Test Strategy Document

Table of Contents

Table of Contents 1

1. Overview 2

2. System Under Test (SUT) 2

3. Requirements Overview 4

4. Overall Testing Strategy 10

4.1 User Need Testing 8

4.2 System-Level Testing 9

4.3 Subsystem-Level Testing 10

5. API Testing 14

6. Facial Recognition Validation 17

7. Test Environment and Tools 19

8. Personnel 20

1. Overview

This document defines the Test Strategy for the Smart Doorbell Camera System. It outlines the approach to validating the system against user needs and technical requirements. The strategy covers the testing levels, including:

- Component-level testing to verify the functionality of individual modules such as the camera, motion sensor, and cloud services.
- System integration testing to assess the interaction between hardware components, mobile applications, and cloud services.
- Performance and security testing to evaluate system efficiency and data protection under various conditions.

Includes also reference to tests automation and focuses on the testing strategy for critical features such as facial recognition, notifications, and motion detection across multiple platforms.

This document provides a framework for executing a comprehensive, scalable, and structured testing process to ensure product readiness for release.

2. System Under Test (SUT)

The **Smart Doorbell Camera System** is a battery-operated, wireless device designed for remote video monitoring and facial recognition. The device incorporates multiple components, including a high-resolution camera, a motion sensor, and a wireless module for network connectivity. It communicates wirelessly with a **local hub**, which acts as a bridge between the doorbell camera and cloud-based services.

The system leverages **cloud infrastructure** for key functionalities such as **video storage** and **facial recognition processing**. Video captured by the camera is transmitted via Wi-Fi to the local hub, which securely uploads the data to the cloud using standard HTTPS protocols. Facial recognition algorithms in the cloud analyse the video streams, tag known individuals, and store the results for later retrieval.

Users interact with the system through a **mobile web application** that provides real-time access to the device. The webapp supports:

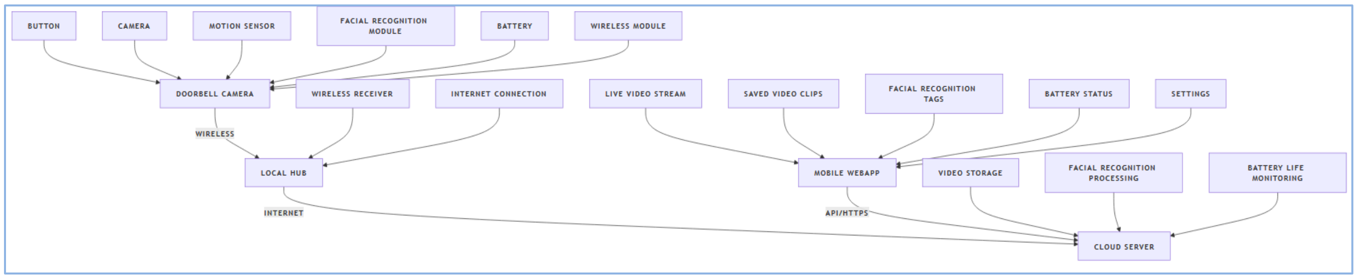
- **Live video streaming** from the camera with minimal latency.
- Access to **stored video clips** and event logs.
- Customization of device settings such as **motion detection sensitivity** and **video quality** (e.g., HD/SD).
- Push notifications for motion detection or when the doorbell button is pressed.

The system architecture ensures **secure data transmission** and **role-based access control** for user management. All communications between the doorbell, hub, and cloud are encrypted, and users can control who has access to video feeds and stored data through the webapp.

System Components

Component	Model Version	Producer
Camera	Model: 1080p-HD	Camera Manufacturer
Motion Sensor	Model: MS-500	Sensor Manufacturer
Local Hub	Model: Hub-200	Hub Manufacturer
Cloud (Facial Recognition)	Version: 2024	Cloud Provider
Cloud (Video Storage)	Version: 2024	Cloud Provider
Mobile Webapp	Version: v1.0	In-House Development
Mobile Webapp Platform	Android, iOS, Web	In-House Development
CLI Interface	Version: DevCLI 1.0	Internal Debugging

The following diagram represents the system architecture, showing how components interact with each other and cloud services:



3. Requirements Overview

The table below provides a detailed **Traceability Matrix** that maps the **User Needs (UN)** to corresponding **System Requirements (SyRS)** and **Subsystem Requirements (SSR)**. This matrix ensures that all user expectations are captured as high-level system requirements and broken down into specific component-level functionalities.

