Stamp/Write in Red Ink: Controlled, Uncontrolled or Working Copy

DOCUMENT COVER SHEET

EP

Document Number: SDD-100.0 Rev. No.: 0 Issue Date: MAY 07 2024

Document Title: SDD for Loom (Gemini Server)

DESCRIPTION OF CHANGES: If this is a new document steps 1 through 5 need not be completed

1) This Revision /Addenda Supersedes Revision No.: ____

2) List the Paragraph / Sections Changed: ____

3) List the Reasons for the Changes: ____

4) Does this Change Affect Other Documents: ____ Yes X No

5) If 'Yes' above, list: ____

REVIEW AND APPROVALS:

Title	Printed Name	Signature	Date
Computer Engineer Prepared By:	Hung Hua	[illegible]	05/07/24
Software Engineer Technical Review By:	Hudson Topping	[illegible]	5/7/2024
 Reviewed By:			
Manager, Engineering Approved By:	Dave Morlan	[illegible]	5/7/2024

Contents

1 Introduction

1.1 Purpose

The purpose of this Software Design Description (SDD) is to provide a comprehensive blueprint for the development, deployment, and maintenance of a distributed client-server application designed to interact with hardware devices, database and data storage interfaces, and external applications. This document outlines the architectural overview, design choices, and technology stack.

1.2 Scope

The scope of this Software Design Description (SDD) encompasses the detailed design and architectural model for a distributed client-server application tailored to interact with specific hardware devices (the Gemini DAUs), database and data storage interfaces, and the external application VOTES Infinity. It covers the structural organization of the application into microservices, each responsible for distinct functionalities such as device communication, data processing, licensing verification, and interaction with VOTES Infinity, ensuring compatibility with designated versions and newer. This document delineates the responsibilities of each microservice. The SDD also outlines the security, deployment, testing strategies integral to the application's successful operation, and basic timeline for the development of the application. The intended audience includes the development team, project managers, and any stakeholders involved in the application lifecycle. The scope explicitly excludes the implementation details.

1.3 Definitions, Acronyms, and Abbreviations

The following are the definitions and abbreviations of terms used within this document:

Gemini - split acquisition system that removes the need for at-the-valve testing at Nuclear Power Plants.

Gemini network - a network whose only nodes are the ones used within Gemini for a single MOV. This network therefore will compose of only three network nodes: one node at the valve (i.e., DAU at the MOV), one node at the MCC (i.e., DAU at the MCC), and one node within Loom (i.e., one of Loom's network interfaces).

Gemini node - a network node within a Gemini network.

DAU node - a network node corresponding to a DAU (Data Acquisition Unit) within a Gemini network.

TCP/IP - Transmission Control Protocol / Internet Protocol: A protocol for sending data over Ethernet and the internet. This protocol includes full transmission error checking and retries logic.

UDP - User Datagram Protocol: A faster protocol than TCP/IP for sending data over Ethernet, but with less error checking.

DHCP – Dynamic Host Configuration Protocol: A protocol for negotiating addresses on a network.

VLAN - Virtual Local Area Network: a separate network within a physical network used for logical network separation; often used to isolate unwanted network traffic within a network.

RPC - Remote Procedure Call: a protocol used to execute functions on a remote system from a local call.

gRPC - google Remote Procedure Call: an RPC implementation by Google using protocol buffers for binary serialization.

MVC - Model View Controller: an architectural design pattern used to separate business logic from the user interface. Typically used for web applications.

PTP - Precise Time Protocol: a protocol used to synchronize clocks within a computer network, with accuracy surpassing NTP (Network Time Protocol). Algorithm specifications described in IEEE1588.

ASP.Net Core MVC - a web framework within the .NET ecosystem that uses MVC.

Seed Vitda - a vitda file generated using specific valve-related information to be used for configuring new tests using DAU data.

MasterTest - a test with all test configuration except that which will be received from the DAUs.

Docker - containerization technology

Kubernetes - container orchestration technology (often abbreviated as "K8s")

1.4 References

SRS-100.0, SRS for Loom (Gemini Server)

FDS-100.0, MOV Online Diagnostic System Functional Design Specification

QEP-100 Product Development Process

QAP-10.0, Inspection/Verification

QAP-12.0, Control of Measuring and Test Equipment

SQAP, Software Quality Assurance Plan

QEP-100, Product Development Process

QEP-103, Design Specifications

GCD-XX, Goals and Constraints for Gemini

IEEE-830, IEEE Standard related to SRS

IEEE-802.3, IEEE Standard related to Ethernet

IEEE-802.11, IEEE Standard related to WiFi

IEEE-1588, IEEE Standard related to PTP

1.5 Overview

This document provides an overview of the architectural design and development strategy for a distributed client-server application, specifically engineered to facilitate robust and secure communication with hardware devices (Gemini DAUs), database and data storage interfaces, and the external software application VOTES Infinity. It presents a blueprint encompassing microservices architecture, design patterns, and a technology stack tailored for performance, scalability, and reliability.

2 System Overview

The system is designed as a distributed client-server application, employing a microservices architecture to ensure scalability, flexibility, and resilience. It integrates closely with hardware devices using a custom protocol and interfaces with the external application VOTES Infinity, leveraging its built-in functionalities for data analysis and data processing. The system supports a broad spectrum of operations, including device communication, data processing, firmware updates, licensing verification, and secure, version-specific interactions with VOTES Infinity. It's architected to handle high volumes of concurrent web client requests efficiently, maintaining performance and reliability. Through the use of Docker (containerization) and Kubernetes (container orchestration), the system is containerized and orchestrated to facilitate ease of deployment and scaling across environments.

2.1 High-Level Architecutre

Loom shall be built using a distributed client-server architecture, structured around a set of four interfaces, each leveraging a set of microservices to achieve the requirements described in the corresponding SRS.

2.2 Design Goals

The design goals of the application are centered on creating a robust, scalable, and secure distributed client-server system that integrates with specific hardware devices and the external application VOTES Infinity. Emphasizing modularity through the usage of microservices, the application aims to ensure flexibility in development, deployment, and maintenance, facilitating easy updates and scalability. Security is also emphasized, with a comprehensive strategy to protect data integrity, authenticate and authorize users, and ensure safe communications within the system and with external interfaces. Performance optimization is targeted to handle high volumes of concurrent requests from web clients efficiently, ensuring low latency.

3 System Architecture

3.1 Architectural Design

The distributed client-server architecture will be built using independent microservices communicating over a network. The microservices exist to support the four primary interfaces that are exposed to valid hardware clients (physical devices with supported network interfaces) and valid software clients (software applications, database management systems, and web clients) that exist on the network. The application will leverage Docker for containerization and Kubernetes for container orchestration; this will ensure scalability, resiliency, modularity, validation, and ease of deployment.

