Software Design Document

Version 2.0

Portcullis

February 27th, 2026

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 show the project's overall architectural design as well as the design of each major model, with a focus on the technical decisions and structures that will guide the implementation of the final product.

Table of Contents

1. Introduction....3
2. Implementation Overview....4
2.1 Raspberry Pi Communication Services....4
2.2 Backend Server....4
2.3 Front End Infrastructure....4
2.4 Security Applications....5
3. Architectural Overview....6
3.1 Infrastructure Purpose....6
3.2 Local Decision-Making and Resilience - G....7
3.3 Secure Event Transmission & Centralized Processing - G & F....7
3.4 IoT Message Routing and Serverless Event Processing - F & E....8
3.5 Secure Database Access and Network Isolation - D & E....8
3.6 Application Backend and Controlled API Access - C & D....9
3.7 Frontend Delivery and Public Access (S3 + CloudFront) - A....9
4. Component-Level Design....11
4.1 Amazon Elastic Container Service....11
4.2 Amazon RDS....12
4.3 Raspberry Pi....13
4.4 PostgreSQL Database Tables....14
5. Implementation Plan....20
6. Risks, Trade-offs, & Mitigation....23
7. Conclusion....24

1. Introduction

General Dynamics Missions Systems (GDMS) is a leading defense manufacturer and contractor that builds and delivers critical mission systems to defense, government, intelligence, and cybersecurity clients. The defense industry plays a critical role in supporting national security, providing advanced and secure communication, surveillance, and access control systems to government and military organizations. As a major defense contractor, GDMS designs and delivers mission systems to the federal government, generating value through government contracts. These contracts may include secure communications, cyber solutions, and security technologies for organizations such as the U.S. Coast Guard. Portcullis' project specifically supports GDMS in developing a door-locking solution for remote Coast Guard facilities, directly impacting the security and protection of critical Coast Guard infrastructure, ensuring mission success.

Currently, the U.S. Coast Guard uses commercial off-the-shelf (COTS) products, which include latches, software, and more to secure sensitive sites. While these products can secure these sites, there are multiple shortcomings that COTS products have, which make them less than optimal for securing Coast Guard installations. Chiefly, these systems must be able to operate without power safely and securely, while remaining up to date with modern security standards. These limitations create operational and logistical challenges for the Coast Guard, motivating GDMS to seek an in-house solution that provides full design control, enhanced security, and lower long-term maintenance costs. The Portcullis project will deliver a secure, reliable, and resilient access control system designed specifically for remote Coast Guard installations. Requirements include power-independent operation and centralized control. Overall, this project consists of three parts. The first part is the administration panel, which gives the admin operator views on the status of all door lock mechanisms in the region. The second part is the communication protocol designed to be secure and resilient under high network latency. Lastly, the third part is the Raspberry Pi, which will be the computer controlling the door lock and will be making all of the decisions (access or no access) locally. All three of these parts will directly address the limitations of existing COTS solutions and improve the protection of critical infrastructure.

2. Implementation Overview

To address the operational and security requirements and the limitations of the existing COTS solution identified in the previous section, the Portcullis system is implementing a cloud-based access control platform using Amazon Web Services (AWS). The implementation prioritizes secure device communication, power-resilient operation, centralized control, and long-term maintainability with persistent storage.

At a high level, the system consists of IoT-enabled door controllers running with Raspberry Pis located in remote U.S. Coast Guard facilities with potential power outages and network latency. These locks, running a local server, will send status updates in secure pipelines to a centrally controlled database accessed by a server to display information on a web-based dashboard. We will call this the infrastructure of our project. Below are the key design elements of the Portcullis system:

2.1 Raspberry Pi Communication Services

- **AWS IoT Core**: Responsible for secure and resilient device connectivity, device certificate validation, and routing rules for lock events.
- **AWS Lambda:** Takes in and validates device messages, normalizes the payloads for the database, and enables potential actions based on events.
- **Amazon Secure Notification Service (SNS)**: A messaging service that immediately alerts the admin when a critical event occurs that requires immediate attention.

2.2 Backend Server

- **Amazon Elastic Container Service (ECS) on Fargate**: Runs the Node/Express API that deals with authentication and database querying.
- **Amazon Elastic Container Registry (ECR):** Stores the Docker image of the backend server that will be pushed to the ECS server.
- **Amazon Relational Database System (RDS) for PostgreSQL:** Stores all logs for door uses, including *NO ACCESS*, *AUTHORIZED ACCESS*, and *UNAUTHORIZED ACCESS*, while also storing credentials and logs determining who has access to which lock.