- The **User Needs** represent the key features and capabilities that users expect from the Smart Doorbell Camera System, such as recognizing individuals, managing access, viewing live video, and receiving notifications.
- The **System Requirements (SyRS)** describe how the overall system will function to meet these needs, ensuring that the necessary features are implemented at a system-wide level (e.g., facial recognition, video storage, notifications).
- The **Subsystem Requirements (SSR)** define how individual components—such as the camera, motion sensor, cloud system, and webapp—will work together to achieve the system requirements, ensuring that each part of the system performs as expected.

This matrix provides full traceability between the user's expectations, system functionality, and component behaviours.

User Need (UN)	System Requirement (SyRS)	Subsystem Requirement (SSR)
UN 1.1: As a user, I want the system to recognize and identify known individuals on arrival and automatically tag video recordings with their names.	SyRS 1.1.1: The system shall capture images and send them to the facial recognition engine to identify individuals upon arrival.	SSR 1.1.1: The camera shall capture images at 1080p and forward them to the facial recognition engine for identification.
	SyRS 1.1.2: The system shall automatically retrieve recognized individuals' names from the facial recognition engine and tag the associated video recordings.	SSR 1.1.2: The cloud system shall interface with the facial recognition API to retrieve recognized individual names and tag the video clips accordingly.
	SyRS 1.1.3: The system shall provide access to these tagged videos via the webapp, allowing users to view the recordings and associated tags.	SSR 1.1.3: The webapp shall display tagged videos with the recognized individual's name for user access.

User Need (UN)	System Requirement (SyRS)	Subsystem Requirement (SSR)
UN 2.1: As a user, I want to view live video of individuals arriving at the doorbell camera via the mobile webapp.	SyRS 2.1.1: The system shall stream live video from the camera to the mobile webapp with minimal latency.	SSR 2.1.1: The camera shall support continuous live streaming at 30fps and 1080p for low-latency viewing.
UN 2.2: As a user, I want the system to store video clips of individuals at the door, whether recognized or not, for later review.	SyRS 2.1.2: The system shall store video clips of all access attempts and arrivals, regardless of recognition status, for at least 30 days.	SSR 2.1.2: The motion sensor shall trigger video recording upon detecting an individual.
		SSR 2.1.3: The cloud storage system shall store video clips for 30 days, including facial recognition tags.
		SSR 2.1.4: The webapp shall allow users to view live and stored video clips with associated tags.

User Need (UN)	System Requirement (SyRS)	Subsystem Requirement (SSR)
UN 3.1: As a user, I want to receive notifications when an individual arrives, and whether they are recognized or not.	SyRS 3.1.1: The system shall send notifications for all arrivals, recognized or unrecognized, within 2 seconds of detection.	SSR 3.1.1: The motion sensor shall trigger notifications within 2 seconds upon detecting an individual.
	SyRS 3.1.2: The system shall include the recognized individual's name or 'unknown' in the notification, along with a timestamp.	SSR 3.1.2: The notification system shall include individual recognition status and timestamp for notifications.
UN 4.1: As a user, I want to configure access permissions for different individuals via the mobile webapp.	SyRS 4.1.1: The system shall allow users to configure access permissions for known individuals via the webapp, specifying who can access the premises.	SSR 4.1.1: The webapp shall support user-defined access permissions, allowing the configuration of access control for recognized individuals.
UN 4.2: As a user, I want to configure video quality settings (e.g., HD/SD) via the mobile webapp.	SyRS 4.1.2: The system shall allow users to toggle between HD and SD video quality for live streaming and recorded clips.	SSR 4.1.2: The camera shall support both HD and SD video quality modes.

