as per needs
ML and Back-end

In progress

Train Baseline HAR Model
Train the HAR Model on the UCI HAR dataset
ML

Create Mobile App Skeleton
Create a Mobile App Skeleton for demonstration purposes
App

Connect App with API
Connect the Mobile App with the Inference API
Back-end and App

Testing/Review

Validate Log-In Feature
Write test-cases and validate log-in feature
Back-end

Check Latency of Server Response
Check latency of server response and find methods for improvement
Back-end

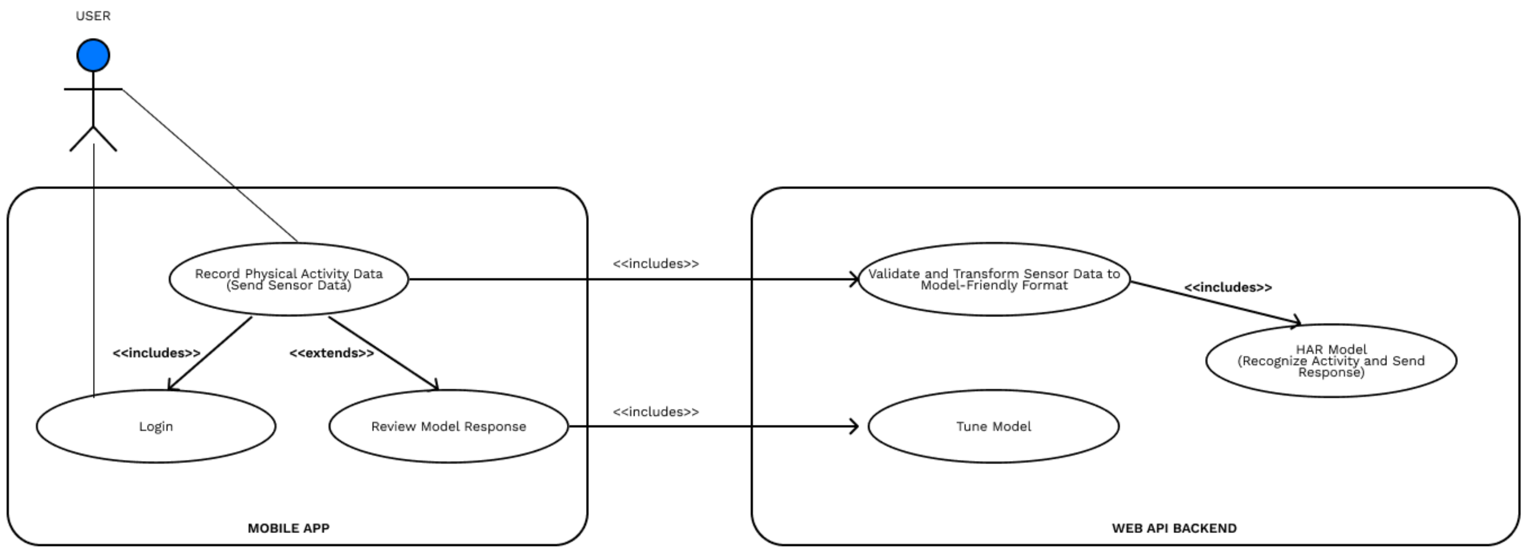
Done

Project Repo
Setup the Project Repo on GitHub
Miscellaneous

Create Initial Documentation
Write the basic documentation for the software.
Miscellaneous

1.8 Use-Case Diagram

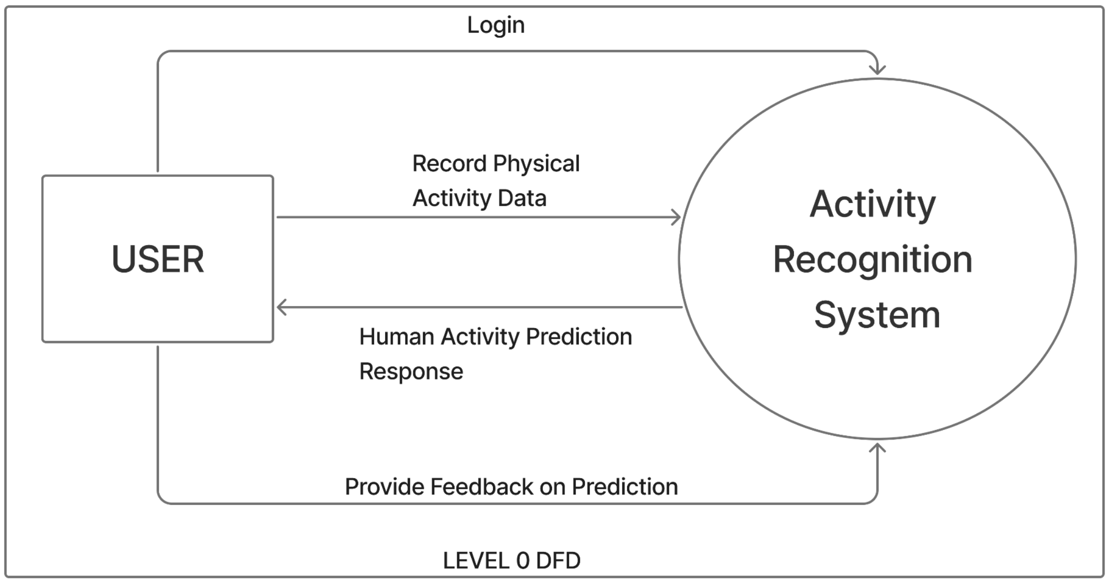
- **Actor:** User who only interacts through the mobile app.
- **Systems:** Mobile App for front-end user interaction and Web API back-end to host pre-trained HAR Model.
- **Actions:** The user can only record physical activity data or log-in to the app.