2.3 Front End Infrastructure

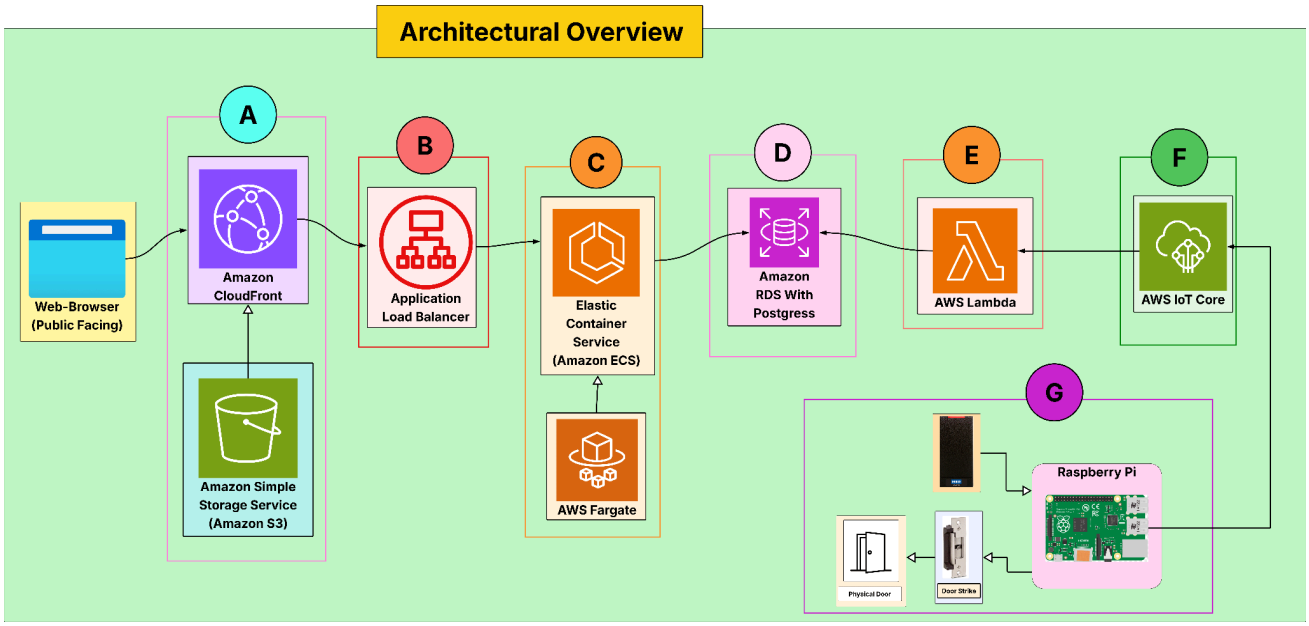
- **Amazon S3**: Stores static React files, including HTML and CSS assets. These files are then loaded into the user's browser,
- **Amazon CloudFront:** Displays the dashboard using HTTPS. Loads static files from the S3 bucket into the user's browser. Cloud Front is also responsible for routing dynamic requests to the Application Load Balancer

2.4 Security Applications

- **AWS IAM Roles**: This service is the manager of roles and policies for all of the different services, adding extra security in the application by determining which service can access secrets and provisioning read/write permissions.
- **AWS Secrets Manager**: Stores confidential information, including database credentials and keys.
- **AWS Certificate Manager:** Issues and manages TLS certificates for HTTPS requirements
- **Application Load Balancer (APB):** The main application that manages the entry point via HTTPS request/response to the application.
- **Amazon Virtual Private Cloud (VPC) :** Manages all security groups and network isolation. Allows different AWS services to communicate over a private subnet.
- **Amazon CloudWatch**: A centralized location where infrastructure errors are located.

Implementation of all parts strongly relies on the proper setup and configuration of the infrastructure. Without the proper backbone, the Raspberry Pi door lock will not be able to send encrypted logs and events to the system. Each part of the system is essential to meet all the requirements set by the client.

3. Architectural Overview



Building on the implementation overview described in the previous section, this section discusses the architecture of the SLAM-Doors solution at a high-level. This discussion will focus on the major components, their description and uses, and how each component interacts with the others within the system. Above is a diagram that illustrates the high-level architecture of the SLAM-Doors system, showing how all of the AWS services interact with each other. Each section of the diagram is appropriately grouped by letter and will be referenced as such below. The services that are not in the diagram above but are listed in the section above are supporting services and do not connect in the diagram. Firstly, we will discuss the purpose of the infrastructure before diving into specific components.

3.1 Infrastructure Purpose

In order to meet the requirements and specifications provided by GDMS, the SLAM-Doors infrastructure is designed to address and overcome several core challenges. These challenges include ensuring secure communication between all parts of the system, maintaining operation over high-latency environments, intermittent connectivity, or power loss. To overcome these obstacles, the SLAM system must centralize access control data without exposing sensitive data infrastructure to the public internet and still provide the functionality to provide real-time data to authorized users.

To address these challenges, the Portcullis team follows a layered cloud architecture built on Amazon Web Services (AWS). This architecture allows the system to offload complex

networking, enforce strict communication, security protocols, and scalability concerns to managed AWS services while still maintaining control over data flow and access boundaries. As a result, we can create a secure back-end system that is fault-tolerant and connected. Since this solution relies heavily on AWS services, most of the connections will be handled via AWS. Abstracting and offloading networking to AWS allows for the system to be responsive, resilient, and scalable.

3.2 Local Decision-Making and Resilience - G

One of the primary challenges in remote access systems is the reliance on continuous network connectivity to make authorization decisions. To account for this, the Slam-Doors system performs door access validation locally. Controllers are equipped with local storage and a battery backup to ensure that the site can remain accessible during power loss and unstable connections. The architectural decision allows doors to maintain an updated entry log and be completely independent systems that can function in uncertain conditions.

At the edge of the system, the Raspberry Pi door controllers (G) are responsible for authenticating users and providing site access to users through assigned RFID cards. These controllers are responsible for local authorization and granting access. Each door controller generates detailed access logs and then packages this data to be sent to the cloud infrastructure using a network connection.

3.3 Secure Event Transmission & Centralized Processing - G & F

While door access requests are determined locally (G), a main component of the system is to store entry logs and intrusion events in the PostgreSQL database (D). AWS handles this transmission using a variety of tools to assist with this transmission. To accomplish this without exposing backend infrastructure to unauthorized devices, all communication between door controllers (G) and the cloud (D) is authenticated and encrypted.