User Need (UN)	System Requirement (SyRS)	Subsystem Requirement (SSR)
UN 5.1: As a user, I want to see the current battery level of the doorbell camera in the mobile webapp.	SyRS 5.1.1: The system shall display the current battery level of the doorbell camera in the webapp.	SSR 5.1.1: The battery monitoring system shall provide real-time battery status updates to the cloud, viewable in the webapp.
UN 5.2: As a user, I want to be notified when the doorbell camera battery is low so I can recharge or replace it.	SyRS 5.1.2: The system shall send low battery notifications when the level drops below 20%.	SSR 5.1.2: The cloud system shall send battery status data and trigger low battery alerts.
UN 6.1: As a user, I want to ensure that all video feeds, access logs, and personal data are securely transmitted and stored.	SyRS 6.1.1: The system shall encrypt all video feeds, access logs, and personal data during transmission and storage.	SSR 6.1.1: The data transmission system shall use HTTPS for video and log streaming and AES-256 for data storage.
UN 6.2: As a user, I want to manage access to my video feed, access logs, and control who can view the recordings.	SyRS 6.1.2: The system shall allow users to manage access to video feeds and logs via role-based access control (RBAC).	SSR 6.1.2: The webapp shall allow users to configure access permissions for video feeds and access logs via RBAC.

User Need (UN)	System Requirement (SyRS)	Subsystem Requirement (SSR)
UN 7.1: As a user, I want the system to wirelessly connect to the local hub for internet access.	SyRS 7.1.1: The system shall wirelessly connect to a local hub for access to the internet and cloud services.	SSR 7.1.1: The wireless module shall support Wi-Fi connectivity to the local hub for internet access.
UN 7.2: As a user, I want to control the doorbell camera settings via the mobile webapp, including motion detection and video quality.	SyRS 7.2.1: The system shall allow users to control camera settings (motion detection sensitivity and video quality) via the webapp.	SSR 7.2.1: The webapp shall provide options for adjusting motion detection sensitivity and toggling video quality (HD/SD).
UN 7.3: As a user, I want the doorbell to have a physical button for individuals to press and notify me of their presence.	SyRS 7.3.1: The system shall notify the user via the webapp when the physical doorbell button is pressed.	SSR 7.3.1: The doorbell button shall trigger a notification to the user's mobile device through the webapp.

4.Overall Testing Strategy

The Smart Doorbell Camera System testing strategy ensures comprehensive validation of the product by covering multiple test levels. Each test level focuses on specific aspects of the system, from end-user scenarios to component-level functionality, guaranteeing that the system meets both functional and non-functional requirements. Below are the test levels used to test the product:

4.1 User Need Testing

Test Objectives:

End-to-End Scenario Simulation: Simulate key real-world user scenarios such as:

- Pressing the doorbell.
- Receiving push notifications when motion is detected, or the doorbell is pressed.
- Viewing the live video stream and accessing saved clips through webapp.
- Confirming that the system recognizes individuals using facial recognition and appropriately tags videos.
- Access Control Validation: Test scenarios in which known individuals are granted or denied access based on pre-configured permissions.
- User Notifications: Verify that users are notified of events (doorbell press, motion detection, access granted/denied) within the specified time frame (2 seconds).
- Data Accessibility: Ensure that the user can retrieve and view the stored access logs and video clips via the webapp.

Approach:

- Simulate end-user actions such as pressing the doorbell, walking into the camera's field of view, and accessing the webapp from a mobile device.

- Test for real-time performance of notifications and live streaming, ensuring that there is minimal latency in video and notifications.
- Run exploratory testing to ensure that all user needs (facial recognition, battery reporting, video access) are met without functional gaps.
- Battery Performance: Include real-world battery scenarios, such as battery drain under typical and heavy usage (e.g., continuous video streaming) to ensure the user is notified when the battery is low, and battery status is correctly reported in the webapp.

Tools:

- Manual testing with **real devices** (mobile phones, doorbell).
- **Automated end-to-end scripts** using tools like **Selenium** to simulate user interactions through the webapp.