USER
Record Physical Activity Data (Send Sensor Data)
<<includes>>
<<extends>>
Login
Review Model Response
MOBILE APP
<<includes>>
<<includes>>
Validate and Transform Sensor Data to Model-Friendly Format
<<includes>>
HAR Model (Recognize Activity and Send Response)
Tune Model
WEB API BACKEND

1.9 Data Flow Diagrams

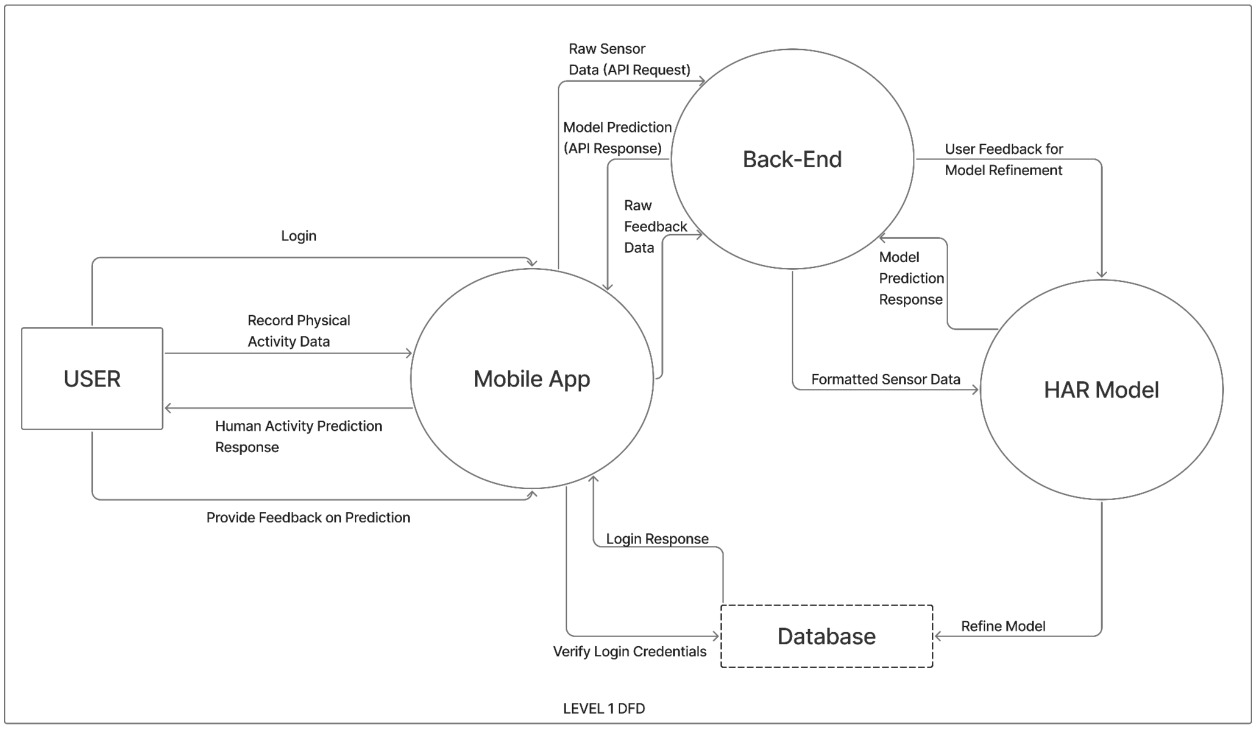
1.9.1 Level-0 DFD

- **User:** Can login in, record physical activity data and provide feedback on prediction.
- **Activity Recognition System:** Handles authentication, processes received data, provides predictions and refines itself on feedback.