3.2 System Components

3.2.1 The Four Primary Interfaces

The application will expose the following four primary interfaces to valid clients:

1. DAU Interface: an interface for DAU clients that use a customized ethernet-based communications protocol
2. Web Interface: an interface for web clients such as web browsers, including Google Chrome, Mozilla FireFox, and Microsoft Edge
3. Database Interface: an interface for database management systems such as Microsoft SQL Server
4. VOTES Infinity Interface: an interface for supported versions of the VOTES Infinity application

See Figure 3.1.

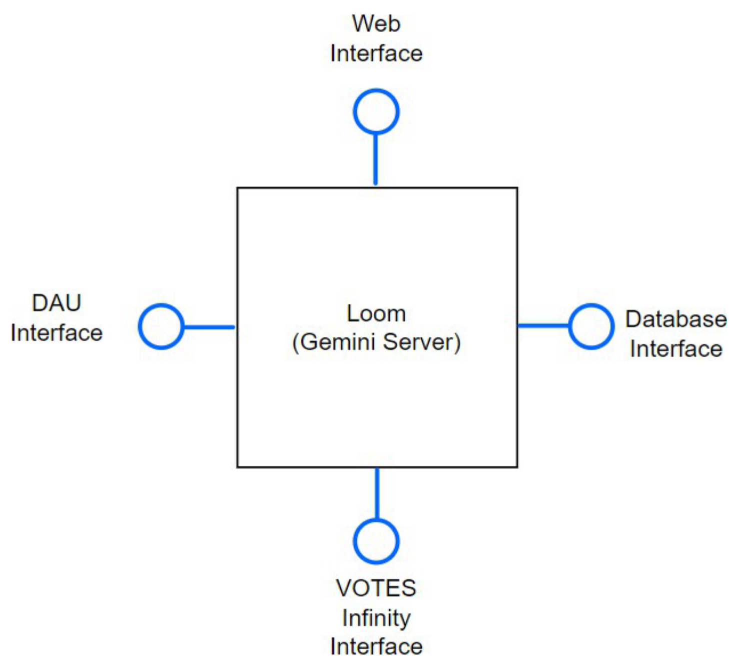


Figure 3.1: Four Primary Interfaces of Loom

3.2.2 The Four Primary Types of Microservices

Each interface leverages services to achieve the functional and nonfunctional requirements outlined in the SRS. Aside from backend microservices that perform general and supportive tasks such as logging and error handling, there will be four primary types of microservices:

1. DAU Microservices: microservices to support the DAU interface
2. Web Microservices: microservices to support the Web interface
3. Database Microservices: microservices to support the Database interface
4. VOTES Infinity Microservices: microservices to support the VOTES Infinity interface

The four types (or categories) of microservices described above will operate independently from one another. That is, each microservice in their respective category will not depend on a microservice from the other three categories. The fifth type of microservices (the backend microservices; not listed above) will consist of microservices that perform general and supportive functions, such as error handling and logging; this fifth type of microservices will also operate independently. However, the four primary microservices may depend on the microservices from this fifth type due to their general functionalities. See Figure 3.2 for the main diagram, and Figure 3.3 for the legend.

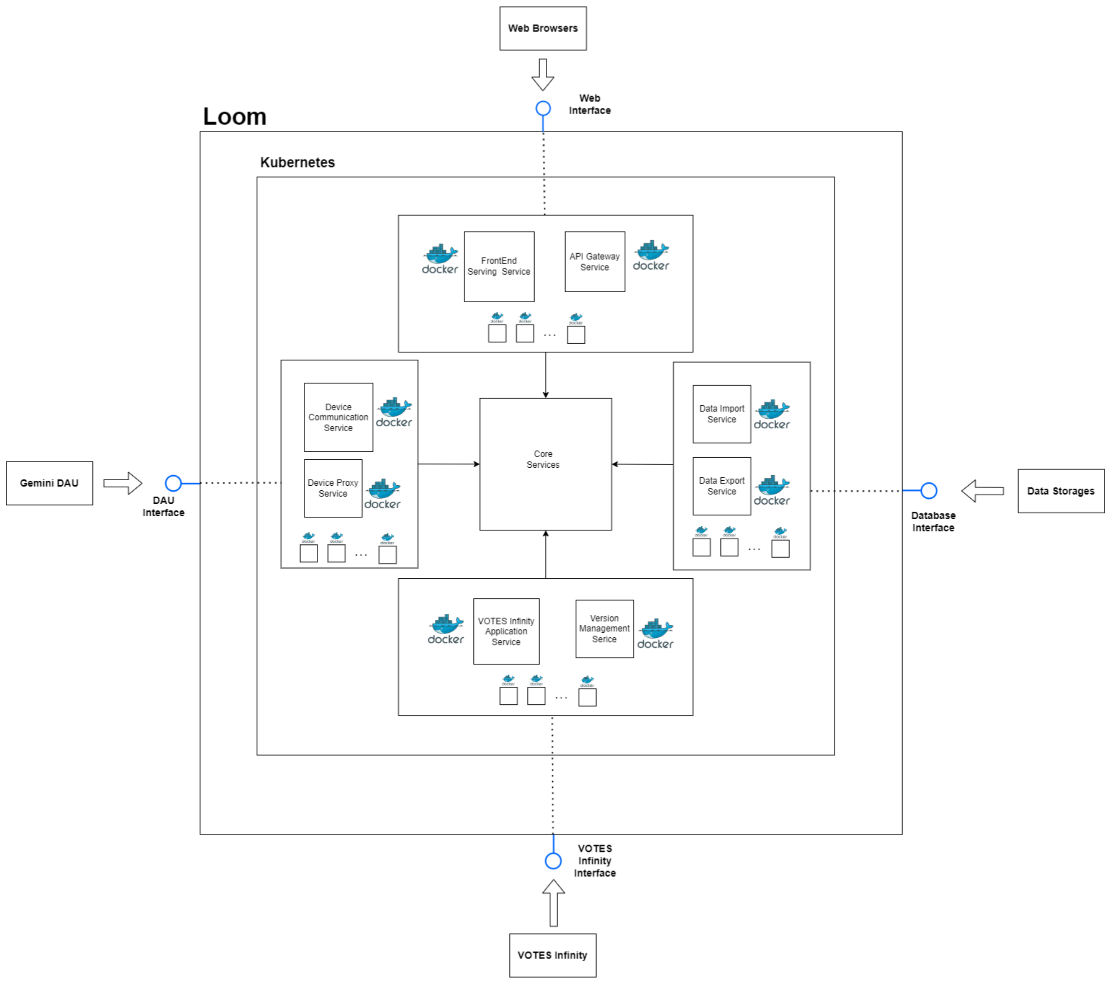


Figure 3.2: Loom: A Distributed Client-Server Application using Microservices. Leverages Docker for containerization and Kubernetes for container orchestration.