4.2 System-Level Testing

Test Objectives:

- Component Integration: Test the interaction between the doorbell camera, local hub, cloud services (facial recognition and video storage), and mobile webapp.
- Performance Testing: Validate that the system performs efficiently, particularly during peak load, such as when multiple notifications are triggered, or video streams are being accessed simultaneously.
- Battery Testing: Ensure that real-time battery status is accurately transmitted from the doorbell to the webapp and that low battery warnings are triggered at the appropriate threshold.
- Latency Testing: Ensure that video streaming and notifications are delivered with minimal delay, meeting user expectations (e.g., live video streams should not exceed a specified latency).

- Security Testing: Ensure that all communications between the components (camera, local hub, cloud) are encrypted and follow proper security protocols (e.g., HTTPS, AES-256 encryption for stored data).

Approach:

- Perform integration testing to verify that all components communicate properly with each other (camera to cloud, cloud to webapp).
- Use load testing tools like JMeter to simulate high volumes of users accessing live video streams and stored clips to verify system stability and performance.
- Run network latency tests to measure the response times for live video streaming and push notifications under various network conditions (e.g., Wi-Fi, 4G).
- Perform security penetration testing to ensure that all data transmission and storage processes meet security requirements.
- Test the battery reporting functionality under various conditions (normal usage, high usage) to ensure the system provides real-time updates to the user.

Tools:

- **JMeter** for performance and load testing.
- **Postman** or **REST Assured** for API testing between components.
- **Wireshark** for analysing network traffic and ensuring secure data transmission.

4.3 Subsystem-Level Testing

Test Objective:

- Camera: Verify that the camera captures images and video in 1080p quality and correctly triggers video recording when motion is detected, or the doorbell is pressed.
- Motion Sensor: Test the motion sensor’s sensitivity settings (low, medium, high) to ensure accurate triggering of events.

- Cloud Services: Validate the interaction between the doorbell camera and cloud services, ensuring that video clips are stored properly, facial recognition results are returned correctly, and access logs are maintained.
- Webapp: Test the webapp's interface and API integration with the cloud to ensure that users can access video feeds, manage access permissions, and view access logs without issues.
- Battery Testing: The subsystem-level battery testing will involve validating the battery status reporting mechanism. This ensures that the doorbell accurately reports battery levels to the cloud and triggers notifications when thresholds are reached.

Approach:

- Perform unit tests on the individual components (e.g., camera, motion sensor, cloud API).
- Use mocking techniques to simulate cloud interactions (e.g., returning facial recognition results from a mock API) to isolate and validate the functionality of components before full system integration.
- Run automated tests to validate different API interactions, ensuring they handle requests and responses correctly (e.g., video retrieval, facial recognition).
- Use CLI-based testing to verify direct system interactions and configurations, including battery levels, event logs, and hardware diagnostics.
- Test the camera’s field of view and motion sensor detection range to meet the specified coverage.

Tools:

- **JUnit** or **PyTest** for unit testing individual components.
- **Mock servers** to simulate cloud responses for API interactions (e.g., using **WireMock**).
- **CLI Testing Tool**: A custom CLI-based tool, which is to be developed by the **engineering team**, will allow direct control over the doorbell camera’s hardware. The tool should provide a **UI interface** for easy interaction with the CLI commands and automate hardware-level testing tasks such as battery diagnostics, camera settings, and motion sensor calibration.

- **Automated UI Scripts**: The CLI and UI tools can be used in combination to automate extensive testing, including hardware configuration and event logging.

5. API Testing

API testing ensures that the **battery-powered doorbell camera** functions seamlessly in communication with **cloud services** and the **mobile webapp**. The APIs are responsible for handling core functionalities such as video uploads, facial recognition, notifications, and battery reporting. Ensuring that these APIs perform correctly and securely is a critical part of validating the system.

Test Objectives:

- **Camera to Cloud Communication**: The objective is to verify that the camera can securely and reliably send video streams and images to cloud storage. This involves testing the correct formatting and transmission of data, as well as ensuring the system handles network disruptions, retries, and uploads without data loss or corruption.
- **Battery Status Reporting**: The goal is to confirm that the system provides real-time battery status updates from the doorbell to the mobile webapp. This includes testing that the API accurately triggers low-battery warnings when required and reflects battery changes in the webapp promptly.
- **Facial Recognition**: The primary objective is to ensure that images sent from the camera to the cloud are processed correctly by the facial recognition system. This involves testing accuracy in identifying known individuals, handling unrecognized faces, and tagging video feeds appropriately.
- **Motion Detection and Button Event Handling**: The objective is to validate that the motion sensor and doorbell button triggers are properly handled by the APIs, ensuring they initiate video recording and send real-time notifications to users without delay.