Login
Record Physical Activity Data
USER
Activity Recognition System
Human Activity Prediction Response
Provide Feedback on Prediction
LEVEL 0 DFD

1.9.2 Level-1 DFD

- **Mobile App:** Handles front-end requests (login, sensor data input, feedback transmission) from the user.
- **Back-End:** Acts as the request and response mediator between the mobile app and the HAR model.
- **HAR Model:** Receives sensor data from the backend and provides prediction. Also fine-tunes itself based on feedback received by the user.

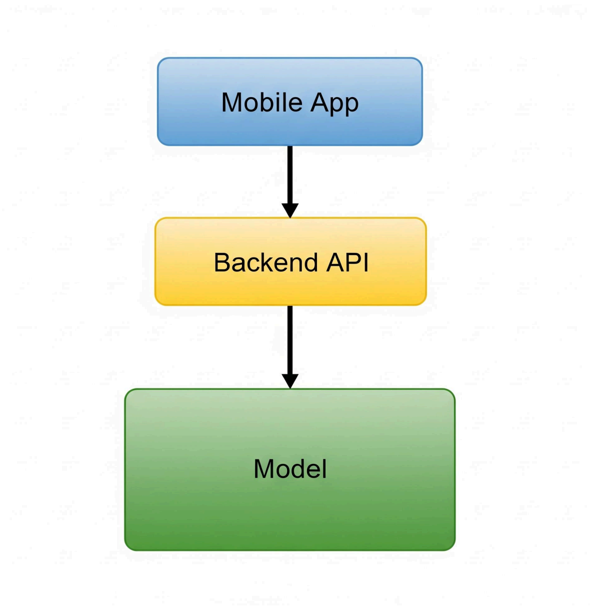

Raw Sensor Data (API Request)
Model Prediction (API Response)
Back-End
User Feedback for Model Refinement
Raw Feedback Data
Login
Model Prediction Response
Record Physical Activity Data
USER
Mobile App
Formatted Sensor Data
HAR Model
Human Activity Prediction Response
Provide Feedback on Prediction
Login Response
Database
Refine Model
Verify Login Credentials
LEVEL 1 DFD

<u>CHAPTER 2 - SOFTWARE ARCHITECTURE AND DESIGN</u>

2.1 Architectural Design Strategy

2.1.1 Architectural Foundation

The system uses a Layered Architectural Style, which organizes components into hierarchical layers.



- **<u>Presentation Layer:</u>** Mobile App (React Native client capturing sensor data).
- **<u>Application Layer:</u>** Backend API Gateway (Validates, authenticates and routes requests).
- **<u>Processing Layer:</u>** Model (PyTorch model performing meta-learning inference).

2.1.2 Design Principles

This structure adheres to the Separation of Concerns principle, which states that complex problems are better handled when subdivided into pieces that can be solved or optimized independently.