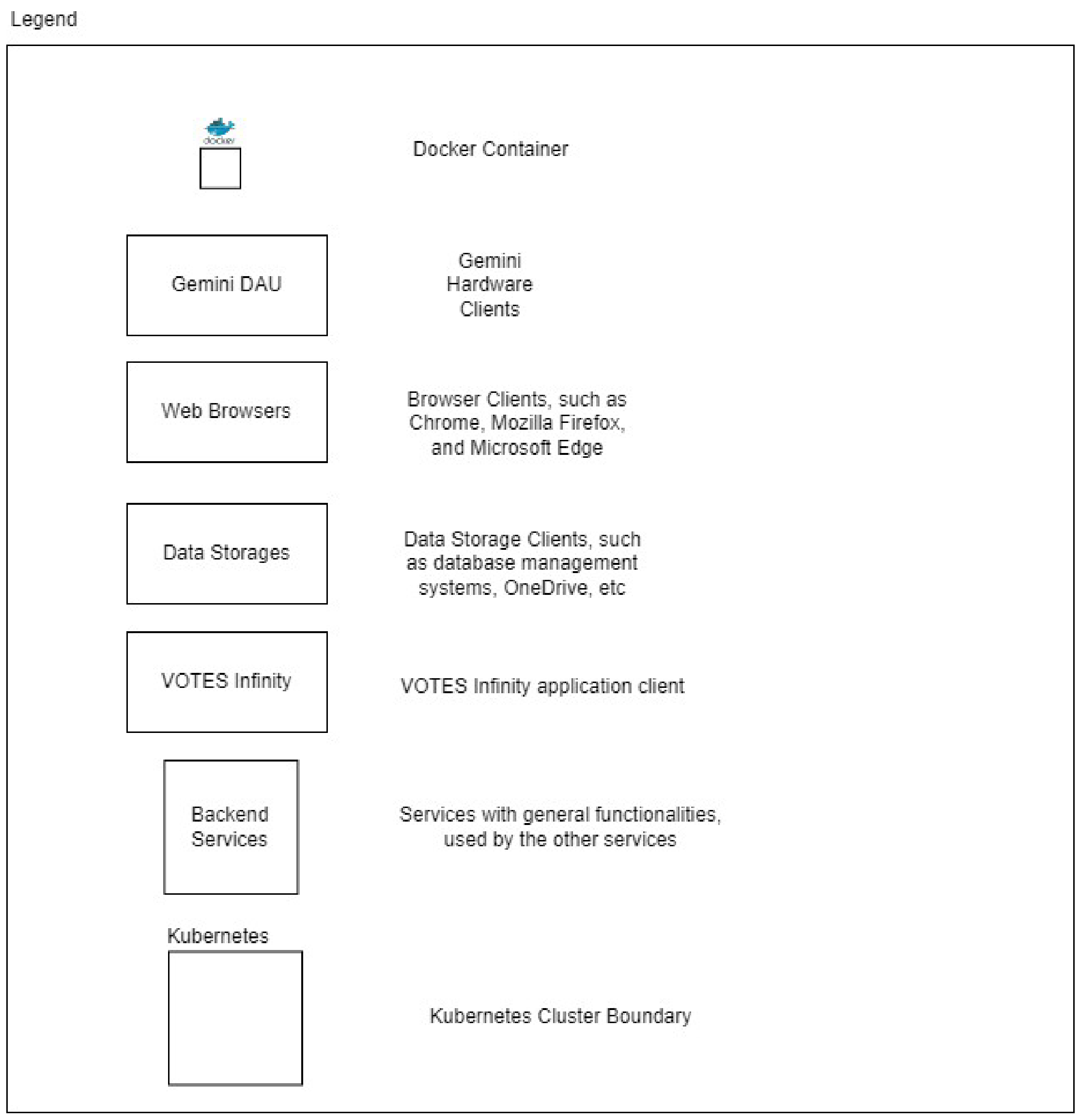
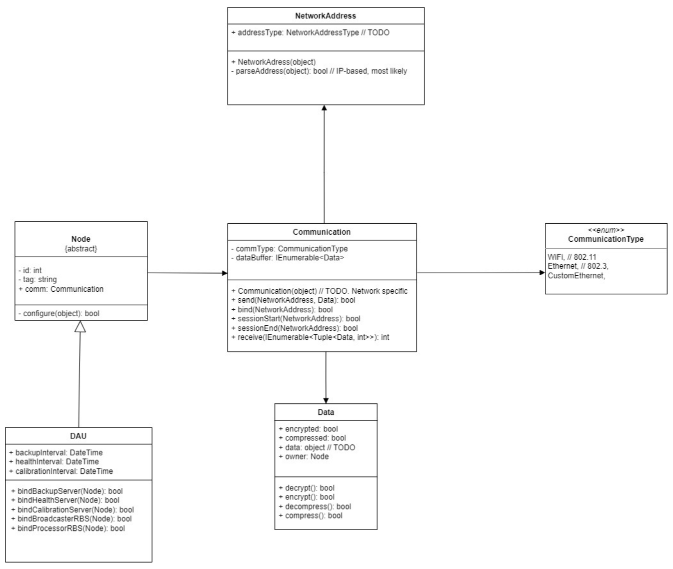

Legend
docker
Docker Container
Gemini DAU
Gemini Hardware Clients
Web Browsers
Browser Clients, such as Chrome, Mozilla Firefox, and Microsoft Edge
Data Storages
Data Storage Clients, such as database management systems, OneDrive, etc
VOTES Infinity
VOTES Infinity application client
Backend Services
Services with general functionalities, used by the other services
Kubernetes
Kubernetes Cluster Boundary

Figure 3.3: Legend for Figure 3.2.

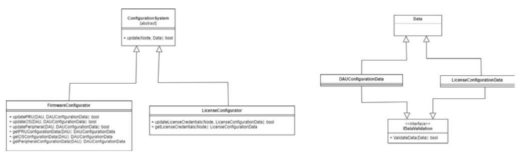
3.2.3 Component Functions

The subsection above outlines the basic types of microservices to be used, and the type of interface it supports. This subsection outlines more detail behind their functionalities. This subsection serves as a reference for the design decisions shown later.

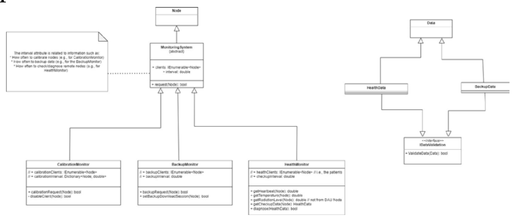
See Figure 3.4 for some of the types that will be used for the microservice implementation described in later sections.


