Technological Feasibility

Portcullis

October 24th, 2025

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 introduce our project, describe the major technical challenges, analyze possible solutions and alternatives to these challenges, and assess the feasibility of these technologies to ensure a reliable, secure, scalable system.

Table of Contents

1 Introduction

General Dynamics Mission Systems (GDMS) is a defense manufacturer that builds and supports mission-critical systems for defense, government, intelligence, & cybersecurity clients. For this project, GDMS is developing a remote locking solution for the U.S. Coast Guard to improve security at isolated and sensitive facilities.

Currently, the Coast Guard relies on commercial off-the-shelf (COTS) hardware and software for securing remote sites. While functional, these products present several shortcomings. They lack the resilience and configurability required to operate in austere power-limited environments, and their reliance on proprietary designs prevents GDMS from maintaining control over production or long-term support. This leads to higher costs, limited customization, and difficulty maintaining compliance with modern security standards.

Our team's proposed solution addresses these challenges by designing an integrated hardware-software locking system centered around a Raspberry Pi control unit. Each unit will operate the latch mechanism, process sensor data, and communicate event logs to a centralized Administrator Application. The application will also send and receive data from a secure cloud-based database, allowing administrators to issue commands, push certificates, and monitor access events in real-time.

This all-in-one solution requires design for multiple domains concerning hardware and software. To break down the design process, we will consider the solution in three domains: **command**, **control**, and **intelligence**. Where **Command** covers the application, including APIs, and servers. **Control** covers the hardware and embedded systems (Raspberry Pi, microSD storage, power source). **Intelligence** the data, databases and analytics systems used for event logging and reporting. We can see there are multiple challenges that the team will face when developing a solution, which include but are not limited to Raspberry Pi components, network connection, web-server technologies, and security protocols.

In this paper, we will go over the technological challenges that the Portcullis team face with hardware and the implications of such challenges. Then a technological analysis will be conducted on hardware and chosen technologies with regard to desired characteristics, alternatives, analysis, chosen approach, proving the feasibility of said technology. Lastly, we will also discuss how all the chosen technologies will integrate and create a scalable, maintainable, and cost-efficient locking system.

2 Technological Challenges

We face an array of challenges that will all affect the success of our project. We can understand these challenges in their respective components:

- **MicroSD**: Selection of reliable microSD storage and stable power supply units
- **Server Infrastructure:** Data, log, and credential storage systems
- **Embedded Platform**: Raspberry Pi configuration and optimization
- **Database Configuration**: Database technologies and their respective expertise
- **Web Application stack**: Utilizing Python-based APIs, web servers, and databases
- **Access Methods:** Selecting access methods to facilitate quick authentication and access
- **Security Protocols**: Implementation of HTTPS/TLS for secure operations

Hardware decisions here are particularly critical, as they determine how long a unit can operate in a power-limited environment, how much data can be stored locally, and how reliably it can communicate with the Administrator Application. Also, a suitable software stack must be chosen to provide infrastructure for the system and meet Coast Guard requirements, which must be secure, scalable, and maintainable. With these challenges framed, the following section will evaluate alternatives for each area using the desired-characteristics criteria.

3 Technology Analysis

3.1 Hardware – microSD

3.1.1 Introduction

A requirement for this project is that the system will need to log events (status changes, access attempts, intrusion detection). To help facilitate logging, each control unit will need an internal storage system to store logs for situations it cannot transmit logs immediately. This can be achieved by utilizing a microSD card. Key factors for consideration include durability, performance, and maintenance.

3.1.2 Desired Characteristics

The storage solution must meet the following criteria:

- **Small form factor**: We will need a system that won't protrude or become too large to fit and be used with the current latches.
- **Durability:** The storage solution must be able to operate in austere environments with minimal maintenance or fear of destruction.
- **Endurance:** Units must be able to log large amounts of data for long periods of time.
- **Performance:** Must have adequate read/write speeds for logging operations.

Recognizing that storage must be **small, durable, endurance,** and **performance**, the team has found a few contenders to analyze and determine which of them are the best fit for our project.

3.1.3 Alternatives

When comparing alternative microSD's we can filter out the majority of products by simply looking at products marketed as 'Endurance', meaning the microSD card was made to handle large workloads and built to be durable. However, we are still left with some competitors to evaluate. Utilizing independent research found on the internet who tested the performance of microSD cards and using available product datasheets, we are able to analyze three cards.

- **SanDisk High Endurance –** Based on market data, this microSD has performance speeds up to 100MB/s read and 40MB/s write. This microSD card is marketed to be temperature, water, drop, wearout, magnet, shock (struck with a hammer), humidity, and x-ray proof. On top of that, the SanDisk High Endurance cards record up to 40,000 hours of video at a capacity of 512GB.
- **SanDisk Max Endurance** – Based on market data, this microSD has performance speeds up to 100MB/s read and 40MB/s write. This microSD card is marketed to be temperature, water, drop, wearout, magnet, shock (struck with a hammer), humidity, and x-ray proof. The SanDisk Max Endurance cards record up to 120,000 hours of video at a capacity of 256GB.
- **Samsung Pro Endurance** – The Samsung microSD offers a robust storage solution. These cards were marketed to be water, temperature, X-ray, Magnetic, drop, and wearout proof. On top of that, the Pro Endurance cards are rated for around 140,000 hours of endurance.

3.1.4 Analysis

The selected cards to compare are the Samsung Pro Endurance and the SanDisk Endurance cards. This is based on the marketing, that the cards were made to be durable and can record large amounts of data for a long time. However, we will need to look deeper into each product using available data from independent studies and data sheets from each of the respective brands. Note that the SanDisk/Samsung performance numbers are **4 KB random R/W** IOPS tests on Pi

4 or 5 platforms and that marketed sequential speeds are not directly comparable to logging workload.

SanDisk High Endurance

- **Performance (r/w @ 4KB blocks):** Marketed Speeds show that the High Endurance can reach speeds of 100/40 MB/s. An independent study shows that this card can read speeds of 8.7/3.4 MB/s on the Raspberry Pi 4 platform.
- **Durability:** There are no independent studies to show this card's durability, thus we'll rely on its data sheet. It can last 72 hours submerged in water, operate in temperatures from -25°C to 85°C, survive drops from 5 meters, handle 10,000 swipes, go through magnets of 5,000 Gauss, and survive x-ray blasts of 100mGy
- **Endurance:** There is no independent research done on the High Endurance card to show its real endurance capabilities. Its data sheet will be used instead. This card is capable of 20,000 hours of endurance at 256GB capacity.
- **Size Factors:** This card comes in 2 form factors
  - microSDHC for 32GB cards
  - microSDXC for 64GB, 128GB, 256GB, and 512GB

SanDisk Max Endurance

- **Performance (r/w @ 4KB blocks):** Marketed Speeds show that the High Endurance can reach speeds of 100/40 MB/s. An independent study shows that this card can read speeds of 11.7/3.8 MB/s on the Raspberry Pi 4 platform.
- **Durability:** There are no independent studies to show this card's durability, thus we'll rely on its data sheet. It can last 72 hours submerged in water, operate in temperatures from -25°C to 85°C, survive drops from 5 meters, handle 10,000 swipes, go through magnets of 5,000 Gauss, and survive x-ray blasts of 100mGy
- **Endurance:** There is no independent research done on the High Endurance card to show its real endurance capabilities. Its data sheet will be used instead. This card is capable of 120,000 hours of endurance at 256GB capacity.
- **Size Factors:** This card comes in 2 form factors
  - microSDHC for 32GB cards
  - microSDXC for 64GB, 128GB, and 256GB

Samsung Pro Endurance

- **Performance (r/w @ 4KB blocks):** Marketed Speeds show that the High Endurance can reach speeds of 100/40 MB/s. Independent studies show actual speeds of 17.7/2.7 MB/s on the Raspberry Pi 5 platform. On the Raspberry Pi 4 platform we see speeds of 10/2 MB/s.
- **Durability:** There are no independent studies to show this card's durability, thus we'll rely on its data sheet. It can last 72 hours submerged in water, operate in temperatures from -25°C to 85°C, survive drops from 5 meters, handle 10,000 swipes, go through magnets of 15,000 Gauss, and survive x-ray blasts of 100mGy
- **Endurance:** There is no independent research done on the High Endurance card to show its real endurance capabilities. Its data sheet will be used instead. This card is capable of 140,000 hours of endurance at 256GB capacity.
- **Size Factors:** This card comes in 2 form factors
    - microSDHC for 32GB cards
    - microSDXC for 64GB, 128GB, and 256GB

3.1.5 Chosen Approach

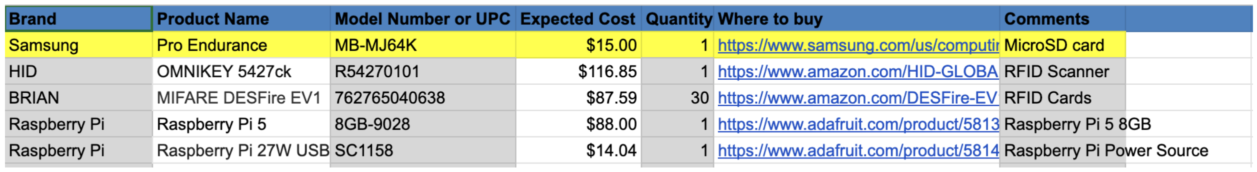
We judged each card based on **Size formats**, **Durability**, **Performance**, and **Endurance** to determine which microSD card is the best for hardware implementation. SanDisk is a close competitor to the Samsung SD card with many capacity options; however, we have more empirical data on the Samsung Pro Endurance.