To establish a trusted communication channel, each Raspberry Pi door controller (G) receives a unique device certificate and private key that identifies the controller, allowing it to communicate with the AWS Internet of Things Core (IoT) (F). These certificates are paired with AWS IoT policies (F), which allow the system to define what actions the door controller (G) is permitted to take. This is a critical component of the system, which only allows verified door controllers to transmit data to the system, improving security.

Once a device has been authenticated, door controllers (G) transmit event data using the MQTT protocol over TLS. This creates an encrypted channel to send access data between the door controller and AWS IoT (F). This communication channel allows the IoT Core (F) to securely receive and route messages to the rest of the system without directly exposing the database (D) or other AWS services to the Raspberry Pi door controllers (G).

3.4 IoT Message Routing and Serverless Event Processing - F & E

While the AWS IoT Core (F) provides device authentication and the ability to receive event logs, it does not perform application-specific logic or directly interact with the database (D). To introduce a scalable processing layer that is capable of handling variable event loads, the Slam-Doors system will utilize a serverless data processing layer using AWS Lambda (E).

To configure data transmission between the IoT core (F) and Lambda (E), the IoT Core Rules Engine (F) will be responsible for evaluating incoming MQTT messages and routing this validated data to functions created within Lambda (E).
AWS Lambda (F) will act as the primary event processor in the system. Since Lambda (E) is a "serverless" service, Lambda (E) automatically scales in response to incoming events. This allows the system to handle bursts of data being sent from multiple door controllers without slowing down the main system. This is specifically important in situations where multiple door controllers go offline and reconnect at the same time. Since this service will be responsible for processing and routing door data to AWS Relational Database Service (RDS) (D), each event will be handled independently to avoid a bottleneck.

To securely transfer this data from Lambda to the Postgres database (RDS) (D), Lambda (E) will be deployed in a private subnet inside of a Virtual Private Cloud (VPC). Lambda (E) functions will be attached to the same private network as RDS (D). This is implemented so all data will be private and never exposed to the public internet.

3.5 Secure Database Access and Network Isolation - D & E

One of the most important requirements of the system is that data sent to and from the RDS database (D) must be strictly controlled and private. Since resources deployed inside a VPC are not reachable from the public internet, Lambda functions (E) must be explicitly attached to the VPC. This allows communication between services to be isolated and protected.

Private network access will also be controlled through the use of security groups. Each service supports the use of security groups that restrict inbound and outbound communication. This communication can be enforced using least-privilege IAM roles. With this component, we can guarantee that only authorized system components are able to connect, write or retrieve data from RDS (D).

Database credentials will never be stored directly in application code or configuration files. Instead, the required database configuration will be stored using AWS Secrets Manager and retrieved by Lambda at runtime using IAM roles. Before data is transferred, Lambda (E) will retrieve the database secrets using its assigned IAM role. Only then will the RDS Security Group allow data to be retrieved and stored in the database (D).

3.6 Application Backend and Controlled API Access - C & D

While Lambda (E) is useful for processing event data, it is not suitable for handling long-lived user interactions for tasks such as user authentication, role-based access control, and other dashboard requests. To allow the system to handle these interactions, the SLAM-Doors system will implement Amazon Elastic Container Service (ECS) within AWS Fargate (C).

The backend application (C) will be responsible for creating and hosting API endpoints using NodeJS. This API will be responsible for enforcing user authentication using JsonWebTokens for session expiration, Bcrypt to hash user passwords to be sure that user passwords are never directly stored in the database. The Node API will incorporate the logic to view access logs, intrusion events, and door status information (C → D).

To allow for consistent deployment of the backend service, the Slam-Door system will use Docker Container Images. Containerization allows for consistent version control, reproducibility, and consistent behavior during deployment. Docker Container Images will be stored and built using AmazonsElastic Container Registry (ECR) (C), which is a private repository that directly integrates with ECS (C).

The backend containers that are deployed using ECS (C) with AWS Fargate (E) will allow the system to perform user authentication and dashboard logic without the need to manage a linux server. Fargate automatically scales based on demand. This is necessary because it allows the backend API to remain responsive as the number of active users increases. All ECS tasks will be deployed within private subnets of a VPC to prevent public access.

All external access to the backend API (C) is routed through an Application Load Balancer (ALB) (B) , which serves as the single entry point for client requests. The ALB (B) will also enforce HTTPS connections and perform TLS termination, which is a key requirement of the system. The TLS termination will be enforced using managed certificates, then the ALB will route the request to the appropriate ECS Target Group for processing (C). Similar to Fargate (E), security groups will be configured on the ECS cluster (C) to only accept inbound traffic from the ALB (B). This creates another layer of security for the backend services.

By separating event-driven processing (Lambda (E)) and user-driven interactions (ECS) (C), the Slam-Doors system is able to support and tolerate high-volume event data and secure administrative access. These components support scalability, security, and maintainability while isolating tasks so that each component is used for the workload it is best suited to handle.

3.7 Frontend Delivery and Public Access (S3 + CloudFront) - A

The Slam-Doors administrator dashboard will provide authorized users with a web-based dashboard to view system status, access logs, intrusion events, and door controller information. Since this interface is accessed through the browser, it does not require any server-side rendering.

The front-end application will be implemented as a static web application and delivered to users using Amazon Simple Storage Service (S3) and Amazon CloudFront (A).

The frontend is built as a static React application and stored in an Amazon S3 bucket (A). S3's only responsibility is to host static assets such as HTML, CSS, JavaScript files (These files do not change per user). This eliminates the need for a publicly exposed server for the application. Team Portcullis chose this design to reduce operational complexity and improve performance.