NetworkAddress
+ addressType: NetworkAddressType // TODO
+ NetworkAdress(object)
- parseAddress(object): bool // IP-based, most likely
Node
{abstract}
- id: int
- tag: string
+ comm: Communication
- configure(object): bool
Communication
- commType: CommunicationType
- dataBuffer: IEnumerable<Data>
+ Communication(object) // TODO. Network specific
+ send(NetworkAddress, Data): bool
+ bind(NetworkAddress): bool
+ sessionStart(NetworkAddress): bool
+ sessionEnd(NetworkAddress): bool
+ receive(IEnumerable<Tuple<Data, int>>): int
<<enum>>
CommunicationType
WiFi, // 802.11
Ethernet, // 802.3,
CustomEthernet,
DAU
+ backupInterval: DateTime
+ healthInterval: DateTime
+ calibrationInterval: DateTime
+ bindBackupServer(Node): bool
+ bindHealthServer(Node): bool
+ bindCalibrationServer(Node): bool
+ bindBroadcasterRBS(Node): bool
+ bindProcessorRBS(Node): bool
Data
+ encrypted: bool
+ compressed: bool
+ data: object // TODO
+ owner: Node
+ decrypt(): bool
+ encrypt(): bool
+ decompress(): bool
+ compress(): bool

(a) Base Types


ConfigurationSystem
{abstract}
+ update(Node, Data): bool
FirmwareConfigurator
+ updatePRU(DAU, DAUConfigurationData): bool
+ updateOS(DAU, DAUConfigurationData): bool
+ updatePeripheral(DAU, DAUConfigurationData): bool
+ getPRUConfigurationData(DAU): DAUConfigurationData
+ getOSConfigurationData(DAU): DAUConfigurationData
+ getPeripheralConfigurationData(DAU): DAUConfigurationData
LicenseConfigurator
+ updateLicenseCredentials(Node, LicenseConfigurationData): bool
+ getLicenseCredentials(Node): LicenseConfigurationData
Data
DAUConfigurationData
LicenseConfigurationData
<<interface>>
IDataValidation
+ ValidateData(Data): bool

(b) Configuration Types


Node
MonitoringSystem
{abstract}
+ clients: IEnumerable<Node>
+ interval: double
+ request(Node): bool
The interval attribute is related to information such as:
* How often to calibrate nodes (e.g., for CalibrationMonitor)
* How often to backup data (e.g., for the BackupMonitor)
* How often to check/diagnose remote nodes (e.g., for HealthMonitor)
CalibrationMonitor
// + calibrationClients: IEnumerable<Node>
// + calibrationInterval: Dictionary<Node, double>
+ calibrationRequest(Node): bool
+ disableClient(Node): bool
BackupMonitor
// + backupClients: IEnumerable<Node>
// + backupInterval: double
+ backupRequest(Node): bool
+ setBackupDownloadSession(Node): bool
HealthMonitor
// + healthClients: IEnumerable<Node> // i.e., the patients
// + checkupInterval: double
+ getHeartbeat(Node): double
+ getTemperature(Node): double
+ getRadiationLevel(Node): double // not from DAU Node
+ getCheckupData(Node): HealthData
+ diagnose(HealthData): bool
Data
HealthData
BackupData
<<interface>>
IDataValidation
+ ValidateData(Data): bool

(c) Monitoring Types

Figure 3.4: UML Diagram Displaying basic relationships among the types that will be used for some of the microservices shown in later sections. Refined further in later subsections that adresses specific design patterns used.

1. DAU Microservices:
   - Interacts with Gemini DAUs using a custom Ethernet-based protocol
   - Converts raw data it acquires from the Gemini DAUs to generate valid signal trace that can get stored in a format usable by the VOTES Infinity application
   - Deliver firmware updates to the Gemini DAUs
   - Perform a health monitoring function according to a predefined schedule
   - Perform calibration according to a procedure and predefined schedule
   - Verify valid licenses
   - Handle and process trigger events
   - Enable or disable Gemini DAUs
   - Handles authentication for communication with Gemini DAUs
   - Handles authorization for communication with Gemini DAUs

- Communicate using payloads with binary serialization.
- Provide a proxy service for interaction among Gemini DAUs

2. Web Microservices:
   - Provides a user interface that can be used to configure and set the following parameters and settings:
     – Device ID
     – Trigger setup parameters for event detection
     – Data acquisition time durations for pre and post triggers
     – Fault detection settings
     – Communications settings
     – Transmission status
     – Security settings
     – Auto update configurations
     – Health monitoring schedules
     – Calibration schedules
   - Routes traffic from FrontEnd to necessary backend functions and services.
   - Handles authorization for communication with web clients
   - Handles authentication for communication with web clients
   - supports common web browsers, such as Google Chrome, Mozilla FireFox, Microsoft Edge, and DuckDuckGo
   - Utilizes the ASP.NET Core framework
3. Database Microservices:
   - Provides a data export service
   - Provides a data import service
   - Provides a data processing service
   - Interacts with Microsoft SQL Server
   - Interacts with accessible mounted drives
4. VOTES Infinity Microservices:
   - Invoke remote procedure calls with valid VOTES Infinity applications

4 Detailed Design

This chapter provides the most important design details for implementation, but implementation specific source code will not be provided. The high-level architecture will be reiterated for completeness, and both the design patterns and technology stack used will be addressed.

The application shall be built using a distributed client-server architecture that employs a set of microservices to support four interfaces exposed by the application. The application shall use Docker for containerization and Kubernetes for container orchestration. The supported four interfaces are:

1. DAU Interface
2. Web Interface
3. Database Interface
4. VOTES Infinity Interface

See prior sections and the corresponding SRS for functional and nonfunctional requirements that these four interfaces shall meet.

Each interface shall be supported by two types of microservices: (1) microservices related solely to that specific interface, and (2) backend microservices that provides general functions, such as error handling and logging. As mentioned in prior sections, the first type of microservices will only depend on (1) microservices of that same type, and (2) the backend microservices. This allows for the major components of the application to be modular, independently deployable, loosely coupled, easy to test and validate, and resilient for the distributed client-server model.

4.1 Microservices Communication

The primary mechanism used for microservice communication shall be gRPC, which uses Protocol Buffers as the interface definition language and HTTP/2 for transport. This RPC framework offers high performance and efficiency, and supports bidirectional streaming and multiplexing (multiple requests over a single connection). It also has built-in support for authentication using TLS. In addition, since the Docker and Kubernetes stack will be the core mechanism that underlies the distributed aspects of the application, it would be beneficial to use a mechanism that each of these technologies support; gRPC is such a mechanism.

