Software Testing Plan

Portcullis

March 21st, 2026

Version 1.0

Sponsor: General Dynamics Mission Systems

Client: Benjamin Walker

Faculty Mentor: Bailey Hall

Team members: Christopher Son, Aiden Seay, Preston Smith, & Ryan Todd

Overview

The purpose of this document is to outline and detail the testing methods that the Portcullis team is using to ensure solution functionality. Tests are used to ensure small-unit and logical behavior. In this document, we will discuss the project, intended behavior, and how unit, integration, and usability testing will enforce behavior and limit defects. We will end with the process and procedure the team uses to address any defects found when testing.

Table of Contents

1. Introduction

The Portcullis solution for the SLAM-Doors project is a secure, cloud-integrated access control solution that is designed to modernize and harden remote Coast Guard facilities and shelters across U.S. coastal, inland waterway, and overseas regions with the use of Raspberry Pis as a cornerstone of the solution. This solution is to replace the old system being used, which utilizes a simple system where access is granted or denied to a secure facility, utilizing a magnetic card, scanner, and controller. This presents vulnerabilities to shelters with the lack of logging for audits, power outages, network degradation, and no alerting. All of which the new solution provides.

The SLAM-Doors solution is a multi-domain project encompassing the user all the way to the web application and the database. This project also utilizes the capabilities of AWS to provide a *highly available infrastructure to host*+- the solution, with the most managed component being the door controller. To ensure that the solution works as intended and that bugs are caught and fixed or documented early in development, the team needs to test the solution at a component level, inter-component level, and at the system level. Even with services and components being handled on AWS, AWS has testing features that can help ensure the functionality of the project. Utilizing unit, integration, and usability testing will show that the individual components work and function as intended, and ultimately prove that the system works when tested together.

Unit testing is used to evaluate the individual subsystems of the system at a logical level, where each subsystem is responsible for a function of the solution. Integration testing is used to evaluate the behavior of and compatibility between two or more subsystems that need to communicate with each other for the purpose of the solution. Usability testing is the evaluation of the system as a whole, and how easy it is to understand, use, and maintain. The main domains that will need to be tested are the AWS components and the hardware.

The AWS infrastructure hosts many of the components that SLAM-Doors use to provide a highly available service. The interconnected nature means that usability tests will be based on the unit tests of the subsystems that users are expected to encounter. These subsystems are listed below with their individual unit and usability tests.

- Web App: Serve web application to clients, render web resources, UI test (Usability)
- Database: Accept well-formed data, reject malformed data
- AWS Lambda: Accept input intended data, output a success message after processing data

These individual components are testable on an immediate timeline during development. These tests will ensure that each component is able to function properly before integration. Integration testing is used to ensure interoperability between components. These major components are AWS-hosted components and local hardware, and the integration tests are listed below:

- AWS components - local hardware: 2-way communication test, access control tests, latency test, power outage tests, network degradation tests.

Because the integration tests require that both components are capable of functioning in isolation, the team will need to wait. Portcullis expects to run these tests when the hardware tests are complete, which should begin by April 7th.

This testing timeline is expected to be completed by the third week of April because of the acquisition issues the team faced at the beginning of the semester. To compensate for this delay, the team has two options: move the time tables or expedite the development of the system with poor testing. The team fully expects to complete the project, but to do it successfully, will need to implement unit, integration, and ultimately usability testing to make sure that the solution is handed over in a usable state.

2. Unit Testing

2.1 Goals For Unit Testing

Unit testing within the Portcullis system is used to verify that important functions, middleware, and route handlers behave correctly in isolation in order to detect and mitigate potential security risks. Given the security-sensitive nature of this project, security and reliability standards must be established and upheld throughout development. To meet this standard, unit testing primarily focuses on backend logic related to authentication, authorization, input validation, and event handling behavior. The Portcullis team is prioritizing unit testing on software components that contain the most important logic and function since defects can directly affect access control decisions, event integrity, and the overall stability of the system. By testing these backend components independently from the database, frontend, and hardware-connected elements, the team can identify logic errors early and patch components appropriately. In addition to backend logic and route testing, Lambda functions and software related to the Raspberry Pi door controller will be tested independently. This is especially important because the Portcullis system depends on correct interaction between cloud services, event-processing logic, and door control hardware. By testing these components independently from the fully deployed environment, the team will be able to continue development knowing that high-risk modules meet expected security and reliability standards.

2.2 Tools & Execution

The Portcullis team will use different testing tools and execution methods depending on the part of the system that is being tested. Since the project includes backend services, AWS Lambda functions, and Raspberry Pi controller software, unit testing must be adapted to each environment. Although the testing environment differs across subsystems, the primary goal of

unit testing will be to validate important logic in isolation. The following subsections describe how unit testing will be executed across each major part of the system.