To deliver this application to users, Amazon CloudFront (A) is used as a content delivery network (CDN) to provide public accessibility. CloudFront (A) will serve as the primary entry point for the administrator dashboard and enforce HTTPS using managed TLS certificates. In order to reduce latency, CloudFront (A) uses servers called edge locations. These servers cache static content at distributed edge locations. CloudFront (A) reduces latency and ensures consistent performance for users accessing the system from different locations.

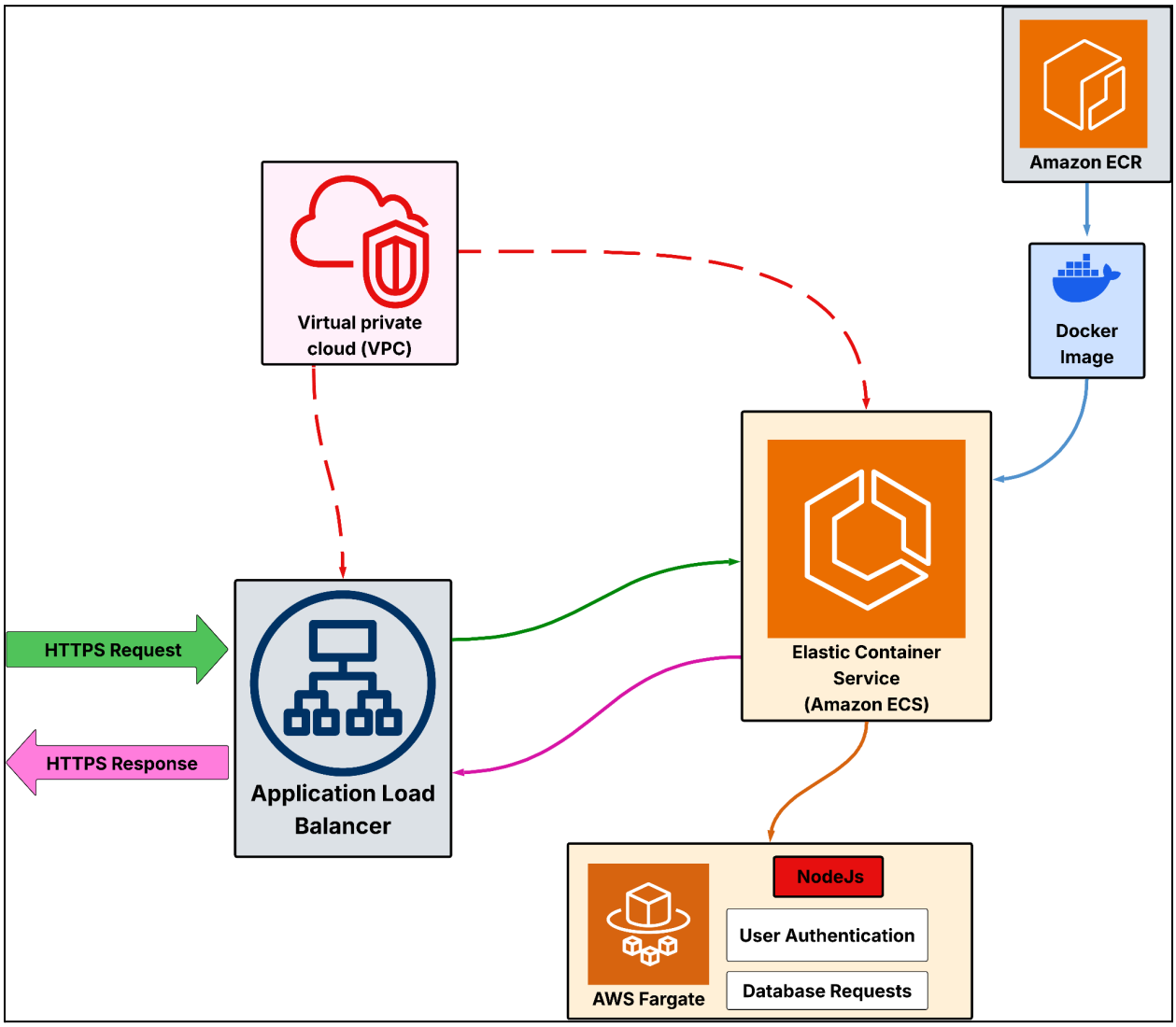
All frontend requests are routed through the CloudFront application (A) to the backend API (C) through the Application Load Balancer (B). This separation is implemented so that no backend services are exposed to the public. CloudFront (A) is configured to only serve static content directly, while dynamic operations (i.e., authentication, database access) are handled by the backend within the Virtual Private Network (VPC).

By separating the frontend content delivery from backend processing, the Slam-Doors system can allow for secure public access managed through the Application Load Balancer (B). This architecture is suited for high performance and distribution, but also allows for strict control over access to internal system components

4. Component-Level Design

In this section, we will look into major components and their internal modules and classes, along with their accompanying interfaces and dependencies. The level of workload responsibility is offloaded onto complementary AWS services, which abstracts internal implementation and reduces our involvement. We will focus on components that are responsible for content and content delivery. The components we will look at are the ECS component, the Aurora/RDS component, and the Raspberry Pi component.