4.2 Microservices for DAU Interface

This subsection lists the microservices to be used for (and by) the DAU interface. We list the microservices below, a short description of their purpose, and the recommended design pattern to enforce:

- Device Communication Service: handles direct communication with DAUs using a custom Ethernet-based protocol. Includes establishing connections, sending commands, and receiving data from DAUs. Uses the Command Pattern. See Figure 4.1.

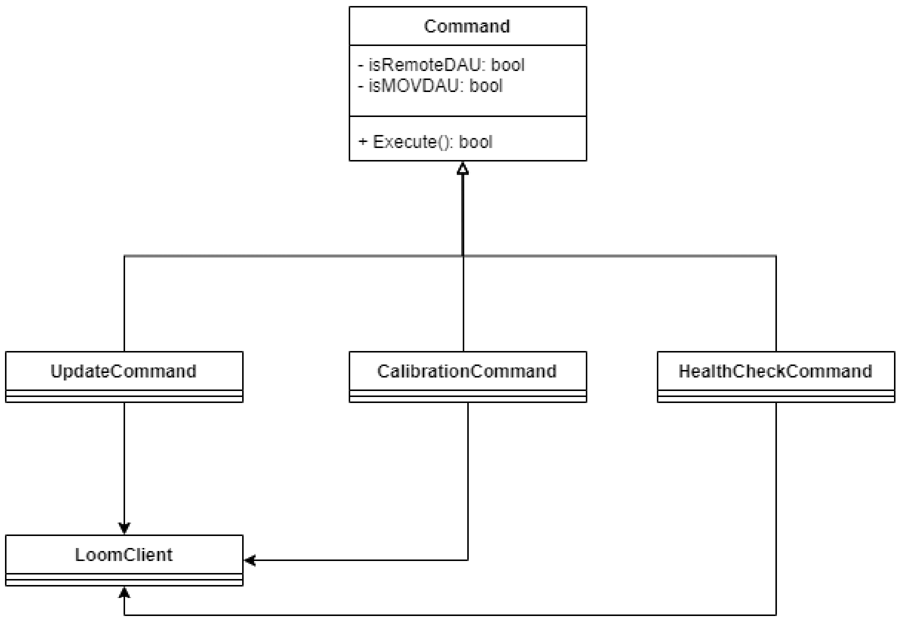


Figure 4.1: Device Communication Service, Basic

- Firmware Update Service: manages firmware updates for DAUs. Handles version tracking, update distribution, and update verification to ensure the latest firmware is being used. Uses the Factory Pattern. See Figure 4.2.

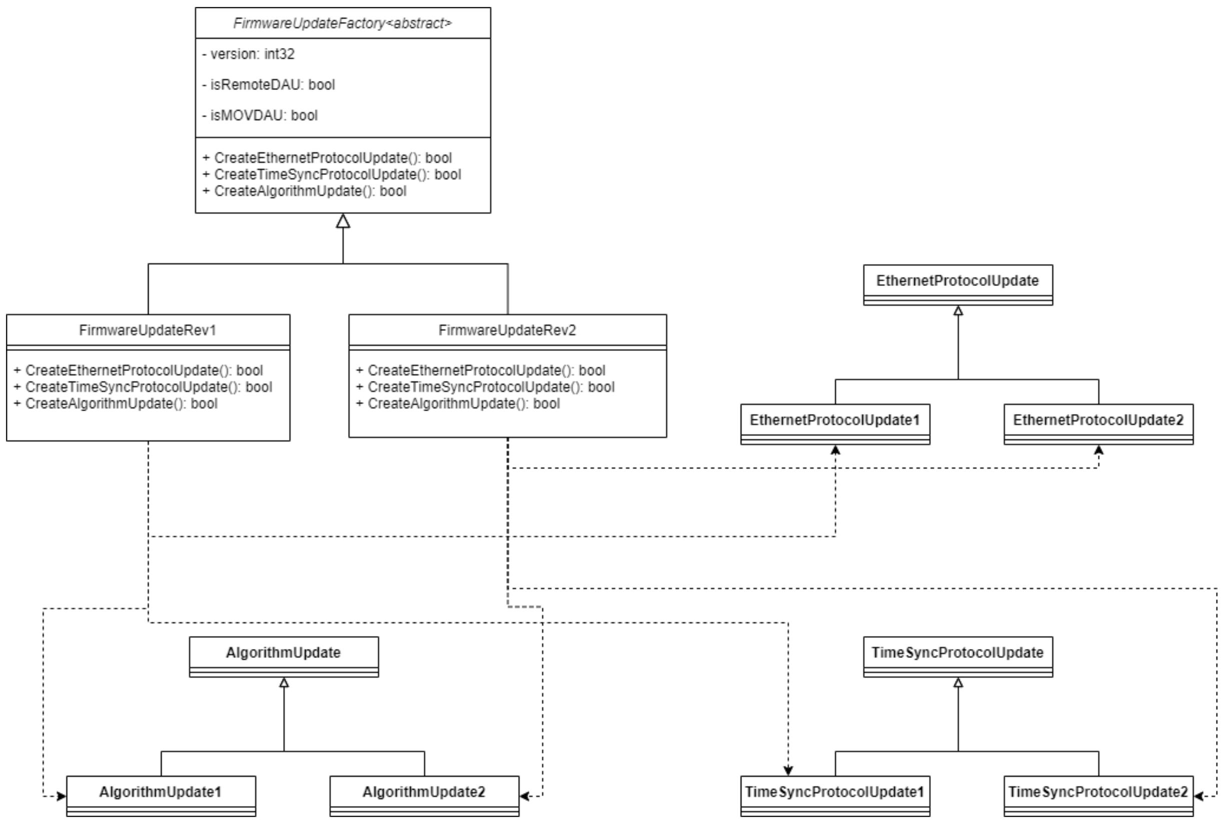


Figure 4.2: Firware Update Service, Basic

- Device Proxy Service: acts as a mediator for hardware-to-hardware communication. This service would enforce security policies and ensure that communication between devices is properly authenticated and authorized, routing messages as needed. This service is needed for the cases where one DAU needs to send information (such as event triggers) to another DAU; in these cases the information is sent (indirectly) from the source DAU to the target DAU through this service. Uses the Proxy Pattern. See Figure 4.3.

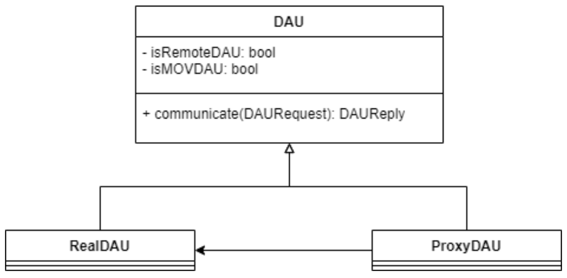


Figure 4.3: Device Proxy Service, Basic

- Data Aggregation Service: weaves multiple sensor data together, potentially transforming raw data into a more meaningful format for further processing or visualization. See Observer Pattern.