Criteria	Size Formats	Performance	Endurance	Durability	Total /20
Samsung Pro Endurance	5	5	5	4	19
SanDisk High Endurance	5	1	1	4	11
SanDisk Max Endurance	5	3	3	4	15

The table above measures how each product stands in each category. 5 – evaluated very well, and 1 – evaluated poorly. Based on the accumulated scores, Samsung Pro Endurance is determined to provide the best performance.

3.1.6 Proving Feasibility

Feasibility testing will be run on a Raspberry Pi 5 platform and IO tests on it. We plan on replicating the IOPS benchmarking methods used by the independent studies. They used iozone on Linux to measure read/write performance in a number of sets, then calculate the average for each category. The team is requesting from GDMS a Samsung Pro Endurance for tests to take place. Below is an acquisition request to GDMS, which is in process to obtain physical materials for this project.



Brand	Product Name	Model Number or UPC	Expected Cost	Quantity	Where to buy	Comments
Samsung	Pro Endurance	MB-MJ64K	$15.00	1	https://www.samsung.com/us/computir	MicroSD card
HID	OMNIKEY 5427ck	R54270101	$116.85	1	https://www.amazon.com/HID-GLOBA	RFID Scanner
BRIAN	MIFARE DESFire EV1	762765040638	$87.59	30	https://www.amazon.com/DESFire-EV	RFID Cards
Raspberry Pi	Raspberry Pi 5	8GB-9028	$88.00	1	https://www.adafruit.com/product/5813	Raspberry Pi 5 8GB
Raspberry Pi	Raspberry Pi 27W USB	SC1158	$14.04	1	https://www.adafruit.com/product/5814	Raspberry Pi Power Source

3.2 Server Infrastructure - Centralized & Decentralized

3.2.1 Introduction

To address the issue presented by GDMS, our server infrastructure will be a critical component of this application. A Linux-based server will act as a control point between all door-mounted Raspberry Pi controllers and the operator's web-based dashboard. It will handle incoming data from control units, maintain the system database, push authorized commands to the off-site devices, and host the administrator web portal, where the admin can see all notifications that the off-site devices push out. This design must therefore ensure high availability, secure communication, scalability, and ease of remote management to meet the long-term operational needs that GDMS requires.

3.2.2 Desired Characteristics

The ideal server infrastructure should have the following characteristics:

- **Reliability**: The server must remain operational with minimal downtime, even during network outages or network latency. This ensures that remote lock controllers can always communicate logs or receive system (lock) updates when connectivity is available.
- **Scalability:** The system must support multiple simultaneous lock controllers across various Coast Guard sites. There must be a simple procedure to set up a lock and push updates to that lock from a centralized server.
- **Security:** All network communication and data storage must be encrypted and authenticated using modern protocols, including but not limited to HTTPS and TLS.

- **Maintainability:** The server should support easy configuration, monitoring, and updates. Logging, health checks, and automated backups are a must to automate the administration as much as possible.
- **Cost Efficiency:** The server infrastructure should be a balance between performance and affordability.
- **Compatibility:** The server must communicate flawlessly with Raspberry Pi controllers.

These characteristics help form the evaluation criteria for comparing different infrastructure configurations.

3.2.3 Alternatives

To identify the most feasible and sustainable server infrastructure for the Portcullis system, several configurations were researched, tested, and compared through small-scale deployments and reading different articles. Below are the following options that are considered:

- **Self-Hosted Linux Server:** This configuration uses a Linux server that can potentially be hosted within GDMS's internal network. Operating systems such as Ubuntu Server 22.04 LTS can be stable, open-source, and widely supported. This approach provides complete administrative control. This option also minimizes 3rd party dependencies but will require physical hardware and routine maintenance.
- **Cloud Hosted VM:** Cloud hosting with AWS provides flexible resources for computational power, automated uptime management, and global accessibility. Services like AWS can run the same Linux stack as on-premises setups but introduce managed redundancy and backup options. While this simplifies deployment and scaling, it may increase recurring costs and introduce dependencies on external networks and third-party compliance.
- **Containerized Deployment (Docker):** Independent of the physical location, containerizing the Admin Server components with Docker allows consistent, reproducible environments across cloud and on-premise systems. Each service, including the web dashboard, API, and database connector, will run in its own container. This will simplify versioning, updates, and rollback procedures.

3.2.4 Analysis

Each service below was researched online, looking through articles and documentation, while the self-hosted linux and Docker sections were tested out on a local computer.

Self-Hosted Linux (Ubuntu Server 22.04 LTS):

- **Reliability:** Solid under emulated link jitter and short outages; requires us to engineer redundancy (dual NICs, UPS, DB replicas). No built-in failover—uptime depends on GDMS facilities.
- **Scalability:** Vertical scaling is trivial (add CPU/RAM); horizontal scaling requires us to add a reverse proxy/load balancer and manage multiple hosts manually.
- **Security:** Strong: full control over OS hardening (CIS baselines), firewalling, and certificate lifecycle. Easiest place to keep sensitive data fully in-house.
- **Maintainability:** Heavier lift: patch cadence, backups, log rotation, and monitoring agents must be owned by the team.
- **Cost Efficiency:** Good long-term total cost if hardware already exists; initial procurement plus redundancy can be non-trivial.
- **Compatibility:** Excellent: we control OS, packages, and network; trivial to expose HTTPS endpoints that the lock servers expect.

Cloud-Hosted Virtual Machine (AWS)

- **Reliability:** High: provider SLAs (Service-Level Agreements), snapshots, managed disks; easy to place behind a managed load balancer for blue/green cutovers.
- **Scalability:** Strong: can resize instances or replicate horizontally; managed DNS/LB makes multi-AZ straightforward.
- **Security:** Very good, but shared responsibility: we harden the guest OS; cloud account policies/IAM must be correctly configured. Extra work for strict compliance/accreditation.
- **Maintainability:** Easier than on-premises: images, backups, and metrics are integrated; still, our job is to patch the OS/app.
- **Cost Efficiency:** Opex model; test environments can be shut down off-hours; prod incurs steady run-cost plus egress for hybrid links.
- **Compatibility:** Excellent: same Linux distro as on-premises; identical API endpoints for lock servers.

Containerized Deployment (Docker)

- **Reliability:** Containers restart automatically on failure; process isolation limits blast radius. Compose/Swarm/K8s add HA when needed.
- **Scalability:** Horizontal replication is straightforward; images are immutable, enabling blue/green or canary updates.

- **Security:** Image signing, provenance, and regular scanning reduce supply-chain risk; still require least-privilege runtime and secret management.
- **Maintainability:** Excellent: identical images across local to staging to cloud; one-command updates/rollbacks.
- **Cost Efficiency:** Minimal overhead vs bare-metal; reduces engineer time during updates and onboarding.
- **Compatibility:** Highest: same container images run on dev laptops, cloud VMs, or on-premises hosts; no code changes across environments.

3.2.5 Chosen Approach

Each alternative in section 3.2.3 was compared against the desired characteristics in section 3.2.2. This table is the result of the analysis in section 3.2.4.

Criteria	Reliability	Scalability	Security	Maintainability	Cost	Compatibility	Total **/30**
Self-hosted Linux	4	3	5	3	4	5	24
Cloud VM	5	5	4	4	3	4	25
Docker Container	5	5	5	5	4	5	29

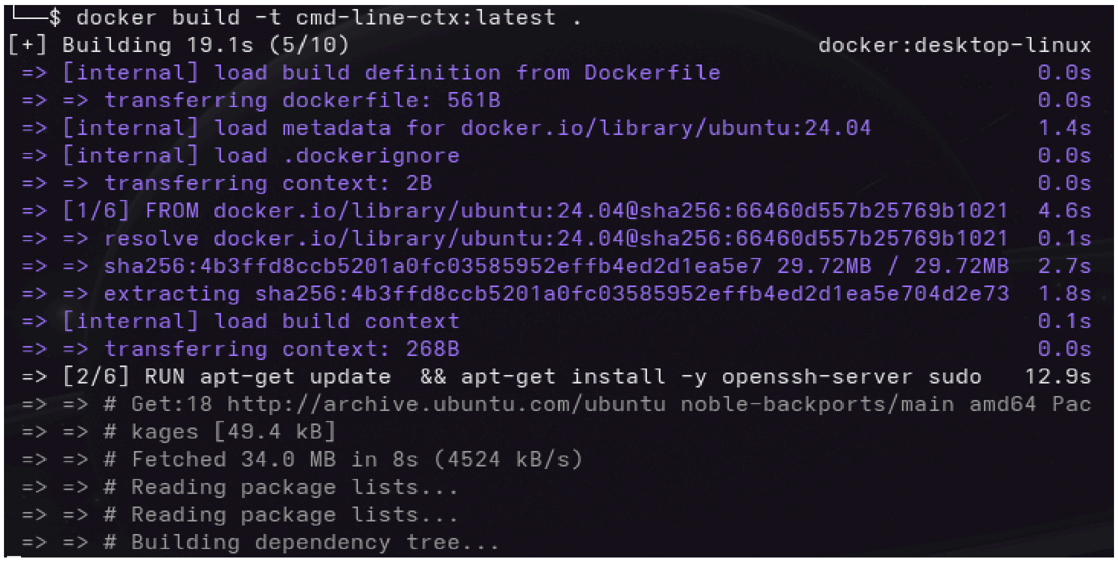
The analysis showed that while a pure cloud or self-hosted approach excels in specific areas, the Hybrid Model enhanced with Docker containers best satisfies GDMS's requirements for reliability, control, and long-term maintainability. Containerization ensures that every deployment can either be on the central server or test environments, runs the same configuration, minimizing human error and easing future scaling.