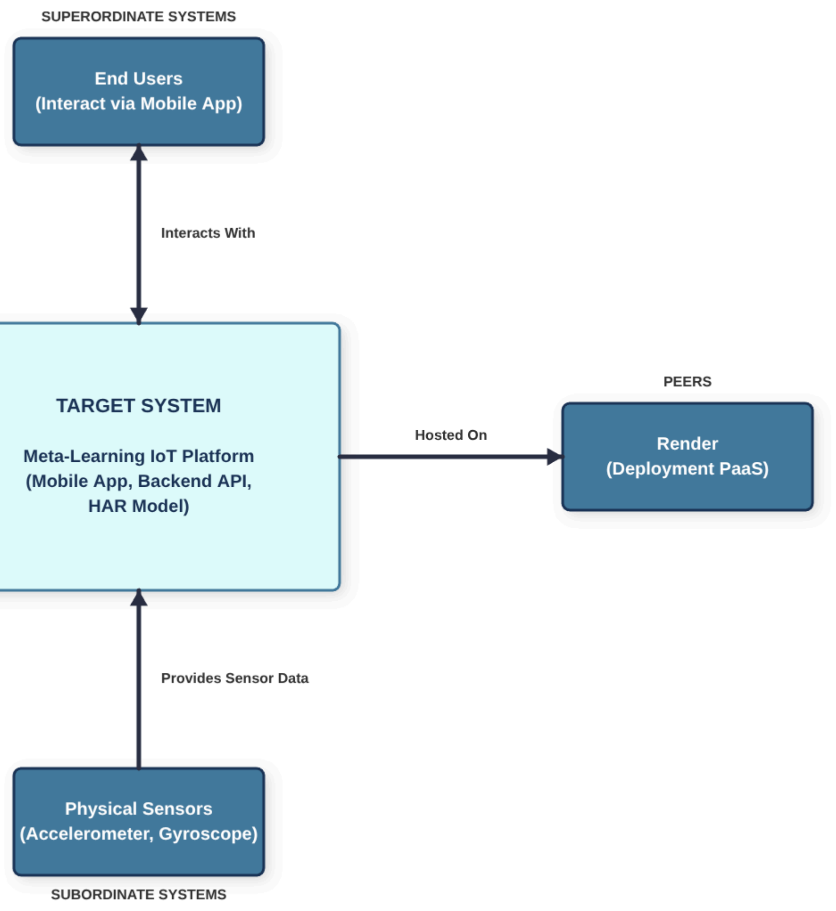
2.1.3 Process Implications

- By enforcing Functional Independence and Low Coupling between the edge client and the cloud server, development on these layers can proceed asynchronously.
- This architectural decoupling allows for continuous, independent work streams rather than rigid, lock-step phases. This justifies the adoption of a Kanban Process Model, as the distinct layers allow tasks to flow through the development pipeline independently.

2.2 Architectural Context Diagram (ACD)

Based on the Architectural Context Diagram methodology, the system is defined by its interaction with four categories of external entities:

- **Superordinate Systems**
    - End Users - Interact with the system via mobile application.
- **Target Systems**
    - Meta-Learning IoT Platform - Mobile app, Backend API, HAR Model.
- **Subordinate Systems**
    - Physical Sensors - Accelerometer and Gyroscope.
- **Peers**
    - Render - Deployment Platform as a Service for Backend API + Model Containers.

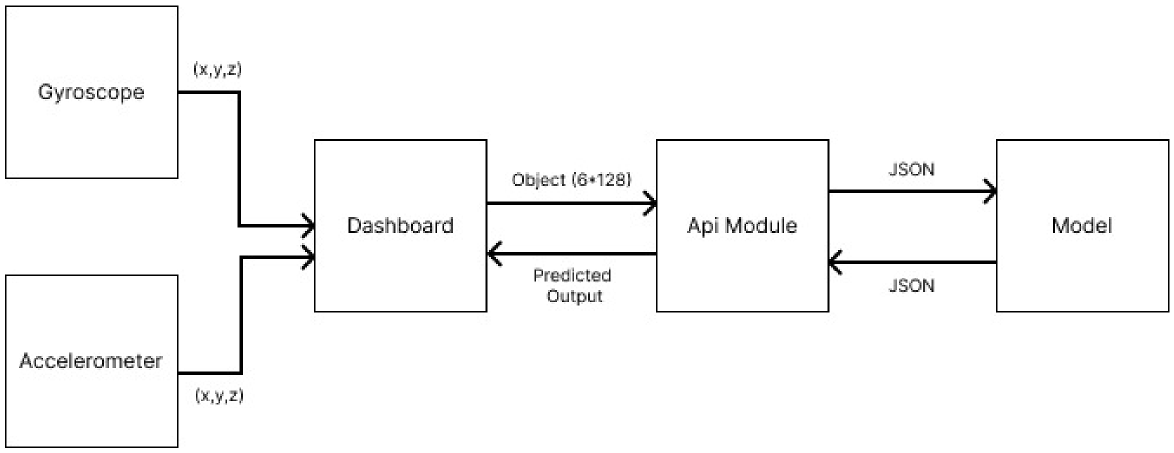

SUPERORDINATE SYSTEMS
End Users (Interact via Mobile App)
Interacts With
TARGET SYSTEM
Meta-Learning IoT Platform (Mobile App, Backend API, HAR Model)
Hosted On
PEERS
Render (Deployment PaaS)
Provides Sensor Data
Physical Sensors (Accelerometer, Gyroscope)
SUBORDINATE SYSTEMS

2.3 Data Flow Architecture

The system internally follows a Pipes-and-Filters style data flow architecture where each stage transforms the sensor data progressively which enables steam-like processing, high modularity and simplified transformations.

- **Filter 1 (Edge Device):**
  - Raw Sensor Signal.
  - Formatted Data.
  - JSON Payload.
- **Pipe (Network Transport):**
  - Data encapsulated in HTTP requests.
  - JSON Payload transferred to Backend gateway.

- **Filter 2 (Backend Server):**
  - Payload Validation.
  - Model Inference.
  - Prediction Output.



2.4 Low Level Design

The system is modelled as a transformation function, converting raw sensor data into meta-learning based predictions and can be decomposed into modules based on the nature of functionality by following the Structured Design Principles:

Module Name	Structured Class Design	Responsibilities
Mobile App	Input Module	Collects accelerometer & gyroscope data, preprocesses it (first-level factoring) and sends payload to server.
Backend API	Coordinate Module	Authenticates (JWT), validates payload schema, routes the request to PyTorch Model Service.

PyTorch Model	Transform Module	Converts input matrix to prediction vector using meta-learning.