2.2.1 Backend Testing

Backend unit tests will be implemented using Vitest and executed locally during development. This framework has been chosen for the Portcullis system since it supports Node.js backend and testing of middleware, helper functions, and selected route handlers in isolation. Testing will focus on high-risk backend logic such as authentication, authorization, internal secret validation, input checking, and event-handling behavior. Additionally, dependencies such as database calls will be mocked so that individual backend components can be verified independently from the rest of the system. Unit tests will be measured using a pass/fail status to determine a function's validity.

2.2.2 Lambda Testing

All Lambda functions and Raspberry Pi connection functions will be implemented using Pytest and will be executed locally during development. This framework has been chosen for the Lambda Portcullis system since Pytest is an extensive testing environment and is extremely useful for unit testing. The testing will focus on all connection mechanisms between the Raspberry Pi server and the main administration server. Dependencies such as mock door entries will be mocked to make sure the main communication protocol works between the local and global servers. Unit tests will be measured using a pass/fail status to determine a function's validity. This is the best metric for this section because, with properly defined parameters, all cases will be defined to represent what situations the functions may face.

2.2.3 Hardware Testing

All hardware tests involving the card reader, door strike, and Raspberry Pi will be implemented using pytest and will be executed locally during development. This framework has been chosen for all hardware components within the Porcullis system because Pytest contains extensive testing tools, and is known to be accurate and reliable, especially with code already written in Python. It will specifically be used to generate mock card scans, intrusion detection, and data transmission to AWS IoT Core. Unit tests will focus on card scanning abilities and intrusion detection, measured using a pass/fail status to determine a function’s validity. These are the best metrics to use for hardware testing because each component has a designated purpose without variation in functionality, and all cases will be defined to represent all plausible cases.

2.3 Units Under Test

There will be three big sections that will be tested under the unit testing category. The following sections include backend, lambda, and hardware. The backend will be responsible for all the administrator server requirements, keeping track of all door locking mechanisms across the region. The lambda section will be responsible for all communication protocols between the local door server and the main administrator server. Lastly, the hardware section will be running all of the mechanical operations of the lock. We didn't include testing the frontend or UI rendering code because that is not the focus of this capstone project. The administration server backend, communication protocols, and hardware functionality are much more important for this project.

2.4 Unit Test Design Approach

2.4.1 Backend Unit Testing

Middleware

Unit Under Test: requireAuth(req, res, next)

Purpose: Verifies the JSON Web Token (JWT) stored in the HttpOnly Cookie. If valid, the function attaches the token to the user payload to req.user and allows requests to continue.

Test Case Categories:

- *Valid inputs:*
    - Request contains a valid JWT in the expected cookie
    - JWT is correctly signed with the server secret
    - JWT contains a valid payload structure (userID, username, role)
- *Boundary cases:*
    - JWT is present and contains a payload, but not signed with the correct secret.
    - JWT is close to expiration but has not expired yet.
    - Request reaches requireAuth function with no prior req.user set.
- *Invalid inputs:*
    - No authentication cookie is present.
    - The authentication cookie exists and is empty.
    - JWT malformed, expired, or missing expected user fields.

Sample Tests:

- Request with no authentication cookie returns HTTP 401 error (unauthorized) and blocks the request.
- Request with malformed, invalid, or expired JTW returns HTTP 401 error.
- Request is a valid JWT in the authentication cookie attached to req.user and calls next()

Unit Under Test: requireRole(role)

Purpose: Verified that an authenticated user has the required role to access the protected route. If the user's role matches the required role, the middleware will allow the request to Otherwise, it blocks access

Test Case Categories:

- *Valid inputs:*
    - req.user exists and contains the required role.
- *Boundary cases:*
    - If the req.user exists and only contains the minimum required field (role), it should remain valid.
    - Required role check is completed immediately after a successful authentication.
- *Invalid inputs:*
    - Req.user is missing and/or req.user.role does not match the required role.

Sample Tests:

- Requests with req.user.role matching the required role should pass and call next().
- Requests with no req.user returns a HTTP 401 error and blocks access.
- Requests with a user role that does not match the required role should return a HTTP 403 (unauthorized) error.

Unit Under Test: requireInternalSecret(req, res, next)

Purpose: Verifies that requests from other internal systems (such as AWS Lambda) include the correct shared secret in the request header.

Test Case Categories:

- *Valid inputs:*
    - Request includes the correct internal secret in the expected header
- *Boundary cases:*
    - Request includes only the required header and no other authentication data
    - Request is not associated with a logged-in user and relies only on the internal secret header for access
- *Invalid inputs:*
    - Required header is missing
    - Provided secret in request does not match the expected secret
    - Internal secret is not configured on the server

Sample Tests:

- Requests with the correct internal secret in the request header call next()
- Requests with an incorrect or missing secret returns a HTTP 401 (unauthorized) error and blocks access

Routes

Unit Under Test: router.post("/door-state-change", requireInternalSecret), async (req, res, next)

Purpose: Accepts door-state events from a trusted internal system (such as AWS Lambda), validates the request body, and verifies that the event matches expected door and controller data.