The team has selected a Dockerized Hybrid Infrastructure as the most practical and sustainable design for the centralized Admin Server. In this configuration, all primary components of the Portcullis system are included. This will contain the web portal, which will be explained in more detail in section 3.5. It will contain the REST API service and, lastly, the database connector. Since we are using Docker, everything will be containerized, ensuring consistent operation across development, testing, and production environments.

During the initial stages of development, these containers will be built, deployed, and tested locally using Docker Compose on our individual laptops. Local containerization allows the team to perform end-to-end integration testing of the entire stack, including API calls, database usage, user authentication, and encryption. Once the local environment is verified, the same Docker images will be pushed to AWS for hosting in the cloud.

3.2.6 Proving Feasibility

Our first initial test of the Docker system will focus on deploying the full Admin server container, which will include a web dashboard, database connection, and API to confirm each container communicates securely with proper TLS encryption to the lock servers. Once this local environment is tested, we can then look into adding this container to AWS, where we can host it in production. We have already tested a general Docker container to run a Linux server. Pictured evidence of building that Linux server is below.



```
└─$ docker build -t cmd-line-ctx:latest .
[+] Building 19.1s (5/10)                                        docker:desktop-linux
 => [internal] load build definition from Dockerfile                             0.0s
 => => transferring dockerfile: 561B                                             0.0s
 => [internal] load metadata for docker.io/library/ubuntu:24.04                  1.4s
 => [internal] load .dockerignore                                                0.0s
 => => transferring context: 2B                                                  0.0s
 => [1/6] FROM docker.io/library/ubuntu:24.04@sha256:66460d557b25769b1021  4.6s
 => => resolve docker.io/library/ubuntu:24.04@sha256:66460d557b25769b1021  0.1s
 => => sha256:4b3ffd8ccb5201a0fc03585952effb4ed2d1ea5e7 29.72MB / 29.72MB  2.7s
 => => extracting sha256:4b3ffd8ccb5201a0fc03585952effb4ed2d1ea5e704d2e73  1.8s
 => [internal] load build context                                                0.1s
 => => transferring context: 268B                                                0.0s
 => [2/6] RUN apt-get update  && apt-get install -y openssh-server sudo   12.9s
 => => # Get:18 http://archive.ubuntu.com/ubuntu noble-backports/main amd64 Pac
 => => # kages [49.4 kB]
 => => # Fetched 34.0 MB in 8s (4524 kB/s)
 => => # Reading package lists...
 => => # Reading package lists...
 => => # Building dependency tree...
```

Linux Server Running in Docker

3.3 Embedded Platform

3.3.1 Introduction

To address the issue presented by GDMS, our embedded platform will be crucial to the success of this application. The embedded platform must reliably interface with all other platforms as well as handle all events related to this system. It must be able to interface with any sensors, be able to access and handle all events locally, and communicate with the central server. These devices are deployed in distributed environments, meaning that they must be optimized to ensure long-term stability, robust security, and reliable performance under limited network availability. The design must balance cost and energy efficiency along with the ability to send and receive encrypted information, all while handling and logging all events within its authority.

3.3.2 Desired Characteristics

The ideal embedded platform should have the following characteristics:

- **Reliability**: Each unit must always be operational, even in high-latency or limited network conditions. It must be able to automatically recover from network and power outages without manual intervention.
- **Performance:** The system must be able to process inputs from several sensors, responsively log all activity, and encrypt/decrypt all data received and sent to the central server in any network conditions.
- **Security:** All communication must be encrypted, employing HTTPS and TLS protocols to ensure secure and strong protection against any possible unauthorized access.
- **Maintainability:** Devices must support remote software and firmware updates that can be completed over poor network connections, ensuring ease of use and easy lifetime support.
- **Cost Efficiency:** The chosen configuration should minimize hardware and power costs while maintaining excellent performance for event handling and encryption protocols.
- **Compatibility:** The device must be able to communicate with any given sensors, physical SD cards for backup logs, and the central server.

These characteristics define the evaluation basis for possible embedded platforms along with the necessary optimization abilities.

3.3.3 Alternatives

Below are some viable options.

- **Raspberry Pi 5:** The Raspberry Pi 5, is one of the latest models of Raspberry Pi and has a significant increase in performance over the previous generations. It offers a quad-core 2.4 GHz processor, along with up to 16 GB of RAM, providing excellent power for all of the required tasks, as well as thermal cooling solutions to maintain consistent performance. It also offers on board ethernet, usb 3.0 and 2.0 ports, as well an SD card slot, which will be highly useful for the GDMS requirements.
- **Raspberry Pi Zero:** This is the most cost and power efficient model of Raspberry Pi, however it is severely limited on its hardware capabilities. It contains a single core 1 Ghz process and only up to 512 MB of RAM. While these are noticeably worse specs than the other models, it outperforms them in every measure of power efficiency. While it may not

be able to handle the whole project, it would be useful in handling simple individual events.

- **Odroid N2+:** This is a direct competitor to the Raspberry Pi line of microcomputers. It offers 2.4 GHz processing speeds and 4 GB of RAM. It excels in thermal cooling, preventing the system from throttling performance, and it also offers an on board ethernet, usb 3.0 , and an SD card slot. It is known for its power efficiency, as well as the ability to add a backup power supply, however it is not nearly as cost efficient as the other solutions.

3.3.4 Analysis

To identify the most feasible and efficient systems, multiple embedded platforms have been analyzed and compared based on manufacturer documentation and specs of the performance of several modules.

Raspberry Pi 5

- **Reliability:** Remains stable under consistently sustained workloads. Improved power management and thermal dissipation. This allows the system to operate under a continuous and consistent workload without the fear of thermal throttling.
- **Performance:** 2.4 GHz processor and 16 GB of RAM. Outperforms many other microcomputers. Can handle all of the requirements with little to no problems and provides capabilities to interact with other pieces of the system with ease.
- **Security:** Full support of TLS protocols along with hardware level encryption optimization, ensuring a strong, secure system
- **Maintainability:** Natively supports a container-based management system which allows easy updates and deployment. This also allows for all updates to be handled remotely with little downtime.
- **Cost Efficiency:** It is more expensive than previous models, it offers much more performance per dollar amount.
- **Compatibility:** Fully compatible with all existing Raspberry Pi hardware, along with sensors and communication protocols. Ensures simple integration across most pieces of hardware

Odroid N2+

- **Reliability:** Excels in handling thermal stability, ensuring sustained performance under consistent workloads. Also natively allows for a backup power solution, making it easy to work with power outages.

- **Performance:** 2.4 GHz processor and 4 GB of RAM. Ensures that processes are handled quickly and efficiently but may struggle with large workloads due to its limited RAM.
- **Security:** Fully supports modern encryption standards at the hardware level as well as the ability to implement them through communication channels.
- **Maintainability:** Supports Ubuntu-based Linux systems which are known to have excellent support, but does not contain the same level of community support as Raspberry Pi's.
- **Cost Efficiency:** Much more expensive per unit, however its long-term thermal capabilities may prevent maintenance in the long-term.
- **Compatibility:** Supports standard communication through ethernet, usb 3.0 ports, and SD interfaces. However, it does not natively support many hardware components, making integration more difficult.

3.3.5 Chosen Approach

Each alternative in section 3.3.3 was compared against the desired characteristics in section 3.3.2.

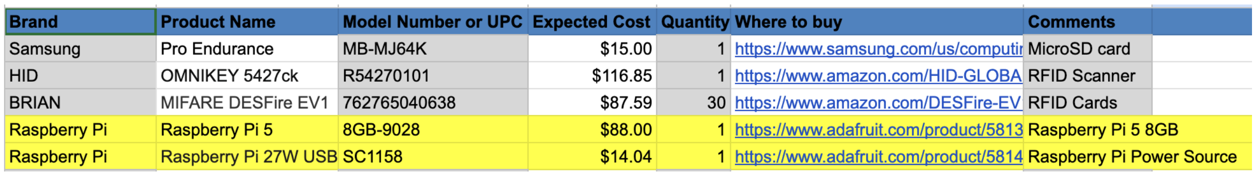
Criteria	Reliability	Performance	Security	Maintainability	Cost	Compatibility	Total **/30**
Raspberry Pi 5	5	5	5	5	3	5	28
Raspberry Pi Zero	4	3	3	5	5	4	24
Odroid N2+	5	5	5	4	2	4	25

After careful consideration, the Raspberry Pi 5 is selected as the best option for our embedded platform for this project. While it is more expensive than most of the other options, it outperforms the other options in the vast majority of areas, providing the services that are required by GDMS.