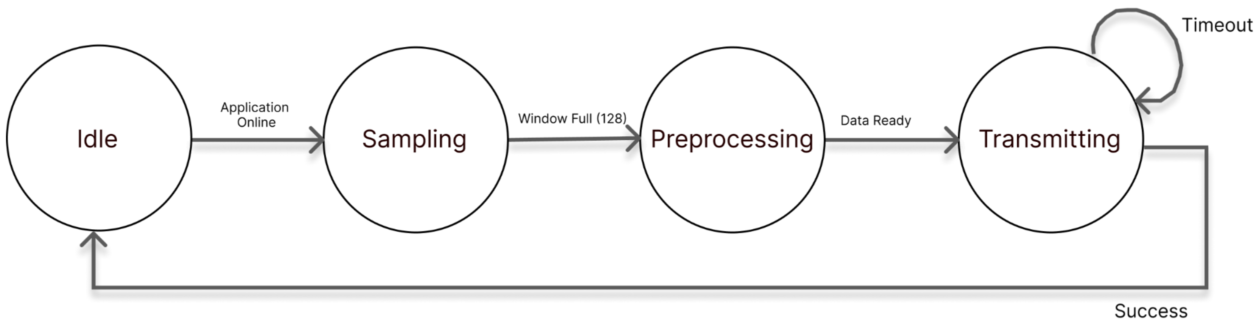
2.4.1 Coupling Strategy

- The system achieves Data Coupling as modules communicate only via data parameters (JSON payloads).
- This ensures Low Coupling.

2.4.2 Cohesion Strategy

- Each module is only responsible for a single, well-defined task (Sensor Module for data capture, Backend Module for routing and request validation and Transform Module for computation), achieving Functional Cohesion.
- This ensures High Cohesion.

2.4.3 Statechart-based Behavioural Modelling



The statechart for the mobile app consists of the following:

- **States:** Represent the conditions in which the system can be in.
    - Idle - When the app is actively being used for a task.
    - Sampling - Collecting data from the sensors.

    - Preprocessing - Sanitizing and formatting input data into required format.
    - Transmitting - Sending data to the backend gateway.
- **Events:** Actions that cause change in state.
    - Application Online - Starting to use the application.
    - Window Full - Stopping data collection after limit is reached.
    - Data Ready - Finishing data formatting and readying for data transmission.

2.5 Design Principles and Quality Attributes

2.5.1 Abstraction

- The client only interacts with high level components without having any knowledge of the underlying algorithms and functions.
- This is known as Functional Abstraction.

2.5.2 Information Hiding

- Internal ML model weights, hyperparameters and logic remain hidden from the client.
- Only validated, sanitized outputs (prediction and confidence) are exposed to the client.
- This reduces attack surfaces and protects knowledge of the inner workings of the application.

2.6 Quality Guidelines

2.6.1 Performance

- Edge preprocessing reduces payload size and latency.
- Server-side batching and async operations optimize inference.

2.6.2 Reliability

- Retry mechanisms implemented in the Coordinate Module.
- Monitoring via health checks, logs and latency metrics.

2.6.3 Security

- Strict payload validation.
- Model service isolated in Render Private Network.

CHAPTER 3 - SOFTWARE TESTING AND VERIFICATION

3.1 Testing Objective

3.1.1 Functional Objective

- Verify correct operation of authentication flows (signup, login, biometric), prediction flow, sensor buffering and start/stop behavior, device/network/location displays, feedback submission, and sign out flow.
- Validate data loader to ensure no data leakage, temporal data of expected shape and form and data values expectation.
- Validate that the model correctly extracts meaningful temporal features from accelerometer and gyroscope streams using its convolutional, recurrent, or attention-based layers (depending on your architecture).
- Ensure that the model outputs the correct predicted class label (e.g., sitting, walking, laying) with acceptable accuracy, precision, recall, and F1-score.
- Verify error handling (empty fields, duplicate signup, invalid credentials, API failure).
- Verify UI actions (buttons, toggles, animation start/stop) and state transitions.

3.1.2 Non Functional Objective

- Validate responsiveness and smoothness of UI animations and sensor updates under normal device load.
- Validate stability of buffer collection and API submission (no crashes on buffer fill or network error).
- Evaluate basic security posture: avoid printing secrets, avoid storing credentials insecurely (note: in this app credentials are stored in-memory for demo; highlight as a risk/future change).

3.2 Testing Scope

- Testing covers the full application lifecycle from pre-login (signup/login/biometrics), through post-login dashboard features (sensor acquisition, prediction API integration, device/network/location readouts), to logout.
- Both functional and basic non-functional aspects (responsiveness, stability, basic performance under normal device conditions) are included.