Test Case Categories:

- *Valid inputs:*
    - Request includes the correct internal secret in the request header and valid event data in the request body.
    - Submitted event data matches the corresponding database record.
- *Boundary cases:*
    - Request contains the minimum required valid fields.
- *Invalid inputs:*
    - Required request fields are missing.
    - Field values are malformed (invalid event ID, timestamp, or state).
    - Referenced event does not exist.

Sample Tests:

- Requests with valid internal secrets and valid event data returns success and confirms that the event was verified within the database.
- Requests that are missing a valid field, such as eventID or doorID returns a validation error.
- Requests with door or controller data that do not match the stored record within the database are blocked.

Unit Under Test: router.get("/:eventId/details", async (req, res, next) => {

Purpose: Retrieves detailed information for a single event and the three most recent events for the same door. This route is called when an intrusion event occurs and is used to display detailed information on the dashboard.

Test Case Categories:

- *Valid inputs:*
    - Request includes a valid eventId for an event that exists in the database.
    - Events are found, and the related previous door events are returned correctly.
- *Boundary cases:*
    - The event exists, but there are no previous events for the same door
    - Events exist, and there are fewer than three previous events for the same door.
- *Invalid inputs:*
    - eventId does not match any stored event.
    - Database query fails while retrieving the event or previous events.

Sample Tests:

- Request with an existing eventId returns HTTP 200 (OK) with the correct event details and three previous door events.
- Request for an event with no previous events on the same door returns HTTP 200 (OK) with an empty previousDoorEvents list.
- Request with a non-existent eventId returns HTTP 404 (Not Found) with an "Event not found" error.

Unit Under Test: router.post("/login", async (req, res, next) => {

Purpose: Authenticates the user by validating the submitted username and password. Then the function checks the stored password hash, issues a JWT, and sets the authentication cookie.

Test Case Categories:

- *Valid inputs:*
    - Request contains a valid username and password and matches a user account stored in the database.
- *Boundary cases:*
    - Request contains the minimum required valid username and password values and no extra fields.
- *Invalid inputs:*
    - Username and/or password are missing.
    - Username or password contains spaces.
    - Username does not exist in the database.
    - Password does not match the stored hash.

Sample Tests:

- Request with a valid username and password returns HTTP 200 (OK), includes the expected user payload, and sets the authentication cookie.
- Requests with missing credentials returns HTTP 400 (Bad Request) with a “Missing credentials” error.
- Request with spaces in username or password returns HTTP 400 (Bad Request) validation error.
- Request with an unknown username or password returns HTTP 401 (Unauthorized) with an “Invalid Credentials” error.

Unit Under Test: setAuthCookie(res, token)

Purpose: Stores the JTW and an HttpOnly authentication cookie with expected expiration and security settings. The security settings provide protections that reduce the risk of client-side token exposures and restrict cross-site use. Allows the user to make authenticated requests.

Test Case Categories:

- *Valid inputs:*
    - Response object is provided, and the token is a valid JWT string.
- *Boundary cases:*
    - Calling setAuthCookie(res, token) creates the authentication cookie with the expected httpOnly, secure, sameSite, and expiration settings
- *Invalid inputs:*
    - Response object does not support the cookie() function.
    - Token is missing or unidentified.

Sample Tests:

- Calling setAuthCookie(res, token) sets a cookie with the correct cookie name and token value.
- Calling the function with a missing token does not silently create an incorrect authentication cookie.

Unit Under Test: router.get("/me", (req, res) =>

Purpose: Determines if the user is currently authenticated by checking for a valid JWT in the authentication cookie. If the token is valid, the route will return the user payload so the front end can restore the user's logged-in session.

Test Case Categories:

- *Valid inputs:*
    - Request includes a valid JWT in the expected authentication cookie.
- *Boundary cases:*
    - JWT contains only the fields needed to identify the authenticated user.
    - Request includes the authentication cookie and no other user session data.
- *Invalid inputs:*
    - The authentication cookie is missing
    - JWT is malformed, expired, or signed with the wrong secret

Sample Tests:

- Request with a valid authentication cookie returns HTTP 200 (ok) and the decoded user payload.
- Request with no authentication cookie returns HTTP 401 (unauthorized) with a "Not authenticated" error.
- Request with an invalid or malformed JWT returns HTTP 401 (unauthorized) with a "Not authenticated" error.

- Request with an expired JWT returns HTTP 401 (unauthorized) and a "Not authenticated" error.