While cost efficiency is important for our project, the other factors are much more significant for the requirements set by GDMS. Security is the number one priority of this system, ensuring that no bad actors can get access to the confidential system. The system must be reliable and maintainable for a long-term implementation. The system must be performant to handle many possible pieces of the system all working in tandem with each other, which can cause strain on underpowered devices. Finally, it must also be extremely compatible with many different kinds of devices, ensuring that every part of the system will be guaranteed to work with our devices. Because of all of these integral parts to the system, the Raspberry Pi 5 balances all of these aspects the best out of all of the options, and is the right choice for our project.

3.3.6 Proving Feasibility

To validate this selection with the GDMS requirements, we will test each component required in this system. First, we will test its ability to function under a simulation of real-world workloads by testing its ability to take in information from a simulated sensor, scrape through that data, format it correctly, encrypt this data, and then send it to a database/central server. Once we have the functionality implemented in a virtual environment and it is performing as expected, we will then begin implementing this into a GDMS environment and make adjustments as needed. As of right now, we have been in contact with our client to submit an acquisition request to GDMS. Below is the acquisition request to GDMS.



Brand	Product Name	Model Number or UPC	Expected Cost	Quantity	Where to buy	Comments
Samsung	Pro Endurance	MB-MJ64K	$15.00	1	https://www.samsung.com/us/computi	MicroSD card
HID	OMNIKEY 5427ck	R54270101	$116.85	1	https://www.amazon.com/HID-GLOBA	RFID Scanner
BRIAN	MIFARE DESFire EV1	762765040638	$87.59	30	https://www.amazon.com/DESFire-EV	RFID Cards
Raspberry Pi	Raspberry Pi 5	8GB-9028	$88.00	1	https://www.adafruit.com/product/5813	Raspberry Pi 5 8GB
Raspberry Pi	Raspberry Pi 27W USB	SC1158	$14.04	1	https://www.adafruit.com/product/5814	Raspberry Pi Power Source

3.4 Database Configuration

3.4.1 Introduction

A secure database is one of the most important parts of the SLAM-Doors system for GDMS. Here, the database will be in charge of all data management, storage, and retrieval processes. The database must efficiently record, store, and organize information from both the centralized admin server and be able to store the routinely updated information from the decentralized lock servers. Information that may be stored on this database, including access control logs, user authentication data for access to the admin server, lock state changes, and system alerts. One of the main challenges we will face is to ensure a reliable synchronization between the lock servers, which can operate offline, and the central database server. The key will be how we maintain data integrity and prevent conflicts once the connectivity is restored.

3.4.2 Desired Characteristics

The ideal database solution for the SLAM-Doors system should have the following characteristics:

- **Reliability**: The database must ensure continuous data integrity even in the event of sudden power loss or network interruptions.
- **Scalability:** The system should support a growing number of locks, users, and events across multiple Coast Guard sites. This requires efficient handling of concurrent transactions.
- **Security:** All stored data must be protected using encryption in transit and at rest. Authentication and authorization mechanisms
- **Maintainability:** Administrators must be able to perform maintenance tasks such as schema updates, backups, and performance monitoring with minimal downtime. Automated tools for backup scheduling and recovery will reduce manual workload and improve reliability.
- **Cost Efficiency:** The solution should offer strong performance and advanced features without significant licensing or hardware costs
- **Compatibility:** The database must integrate seamlessly with Docker containerization and Raspberry Pi-based lock servers.

These characteristics form the foundation for evaluating potential database systems and ensuring that the chosen configuration meets the operational, security, and scalability standards required by GDMS.

3.4.3 Alternatives

To identify the most feasible database configuration for the SLAM-Doors system, several solutions were researched. Each alternative was evaluated on reliability, scalability, security, maintainability, cost efficiency, and lastly compatibility. Below is an overview of the options that were considered:

- **SQLite:** SQLite is a lightweight, file-based database engine designed for embedded systems and small-scale applications. It is widely used in IoT and mobile environments because it requires no external server process and has minimal configuration. SQLite was considered for use within each Lock Server due to its simplicity and low resource consumption.
- **MySQL:** MySQL is another popular open-source relational database used in web development. This system is known for their stability and ease of use. Some downsides of MySQL is that it doesn't follow the ACID guarantees (Atomicity, Consistency, Isolation, Durability) by default. There is also limited JSON support which may be a limiting factor in our project.
- **PostgreSQL:** PostgreSQL is again another open source relational database system. This database is known for its ACID compliance, JSON data storage, and flexible search

methods. It is well known to be part of the PERN (PostgreSQL, Express JS, React JS, Node JS) stack in web development.

3.4.4 Analysis

To identify the most feasible database configuration for the SLAM-Doors system, several relational database solutions were researched and compared based on reliability, performance, scalability, security, maintainability, cost efficiency, and compatibility. The following three options were examined in greater depth through reading both documentation and online articles. The only system that was tested was PostgreSQL.

SQLite

- **Reliability:** Stable for single-user or embedded use cases; all data is stored in a single file, minimizing corruption risk when properly closed. However, it lacks transactional durability under concurrent writes and power loss.
- **Scalability:** Poor horizontal scalability; handles small datasets well but struggles beyond a few concurrent connections. Not suitable for multi-client write operations from multiple locks.
- **Security:** Limited; no built-in authentication or encryption; requires external filesystem-level encryption for protection.
- **Maintainability:** Extremely lightweight; requires no server setup or configuration. However, backups and synchronization must be manually implemented.
- **Cost Efficiency:** Excellent; free, zero setup overhead, and runs on minimal hardware resources.
- **Compatibility:** High for embedded systems such as Raspberry Pi lock controllers; ideal for local offline caching, but not for the centralized Admin Server.

MySQL

- **Reliability:** Reliable for general web applications with mature replication and backup tools. However, by default, it is not fully ACID-compliant which means that transactions must be explicitly configured for durability.
- **Scalability:** Moderate; supports read replicas and sharding, but scaling writes efficiently requires advanced configuration.
- **Security:** Good; this supports TLS connections, user authentication, and role-based access, though encryption features require more manual setup than PostgreSQL.

- **Maintainability:** Well-documented and easy to manage; compatible with most admin tools such as phpMyAdmin and Workbench. Schema migrations can become cumbersome with large datasets.
- **Cost Efficiency:** Free and open-source; abundant community resources keep operational costs low.
- **Compatibility:** Excellent with standard web stacks (LAMP), though limited JSON support and non-standard SQL extensions may restrict flexibility for the SLAM-Doors data model.

PostgreSQL

- **Reliability:** Excellent; fully ACID-compliant with write-ahead logging (WAL) and automatic recovery.
- **Scalability:** Strong; supports both vertical scaling and horizontal replication. Built-in partitioning and connection pooling enable efficient growth across multiple sites.
- **Security:** Outstanding; TLS encryption, role-based access control, certificate-based authentication, and row-level security policies available natively.
- **Maintainability:** Very high; supports Dockerized deployment, automated backups including tools like pg_dump and pg_basebackup, and structured schema migrations. A large developer ecosystem ensures long-term sustainability.
- **Cost Efficiency:** Free and open-source; offers enterprise-grade performance without licensing costs. Minimal hardware requirements for high reliability.
- **Compatibility:** Excellent; works seamlessly with the project's Docker infrastructure and lock server communication stack. Fully supports JSONB for flexible data structures.

3.4.5 Chosen Approach

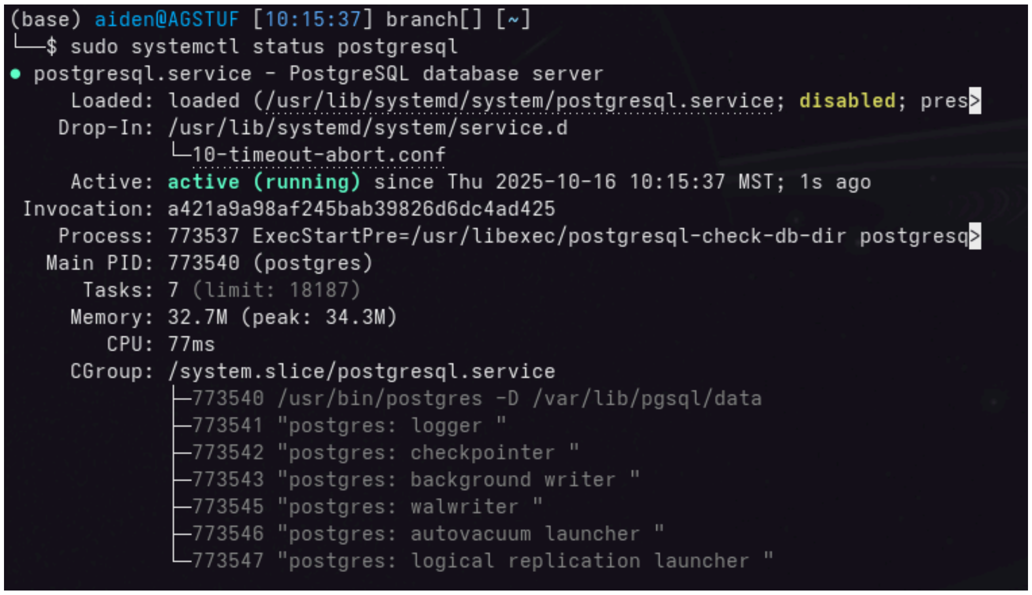
Each alternative in section 3.4.3 was compared against the desired characteristics in section 3.4.2. The following table summarizes the results of the analysis presented in section 3.4.4.