- Configuration Management Service: allows for the configuration of DAUs, such as the modification of update alerts, frequency of health checks, and threshold/parameter adjustements. Uses the Strategy Pattern.

- License Verification Service: manages the verification of licenses for each device, ensuring that only devices with valid licenses can access certain functionalities or data. Uses Decorator Pattern. See Figure 4.4.

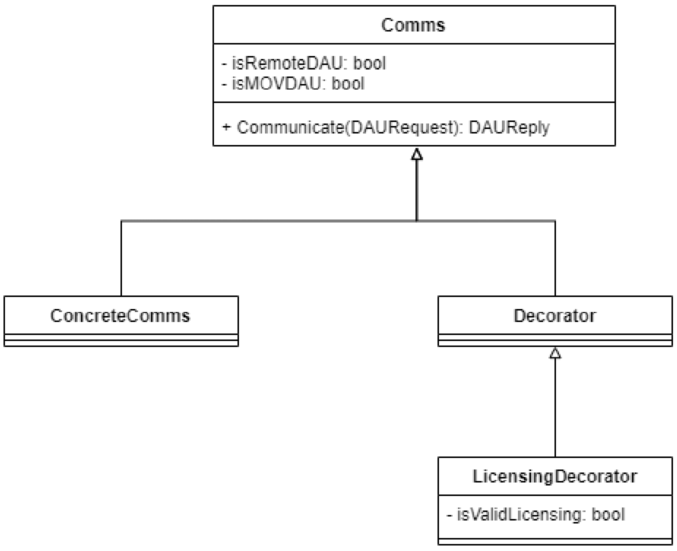


Figure 4.4: License Verification Service, Basic

- Trigger Event Handling Service: responsible for triggering actions based on the events. Uses the Observer Pattern.

- Device Control Service: allows for enabling or disabling devices remotely. This service can change the operational state of a device based on commands from the application or according to certain rules or schedules. It is better to keep this as a separate service rather than lumping it with the Configuration Management Service for improved security and robustness. Uses the State Pattern.

- Calibration Service: handles the calibration of DAUs, ensuring accurate measurements and operations. This could involve managing calibration parameters, schedules, and storing calibration results. Uses the Strategy Pattern. See Figure 4.5.

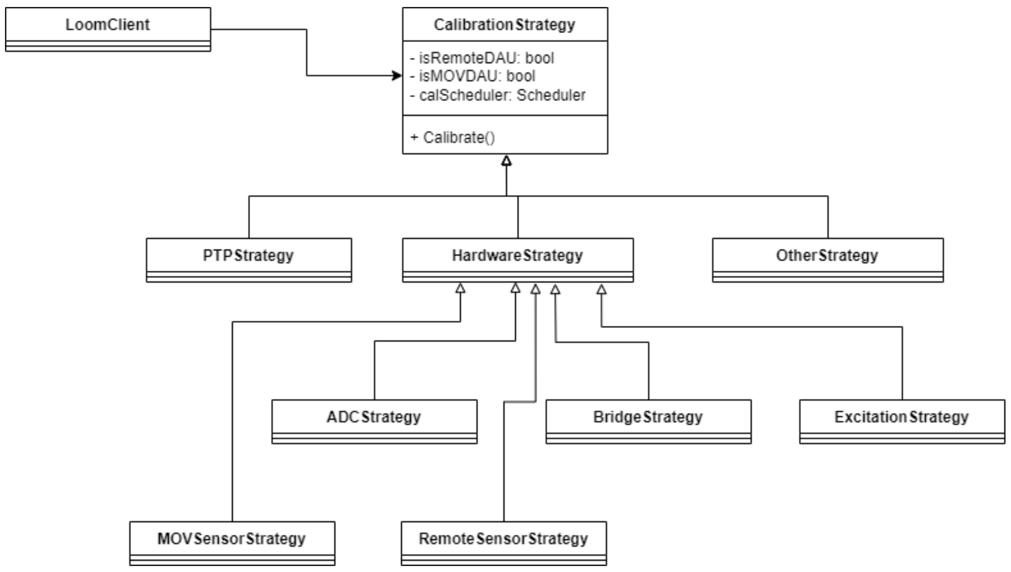


Figure 4.5: Calibration Service, Basic

The implementation of the microservices listed above will be achieved using the following technologies:

- .NET 8: cross-platform framework for building applications
- Docker: containerization technology
- Kubernetes: container orchestration technology
- gRPC: high performance RPC framework for service-to-service communication
- MySQL: database management system used to store configuration data, firmware versions, cached aggregated sensor data
- Prometheus: tool for comprehensive monitoring of microservice performance and health (Kubernetes provides such a feature already, but primarily at the level of the pod, rather than at the level of the service internals)
- Coravel: scheduler to manage tasks such as license verification checks, scheduled device calibrations, or enabling/disabling devices at specified times
- HashiCorp Vault: managing and protecting sensitive information, such as calibration parameters, licensing information, and device configuration details
- Doxygen: documentation generator; it parses source code and comments from various programming languages (C, C++, C# included) to generate readable documentation in various formats (LaTeX, HTML, XML, etc.)

4.3 Microservices for Web Interface

This subsection lists the microservices to be used for (and by) the Web interface. We list the microservices below, a short description of their purpose, and the recommended design pattern to enforce:

- Frontend Serving Service: serves the static files (HTML, CSS) the web interface (dynamic content handled by the Content Management Service). It acts as the entry point for all user interactions. Uses the Singleton Pattern and Repository Pattern.
- API Gateway Service: handles requests from the frontend and routes them to the appropriate backend services. It can aggregate results from multiple services and provides a single point of access for the frontend. Uses the Strategy Pattern and Command Query Responsibility Segregation Pattern.
- Authentication and Authorization Service: managers user identities, authentication processes, and permissions. Ensures that users can securely access the web application and that their interactions are within their allowed roles and rights. Uses the Strategy Pattern and Singleton Pattern.
- User Preferences and Session Management Service: stores and retrieves user-specific settings and session states. Personalizes content and remembering user interactions. Uses the Repository Pattern and Singleton Pattern.
- Content Management Service: manages dynamic content that the web server displays. Uses the Repository Pattern and MVC Pattern.

The implementation of the microservices listed above will be achieved using the following technologies:

- Docker: containerization technology
- Kubernetes: container orchestration technology
- ASP.NET Core: framework for building the web server's backend. Supports MVC and Web API development patterns.
- Blazor: building dynamic and interactive Web UIs using C# rather than JavaScript
- OpenIddict: standard-compliant OAuth2.0/OpenID Connect framework for ASP.NET Core, used for authentication and authorization
- Entity Framework Core: for data access, works with MySQL database management system
- MySQL: database management system used for the various services such as User Preferences and Session Management Service. Supports the services enforcing Repository Pattern.