3.3 Testing Environment

3.3.1 Hardware Environment

- **Mobile Device**
    - Physical devices:
        - Android phone: sensors & biometrics validated.
        - Required for:
            - Real Accelerometer data
            - Real Gyroscope data
            - Biometric authentication
    - Emulators:
        - Android Emulator (API 31+): used for functional testing (note: some biometric behaviors require physical device).
            - Limitations:
                - No real accelerometer/gyroscope
                - Limited biometric simulation
- **Backend Hardware**
    - Cloud server hosting the PyTorch model
        - Stable internet connection required

3.3.2 Software Environment

- **Mobile Application Software**
  - Expo SDK
  - Device OS:
    - Android 10+ (tested)
- **Backend & Model Software**
  - Python 3.11+
  - PyTorch (Latest stable release)
    - Used for:
      - Model definition
      - DataLoader and Dataset definition
      - Meta-learning training loop
      - Forward-pass inference
  - Torchvision / Numpy
    - For preprocessing signals
  - FastAPI backend
    - Receives payload from mobile
    - Converts JSON → Tensor
    - Runs inference
    - Returns prediction + confidence

3.4 Software Description

3.4.1 System Features

- **Authentication**
  - Signup (username + password).
  - Login (username + password).
  - Biometric login (fingerprint/FaceID if enrolled).
- **Sensor Acquisition**
  - Accelerometer and Gyroscope listeners with configurable update intervals.

    - In-memory circular buffer until 128 samples are collected.
- **Prediction**
    - Local client prepares 6-channel input (acc x/y/z, gyro x/y/z) and posts to the prediction API.
    - Prediction API classifies inputs into activity labels(namely walking, walking upstairs, walking upstairs, sitting, standing, laying).

3.4.2 Modules Description

- **Auth Module**
    - <u>handleSignUp()</u> **:** Validates fields, checks duplicate usernames, stores users in component state.
    - <u>handleBiometricLogin()</u> **:** Validates credentials, checks enrolled biometrics, invokes LocalAuthentication.authenticateAsync
- **Sensor Module**
    - Accelerometer & Gyroscope listeners set with Accelerometer.setUpdateInterval() and Gyroscope.setUpdateInterval()
    - Buffers stored in accBuffer.current / gyroBuffer.current; when both reach 128, sendPredictionRequest() is invoked**.**
- **Prediction Module**
    - <u>Prepares payload:</u> six arrays of 128 values each and POSTs to the prediction endpoint; parses predicted class & confidence
    - <u>Loads Model:</u> envokes loadModel() to load prediction model.
    - The model is used to calculate Prototypes and get logits(log odds) of activity.
- **Feedback Module**
    - Rating state and handleSubmit validates prediction exists and rating selected.

3.5 Testing Strategy

3.5.1 Unit Testing

- **Mobile App Testing**
    - handleSignUp()
        - Validates field checks, duplicate user detection, and user creation.
    - handleBiometricLogin()
        - Verifies credential validation and biometric flow invocation.
    - sendPredictionRequest()
        - Tests buffer slicing, payload creation, API call structure, and error handling.
- **Prediction Model Unit**
    - loadModel()
        - Validates model file loading, version compatibility, and initialization failures.
    - preprocessInput()
        - Ensures correct normalization, reshaping, and rejection of malformed or empty inputs.
    - runInference()
        - Verifies output dimensions, deterministic predictions, probability validity, and thresholding logic.
    - postprocessOutput()
        - Confirms correct label extraction, score formatting, and handling of edge-case outputs.
    - batchAndBufferHandling()
        - Tests batching logic, buffer slicing consistency, and correct handling of incomplete or large input buffers.
    - errorHandling()
        - Validates graceful failures for corrupted models, runtime errors, invalid inputs, and fallback logic.

3.5.2 Integration Testing

- Integration testing ensures all components interact correctly across both mobile app and model backend.
- **Mobile App Integration Tests**
    - <u>Sensor Buffers → Prediction API</u>
        - Validate that accelerometer and gyroscope listeners correctly populate buffers and trigger inference when both reach 128 samples.
    - <u>Prediction Response → UI</u>
        - Confirm prediction labels and confidence to update the UI correctly.
    - <u>Authentication → Dashboard Transition</u>
        - Ensure successful login leads to full dashboard access.
- **Prediction Model Integration Tests**
    - <u>Preprocessing → Model Inference Flow</u>
        - Validate that raw accelerometer/gyroscope windows are correctly normalized, reshaped, and forwarded to the model.
    - <u>Model Output → API Payload Mapping</u>
        - Ensure predicted label, confidence scores, and metadata are correctly placed into the API response payload.
    - <u>Batch/Buffer Handling → Inference Trigger</u>
        - Confirm that the model receives properly sliced 128-sample windows with 6 features and that no frames are skipped or duplicated.
    - <u>Model Versioning → Response Validation</u>
        - Verify that the correct model version is returned in the API response and displayed in app/debug logs.
    - <u>Error Propagation → UI/Client Handling</u>
        - Ensure model errors (invalid input window, corrupted data, inference failure) return structured error responses without crashing the app.