Criteria	Reliability	Scalability	Security	Maintenance	Cost	Compatible	Total **/30**
SQLite	3	2	2	4	5	4	20
MySQL	4	3	4	4	5	4	24
PostgreSQL	5	5	5	5	5	5	30

PostgreSQL is selected as the primary database for the centralized admin server. Its full ACID compliance ensures data integrity during concurrent operations and system recovery, while native JSONB support enables efficient storage of semi-structured sensor and log data. PostgreSQL's strong Docker compatibility and wide ecosystem support also make it ideal for seamless integration with the project's containerized environment.

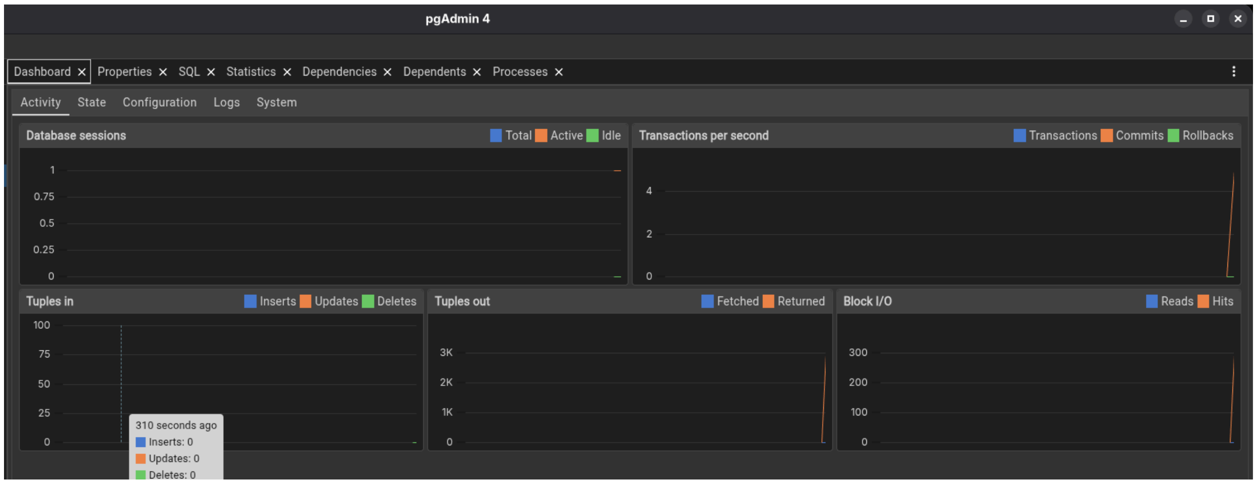
3.4.6 Proving Feasibility

Initial testing has already confirmed that PostgreSQL can be successfully deployed and managed within the project's development environment. The team has demonstrated the ability to host a PostgreSQL instance locally using Docker, verifying container stability, data persistence, and schema configuration. In addition, pgAdmin4 was installed and configured to provide a visual interface for monitoring queries, viewing active connections, and managing database objects. This verified that PostgreSQL can integrate seamlessly into our existing development workflow. Both images below are proof of concept and prove that we can use this technology. Later, we can test the connection between the database and the admin server in separate Docker containers to verify that inter-container networking, authentication, and secure data transmission over TLS function correctly within the Docker environment.



```
(base) aiden@AGSTUF [10:15:37] branch[] [~]
└─$ sudo systemctl status postgresql
● postgresql.service - PostgreSQL database server
     Loaded: loaded (/usr/lib/systemd/system/postgresql.service; disabled; pres>
    Drop-In: /usr/lib/systemd/system/service.d
             └─10-timeout-abort.conf
     Active: active (running) since Thu 2025-10-16 10:15:37 MST; 1s ago
 Invocation: a421a9a98af245bab39826d6dc4ad425
    Process: 773537 ExecStartPre=/usr/libexec/postgresql-check-db-dir postgresq>
   Main PID: 773540 (postgres)
      Tasks: 7 (limit: 18187)
     Memory: 32.7M (peak: 34.3M)
        CPU: 77ms
     CGroup: /system.slice/postgresql.service
             ├─773540 /usr/bin/postgres -D /var/lib/pgsql/data
             ├─773541 "postgres: logger "
             ├─773542 "postgres: checkpointer "
             ├─773543 "postgres: background writer "
             ├─773545 "postgres: walwriter "
             ├─773546 "postgres: autovacuum launcher "
             └─773547 "postgres: logical replication launcher "
```

Hosting server locally on a laptop



Using pgAdmin4 to view the database

3.5 Web Application Stack

3.5.1 Introduction

The web application stack will serve as a centralized management system where authorized personnel can view, control, and audit all connected door units without needing direct access to the hardware controllers. It provides real-time communication with the Linux-based backend server, which aggregates data from remote Raspberry Pi controllers. While the database layer (detailed in Section 3.4) manages event storage and long-term logging, the web application stack focuses on secure data exchange, visualization, and command routing through encrypted communication channels.

3.5.2 Desired Characteristics

These characteristics have been selected by the Portcullis team to provide reliable connection and maintain system security. By meeting these criteria, the system guarantees that only authorized personnel can access and command the platform without risking data integrity or availability. The API layer will serve as the controlled interface between the web dashboard and the backend server, the selected framework must also have the capability to integrate with our other technologies and satisfy the requirements provided by GDMS.

The ideal web application stack for the Portcullis system should have the following characteristics.

- **Real-Time Communication:** The web interface will provide GDMS with a centralized, realtime interface to observe door status, intrusion events, and sensor activity across multiple remote facilities.
- **Reliability and Network Resilience:** The web application must maintain continuous functionality and automatic recovery under high-latency or interrupted network conditions to ensure operation can continue without interruption. The backend shall accept delayed data uploads from field controllers to ensure all locally stored events are eventually logged in the central system. Automatic reconnection and session restoration shall occur without the need for operator input.
- **User Authentication and Role Management:** The application dashboard will enforce user authentication by role. The application shall validate user credentials through a centralized authentication process that supports session management and token expiration. All credential handling, password storage, and authorization tokens must comply with modern encryption and privacy standards.The authentication system should integrate seamlessly with the backend API and support secure communication during login and session renewal.

- **Secure Communication:** All communications between the web dashboard and backend server will be encrypted using industry standard protocols. This will ensure that sensitive information such as event logs, user credentials, and door-status information is secure and reduces exposure to potential threats. The web application will never directly communicate with door controllers, all exchange of information will be routed through the backend server to achieve this.
- **Event Retrieval:** The web application stack must support consistent deployment, configuration, and maintenance across multiple environments. The web dashboard shall request event data from the backend through authenticated interfaces that support filtering by date, location, or event type. Event data will be retrieved from the **PostgreSQL** database using secure, parameterized queries to prevent unauthorized access.

3.5.3 Alternatives

To determine the optimal method for meeting the requirements specified by GDMS, the Portcullis Team has compared and tested the following frameworks to identify the most suitable solution.

- **FastAPI:** FastAPI is an asynchronous python framework used to build and API endpoints quickly and efficiently. This framework supports WebSockets and background testing which is a critical component of the system infrastructure. FastAPI also supports OAuth2 and JTW authentication for user verification,
- **Node.js:** Node is an event-driven framework that is suitable for real time communication. Node has a vast ecosystem and is compatible with libraries such as Express and Socket.IO. Node also easily integrates TLS 1.3 for encryption and communication between services.
- **Flask:** Flask is a lightweight modular framework that is more flexible than other frameworks, Flask can be extended using third party libraries to allow for asynchronous features. Flask is known for its simplicity and is commonly used in small scale modular deployments. This design makes the flask highly adaptable but requires more configuration for complex systems.

3.5.4 Analysis

Through documentation, reviews, and personal testing of these frameworks, the team has thoroughly evaluated each framework against each other to find the best match for the Porucullis system.