- Prometheus: tool for comprehensive monitoring of microservice performance and health (Kubernetes provides such a feature already, but primarily at the level of the pod, rather than at the level of the service internals)
- Grafana: visualization tool, typically coupled with Prometheus, for user-friendly displays and interactions
- Doxygen: documentation generator; it parses source code and comments from various programming languages (C, C++, C# included) to generate readable documentation in various formats (LaTeX, HTML, XML, etc.)

4.4 Microservices for Database Interface

This subsection lists the microservices to be used for (and by) the Database interface. We list the microservices below, a short description of their purpose, and the recommended design pattern to enforce:

- Data Import Service: creates new MasterTest data from seed data that exists in the .vitda file formats. Uses Builder Pattern.
- Data Export Service: exports formatted file format or in appropriate versions to a database server. Uses Adapter Pattern and Strategy Pattern.
- Data Processing Service: processes the weaved sensor data before sending it to external databases; this includes filtering, aggregation, and transformation operations (e.g., scaling, unit conversions, etc.). Uses the Chain of Responsibility Pattern and Observer Pattern.

The implementation of the microservices listed above will be achieved using the following technologies:

- .NET 8: cross-platform framework for building applications
- Docker: containerization technology
- Kubernetes: container orchestration technology
- Entity Framework Core: for cases where interacting (e.g., facilitating the creation and export of data models) with SQL databases (as opposed to flat directories) is necessary
- Apache Kafka: performant and scalable management of data streams and data processing pipelines
- Hangfire: for background processing tasks such as data imports and exports that can be scheduled or triggered based on events
- Doxygen: documentation generator; it parses source code and comments from various programming languages (C, C++, C# included) to generate readable documentation in various formats (LaTeX, HTML, XML, etc.)

4.5 Microservices for VOTES Infinity Interface

This subsection lists the microservices to be used for (and by) the VOTES Infinity interface. We list the microservices below, a short description of their purpose, and the recommended design pattern to enforce:

- VOTES Infinity Application Service: serves as the primary bridge between Loom and the VOTES Infinity application, handling all interactions, including data sending for processing and invoking VOTES Infinity's internal functions. Used the Adapter Pattern.
- Version Management Service: managers and validates the version compatibility of VOTES Infinity before any interaction is initiated. Uses the Proxy Pattern.
- Licensing Verification Service: responsible for verifying the licensing status of the application or the user before allowing any interaction with VOTES Infinity. This service checks if the user or the system has a valid license to access VOTES Infinity functionalities and processes requests accordingly. Uses Decorator Pattern.
- Data Processing and Transformation Service: prepares and transforms data to meet VOTES Infinity's expected formats for processing, potentially aggregating or transforming data before sending it to VOTES Infinity. Uses Strategy Pattern.

The implementation of the microservices listed above will be achieved using the following technologies:

- .NET 8: cross-platform framework for building applications
- Docker: containerization technology
- Kubernetes: container orchestration technology
- gRPC: high performance RPC framework for service-to-service communication
- Doxygen: documentation generator; it parses source code and comments from various programming languages (C, C++, C# included) to generate readable documentation in various formats (LaTeX, HTML, XML, etc.)

4.6 Summary of Technologies Used

The technologies listed above for the various types of microservices overlapped for a few types of microservices. We reiterate this list by summarizing the technologies in a single list here:

- .NET 8: cross-platform framework
- Docker: containerization technology

- Kubernetes: container orchestration technology
- gRPC: an RPC framework
- MySQL: lightweight database management system
- Prometheus: microservice monitoring
- Coravel: scheduler for periodic checks
- HashiCorp Vault: managing sensitive information
- Doxygen: documentation generator by parsing source code and comments
- ASP.NET Core: web framework
- Entity Frameworks Core: framework for database interaction using .NET objects
- Grafana: visualization tool for application/service monitoring
- Apache Kafka: efficient management of data streams and pipelines
- Hangfire: handler for background processes

5 Security

This chapter describes the security measures in place for Loom. See Figure 5.1.

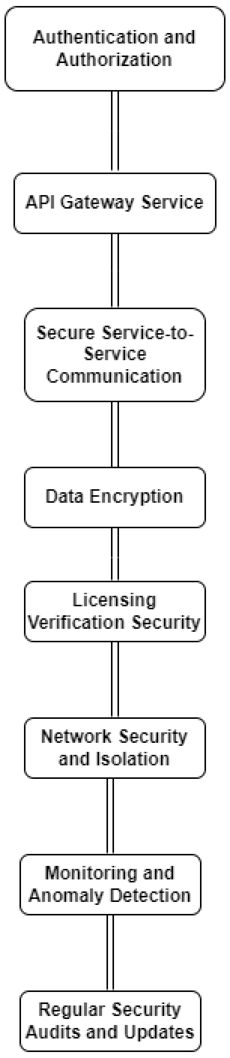

Authentication and Authorization
API Gateway Service
Secure Service-to-Service Communication
Data Encryption
Licensing Verification Security
Network Security and Isolation
Monitoring and Anomaly Detection
Regular Security Audits and Updates

Figure 5.1: Security Measures

5.1 Authentication and Authorization

The Authentication and Authorization Service from the Web Microservices shall achieve authentication and authentication through token-based authentication for services and users.

5.2 API Gateway Security

The API Gateway Service from the Web Microservices shall act as the entry point for all inbound requests and will enforce TLS termination to ensure encrypted communication.

5.3 Secure Service-to-Service Communication

The services shall communicate over secure channels using TLS since gRPC supports it already.

5.4 Data Encryption

All data will be encrypted at rest and in transit. AES-256 will be used for data at rest, and TLS will be used for data in transit, utilizing AES-128 underneath.

5.5 Licensing Verification Security

The Licensing Verification Service will securely store and manage licensing information, implementing secure API calls to licensing servers with proper authentication.

5.6 Network Security and Isolation

Kubernetes network policies will be configured to isolate microservices from each other, allowing only necessary communication paths.

5.7 Monitoring and Anomaly Detection

Logging, monitoring, and anomaly detection will be used to identify and respond to security incidents in real-time. Several services outlined in prior sections will be using Prometheus for the exact purpose.

5.8 Regular Security Audits and Updates

Regular security audits of codebases, dependencies, and infrastructure will be performed.

6 Quality Assurance

This chapter describes the Quality Assurance (testing and validation) to be used for Loom. See Figure 6.1.