4.1 Amazon Elastic Container Service

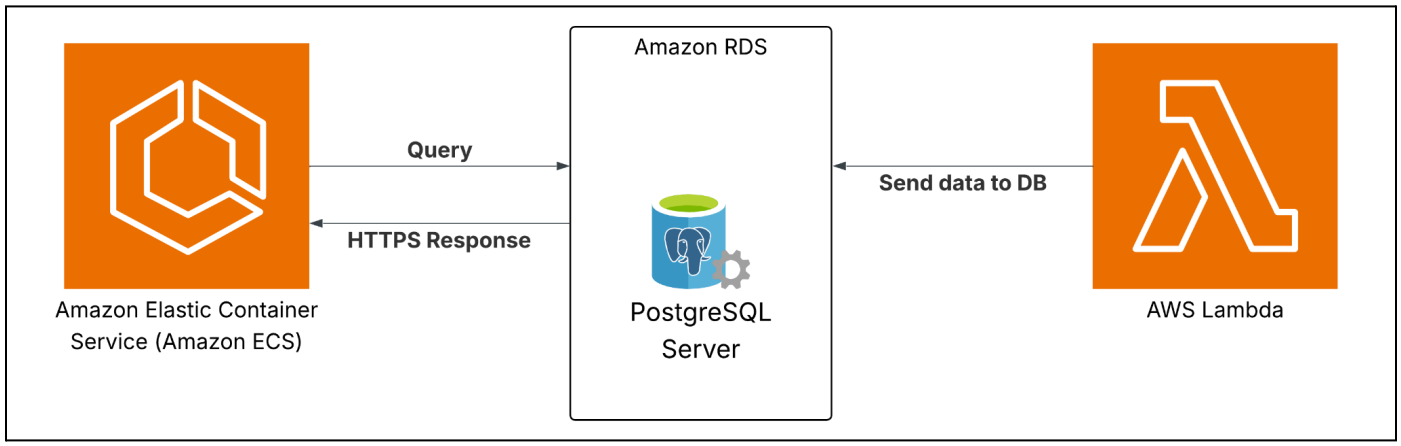


This component is responsible for hosting the administrator application. The component runs a Docker image of the Node.js backend, meaning that the team will only have to worry about updating and patching the image that is provided. The ECS-Hosted backend communicated with

the PostgreSQL database over private network connections with the Amazon Virtual Private Cloud (VPC), an AWS service that creates a private network for communication between infrastructure components. ECS also receives door controller logs from and sends commands to the Door Controllers using AWS Lambda, a serverless function-based system to process commands. Commands are then transferred via AWS IoT Core. This service is responsible for routing commands to individual Raspberry Pi door controllers using an encrypted communication channel. Before the ECS component receives any events to process, the event must go through a forward-facing load balancer, called the Application Load Balancer (ALB), which is the forward-facing application that routes valid external traffic to internal services within ECS.

Because the ECS hosts the Node.js backend, the server will be interacting with all sorts of data from both internal and external to the infrastructure. The ECS will have the responsibility of processing authenticated data flows, querying the database, accepting incoming requests to view door controller data, and sending logs and execution commands. This responsibility makes it so that important events are properly and securely handled.

4.2 Amazon RDS

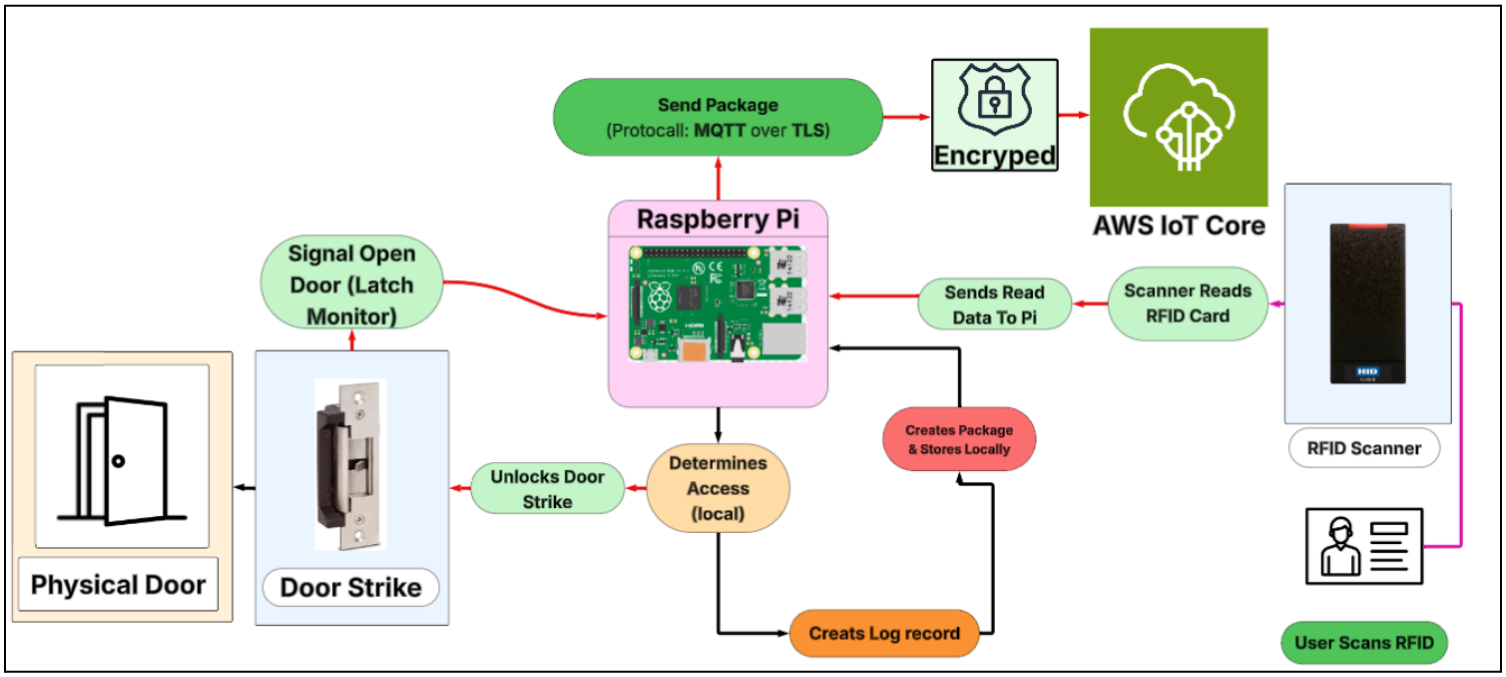


This component is responsible for the permanent storage of system-generated data, which is described below. This component receives JSON data from the administrator application in the ECS, which is then placed in their appropriate tables, so long as they are not malformed. The tables are ‘users’, ‘logs’, ‘access’, ‘rfid_cards’, ‘door_controllers’, and ‘system_commands’. This component also sends queried data to the ECS component for analytical purposes when queries and commands come from the administrator application.