FastAPI (Python)

- **Real-time Communication:** FastAPI is a python framework designed to build APIs based on Python type hints. FastAPI is a modern python framework that natively supports WebSocket connections for real-time communication between the backend and web dashboard. Evaluation is based on the official FastAPI documentation.
- **Reliability and Network Resilience:** Through the use of FastAPI's asynchronous I/O design, this framework supports background tasks and queued message handling. This allows for FastAPI to be effective throughout a network outage by allowing delayed controller data to update automatically after an outage.
- **User Authentication & Role Management:** FastAPI includes native support for OAuth2 and JWT (JSON Web Token) authentication. These services provide secure, token based access control and session management capabilities.
- **Secure Communication:** Fully compatible with TLS 1.3 and HTTPS. When deployed to a linux environment FastAPI has the ability to encrypt data transfer to and from the backend server which is a critical component of the requirements provided by GDMS
- **Event Retrieval:** FastAPI supports RESTful API endpoints that allow filtered data retrieval for event auditing and reporting. Through its built-in validation system, it safeguards against unauthorized queries and injection attacks while maintaining secure access to event logs and system records. FastAPI's architecture and compatibility with Docker allow for quick updates, versioning, and replication across multiple door access controllers.

Node.js

- **Real-time Communication:** Node.js is built on an event-driven, TypeScript-based architecture designed for creating scalable and maintainable server-side applications. According to the official Node.js documentation it natively supports WebSocket configuration with the addition of MQTT integration. This allows for a secure two way communication between the web server and calling of the API to the linux server.
- **Reliability and Network Resilience:** Through event-driven design model Node will allow the server to send and receive data instantly from the Linux backend allowing for up to date information about door access and entry attempts. Node's infrastructure is designed to handle high concurrency, enabling it to support many users and devices simultaneously without sacrificing performance.
- **User Authentication & Role Management:** For authentication Node.js integrates with Passport.js and JWT Authentication to control token access, password hashing and roll

based access. These tools provide administrators with the ability to limit access and uphold security using session based access control
- **Secure Communication:** Node.js also supports deployment over HTTPS / TLS 1.3 to encrypt communication. When Node is deployed to a Linux environment, it functions behind reverse proxies such as *Nginx* to enforce security protocols presented by GDMS.
- **Event Retrieval:** Supports structured RESTful API endpoints but also includes support for GraphQL integration. GraphQL has the ability to reduce the number of overall API calls by requesting the exact data that is needed. This is accomplished by using a single API endpoint to limit network traffic.

Flask (Python)

- **Real-time Communication:** Flask is a lightweight Python microframework that relies on extensions to provide advanced functionality. It does not include native support for WebSockets; however, by integrating extensions such as Flask-SocketIO or Flask-Sock, Flask can enable real-time, bidirectional communication between the web client and the backend server.
- **Reliability and Network Resilience:** Flask primarily operates as a synchronous framework. This means Flask can only process one request at a time. This can limit performance on high latency networks. To overcome this challenge Flask utilized tools such as Celery or message queuing to delay controller data and to synchronize after network connection is re-established.
- **User Authentication & Role Management:** Flask does not include built in features for user authentication. Tools such as Flask-Login or Flask-JWT can be used to handle authentication. This modular design allows flexibility during development but requires a lot more manual overhead compared to Node and FastAPI systems.
- **Secure Communication:** Flask supports communication over HTTP/TLS. This can be accomplished configuring SSL certificates directly within the application. In production environments Flask applications are typically served though WSGI servers such as Gunicorn which is responsible for encryption and certificate management.
- **Event Retrieval:** Flask's flexibility enables the implementation of RESTful API endpoints, allowing authorized users to access and filter event data for auditing and reporting purposes. By leveraging parameterized queries and database ORM tools like SQLAlchemy, the framework ensures secure retrieval and presentation of event logs, intrusion reports, and user access records. Its lightweight and modularity allows for easy integration with Docker for deployment in Linux environments, promoting version

control, scalability, and consistent configuration across development and production systems.

3.5.5 Chosen Approach

To evaluate the most suitable framework, a criteria table was developed to assess the effectiveness of each alternative solution identified in Section 3.5.3.

Criteria	Real-time Communication	Reliability and Network Resilience	User Authentication & Role Management	Secure Communication	Event Retrieval	Total **/25**
FastAPI	4	4	5	5	4	22
Node.js	5	5	5	5	5	25
Flask	3	3	3	4	3	16

After evaluating each option, Node.js was chosen as the optimal framework for the Portcullis web application. Node.js showed superior performance across all categories. Node has been selected for its compatibility with Socket.IO, JWT authentication, and TLS 1.3 encryption ensures that the developed system efficiently meets GDMS's operational and security requirements.

3.5.6 Proving Feasibility

To prove the feasibility of Node.js, the Portcullis Team simulated the following tests to verify findings from 3.5.4. The team conducted a controlled simulation using a locally hosted Docker based Linux environment to simulate the conditions of the chosen backend server. Within this environment we also configured our database (PostgreSQL) to accurately test the Node infrastructure by simulating 5 containerized door controllers.

Door controllers were simulated using a constant data stream that communicated door status updates to the backend via Socket.IO connections using JSON formatting. Multiple controllers were deployed as Docker containers to model how Node connects to physical door controllers.

To simulate our system under varying latency and connection conditions data was collected across door controllers to measure Node's capabilities. Each controller operated under simulated network latency ranging from 100–400 ms with an additional **±100 ms jitter** to emulate unpredictable real-world network conditions. Across a total of **170 simulation logs**, we recorded an average response time of **188 ms** and a maximum observed delay of **439 ms**, Node maintained stable communication well within the defined ≤ 400 ms tolerance specified by the project requirements. The standard deviation (**121 ms**) proves the system's ability to handle moderate jitter. Additionally, no buffered events were recorded, showing that the framework successfully maintained continuous communication without message loss or the need for manual recovery.

To verify that Node can meet the requirements for user authentication and role management, the portcullis team simulated entry logs based on user access levels. Passwords were then hashed using JWT and bcrypt to be securely stored in the PostgreSQL database.

Based on the results of simulation testing, Node.js has been verified and proves the capabilities for the Portcullis web application. It successfully fulfilled all functional and non-functional requirements. By analyzing the collected data, the system successfully demonstrated real-time communication, allowing for instant two-way data exchange between distributed door controllers and the central web dashboard through Socket.IO with consistent response times were well below the 400 ms tolerance. Finally, event retrieval and auditing were verified through PostgreSQL-backed API routes that securely returned filtered and timestamped logs, demonstrating both data integrity and persistence across container restarts.

3.6 Access Methods

3.6.1 Introduction

Security is the top priority for the GDMS requirements; our system will require a quick and easy-to-use authentication method to gain access and prevent unauthorized access to the building. This system will have a method to input user identity, and the system will quickly verify if they are listed in the database and decide whether or not to give them access. The final solution should be able to work independently of network availability, be easy to integrate, and be able to work with the embedded system.

3.6.2 Desired Characteristics

The ideal access method should have the following characteristics:

- **Security**: The tool must be entirely secure, preventing people from fraudulent imitation. Specifically, it should be resistant to cloning, spoofing, and replay attacks.

- **Authentication accuracy:** The system must be highly reliable to distinguish between authorized and unauthorized users.
- **Ease of use:** The system must be a system that every user can understand with little to no training, while offering quick responses to the central server.
- **Compatibility:** The method must be able to communicate and send all information to a Raspberry Pi 5, allowing it to communicate with all other parts of this overall system.
- **Cost Efficiency:** The chosen method should be able to handle all previous requirements while keeping costs low.

These characteristics help form the evaluation criteria for comparing different access methods.

3.6.3 Alternatives

Below are some viable options that we will judge based off our desire characteristics.

- **Keypad:** Keypads are very common but very simple authentication methods. They allow the user to input a PIN code, which the system would then use to validate their identity. They are very easy to install and add users' credentials from an admin portal, but are highly vulnerable to spoofing.
- **Biometric Scanner:** Biometric systems authenticate users from physical traits that are always unique to each person. They are highly secure and resistant to attacks, but they are vulnerable to environmental conditions and are very expensive
- **RFID Scanner:** RFID Scanners utilize radio waves/frequencies from a wireless card or fob to communicate with a system to validate identity. They are known for being highly secure and quick, especially for distributed systems. They are very simple to use and provide immediate responses.

3.6.4 Analysis

Each of these methods have been analyzed in depth with user reviews and technical reviews of security systems.

Keypad

- **Security**: Keypads are notorious for their lack of security; codes can easily be spoofed or copied by any outside observer, and giving anyone access to the system is a risk if the user does not hide their codes
- **Authentication accuracy:** Consistent, but codes can be forgotten or shared, or observed by a malicious third party.
- **Ease of use:** Familiar tool to most people, and can be easily updated on the admin portal.
- **Compatibility:** Compatible with most embedded systems via I/O connections.
- **Cost Efficiency:** Very low cost and low power consumption, and almost no maintenance.

Biometric Scanner

- **Security**: Almost impossible to spoof these systems. The nature of these systems ensures a unique identity for every person that cannot be replicated.
- **Authentication accuracy:** With a proper and high-quality version, these are very accurate, being able to identify the user without error quickly.
- **Ease of use:** Generally easy for users to understand, but can be difficult with poor environmental factors, such as dust kicked up onto a scanner.
- **Compatibility:** Is compatible with many embedded systems, but may require external drivers or libraries, which can be difficult to set up.
- **Cost Efficiency:** Very expensive and high energy consumption, and may require frequent maintenance.