Unit Under Test: router.post("/logout", (req, res)

Purpose: Clears the existing authentication cookie and ends the user's session, returns a successful logout response. This requires the user to sign in again to make authenticated requests.

Test Case Categories:

- *Valid inputs:*
    - Request reaches the logout route, and the authentication cookie can be cleared using the cookie settings.
- *Boundary cases:*
    - Logout is requested even if the auth cookie is already missing
    - Cookies are cleared using the same name and path settings used when the cookie was created.
- *Invalid inputs:*
    - The response object fails while clearing the cookie.

Sample Tests:

- Request clears the authentication cookie and returns HTTP 200 (OK).
- The logout request still returns an HTTP 200 even if no authentication cookie is present.
- Error during cookie clearing is handled if the response behavior fails.

2.4.2 Lambda Unit Testing

Unit Under Test: connect_to_database()

Purpose: Establishes a connection to the lock database using the provided configuration parameters

Test Case Categories:

- *Valid inputs:*
    - Correct database credentials
    - Reachable database host
    - Properly formatted database connection parameters
- *Boundary cases:*
    - Database connection with the minimal required parameters
    - Connection attempts again and again will specified timeout values.

        - Repeated connection attempts in a short timeframe.
    - *Invalid inputs:*
        - The response object fails while clearing the cookie.

Sample Tests:

- Successful connection returns a valid database connection object.
- Displays [DB] Connected to PostgreSQL.
- Connection attempt with incorrect credentials returns an authentication error.
- Connection attempt to an invalid host results in a connection failure.

Unit Under Test: connect_to_iot()

Purpose: Establishes a connection between the Docker container running on the Raspberry Pi and the AWS IoT Core service.

Test Case Categories:

- *Valid inputs:*
    - IOT_ENDPOINT defines the address where packages will be sent
    - IOT_CERT defines the public certificate for authenticity
    - IOT_PRIVATE_KEY defines the private certificate for authenticity
    - IOT_ROOT_CA defines a root certificate for authenticity
    - IOT_CLIENT_ID defines the ID of the client using the service
- *Boundary cases:*
    - AWS IoT is disabled through IOT_ENABLED, so the function exits early without attempting a connection.
    - Prior session state may persist across reconnects.
    - The keep-alive interval is set to 30 seconds, which should allow a valid connection.
- *Invalid inputs:*
    - One or more certificates are missing, the endpoint will be unreachable
    - The TLS handshake or MQTT connection attempt raises an exception.

Sample Tests:

- Function returns None and prints "[IOT] AWS IoT is disabled" when IOT_ENABLED is False.
- Function returns None and resets mqtt_connection to None when the certificate path is invalid.
- Function returns None and resets mqtt_connection to None when the endpoint is unreachable.

- Function logs the connection attempt and success messages when the connection completes successfully.

Unit Under Test: build_event_payload(event, user_id)

Purpose: Generates a structured event payload for IoT messaging.

Test Case Categories:

- *Valid inputs:*
    - A valid event string containing “authorized”, “no_access”, or “unauthorized”.
    - Valid and real user_id that can be found in the main user database
- *Boundary cases:*
    - User_id is provided as different data types, including integer, UUID, and string
    - Event is an empty string, but still included in the payload
- *Invalid inputs:*
    - Event is none or an invalid type
    - User_id is none and cannot be converted into a string

Sample Tests:

- Function returns a valid UUID string as event_id and a properly structured payload dictionary with valid arguments.
- What will the function return if there is an invalid event or payload type?

Unit Under Test: publish_payload(payload)

Purpose: Publishes a JSON-formatted event payload to AWS IoT Core over MQTT.

Test Case Categories:

- *Valid inputs:*
    - A properly scheduled payload from the build_event_payload function.
    - AWS IoT is enabled, and a valid MQTT connection is already established.
    - AWS IoT is enabled, and the function successfully establishes a connection before publishing.
- *Boundary cases:*
    - mqtt_connection is initially none, which will trigger a call to connect_to_iot before publishing to the service
    - Multiple concurrent calls are made

        - Payload contains more than what's required
- *Invalid inputs:*
    - The payload is missing specific values, such as the event_id
    - mqtt_connection .publish() raises an exception
    - IOT_TOPIC holds an invalid value

Sample Tests:

- Function returns True when a valid payload is successfully published to AWS IoT Core.
- Function returns False when a connection cannot be established and logs a not connected message.

Unit Under Test: queue_event(event_id, payload)

Purpose: Inserts an event into the outbound_event_queue database table with a status of "pending".

Test Case Categories:

- *Valid inputs:*
    - A valid event_id (unique identifier)
    - A properly structured payload JSON object.
- *Boundary cases:*
    - Duplicate event_id is provided
    - Payload doesn't have all of the required fields
    - Payload has more than the required fields
    - Inserts into the queue are done rapidly, all at once
- *Invalid inputs:*
    - Payload is a non-JSON type
    - SQL execution fails due to a schema mismatch

Sample Tests:

- Function successfully inserts a new event into outbound_event_queue with status "pending".
- Function does not insert a duplicate row when the same event_id is provided again.
- Payload is correctly stored as a JSON object in the database.