Much of the technology of this component is abstracted by AWS, apart from configuration. This component is directly networked with the ECS component by use of Amazon Virtual Private Network (VPC). This network implementation secures communication between the two components, making man-in-the-middle attacks near impossible. Keeping in mind that the

database is internal and is only accessible via the ECS, this makes the component isolated from any public network.

4.3 Raspberry Pi



This component is responsible for generating door event logs. These logs come in the form of JSON, taking the schema as described above. This component generates payloads for the following tables: ‘door_controller’, ‘access_table’, and ‘log’. However, this component will have a local PostgreSQL database that replicates the ‘users’ and ‘rfid_cards’ for authentication purposes. The Raspberry Pi will send and receive data via AWS IoT Core, which handles encryption. This component also executes commands from the administrator application, including reboot, update, open, lock, and test.

Along with generating the door event logs, this component is also responsible for controlling all hardware components. The Raspberry Pi contains GPIO pins that can take in and send out simple electrical signals, and will be used to interact with the door strike and door latch monitor, which is built into the door strike. The door locking mechanism will be powered from an external 12V power supply, and a MOSFET transistor will receive a signal from one of the Raspberry Pi’s 3.3V GPIO pins, to allow the full 12V to flow to the strike, unlocking the door. Similarly, the door latch monitor will be connected to a separate GPIO pin that will send a signal to the Raspberry Pi if the strike is opened. This will help detect unauthorized intrusion attempts. Finally, the last piece of hardware it will interact with is the card reader itself. The Raspberry Pi does not natively support serial interfaces, so we will be utilizing a USB to RS-485 serial

converter, which will allow us to read input from the card reader, which sends packages in a JSON format, and will be automatically analyzed with the local database, and determine if access is granted, and immediately generate the event log.

4.4 PostgreSQL Database Tables

The database holds data of five types, each with its own table. The first is the ‘user’ table, then ‘log’, ‘access_table’, ‘rfid_cards’, ‘door_controllers’, and ‘system_commands’. Each table, and by extension, its data, will be described below as abbreviated JSON schemas.

Users:

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "Users",
  "type": "object",
  "properties": {
    "user_id": { "type"  "integer" },
    "name": { "type"  "string" },
    "password": { "type"  "string" },
    "last_login": { "type"  "string", "format": "date-time" },
    "role": { "type"  "string" }
  },
  "required": ["user_id", "name", "password", "role"]
}
```

This table is used for the users table within the database. The data within this schema is not to be transmitted over a network outside of the infrastructure, only between the administrator application and the database. When user information is being transmitted, this schema is expected; any malformed JSON payload will be dropped by the receiving component of the system.

Logs:

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "AccessLog",
  "type": "object",
  "properties": {
    "event_id": { "type"  "integer" },
    "door_id": { "type"  "integer" },
    "date_time": { "type"  "string"  "format"  "date-time" },
    "access_type": { "type"  "string" },
    "user_id": { "type"  "integer" }
  },
  "required": ["event_id"  "door_id"  "date_time"  "access_type"]
}
```

This table is used to store and retrieve logs generated by the door controllers. Payloads with this schema are meant to be transmitted over a secure network from door controllers to the infrastructure of the system. When log information is being transmitted, this schema is expected; any malformed JSON payload will be dropped by the receiving component of the system.

Access Table:

```
{
 "$schema": "https://json-schema.org/draft/2020-12/schema",
 "title": "AccessPermission",
 "type": "object",
 "properties": {
  "given_access": {
   "type"  "boolean"
  },
  "access_duraction": {
   "type": "integer",
   "minimum": 0,
   "description": "Duration in seconds"
  },
  "user_id": {
   "type"  "string"
  }
  "door_id": {
   "type"  "string"
  },
  "date  : {
   "type": "string",
   "format"  "date"
  }
 },
 "required": ["given_access", "user_id", "door_id", "date"]
}
```

The purpose of this table is to log user access attempts in relation to doors and how long access was given, if access was given. Payloads with this schema are meant to be transmitted over a secure network from door controllers to the infrastructure of the system. When access information is being transmitted, this schema is expected, otherwise any malformed JSON payload will be dropped by the receiving component of the system.

RFID Cards:

```
{
 "$schema": "https://json-schema.org/draft/2020-12/schema",
 "title": "RFIDCard",
 "type": "object",
 "properties": {
  "card_id": {
   "type": "string",
   "description"  "Primary key"
  },
  "user_id": {
   "type": ["string", "null"],
   "description"  "Nullable FK → users.user_id"
  },
  "is_active": {
   "type": "boolean"
  },
  "issue_date": {
   "type": "string",
   "format": "date-time"
  },
  "revoke_date": {
   "type": ["string", "null"],
   "format": "date-time"
  }
 },
 "required": ["card_id", "is_active", "issue_date"]
}
```

This table is used to store active RFID cards given to personnel in the field who need to access secured shelters. This schema is meant to be stored in the database and propagated to door controllers for local authentication. Payloads with this schema are meant to be transmitted over a secure network from the administrator application to the door controllers. When RFID card information is being transmitted, this schema is expected, otherwise any malformed JSON payload will be dropped by the receiving component of the system.