RFID Scanner

- **Security**: Very secure to use, the communication between the card and reader can be easily encrypted to prevent malfeasance, and it is difficult to clone a user's credentials from the card.
- **Authentication accuracy:** Very accurate, each card has a unique ID along with the credentials of the user, which can be easily compared within the database to ensure consistent identification.
- **Ease of use:** A very familiar tool, especially within GDMS systems, and requires little to no training to understand.
- **Compatibility:** Very compatible with embedded systems, with many Raspberry Pi solutions available.
- **Cost Efficiency:** Has a moderate cost and power demand, but requires almost no maintenance.

3.6.5 Chosen Approach

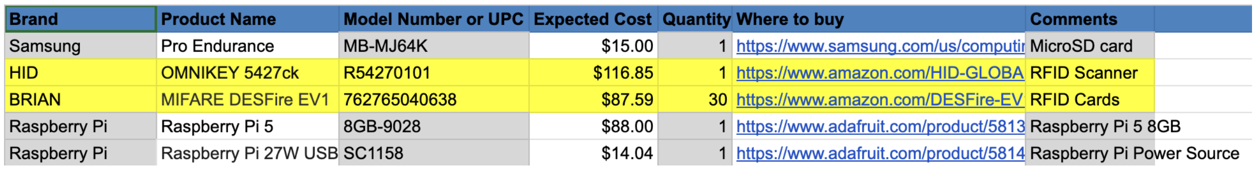
Each alternative in section 3.6.3 was compared against the desired characteristics in section 3.6.2.

Criteria	Security	Accuracy	Ease of Use	Compatibility	Cost	Total **/25**
Keypad	3	4	5	5	5	22
Biometric Scanner	5	5	4	3	2	19
RFID Scanner	5	5	5	5	4	24

After careful consideration, we have decided to use an RFID scanner for our access method. For this part of the project, the key requirement for GDMS systems is security and then cost efficiency. RFID scanners almost perfectly balance all of the desired characteristics in this piece of the project. It offers very quick responses to the system, with high security measures that prevent any bad actors from gaining access to the system. They are very intuitive, so users will not have difficulty using the system. They have great compatibility with our selected embedded system, and while they are more expensive than other options, the benefits of this system are impressive, and are more cost-efficient than other options that may be marginally more secure.

3.6.6 Proving Feasibility

To validate this selection with the GDMS requirements, we will test the RFID scanners along with the embedded system to ensure its ability to perform as expected. We will start by loading test credentials into the database and assigning cards to several of these credentials, along with some with false credentials, to ensure the system is secure, accurate, easy to use, and that it will work as we expect it to with the Raspberry Pi 5. We will also test it with a handful of different RFID scanners to test which one performs all of these tasks the quickest while still being reliable and secure. Currently, the team is requesting from the client an acquisition form to submit to GDMS. Below is the acquisition request to GDMS. We have chosen to use the HID OMNIKEY 5427ck for the development of a prototype to begin testing the reliability of simple RFID cards, and then possibly move to a more permanent installment, designed for door security if the tests prove successful.



Brand	Product Name	Model Number or UPC	Expected Cost	Quantity	Where to buy	Comments
Samsung	Pro Endurance	MB-MJ64K	$15.00	1	https://www.samsung.com/us/computir	MicroSD card
HID	OMNIKEY 5427ck	R54270101	$116.85	1	https://www.amazon.com/HID-GLOBA	RFID Scanner
BRIAN	MIFARE DESFire EV1	762765040638	$87.59	30	https://www.amazon.com/DESFire-EV	RFID Cards
Raspberry Pi	Raspberry Pi 5	8GB-9028	$88.00	1	https://www.adafruit.com/product/5813	Raspberry Pi 5 8GB
Raspberry Pi	Raspberry Pi 27W USB	SC1158	$14.04	1	https://www.adafruit.com/product/5814	Raspberry Pi Power Source

3.7 Secure Communication Protocols

3.7.1 Introduction

A core component of the door controller infrastructure is upholding the integrity, confidentiality, and availability of all communications within the Portcullis architecture. To achieve this, secure communication protocols must be implemented between the Raspberry Pi door controllers, the backend server, and the web administrator interface.

This connection is responsible for exchanging sensitive data such as authentication requests, lock/unlock commands, and access event logs. All transmissions must be encrypted using current industry standards to protect data from interception and tampering.

3.7.2 Desired Characteristics

The ideal secure communication protocol should have the following characteristics:

- **End-To-End Encryption:** All communication must be encrypted between door-controllers, backend server, and the web interface. This guarantees sensitive information is only accessible by authorized GDMS personnel.
- **Authentication:** Each communication session must verify the identity of both parties before data exchange can be made. Implementing security measures such as passwords, tokens, or digital certificates is crucial for network security. This prevents unauthorized controllers and clients from connecting to the network, submitting false data, or accessing sensitive information.
- **Secure Monitoring:** The developed system will allow for secure monitoring and reporting of network and device status. This is to be achieved via SNMPv3 or a comparable protocol. SNMPv3 protects the confidentiality of all telemetry and intrusion reports from door units. The system must be protected from interception or manipulation through encrypted message exchange and authentication.
- **Update Integrity:** The system must protect against potential threats during software and firmware and updates. Updates will be securely transferred using encrypted channels and validated before installation takes place
- **Scalability:** The security architecture must be scalable and support hundreds of connected door controllers while maintaining encryption protocols without sacrificing performance. As the number of connected devices grows, encryption key management, certificate renewal, and authentication processes must remain efficient and manageable to support long-term deployment across multiple facilities.

3.7.3 Alternatives

Below are some viable options.

- **TLS 1.3/HTTPS:** TLS 1.3 is a modern transport-layer security protocol. When paired with HTTPS it secures and encrypts communications between the backend server, web dashboard, and connected door controllers. TLS is proven to be reliable and compatible with Docker and Node.js, making it a suitable option for securing application level communication across different systems

- **DTLS 1.3:** DTLS is used to extend the security features TLS to datagram-based UDP communication. This process encrypts communications and focuses on operating during high-latency or unstable networks, making it reliable to real time data transmission
- **IPsec**: IPsec is a network-layer security framework designed to protect all data transferred between networked devices. Unlike many application level encryption methods, IPsec secures traffic directly at the IP level securing all packets between the API endpoints.

3.7.4 Analysis

To ensure the best fit with GDMS requirements, the team researched each of these protocols to ensure the system is as secure as possible. The team looked at documentation for each method along with reviews to see which one will provide the most secure communication protocols.

TLS 1.3/HTTPS

- **End To End Encryption:** TLS 1.3 is a platform that provides encrypted communication between web and API services. It uses cipher suites to confidentiality transmit data between application layers. This prevents attackers from intercepting or altering sensitive information such as lock commands, event logs, and user credentials during transmission.
- **Authentication:** TLS supports secure authentication using X.509 certificates and supports token based credentials. During the TLS handshake both the client and server will verify each other's encrypted identity before transmitting data.
- **Secure Monitoring:** While TLS itself does not monitor system status, encrypted channels can be configured to use monitoring protocols such as SNMPv3 or HTTPS telemetry.
- **Update Integrity:** HTTP connections can be secured using TLS and can be used to download and verify software or firmware updates. This process is transmitted over a secure channel that can not be modified or intercepted
- **Scalability:** TLS has been used to secure data transmissions across many modern cloud platforms and it is a viable option for docker and Node.js applications since TLS can easily manage multiple connections.

IPsec (Internet Protocol Security)

- **End To End Encryption:** IPsec is an encryption used at the network layer. This protocol secures all IP traffic between devices regardless of the application in use. IPsec uses Encapsulating Security Payload (ESP) mode, this means that IPsec encrypts all communication between door controllers and the backend server keeping data protected from eavesdropping or packet tampering.

- **Authentication:** Through the use of the Internet Key Exchange (IKEv2) protocol, devices can be authenticated using pre-shared keys and digital certificates. In production this would allow only trusted controllers to connect and communicate with other systems in the Portucullis infrastructure.
- **Secure Monitoring:** Since IPsec operates below the application layer, it automatically secures monitoring data such as SNMPv3 messages or HTTPS traffic without requiring changes to the application..
- **Update Integrity:** While IPsec encrypts data packets during transfer to protect firmware and configuration updates, application-level signing and checksum validation are still required to verify the update's authenticity.
- **Scalability:** IPsec offers strong protection, it can become complex to manage at scale. Each tunnel requires configuration and key management, which increases administrative overhead and potentially increases traffic across the network.

DTLS 1.3

- **End To End Encryption:** DTLS 1.3 supports TLS-Equilivent encryption but for UDP base communication. This allows for secure message exchange even in high-latency or lossy network environments. Each datagram is individually encrypted and authenticated.
- **Authentication:** DTLS natively supports mutual authentication using pre-shared keys, similar to other protocols mentioned above. DTLS is considered to be a lightweight handshake process and would be well suited for remote controllers.
- **Secure Monitoring:** Event messages can be securely sent and transmitted using telemetry over UDP connections. This allows for real-time monitors similar to HTTPS but with higher tolerance for packet loss.
- **Update Integrity:** Software updates are transmitted over UDP. They remain encrypted and verified through the transmission process. This provides protection against injection attacks during updates and configuration changes.
- **Scalability:** DTLA is lightweight and efficient, its reduced handshake time is scalable across hundreds of endpoints using minimal resources.