Unit Under Test: log_event(event, user_id)

Purpose: Generates a structured event payload for IoT messaging.

Test Case Categories:

- *Valid inputs:*
    - A valid event string containing “authorized” “no_access”, “no_access”, or “unauthorized”.
    - Valid and real user_id that can be found in the main user database
- *Boundary cases:*
    - Event is an empty string, but still gets logged and queued
    - user_id is provided in a different type other than UUID
    - The flush queue function is called even when there is only the event that is newly inserted.
- *Invalid inputs:*
    - build _event_payload fails and returns invalid data
    - Database insertion into the log table fails
    - The queue event fails when inserting into the outbound queue

Sample Tests:

- Function calls build_event_payload() and receives a valid event_id and payload.
- Function inserts a new row into the log table with event_id, door_id, event, and user_id.
- Function prints a log message containing the event type and event ID.

Unit Under Test: flush_queue(limit=20)

Purpose: Retrieves pending events from the outbound queue in chronological order and attempts to publish them to AWS IoT Core.

Test Case Categories:

- *Valid inputs:*
    - Pending events exist in the outbound_event_queue table.
    - Payloads are valid and can be processed by publish_payload().
- *Boundary cases:*
    - No pending events exist, so the function exits early without error.
    - The number of pending events exceeds the limit, ensuring only the oldest limit is processed.
- *Invalid inputs:*

    - Database query fails, such as a table missing or a connection lost.
    - Payload retrieved from the database is malformed or invalid.
    - Update statements fail due to schema or connection issues.

Sample Tests:

- Function retrieves pending events and successfully publishes all, updating their status to "sent".
- Function processes only up to the specified limit of events.
- Function logs appropriate messages for both successful sends and retries.

Unit Under Test: portcullis-ingest-events (AWS Lambda Function)

Purpose: Processes an incoming IoT event in AWS Lambda by validating required fields, inserting the event into the PostgreSQL log table, and notifying an internal backend API.

Test Case Categories:

- *Valid inputs:*
    - The event contains all required fields: event_id, door_id, controller_serial, event_type, and timestamp.
- *Boundary cases:*
    - Event values are valid but minimal, such as short strings for IDs and event type.
    - Only one required field is missing, producing a targeted validation error.
- *Invalid inputs:*
    - One or more required event fields are missing or empty.
    - SQL insert fails due to schema mismatch, constraint violation, or invalid data type.
    - Internal API returns an HTTP error.

Sample Tests:

- Function returns status code: 200 when the event is valid, the database insert succeeds, and the internal API is notified successfully.
- Function returns status code: 400 and lists missing required fields when one or more required event fields are absent.
- Function returns status code: 502 when the database insert succeeds, but the internal API returns an HTTP error.
- Function returns status code: 500 when the database connection fails or another unexpected exception occurs.

2.4.3 Hardware Unit Testing

Unit Under Test: parse_card_event(event):

Purpose: Extracts and formats raw card event data from the OSDP reader into a standardized dictionary that other functions can use as input.

Test Case Categories:

- *Valid inputs*:
  - The event dictionary contains a valid card read event type with raw byte data, bit length, format, reader number, and direction.
- *Boundary cases*:
  - DB_PORT is not set, so the function uses the default port "5432".
  - Card data is the shortest possible valid byte length.
- *Invalid inputs*:
  - Event is None.
  - Event type is not a card read event.

Sample Tests:

- A valid card read event returns a dictionary with the card_id correctly formatted as a hex string.
- An invalid card read event returns None with an error message.
- An event with a non-card event type returns None with an error message.

Unit Under Test: get_card_access(card_id, Cards):

Purpose: Checks whether a scanned card's hex ID exists in the authorized cards dictionary and returns a formatted access result dictionary containing authorization status, label, user_id, and event type for use by logging and door control functions.

Test Case Categories:

- *Valid inputs*:
  - card_id exists in the cards dictionary and contains a valid label and user_id.
- *Boundary cases*:
  - The card's dictionary contains exactly one entry, and the scanned card matches it.
  - The card's dictionary is empty, and an unknown card is scanned.
- *Invalid inputs*:
  - card_id does not exist in the cards dictionary.

    - The cards dictionary is empty.

Sample Tests:

- A card_id present in the authorized dictionary returns authorized set to True with the correct label and user_id.
- A card_id not present in the authorized dictionary returns authorized set to False with NO_ACCESS_USER_ID and event_type set to no_access.
- An empty cards dictionary with any card_id returns authorized set to False.