- End-to-End Mini Pipeline Test
    - Run input window → preprocessing → inference → post-processing → prediction response to confirm consistency across all components.

3.5.3 System Testing

- System Testing validates the entire system as a whole — app + sensors + backend + model.
- **Mobile App System Tests**
    - Signup → login → start sensors → API → prediction → feedback → logout
      Verified end-to-end flows with multiple users.
    - Sensor start/stop stability
      Ensured no crashes or buffer inconsistencies.

3.5.4 Acceptance Testing

- **Alpha Testing (Internal)**
    - Performed by developers on:
        - Android Emulator
        - Physical Android Device
    - Validate:
        - Walking
        - Laying
        - Walking upstairs
        - Walking downstairs
        - Standing
        - Sitting
- **Beta Testing (External Testers)**
    - Testing by real users running real activities:
        - Walking
        - Laying
        - Walking upstairs
        - Walking downstairs
        - Standing

        - Sitting
    - Feedback was collected
- **Acceptance Criteria**
    - The model must classify activities with reasonable accuracy.
    - The prediction pipeline must complete end-to-end in under 1 second.
    - No crashes allowed during sensor collection or API calls.
    - Authentication must be secure and consistent.

3.6 Test Data

3.6.1 Authentication Test Data

- Default user:
    - { "username": "ad", "password": "123" }
- Additional test users:
    - **Valid**: "user1" / "pass123"
    - **Invalid**: "", " ", malformed inputs

3.6.2 Sensor Data

- **Real Accelerometer/Gyroscope signals**
    - Walking
    - Walking Upstairs
    - Walking Downstairs
    - Sitting
    - Standing
    - Laying

- **Simulated Sensor Data**
    - **Mock data for emulator**
        - **Mock Data for Mobile Application**

```
function syntheticData(){
    const rows = 6;
    const cols = 128;
    const data = [];
    for (let i = 0; i < rows; ++i){
        const row = [];
        for (let j = 0; j < cols; ++j){
            row.push(Math.random()) //0-1
        }
        data.push(row)
    }
    return data;
}
```

        - **Mock Data for Model**

```
import random
def syntheticData():
    data = [
        [random.random() for _ in
range(128)] #0-1
        for _ in range(6)
    ]
    return data
```

3.7 Testing Techniques

Testing techniques were selected to ensure the application and the prediction model behave correctly under real operational conditions. Both Black-Box (user-focused) and White-Box (internal logic-focused) techniques were applied.

3.7.1 Black Box Testing Techniques

- **Equivalence Partitioning**
  - Inputs were divided into valid and invalid partitions based on real user and sensor behavior.

Input	Valid Partition	Invalid Partition
Username	Non-Empty, Alphanumeric	Empty, Whitespace
Password	Non-Empty	Empty
Rating	1, 2, 3, 4, 5	No Selection
Sensor Window	Real Accelerometer and Gyroscope	Wrong Length, Missing Channel, Corrupted Arrays

  - Invalid partitions come from realistic failure cases, bad sensor windows or data corruption.
  - Valid partitions were chosen because all valid inputs have the same output.
- **Boundary Value Analysis**
  - <u>Sensor Buffer Size</u>
    - 128 samples → valid complete window
    - 127 samples → slight underflow due to sensor delay : invalid
    - 129 samples → slight overflow (practical buffer overshoot) : invalid
  - <u>Real-world edges tested:</u>
    - Device has biometrics → valid
    - Device has biometrics but no fingerprints enrolled → invalid
    - User cancels biometric prompt → invalid

3.7.2 White Box Testing Techniques

- Statement Coverage
- Branch Coverage

3.8 Test Execution

This phase validates whether all units, modules and integrated components behave as expected under real-word use conditions.

3.8.1 Unit Testing

- **Authentication Module Execution**

Scenario	**Description**	**Result**
Login with valid credentials	Correct username and password	**PASSED**
Login with invalid credentials	Wrong password	**PASSED**
Empty fields	Username/Password not given	**PASSED**
Signup with new user	Adds new entry	**PASSED**
Signup with existing user	Duplicate user detected	**PASSED**
Biometric authentication success	Fingerprint match	**PASSED**
Biometric authentication failure	Wrong finger or cancelled	**PASSED**

- **Sensor Acquisition and Buffer Execution**

Scenario	**Description**	**Result**
Start sensor capture	Accelerometer + Gyroscope start sampling	**PASSED**
Continuous sampling	Stable at 200ms interval	**PASSED**
Buffer fill	Both buffer reach 128 samples	**PASSED**
Rapid start/stop	User presses start/stop repeatedly	**PASSED**
Buffer reset	After each prediction	**PASSED**