3.7.5 Chosen Approach

Each protocol presented in section 3.7.3 was evaluated against the desired characteristics defined in 3.7.2. The table below summarizes this comparison, rating each option based on its encryption strength, authentication reliability, monitoring capability, update integrity, and scalability within the Portcullis system. The scores represent each protocol's ability to integrate into our chosen system and to meet the requirements set by GDMS.

Criteria	End To End Encryption:	Authentication	Secure Monitoring	Update Integrity	Scalability	Total **/25**
TLS 1.3 / HTTPS	5	5	4	5	5	24
IPsec	5	5	5	4	3	22
DTLS 1.3	4	4	4	4	5	21

TLS 1.3/HTTPS is the most balanced and practical solution for securing Portcullis system communication. It offers strong encryption, seamless authentication, and efficient scalability in Docker and Node.js. TLS 1.3 simplifies certificate management and ensures update integrity via secure HTTPS, aligning with GDMS security requirements.

3.7.6 Proving Feasibility

To validate that HTTPS/TLS 1.3 can meet the Portcullis system's security requirements, the development team implemented a fully containerized simulation using Docker. The simulation environment consisted of a Node.js backend server, a controller-simulator service (representing Raspberry Pi door units), a PostgreSQL database, and a web interface connected through an internal Docker network. Security testing was conducted within the same simulated environment created in section 3.5.

The backend server was configured with self-signed TLS certificates generated using OpenSSL. The simulation was used to confirm encrypted communication could be maintained with latency values between 50–400 ms. In addition to latency testing, TLS handshakes averaged approximately 120ms per connection, in other words transmission was able to maintain a throughput of 30-40 messages per second. Testing allowed the portcullis team to confirm that HTTPS/TLS 1.3 is compatible with the system's containerized structure.

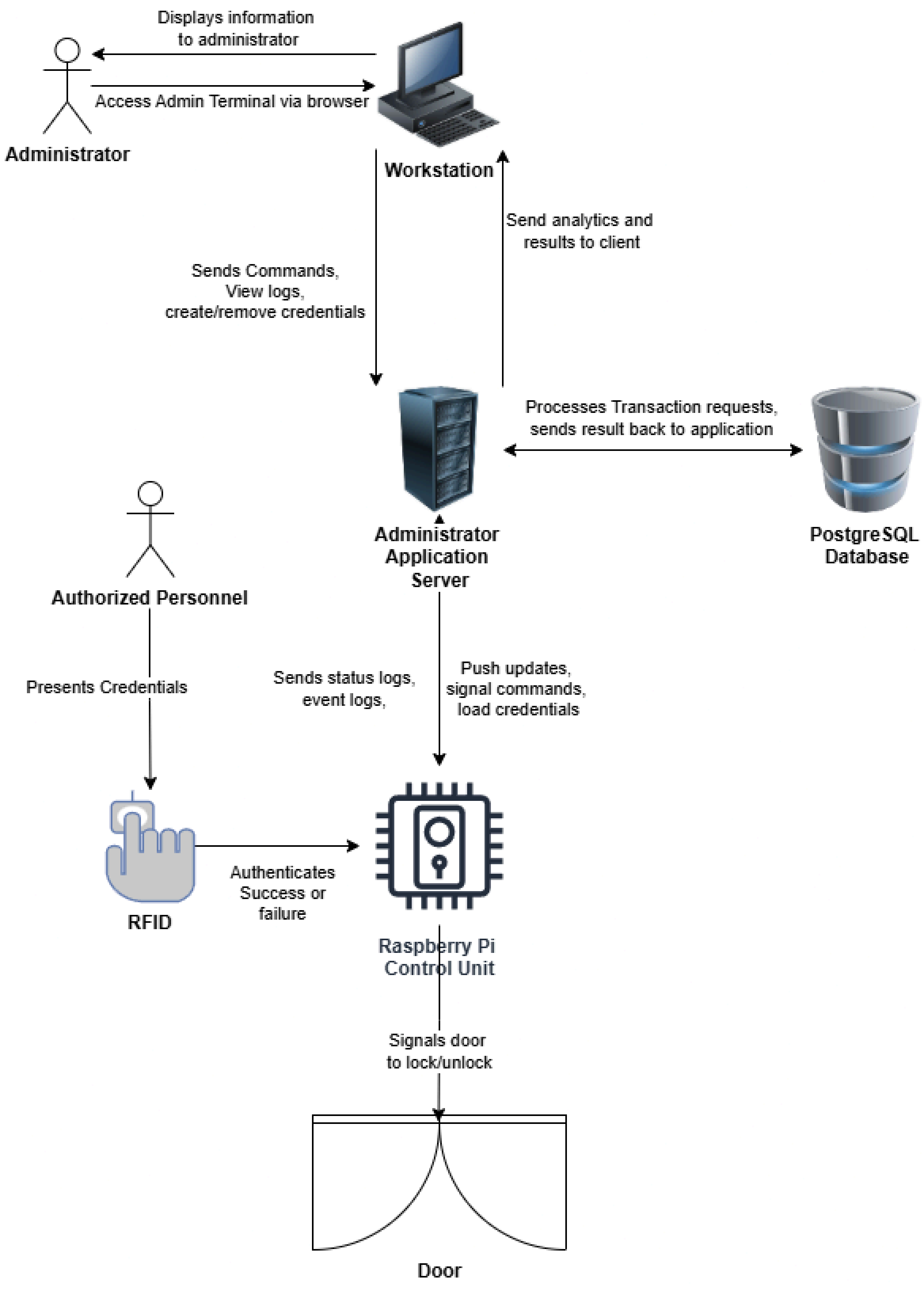
4 Technological Integration

Portcullis plans to integrate all these hardware and software components to create a unified, secure and scalable remote locking system. In this section, we will explain how each subsystem communicates and works with each other to effectively secure locations while also fulfilling requirements set out by GDMS and the Coast Guard. At the highest level, the system is composed of three domains: **Command**, **Control**, and **Intelligence**.

The **Command** domain interacts with and manages all parts of the system. Administrators of the system will do this by interfacing with the **Administrator Terminal** via web browser. The application will be a dockerized Node.js environment **,** called the **Administrator Application Server**. From the terminal, administrators can issue lock/unlock commands, register credentials , direct logs from control units to the database, and analyze system logs. The **Command** component will also use secure communication methods, **HTTPS/TLS 1.3**, to communicate with the other components.

The **Control** domain is the on-site hardware that functions to authenticate and allow access to secure facilities of the Coast Guard. This includes the **Raspberry Pi 5** unit, the **RFID scanner**, and locks themselves. When the **RFID scanner** is presented with credentials, the scanner will send this event to the **Raspberry Pi 5** control unit which will log the event. From there, validation is determined by requesting credential verification from the **Administrator Application Server**, which will communicate with the **PostgreSQL** server to determine this. If the control unit is not on the network, it will utilize the credentials in local storage to determine if the presented credentials are valid or not. If the credentials are valid, the control unit will send a signal to the latch and open the door, or in the case of invalid credentials, the control unit will not open the latch. In both cases, a log will be generated and sent to the **Administrator Application Server**.

The **Intelligence** domain encompassess all control and access data which is stored in the **PostgreSQL** server. This database is hosted in a **Docker container**, and will be responsible for storing access logs, user credentials, system events, and certificate data as well as process transaction requests from the **Administrator Application Server**. The Intelligence domain will only be accessed via requests from the **Administrator Application Server.**


Displays information to administrator
Access Admin Terminal via browser
Administrator
Workstation
Send analytics and results to client
Sends Commands, View logs, create/remove credentials
Processes Transaction requests, sends result back to application
Administrator Application Server
PostgreSQL Database
Authorized Personnel
Presents Credentials
Sends status logs, event logs,
Push updates, signal commands, load credentials
RFID
Authenticates Success or failure
Raspberry Pi Control Unit
Signals door to lock/unlock
Door

5 Conclusion

In conclusion, General Dynamics Mission Systems needs to develop a solution for their client, the U.S. Coast Guard. Portcullis proposes a solution to create a secure, scalable, maintainable, and cost-efficient system which will give GDMS more control over the maintenance of this system. We intend to create this system from the ground up using the Raspberry Pi 5 as a base to control the locks, databases to store all log data and security certificates, and all controlled using an Administrator application which can be accessed via a web browser.

Of the technologies presented and analyzed here, our team made sure to select products which can meet the needs of the Coast Guard and be able to operate in environments where attacks from hostile actors is possible and power can be cut-off at any given time. For this we selected the Raspberry Pi 5 as the basis of our system because of its affordability and modularity. Adding the Samsung microSD for local storage also improves security and operability because of its robust design. The Node.js allows for quick and reliable communication between the control units and the Administrator application even in high latency situations. A postgreSQL server will provide a long-term storage solution that can easily be maintained and scaled using Docker containerization. The system will be the foundation for allowing the Coast Guard to secure their sites, with a cost-effective and reliable system.