Door Controllers:

```
{
 "$schema"  "https://json-schema.org/draft/2020-12/schema",
 "title": "DoorController",
 "type": "object",
 "properties": {
  "controller_id": {
   "type": "string",
   "description": "Primary key"
  },
  "serial_number": {
   "type": "string"
  },
  "location": {
   "type": "string"
  },
  "firmware_version": {
   "type": "string"
  },
  "last_online": {
   "type": "string",
   "format": "date-time"
  },
  "status": {
   "type": "string",
   "enum": ["online", "offline", "degraded"]
  }
 },
 "required": ["controller_id", "serial_number", "status"]
}
```

This table holds the door controllers in the system, both active and inactive. This schema is to be generated and transmitted securely from the on-site door controllers, read by the administrator application, and stored in the database. When door controller information is being transmitted, this schema is expected, otherwise any malformed JSON payload will be dropped by the receiving component of the system.

System Commands:

```
{
 "$schema": "https://json-schema.org/draft/2020-12/schema",
 "title": "SystemCommand",
 "type": "object",
 "properties": {
  "controller_id  : {

 "type": "string",
    "description": "FK → door_controllers.controller_id"
  },
  "access_type": {
    "type": "string",
    "enum": ["authorized", "unauthorized", "no_access"]
  },
  "command_type": {
    "type": "string",
    "enum": ["lock", "unlock", "reboot", "update"]
  },
  "command_date": {
    "type": "string",
    "format": "date-time"
  }
 },
 "required": ["controller_id", "command_type", "command_date"] }
```

This table's purpose is to hold a history of commands sent from the administrator console to the door controllers. This schema is generated from the administrator console and is transmitted to both the database and door controller, the latter of which executes any of the enumerated commands. When system command information is being transmitted, this schema is expected; any malformed JSON payload will be dropped by the receiving component of the system.

These schemas and commands are strict structures that will be used within the system. The actual implementation is specific to each component. The PostgreSQL database will be able to seamlessly store these tables so long as the payload is not malformed.

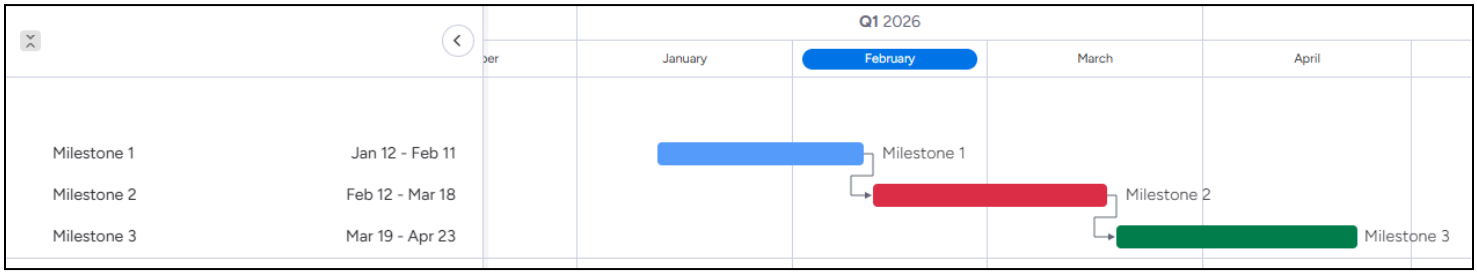
JSON was chosen as the table of choice because of its readability and ease of use. Compared to other tables like XML, the readability of JSON becomes more apparent as XML is verbose, using tags, attributes and declarations. JSON, however, averts the complexity of XML by

utilizing Key:Pair values to define attributes and their expected values. JSON also works well with REST APIs, web applications, and with real-time data transmissions.

While JSON doesn't support all data types that are used in the database, strings within the JSON payload can easily be converted to their appropriate data types in PostgreSQL. So long as the attributes are matched with their appropriate column in the database. These reasons make JSON an excellent choice as the table of choice.

5. Implementation Plan

Throughout the remainder of the development of this project, there are 3 key milestones planned to create the final Portcullis system in stages, with adequate time for quality testing of each milestone. The figure below showcases a Gantt chart of the envisioned development cycles and dedicated times for testing.



The chart references 3 key milestones, Milestone 1 to create the Initial Admin Portal, Milestone 2 to begin hardware integration and develop the security and reliability functionality of the project, and finally Milestone 3 to add the final polishing and full hardware integration.
The first Milestone features simulated data sent to and from the administrators portal. This milestone assumes that this data will mirror real-world data that the Portcullis system will utilize and that there is a consistent development environment that each of the team members can access. This requires the use of some AWS services, specifically ECS, ECR, VPC, application load balancer, and fargate, as well as a Docker container to containerize the software. At this stage the system is constrained by the fact all of the data used is entirely simulated, and the team must wait for access to the critical AWS infrastructure.

Milestone 2 begins the hardware integration and the development of security and reliability features. Here, the team plans to focus on ensuring the system will stay operable even through poor network connections, or a loss of access to direct power, adding encryption between every data transmission, and adding intrusion detection notifications. This assumes that the team will be able to simulate network and power degradation, that the team has the ability and knowledge to add encryption protocols with AWS, and that the hardware is obtained and functional. This

part depends on more access to AWS services, such as SNS, IoT core, secrets manager, and lambda, as well as the direct access to all required hardware. This system is still constrained by simulated events, as well as being limited to text and email notifications for intrusion detection.