- **Prediction Model (Execution + Backend App)**

Scenario	**Description**	**Result**
Sensor feature missing	One or more sensor fields are missing	**PASSED**
Sensor window size	Input temporal window size not matching required	**PASSED**
Low inference log odds	Model having trouble understanding the activity	**PASSED**
Noisy sensor value	Unexpected input value	**PASSED**

- **Feedback Execution System**

Scenario	**Description**	**Result**
Submit Rating	1-5	**PASSED**
Submit without Rating	Error	**FAILED**
Submit without Prediction	Error	**FAILED**

3.8.2 Integration Testing

- **Authentication -> Dashboard Navigation**

Scenario	**Description**	**Result**
Login success leads to dashboard	Correct credentials allow user to enter main app	**PASSED**
Login fail prevents dashboard access	Wrong credentials keep user on login page	**PASSED**
Signup success redirects to login	New user created and returned to login screen	**PASSED**
Biometric success redirects to dashboard	Biometric match logs in user	**PASSED**
Biometric fail stays on login	Failed/Cancelled biometric does not bypass login	**PASSED**

- **Sensor System -> Buffer -> Prediction Pipeline**

Scenario	**Description**	**Result**
Start sensors → buffer fills → prediction triggered	Full pipeline executed without breaking	**PASSED**
Buffer full → payload sent to API	128 samples across 6 channels triggers request	**PASSED**
API response → prediction displayed	Prediction + confidence updates UI	**PASSED**
Missing sensor channel → server error handled	App shows error without crash	**PASSED**
Slow network → delay handled	App waits and displays result properly	**PASSED**

3.9 Debugging

- **Debugging Steps Used**
  - Debugging followed the systematic process outlined in software engineering testing standards:
  - Problem Identification
  - Fault Isolation
  - Fault Fixing
  - Verification of Fix
- **Issues**
  - Below is a complete list of issues found in app and how they were debugged:

        - **Issue 1:** Duplicate Signup Detection Not Working
            - Added console logs
            - Used breakpoint-style logs:
                - console.log("Existing users:", user);
            - **Status:** Resolved
        - **Issue 2:** Feedback submitted without rating
            - **Status:** Not Resolved
        - **Issue 3:** Buffer not clearing after prediction
            - Printed buffer lengths every update
            - Verified using logs
                - console.log(accBuffer.current.length, gyroBuffer.current.length)
            - Moved buffer clearing to occur *immediately* after slicing:
            - **Status:** Resolved
        - **Issue 4:**Model returning error due to incorrect payload
            - Used Postman to manually send requests with:
                - Correct shape (6 × 128)
                - Incorrect shape (6 × 100)
            - Corrected app payload formatting
            - **Status:** Resolved
        - **Issue 5:** Model having low log odds
            - Checked sensor Values if they are noisy and behave unexpectedly
            - Normalized values to fit in model expectation
            - **Status:** Resolved
- **Remaining Risks**
    - **Feedback Submission Bug (High Priority)**
        - Feedback can still be submitted without selecting a rating under certain conditions.
          This impacts data quality and user experience
    - **Local Credential Storage (High Risk – Security)**
        - Credentials are stored directly in component state and not secured.

A proper authentication backend with hashed passwords or JWTs is needed for production.

- **Dependency on External Prediction API (Medium Risk)**
  - The system relies on an internet connection for prediction. Server downtime, network latency, or API failure will immediately impact functionality.

CONCLUSION

This project successfully demonstrated the application of Meta-Learning for Human Activity Recognition (HAR) on small, resource-limited devices. Utilizing Prototypical Networks, a system was developed to quickly learn new activities from minimal examples. This effectively addressed the challenge of deploying intelligent systems without excessive device resource consumption.

The project employed the Kanban process and a clear layered architecture (React Native, FastAPI, PyTorch) to ensure flexibility and maintainability. Adherence to core software principles, specifically low coupling and high cohesion, resulted in a highly structured and scalable system.

Through comprehensive testing (Unit, Integration, System, and Acceptance), correct system operation was verified, confirming the achievement of key performance goals, including real-time predictions within the one-second limit. While the project is a significant success, future work requires attention to outstanding issues in the user feedback system and the implementation of a secure, production-ready login mechanism.

In summary, this work strongly establishes that meta-learning and robust software design are capable of handling complex tasks within the Internet of Things (IoT). It provides a foundational basis for continuously learning, personalized applications in domains such as health and smart homes.

REFERENCES

1. Software Engineering: A practitioner’s Approach, Roger S Pressman, Seventh Edition, McGrawHill International Edition, 2010.

2. Software Engineering, Ian Sommerville, Eighth Edition, Pearson Education, 2017.