- **Notifications**: The key objective is to ensure that notifications triggered by events (e.g., motion detection, button press, low battery) are sent to the user in real-time. Testing ensures that the notifications include the necessary details and are delivered promptly across different scenarios, including simultaneous event triggers.

Test Environment for API Testing:

The API test environment simulates real-world interactions between the doorbell camera, cloud services, and mobile webapp, allowing thorough validation of the system's behaviour under various conditions.

- **Test API Server**: A test server replicates the production environment's API endpoints, allowing video uploads, facial recognition, notifications, and battery reporting to be tested comprehensively before live deployment.
- **Real or Simulated Doorbell Hardware**: The test environment includes real or simulated hardware that allows testing of API behaviour in response to real-world events like motion detection and battery depletion. Automated scripts are used to simulate these events, ensuring consistency and reducing manual intervention.
- **Security and Encryption Testing**: Using tools like **Wireshark**, automated tests are run to verify that all communications between the camera, cloud services, and webapp are encrypted with HTTPS/TLS, preventing unauthorized access or data breaches.
- **Mock Services**: **WireMock** and other mock services are used to simulate cloud responses, allowing tests under various scenarios, such as delayed responses or cloud service failures. This helps ensure that the system is resilient under different conditions and handles errors appropriately.
- **Performance and Load Testing**: Automated load testing using **JMeter** simulates high-traffic conditions to validate that the APIs can handle concurrent requests without impacting system

performance. This ensures that the system can scale efficiently as usage increases, such as when multiple video uploads or notifications occur simultaneously.

Automation Strategy:

Automation is central to the API testing process, ensuring efficiency and consistency across tests. The strategy is designed to minimize manual intervention, increase test coverage, and ensure continuous validation throughout the development lifecycle.

- **Postman and REST Assured**: These tools are used to automate API tests, verifying correct request-response behaviour, payload validation, and status codes. They also handle specific edge cases such as retries for battery reporting or video uploads, ensuring the system responds correctly under different conditions.
- **Continuous Integration (CI)**: The API tests are integrated into a CI pipeline (e.g., Jenkins, GitLab CI), allowing automated tests to be triggered with every code change or new feature. This ensures early detection of issues and reduces the risk of regression.
- **Automated Event Simulations**: Test scripts are created to simulate real-world events, such as button presses, motion detection, and battery depletion. This ensures that APIs handling these events can be tested automatically, replicating real user interactions without manual testing.
- **Load Testing**: Tools like **JMeter** are employed to automate load tests, simulating high volumes of requests, such as simultaneous motion detection events or video uploads. This ensures that the system remains performant under heavy loads and that APIs maintain acceptable response times.

6. Facial Recognition Validation

Facial recognition is a critical feature of the **Smart Doorbell Camera System**, powered by a third-party service for real-time image processing. While the provider has validated their service, in-house testing is required to ensure proper system integration and reliable performance under real-world conditions. The following scenarios must be considered to validate the accuracy and robustness of the system's facial recognition:

Known Faces:

Testing with registered individuals ensures consistent identification under various conditions:

- **Lighting conditions** (e.g., bright sunlight, low light) and **varying distances** from the camera are tested to validate that the system accurately recognizes individuals regardless of environmental factors.
- The system's ability to handle **appearance changes**, such as glasses, hats, or facial hair, is tested to ensure that minor variations do not impact recognition accuracy.
- Performance with **different face orientations** (e.g., side profiles, tilted heads) is validated to confirm accurate identification even when individuals are not directly facing the camera.

Unknown Faces:

- The system's ability to handle unregistered individuals without producing false positives is crucial:
- It should consistently flag unregistered individuals as "unknown," and testing will focus on ensuring a low false positive rate under various conditions.
- Mixed scenarios, where both known and unknown individuals appear in the camera's field of view, are tested to ensure the system correctly distinguishes between them.