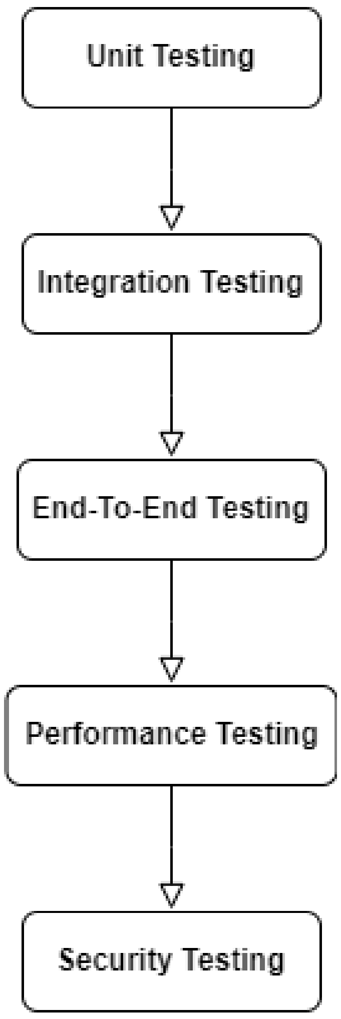


Figure 6.1: Testing Strategy

6.1 Unit Testing

Unit Testing will be used to test individual functions or methods within the microservices outlined above, ensuring proper behavior in an isolated (non-interacting) environment.

6.2 Integration Testing

Integration Testing will be used to test the interaction(s) among the microservices and its dependencies. As these microservices will be inside Docker containers, Docker Compose will be the tool of choice for Integration Testing.

6.3 End-to-End Testing

End-to-End Testing will be used to test the system as a whole, simulating real user scenarios from start to finish.

6.4 Performance Testing

Performance Testing will be used to measure the system's performance, scalability, and reliability under varying loads.

6.5 Security Testing

Security Testing will be used to identify the vulnerabilities in the application in various scenarios, from the web interface to the underlying microservices and infrastructure.

7 Deployment

This chapter describes the strategy for the deployment of Loom. See Figure 7.1.

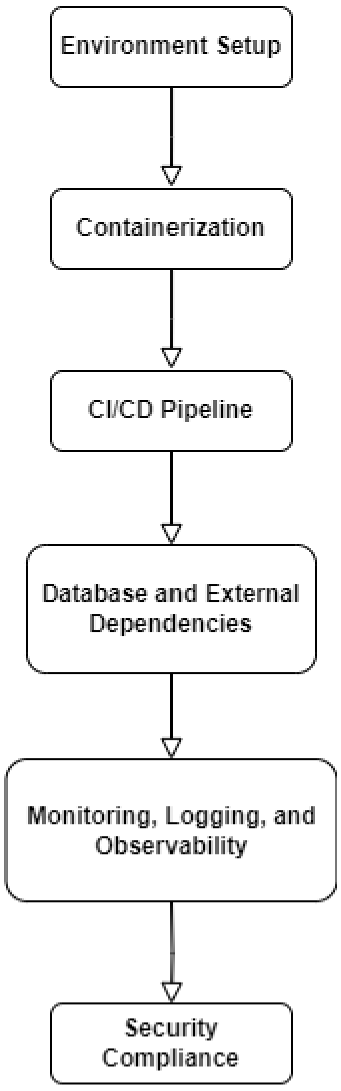

Environment Setup
Containerization
CI/CD Pipeline
Database and External Dependencies
Monitoring, Logging, and Observability
Security Compliance

Figure 7.1: Deployment Strategy

7.1 Environment Setup

A common environment will be used for all deployment purposes. Accordingly, local setups or shared environments will mimic production as closely as possible. Docker and Kubernetes will be used extensively for this purpose.

7.2 Containerization

Each microservice will be packed into its Docker container. Dockerfiles will be defined for each microservice, specifying the base images, environments, dependencies, and build instructions.

7.3 CI/CD Pipeline

Git with Azure DevOps will be used for the Continuous Integration and Continuous Deployment Pipelines.

7.4 Database and External Dependencies

Databases will be prepared and configured as needed by the application. Managed database services will be used for scaling and maintenance.

7.5 Monitoring, Logging, and Observability

Along with pod monitoring using Kubernetes, Prometheus and Grafana will be used to track the health and performance of all microservices. Log files from all services will be aggregated for analysis and troubleshooting.

7.6 Security Compliance

Security measures will be implemented and verified. Kubernetes will configure the network policies, sensitive information maangement will be handled by HashiCorp Vault, and all communications will be encrypted using TLS.

7.7 Rollout Strategy

The Blue-Green Deployment strategy will used to minimize downtime. Accordingly, two identical environments will be run, with only one serving production.

7.8 Post-Deployment

A feedback loop will be established. This feedback loop will have monitoring and alerting to quickly identify and address any issues that arise post-deplyment.

8 Summary of Services

This chapter provides a short summary of the microservices used. A short description and recommended design pattern is also provided for each microservice.

8.1 DAU Microservices

This section provides a summary of the DAU Microservices. See Table 8.1.

Microservice	**Short Description**	**Design Pattern**
Device Communication Service	custom protocol	Command
Firmware Update Service	firmware updates	Factory
Device Proxy Service	proxy for DAU to DAU comms	Proxy
Data Aggregation Service	weave separate sensor data	Observer
Configuration Management Service	configure DAUs	Strategy
License Verification Service	license verification	Decorator
Trigger Event Handling Service	event triggers	Observer
Device Control Service	enable/disable DAUs	State
Calibration Service	calibrate DAUs	Strategy

Table 8.1: DAU Microservices

8.2 Web Microservices

This section provides a summary of the Web Microservices. See Table 8.2.

Microservice	**Short Description**	**Design Pattern**
Frontend Serving Service	web frontend; user entrypoint	Singleton, Repository
API Gateway Service	routes frontend reqs to backend	Strategy, CQRS
AA Service	authentication and authorization	Strategy, Singleton
User Pref. and Sess. Mgmt Service	personalizes content	Repository, Singleton
Content Management Service	dynamic content	Repository, MVC

Table 8.2: Web Microservices

8.3 Database Microservices

This section provides a summary of the Database Microservices. See Table 8.3.

Microservice	**Short Description**	**Design Pattern**
Data Import Service	data imports	Builder
Data Export Service	data exports	Adapter, Strategy
Data Processing Service	data processing	CR, Observer

Table 8.3: Database Microservices

8.4 VOTES Infinity Microservices

This section provides a summary of the VOTES Infinity Microservices. See Table 8.4.

Microservice	**Short Description**	**Design Pattern**
VI Application Service	VI integration	Adapter
Version Mgmt Service	version mgmt	Proxy
Licensing Verif. Service	license verif.	Decorator
Data Processing and Transf. Service	VI format converter	Strategy

Table 8.4: VOTES Infinity Microservices