For the final step, Milestone 3, the team plans to add the final touches, and full hardware integration. Here all data should no longer be simulated with a reliable and operable system, and with the ability to push remote, manual updates as needed. Here, the team assumes that all AWS services are properly deployed, and specifically that the Raspberry Pi is connected to the IoT core, as well as that the OSDP protocol is operational to communicate between the Raspberry Pi and card reader. This stage depends on the team understanding and utilizing Python and C libraries for access to OSDP protocols, and access to the GPIO pins on the Raspberry Pi to properly control all hardware components. At the end, the system will still be constrained with the inability to test multiple scenarios simultaneously, and remote updates require a reliable and stable internet connection, so the team also needs to account for failed updates, specifically needing the ability for the system to automatically revert back to an operable state in case of any failure.

Milestone 1: Initial Admin Portal (Week 6)

Objectives:

- Develop Administrator Portal
- Automate Event Logs (Simulated)
- Automate Intrusion Detection (Simulated)

Assumptions:

- Simulated events can mirror real-world events
- Development environment is consistent and setup

Dependencies:

- AWS Services: ECS, ECR, VPC, ALB, and Fargate
- Docker

Constraints:

- All data/events are simulated
- Wait times for access to AWS services

Milestone 2: Security/Reliability Development + Initial Hardware Integration (Week 10)

Objectives:

- Ensure Functionality in Poor Network/Weather Conditions
- Integrate Network Encryption
- Add Intrusion Detection Notifications

- Ensure Encryption Between Each Component

Assumptions:

- Network/Power degradation can be reliably tested
- Hardware is obtained and functional, but not fully integrated
- The team understands how to communicate and encrypt channels between AWS services

Dependencies:

- AWS Services: SNS, IoT Core, Secrets Manager, and Lambda
- Hardware is in the teams possession

Constraints:

- Testing is still mainly simulated
- Notifications are limited to text/email

Milestone 3: Final Polishing/Hardware Integration (Week 14)

Objectives:

- Integrate Hardware Components
- Ensure Consistent System Operability
- Test Remote, Manual Updates

Assumptions:

- The Raspberry Pi is securely connected via AWS IoT Core
- The OSDP protocol is set up between the Raspberry Pi and card reader

Dependencies:

- OSDP and DPIO Python/C Libraries
- Wiring diagrams for the door latch and card reader

Constraints:

- No ability for simultaneous testing of multiple scenarios
- Remote updates require stable internet connection and the ability to revert back into an operable state in case of failure

6. Risks, Trade-offs, & Mitigation

The Portcullis software design solution is largely complex, even with the steps taken to reduce responsibility and intricacy by offloading functionality to AWS. This complexity opens the solution to risks from attackers, faults in infrastructure, and delays. However, the team has taken steps in hardening the software and hardware to avoid critical failures in the service.

The first risk that the Portcullis solution faces is data inconsistency. The system generates and processes data across multiple components including the Door Controller, AWS Lambda functions, the ECS-hosted backend, and the PostgreSQL database. Because these components transmit JSON payloads between each other, any inconsistencies will cause the system to store and present incorrect data to administrators, break foreign key relationships in the database, corrupt logs, and create an untrustworthy solution. Using JSON as a flexible and interoperable data exchange format improves communication between the components and simplifies communication. However, flexibility increases the possibility of data inconsistency if the data is not created correctly and expected from the system. To mitigate this, the system needs to enforce strict schema validation for the JSON payloads and their different types. Another mitigating method used is the usage of constraints in the database, this will make sure that no malformed or bad data can be stored into the database.

The second risk that comes into consideration is that the Portcullis backend infrastructure is hosted entirely within AWS, this includes IoT Core, Lambda, RDS, ECS,S3 and the Application Load Balancer, etc. A regional AWS outage can affect these services and could temporarily disrupt monitoring, admin actions, event processing, and log transmission. Importantly, the door controllers will remain operational to facilitate access control functions at local shelters and facilities. Leveraging AWS to host and manage services significantly reduces operational burden, maintenance costs, and scalability concerns. However, reliance on cloud-infrastructure introduces dependency on AWS regional availability and limits direct physical control over hosting hardware. To mitigate this risk local Door Controller units will be able to store events and logs locally whenever a backend outage occurs. The control units will then be able to send the buffered data to the backend via IoT Core whenever the services are restored. This ensures that continuity is kept when real-time events occur and that they are stored in the system. A second mitigating strategy to reduce downtime is to utilize AWS failover routing to increase fault tolerance, which is the practice of moving solution operations to another region to minimize downtime. By allowing local door controllers to operate and log access events and failover strategies, the system preserves operational continuity at remote locations even during infrastructure disruptions and improves backend resilience.

7. Conclusion

In this document, Portcullis discusses the software design for the SLAM-Doors project, developed by the Portcullis team in support of General Dynamics Mission Systems (GDMS) and the U.S. Coast Guard. The project addresses the limitations of the commercial off-the-shelf solution due to it being in a remote, power and network-constrained environment by proposing a secure, resilient, and centralized managed door-locking system.

The proposed solution leverages local decision-making at the door controller and a centralized, AWS-supported cloud infrastructure to deliver a secure, resilient, and scalable access control system that maintains operation during network or power disruptions while providing centralized monitoring, auditability, and administrative control.

Currently, the Portcullis capstone team has completed the system architecture and component-level design, which will lead to a firm base for future implementation and deployment. This design puts the SLAM-Doors project to meet GDMS’s requirements.