Unit Under Test: check_intrusion(door_unlocked, unauthorized_logged):

Purpose: Determines whether a latch monitor signal should be treated as an unauthorized entry based on the current door state and whether an intrusion has already been logged for the current event. Prevents duplicate alerts and suppresses detection when the door is legitimately unlocked.

Test Case Categories:

- *Valid inputs*:
    - door_unlocked is False and unauthorized_logged is False, representing a genuine intrusion attempt.
- *Boundary cases*:
    - Signal is received immediately after the door lock is re-engaged, so door_unlocked transitions from True to False.
    - Signal is received multiple times in rapid succession for a single door open event.
- *Invalid inputs*:
    - door_unlocked is True, meaning the door was legitimately opened by an authorized card.
    - unauthorized_logged is True, meaning the intrusion has already been recorded for this event.
    - Both door_unlocked and unauthorized_logged are True.

Sample Tests:

- When door_unlocked is False and unauthorized_logged is False, the function returns True, confirming an intrusion is detected.
- When door_unlocked is True, the function returns False, confirming no intrusion is logged for a legitimate door open.
- When unauthorized_logged is True, the function returns False, confirming duplicate intrusion events are suppressed.

- When both door_unlocked and unauthorized_logged are True, the function returns False without raising an exception.

3. Integration Testing

3.1 Approach to Identifying Integration Points

Integration testing for this capstone project verifies that independently functioning components continue to operate correctly when combined into the full SLAM-Doors system. The system that we have contains multiple components that must work together in production environments. It is important to identify the major system boundaries. There are multiple parts of the project that are responsible for different things. The best way to identify the different sections is to look at how each component accomplishes a different functionality of the project. With this project in particular, it is easy to identify these integration points because they are physical. With multiple different physical devices, we as developers need to integrate them together. In the next section, we will discuss the key components that need to be integrated with one another.

3.2 Key Integration Points

The first boundary that needs to be integrated is between the Raspberry Pi running the local lock server and the main administration database responsible for holding data of locks all across the region. This integration is important because the admin server running is directly responsible for logging what data the lock servers are sending. This integration is responsible for two things. The first is that this integration is what shares the authorized/unauthorized entry from the local server to the main administration server. The second is that this integration is what shares the intrusion detection message. Both of these are explained in more detail below in section 3.4.

The second boundary that needs to be integrated is between the main administration server and the database. The main administration server is what pushes data to the front-end to the administrator, and the server is what stores all of the data. This integration is important because it is how all of the information received from the Raspberry Pis gets displayed into useful information that the administrator can use.

3.3 How/Where Integration Tests will be Executed

Now that we know where the key integration points are, we need to figure out how these integration tests will be executed. Because the backbone of this project is AWS, we will be using their AWS Lambda service. Within this service, developers have the opportunity to send test cases to execute these Lambda functions. Data can either be generated with the card read with Raspberry Pi, or it can come from a script that generates messages from a user typing into a

prompt. Both of these functionalities will be found in the Docker container, so it is very portable. All of the testing will be tied to a testing account, so it is easy to determine real data from testing data. This would also make it easy to remove all testing data from the database since it is tied to a test user. Between a Docker container running the Raspberry Pi server and the Lambda function tests, the environment is stable to produce stable and repeatable results.

3.4 End-To-End Integration Scenarios

Integration Point: Card Reader -- Raspberry Pi -- Local Database -- Door Strike -- AWS IoT Core

Feature: Authorized Card Scan, Door Unlock, and Logging

Scenario Description: An authorized user scans their assigned access card to the card reader.

Integration Steps:

- User presents the badge to the card reader
- The card reader detects the badge and sends the raw data over the OSDP protocol to the Raspberry Pi
- Raspberry Pi receives data and validates it against its local database, checking for a registered card ID
- Raspberry Pi authorizes the user to access the shelter and sends a signal to unlock the door for five seconds
- Raspberry Pi creates a log, storing information about the user, location, and time of access within its local database
- Raspberry Pi queues the log to be published to AWS IoT Core via the MQTT Client

Expected Results:

- Card data is correctly transmitted, parsed, and verified with registered users
- Door strike is unlocked for 5 seconds only
- Logs are immediately inserted into the local database
- Logs are queued to be published to AWS IoT Core

Failure Handling:

- Invalid card scan results in denied access to the shelter
- If the card reader disconnects, the reader goes offline until a connection can be established
- If the AWS IoT Core connection fails, the queued logs will be transmitted when the connection is reestablished

Integration Point: Door Strike -- Raspberry Pi -- Local Database -- AWS IoT Core

Feature: Intrusion Detection and Logging

Scenario Description: An unauthorized person physically forces the door latch open without authorization from the control system.