Image Variation:

- The system must manage different facial variations to ensure reliable performance:
- Facial expressions, including smiling, frowning, and neutral, are tested to ensure consistent recognition regardless of expression changes.

- Face angles (e.g., side profiles) and the presence of multiple faces in a single frame are validated to ensure the system accurately processes real-world interactions where more than one person may be present.

Additional Scenarios:

Real-world conditions are tested to ensure the system's robustness across different environments:

- Outdoor conditions, including various weather scenarios, and night-time performance using infrared lighting, are tested to ensure consistent recognition in diverse environments.
- Scenarios involving masks and partial facial obstructions are tested to verify that the system flags such cases as "unknown" while avoiding incorrect identification.
- Movement scenarios are validated to ensure that the system can accurately capture and recognize individuals walking toward or past the camera.

Family Members and Lookalikes:

The system's ability to distinguish between family members or individuals with similar facial features is critical to prevent confusion and ensure accurate identification in such cases.

Age Variation Testing:

As users age, their appearance changes. The system's performance is validated to ensure it continues to recognize individuals accurately over time, even as facial features evolve.

<u>Automation strategy</u>

The automation strategy for facial recognition focuses on validating the system's ability to handle various image scenarios, ensuring accuracy and consistency under real-world conditions. Automation provides a repeatable and scalable method to test the system's response to diverse image inputs, reducing manual testing efforts.

Scripts simulate diverse conditions such as **lighting variations**, **facial orientation**, **appearance changes**, and **image obstructions**. By automatically feeding these images into the system, the tests cover edge cases like low-light environments, partial obstructions (e.g., masks), and accessories. This approach confirms that the system can identify registered individuals correctly and flag unknown faces as required.

Key image scenarios, including **multiple faces**, **side profiles**, and **facial expressions** (smiling, neutral, frowning), are tested to validate consistent system performance across these variations. Automated tests also assess **known vs. unknown individuals**, ensuring a low false positive rate when unregistered faces are processed.

Automated tests further handle **age progression** and **family lookalikes** by using image sets that simulate changes over time or subtle differences in facial features. This confirms the system's ability to distinguish between similar-looking individuals and recognize users over time.

7. Test Environment and Tools

The test environment includes the following hardware, software, and tools:

- **Hardware**: Doorbell camera, motion sensor, local hub, and mobile devices (iOS and Android).
- **Cloud Infrastructure**: Cloud services for video storage and facial recognition.
- **Testing Tools**:
  - **Postman** and **REST Assured** for API testing.
  - **Selenium** for automated UI testing of the webapp.
  - **JUnit** and **PyTest** for unit and subsystem-level tests.
  - Mocking tools like **WireMock** to simulate interactions with cloud services.

The **CLI Interface** will include a **Graphical User Interface (GUI)** for easier tester interaction. This GUI will allow testers to control and configure the doorbell camera, adjusting settings such as motion sensitivity or video resolution and capturing logs for debugging. It will be integrated with the automation framework, enabling both manual and automated test cases to interact with the camera. This setup allows for flexible testing, combining hands-on control with automated execution for more efficient and thorough validation.

8. Personnel

The successful execution of the test strategy will require the following key roles, each responsible for specific aspects of the testing process:

QA Lead

The QA Lead will be responsible for overseeing the overall testing strategy, ensuring that all test activities are aligned with project goals and timelines. This role involves managing the test environment, coordinating resources, and maintaining communication between the QA, development, and product teams. The QA Lead will ensure all testing phases are completed as planned and will address any risks or challenges that arise during the process.

Test Automation Engineers

Test Automation Engineers will focus on developing and maintaining automated test scripts for the system, covering API testing, UI testing, and specific facial recognition scenarios. Their work will ensure efficient test execution across key components, including the webapp and cloud integrations. Automation engineers will also integrate the CLI interface into the automation framework, ensuring consistent and repeatable tests for both manual and automated workflows.

Development Team Support

The development team will provide support for specific technical aspects, such as debugging hardware-software integration and enhancing the CLI interface for testing needs. Their involvement will be essential for resolving any issues identified during testing, as well as ensuring smooth integration of automation tools and frameworks into the system.