Integration Steps:

- The switch is activated on the door strike latch monitor, sending a signal to the Raspberry Pi indicating an intrusion attempt
- Raspberry Pi receives a signal and checks for a recent authorized card scan
- If no authorized users have been verified within the given time frame, the Raspberry Pi marks this attempt as unauthorized
- Raspberry Pi creates a log, storing information about the intrusion, including location and time of intrusion, within its local database
- Raspberry Pi queues the log to be published to AWS IoT Core via the MQTT Client

Expected Results:

- Raspberry Pi receives a signal from the door latch monitor
- Raspberry Pi validates intrusion attempt detection
- Door Strike remains in a locked state
- Intrusion alert is immediately stored in the local database
- Intrusion alert is queued to be published to AWS IoT Core

- If the AWS IoT Core connection fails, the queued alert will be transmitted when the connection is reestablished

Failure Handling:

- If the door is unlocked when the signal is received, no intrusion report is generated
- When the signal drops from the door latch monitor, the Raspberry Pi is primed again, available to scan a card or report another intrusion
- If the AWS IoT Core connection fails, the queued alert will be transmitted when the connection is reestablished.

Integration Point: Front-end – Back-end API – Database

Feature: Data retrieval from database, display data to users/administrators, send data from web app to database.

Scenario Description: Administrator/User opens up a web application client via a web browser, authenticates, and views current door statuses and logs.

Integration Steps:

- User/Administrator submits credentials from the web client
- Front-end sends a POST request to confirm authentication, then issues a JWT cookie
- Once logged in as a user/administrator, front-end sends a POST request to the back-end to receive door information from the database
- Front-end displays up-to-date information to the user via the web client

Expected Results:

- API validates the request and sends an SQL request to the database once validation is complete
- The database sends the response body back with appropriate information
- Database information stays the same
- Returned data is reflected in the web client UI

Error Handling:

- Invalid credentials result in the client UI notifying the user of invalid credentials
- The database sends error codes, with appropriate messages, when invalid SQL commands are received
- User/administrators are unable to view system information

4. Usability Testing

Usability testing is used with the goal of ensuring a minimum standard for when users interface with the solution in some sort of way. The goals of which are to minimize confusion and maximize the achievement of user intent, while also not allowing unintended uses. Usability testing in the context of SLAM-Doors involves the cases where human interaction with the system occurs. Portcullis has identified two scenarios where this happens: the user presents an RFID card for access, and administrators/users interact with the Web Application and evaluate how easy it is to understand the information provided and navigate the web app.

4.1 Context and Assumptions

The usability tests have key environment and system configuration assumptions before tests occur. These assumptions are:

- User/Administrator is registered in the system
- User has a registered RFID card
- The Raspberry Pi Control Unit has a local and updated access table
- Backend is properly functioning, and the connection from local hardware to the backend is correctly configured
- There are no extenuating environmental factors (hurricanes, power outage, etc.)

These assumptions are present in scenarios where there are no adverse factors. This assumption is inherently risky as it doesn't take into account any possible situations where components can possibly be degraded or taken offline. Also, edge cases are not evaluated with these assumptions,ons such as when the user is registered with an unregistered card, and the user is registered with a registered card. Other test cases not assumed include administrator logging into the system during outages. The Portcullis team decided to work on these assumptions as the basis of usability testing because this ensures that, in good condition, the system is able to function properly. Then the team will be able to focus on situations where issues are present, such as outages or invalid use cases.

4.2 Usability Testing Methods

Usability testing for the Portcullis system will be conducted across two short task-based sessions scheduled after Design Review 2 and before final acceptance. During these sessions, participants will be given a specific goal to complete while the team observes how effectively they are able to use the interface. The assigned tasks will include logging into the system, interpreting access events for a specific door controller, and reviewing door status information. In addition, the team will trigger an intrusion alert so the participant can attempt to identify the door controller and user related to the event. Each session will conclude with the participant logging out of the system. Lastly, the user will be able to present a valid RFID card to a secured door and be given access.

The goal of this test will be to determine if participants can navigate the system efficiently and understand the presented information without unnecessary confusion. Feedback will be collected through a live observation, participant comments, and team notes. The team will record task start and completion time, navigation errors, or sections where a participant misunderstands the information that is being presented. The team's client will be the primary participant during these sessions to ensure that the user interface is best suited for the client's preferences. This will allow the Portcullis team to identify usability issues and allow the team to make the necessary changes for future development cycles and finally acceptance.

4.3 Fitting Usability Testing Into Development Cycles

Users: System administrators (primary), other users (secondary)

Goals: Ensure systems logs are transmitted and usable in the system portal

Method: Short, task-based usability sessions and informal client feedback

Sessions:

- 1 expert review (GDMS client)
- 1 client representative (other GDMS user)

Tasks:

- Log in and access the dashboard
- Locate and interpret access events for a specific door
- Review door status and event logs
- Log out of the system

Measures: Ability to identify and interpret event logs, errors/confusion points, and user comments

Follow-up: Prioritize high-impact issues identified for the next development cycle.

Users: System administrators (primary), other users (secondary)

Goals: Ensure systems logs are transmitted and usable in the system portal

Method: Short, task-based usability sessions and informal client feedback

Sessions:

- 1 expert review (GDMS client)
- 2 client representatives (other GDMS users)

Tasks:

- Log in and access the dashboard
- Locate and interpret access events for a specific door
- Review door status and event logs
- Log out of the system

Measures: Ability to identify and interpret event logs, errors/confusion points, and user comments

Follow-up: Prioritize high-impact issues identified for the next development cycle.

Users: Authorized personnel

Goals: Ensure the system grants access to shelters to authorized users

Method: Short, task-based usability sessions and informal client feedback

Sessions:

- 1 expert review (GDMS client)
- 1 client representative (other GDMS user)

Tasks:

- Present the badge to the card reader
- Attempt to access the shelter
- Log in and access the dashboard

- Review and interpret the event log
- Log out of the system

Measures: Ability for authorized personnel to access shelter, errors/confusion points, and user comments

Follow-up: Prioritize high-impact issues identified for the next development cycle.

5. Testing Workflow and Quality Controls

The Portcullis team needs a system that can find, document, and address any defects found at the unit, integration, and usability levels. This purpose can be handled with GitHub Issues tracker, where team members can open issues for bugs, categorize them, document remedial steps, and then update the statuses of issues based on if bugs are fixed or not. The workflow of such a process should be formed like this:

Title: A concise label to specify the bug/issue.

Description: A brief description of the error, where to find it, how to reproduce it, and the list of steps already taken to remediate it. Also, link to any other issues that may relate to it.

Labels: One-word tags that must state the severity of the issue it's tagged to, what component it relates to, the User Story it relates to, if applicable, and the status.

1. Severity:
    a. HIGH - Issue that will affect the entire system and is expected to have a major impact; users are directly affected
    b. MEDIUM - Issue that will affect individual components or subsystems and has a moderate impact on the system; users may be affected
    c. LOW - Issues that are trivial and may at most affect individual subsystems, and have a negligible impact on the system. Users are not expected to be affected
2. Component:
    a. AWS - Amazon Web Service related component
    b. Hardware - Hardware-related component

        i. Raspberry Pi
        ii. Card/Card reader
    c. WebApp - Web application-related component
        i. API
        ii. Middleware
        iii. Registration
3. User Story:
    a. User Story number: (Ex. US-3)
4. Status:
    a. Opened - Defect is identified as unique, documented, and needs to be assigned
    b. Assigned - Issue is being worked on by a team member
    c. Closed - Issue has been remediated, where a solution is well documented

When issues are created that identify a bug, they should be handled based on the severity, with HIGH severity being prioritized in descending order. This makes sure that trivial issues do not take up time that should be used on issues that are expected to impact users and the system negatively. When issues are being worked on, the assigned team member must document remedial steps taken to address the issue. Once a solution is found, it should be clearly stated in the issue description, ensuring that ALL tests at all levels still pass. Then the issue can be closed when the solution is applied to the solution. The flow goes as follows:

1. Identify defect
2. Open issue on GitHub
3. Document remediation steps
4. Ensure all tests (Unit, Integration, Usability) pass if a solution is found
5. If step 4 is complete, document the solution and integrate the solution into the codebase

6. Conclusion

The complexity of the SLAM-Doors solution requires that solid testing be conducted on all subsystems of the project to ensure that the system is secure and fulfills the requirements of the project. At the most fundamental, atomic level, this is done with unit testing, where logical

divisions of the system can be tested, such as API, Middleware, Lambda functions, and hardware components.

But, Portcullis cannot rely on unit testing alone, as the team needs to know if related subsystems can work together to satisfy the project's behavior that the project needs and contribute to the achievement of the project requirements. This can be done with integration testing, where the goal is to connect related subsystems together and ensure that the expected target functionality is achieved. This includes subsystems being able to ingest data, process it, and output it into a target form, and be able to feed that data into another subsystem. For example, integrating the Raspberry Pi component with the database component via IoT Core and Lambda functions.

Ultimately, the full functionality of the system will be tested with a usability test where users are able to test the solution design with hands-on tests on the project. The goal of which is intended to holistically test each subsystem and subcomponent, and see if it can fulfill the requirements of the project. For example, users are able to scan a registered card, and then are given access to a door.

The three tiered approach to testing the project builds trust of the solution from the ground up, ensuring that subsystems at the foundation-level acts as expected, that subcomponents behave by being able to ingest input and process output, and holistically ensure that the system works and fulfills its requirements. This ground up approach provides the necessary baseline to provide and establish a maximally error-free and functioning system that users are able to interact with without issues.